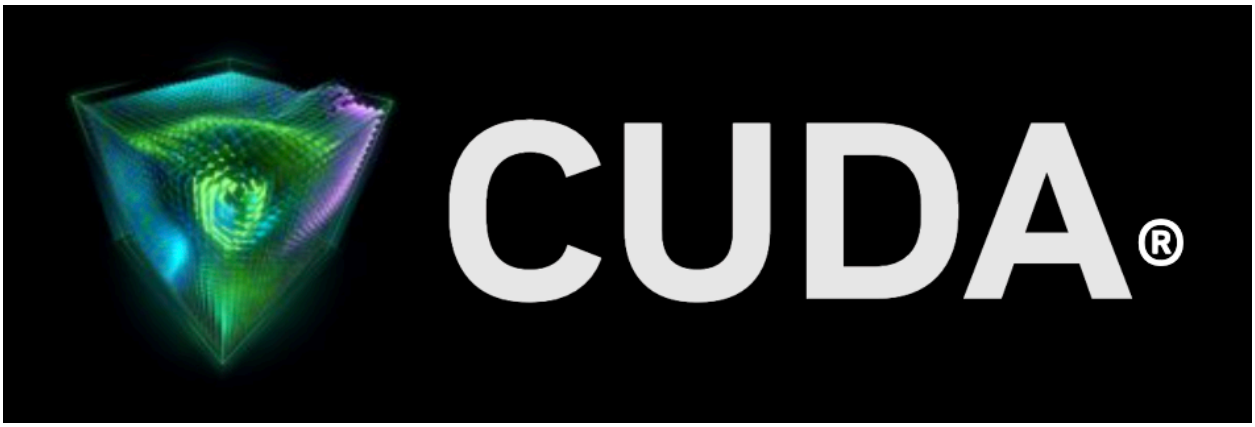


CUDA C++ Programming Guide

Release 13.4

NVIDIA Corporation

Oct 12, 2026

Contents

1 Overview **1**

2 What Is the CUDA C Programming Guide? **3**

3 Introduction **5**
3.1 The Benefits of Using GPUs . . . 5
3.2 CUDA®: A General-Purpose Parallel Computing Platform and Programming Model . . . 6
3.3 A Scalable Programming Model . . . 6

4 Changelog **9**

5 Programming Model **11**
5.1 Kernels . . . 11
5.2 Thread Hierarchy . . . 12
5.2.1 Thread Block Clusters . . . 14
5.2.2 Blocks as Clusters . . . 16
5.3 Memory Hierarchy . . . 16
5.4 Heterogeneous Programming . . . 18
5.5 Asynchronous SIMT Programming Model . . . 18
5.5.1 Asynchronous Operations . . . 18
5.6 Compute Capability . . . 20

6 Programming Interface **23**
6.1 Compilation with NVCC . . . 23
6.1.1 Compilation Workflow . . . 24
6.1.1.1 Offline Compilation . . . 24
6.1.1.2 Just-in-Time Compilation . . . 24
6.1.2 Binary Compatibility . . . 25
6.1.3 PTX Compatibility . . . 25
6.1.4 Application Compatibility . . . 25
6.1.5 C++ Compatibility . . . 27
6.1.6 64-Bit Compatibility . . . 27
6.2 CUDA Runtime . . . 27
6.2.1 Initialization . . . 28
6.2.2 Device Memory . . . 28
6.2.3 Device Memory L2 Access Management . . . 32
6.2.3.1 L2 Cache Set-Aside for Persisting Accesses . . . 32
6.2.3.2 L2 Policy for Persisting Accesses . . . 32
6.2.3.3 L2 Access Properties . . . 34
6.2.3.4 L2 Persistence Example . . . 34
6.2.3.5 Reset L2 Access to Normal . . . 35
6.2.3.6 Manage Utilization of L2 set-aside cache . . . 35
6.2.3.7 Query L2 cache Properties . . . 36
6.2.3.8 Control L2 Cache Set-Aside Size for Persisting Memory Access . . . 36

6.2.4 Shared Memory . . . 36
6.2.5 Distributed Shared Memory . . . 41
6.2.6 Page-Locked Host Memory . . . 44
6.2.6.1 Portable Memory . . . 45
6.2.6.2 Write-Combining Memory . . . 45
6.2.6.3 Mapped Memory . . . 45
6.2.7 Memory Synchronization Domains . . . 46
6.2.7.1 Memory Fence Interference . . . 46
6.2.7.2 Isolating Traffic with Domains . . . 47
6.2.7.3 Using Domains in CUDA . . . 47
6.2.8 Asynchronous Concurrent Execution . . . 48
6.2.8.1 Concurrent Execution between Host and Device . . . 49
6.2.8.2 Concurrent Kernel Execution . . . 49
6.2.8.3 Overlap of Data Transfer and Kernel Execution . . . 50
6.2.8.4 Concurrent Data Transfers . . . 50
6.2.8.5 Streams . . . 50
6.2.8.6 Programmatic Dependent Launch and Synchronization . . . 54
6.2.8.7 CUDA Graphs . . . 57
6.2.8.8 Events . . . 84
6.2.8.9 Synchronous Calls . . . 85
6.2.9 Multi-Device System . . . 85
6.2.9.1 Device Enumeration . . . 85
6.2.9.2 Device Selection . . . 85
6.2.9.3 Stream and Event Behavior . . . 86
6.2.9.4 Peer-to-Peer Memory Access . . . 86
6.2.9.5 Peer-to-Peer Memory Copy . . . 87
6.2.10 Unified Virtual Address Space . . . 88
6.2.11 Interprocess Communication . . . 88
6.2.12 Error Checking . . . 89
6.2.13 Call Stack . . . 89
6.2.14 Texture and Surface Memory . . . 90
6.2.14.1 Texture Memory . . . 90
6.2.14.2 Surface Memory . . . 95
6.2.14.3 CUDA Arrays . . . 98
6.2.14.4 Read/Write Coherency . . . 98
6.2.15 Graphics Interoperability . . . 98
6.2.15.1 OpenGL Interoperability . . . 99
6.2.15.2 Direct3D Interoperability . . . 101
6.2.15.3 SLI Interoperability . . . 107
6.2.16 External Resource Interoperability . . . 108
6.2.16.1 Vulkan Interoperability . . . 109
6.2.16.2 OpenGL Interoperability . . . 116
6.2.16.3 Direct3D 12 Interoperability . . . 117
6.2.16.4 Direct3D 11 Interoperability . . . 123
6.2.16.5 NVIDIA Software Communication Interface Interoperability (NVSCI) . . . 132
6.3 Versioning and Compatibility . . . 137
6.4 Compute Modes . . . 138
6.5 Mode Switches . . . 139
6.6 Tesla Compute Cluster Mode for Windows . . . 139

7 Hardware Implementation 141
7.1 SIMT Architecture . . . 141
7.2 Hardware Multithreading . . . 143

8 Performance Guidelines **145**
8.1 Overall Performance Optimization Strategies . . . 145
8.2 Maximize Utilization . . . 145
8.2.1 Application Level . . . 146
8.2.2 Device Level . . . 146
8.2.3 Multiprocessor Level . . . 146
8.2.3.1 Occupancy Calculator . . . 148
8.3 Maximize Memory Throughput . . . 150
8.3.1 Data Transfer between Host and Device . . . 151
8.3.2 Device Memory Accesses . . . 152
8.4 Maximize Instruction Throughput . . . 155
8.5 Minimize Memory Thrashing . . . 155

9 CUDA-Enabled GPUs **157**

10 C++ Language Extensions **159**
10.1 Function Execution Space Specifiers . . . 159
10.1.1 __global__ . . . 159
10.1.2 __device__ . . . 160
10.1.3 __host__ . . . 160
10.1.4 Undefined behavior . . . 160
10.1.5 __noinline__ and __forceinline__ . . . 161
10.1.6 __inline_hint__ . . . 161
10.2 Variable Memory Space Specifiers . . . 161
10.2.1 __device__ . . . 161
10.2.2 __constant__ . . . 162
10.2.3 __shared__ . . . 162
10.2.4 __grid_constant__ . . . 163
10.2.5 __managed__ . . . 164
10.2.6 __restrict__ . . . 164
10.3 Built-in Vector Types . . . 165
10.3.1 char, short, int, long, longlong, float, double . . . 165
10.3.2 dim3 . . . 167
10.4 Built-in Variables . . . 167
10.4.1 gridDim . . . 167
10.4.2 blockIdx . . . 167
10.4.3 blockDim . . . 167
10.4.4 threadIdx . . . 168
10.4.5 warpSize . . . 168
10.5 Memory Fence Functions . . . 168
10.6 Synchronization Functions . . . 171
10.7 Mathematical Functions . . . 172
10.8 Texture Functions . . . 172
10.8.1 Texture Object API . . . 172
10.8.1.1 tex1Dfetch() . . . 172
10.8.1.2 tex1D() . . . 172
10.8.1.3 tex1DLod() . . . 173
10.8.1.4 tex1DGrad() . . . 173
10.8.1.5 tex2D() . . . 173
10.8.1.6 tex2D() for sparse CUDA arrays . . . 173
10.8.1.7 tex2Dgather() . . . 173
10.8.1.8 tex2Dgather() for sparse CUDA arrays . . . 174
10.8.1.9 tex2DGrad() . . . 174
10.8.1.10 tex2DGrad() for sparse CUDA arrays . . . 174

10.8.1.11 tex2DLod() . . . 174
10.8.1.12 tex2DLod() for sparse CUDA arrays . . . 174
10.8.1.13 tex3D() . . . 175
10.8.1.14 tex3D() for sparse CUDA arrays . . . 175
10.8.1.15 tex3DLod() . . . 175
10.8.1.16 tex3DLod() for sparse CUDA arrays . . . 175
10.8.1.17 tex3DGrad() . . . 175
10.8.1.18 tex3DGrad() for sparse CUDA arrays . . . 176
10.8.1.19 tex1DLayered() . . . 176
10.8.1.20 tex1DLayeredLod() . . . 176
10.8.1.21 tex1DLayeredGrad() . . . 176
10.8.1.22 tex2DLayered() . . . 176
10.8.1.23 tex2DLayered() for Sparse CUDA Arrays . . . 177
10.8.1.24 tex2DLayeredLod() . . . 177
10.8.1.25 tex2DLayeredLod() for sparse CUDA arrays . . . 177
10.8.1.26 tex2DLayeredGrad() . . . 177
10.8.1.27 tex2DLayeredGrad() for sparse CUDA arrays . . . 177
10.8.1.28 texCubemap() . . . 178
10.8.1.29 texCubemapGrad() . . . 178
10.8.1.30 texCubemapLod() . . . 178
10.8.1.31 texCubemapLayered() . . . 178
10.8.1.32 texCubemapLayeredGrad() . . . 178
10.8.1.33 texCubemapLayeredLod() . . . 179
10.9 Surface Functions . . . 179
10.9.1 Surface Object API . . . 179
10.9.1.1 surf1Dread() . . . 179
10.9.1.2 surf1Dwrite . . . 179
10.9.1.3 surf2Dread() . . . 180
10.9.1.4 surf2Dwrite() . . . 180
10.9.1.5 surf3Dread() . . . 180
10.9.1.6 surf3Dwrite() . . . 180
10.9.1.7 surf1DLayeredread() . . . 181
10.9.1.8 surf1DLayeredwrite() . . . 181
10.9.1.9 surf2DLayeredread() . . . 181
10.9.1.10 surf2DLayeredwrite() . . . 182
10.9.1.11 surfCubemapread() . . . 182
10.9.1.12 surfCubemapwrite() . . . 182
10.9.1.13 surfCubemapLayeredread() . . . 182
10.9.1.14 surfCubemapLayeredwrite() . . . 183
10.10 Read-Only Data Cache Load Function . . . 183
10.11 Load Functions Using Cache Hints . . . 183
10.12 Store Functions Using Cache Hints . . . 184
10.13 Time Function . . . 184
10.14 Atomic Functions . . . 184
10.14.1 Arithmetic Functions . . . 187
10.14.1.1 atomicAdd() . . . 187
10.14.1.2 atomicSub() . . . 188
10.14.1.3 atomicExch() . . . 188
10.14.1.4 atomicMin() . . . 188
10.14.1.5 atomicMax() . . . 189
10.14.1.6 atomicInc() . . . 189
10.14.1.7 atomicDec() . . . 189
10.14.1.8 atomicCAS() . . . 189
10.14.1.9 __nv_atomic_exchange() . . . 190

10.14.1.10 __nv_atomic_exchange_n() . . . 191
10.14.1.11 __nv_atomic_compare_exchange() . . . 191
10.14.1.12 __nv_atomic_compare_exchange_n() . . . 191
10.14.1.13 __nv_atomic_fetch_add() and __nv_atomic_add() . . . 192
10.14.1.14 __nv_atomic_fetch_sub() and __nv_atomic_sub() . . . 192
10.14.1.15 __nv_atomic_fetch_min() and __nv_atomic_min() . . . 193
10.14.1.16 __nv_atomic_fetch_max() and __nv_atomic_max() . . . 193
10.14.2 Bitwise Functions . . . 193
10.14.2.1 atomicAnd() . . . 193
10.14.2.2 atomicOr() . . . 194
10.14.2.3 atomicXor() . . . 194
10.14.2.4 __nv_atomic_fetch_or() and __nv_atomic_or() . . . 194
10.14.2.5 __nv_atomic_fetch_xor() and __nv_atomic_xor() . . . 195
10.14.2.6 __nv_atomic_fetch_and() and __nv_atomic_and() . . . 195
10.14.3 Other atomic functions . . . 195
10.14.3.1 __nv_atomic_load() . . . 195
10.14.3.2 __nv_atomic_load_n() . . . 196
10.14.3.3 __nv_atomic_store() . . . 196
10.14.3.4 __nv_atomic_store_n() . . . 197
10.14.3.5 __nv_atomic_thread_fence() . . . 197
10.15 Address Space Predicate Functions . . . 197
10.15.1 __isGlobal() . . . 197
10.15.2 __isShared() . . . 198
10.15.3 __isConstant() . . . 198
10.15.4 __isGridConstant() . . . 198
10.15.5 __isLocal() . . . 198
10.16 Address Space Conversion Functions . . . 198
10.16.1 __cvta_generic_to_global() . . . 199
10.16.2 __cvta_generic_to_shared() . . . 199
10.16.3 __cvta_generic_to_constant() . . . 200
10.16.4 __cvta_generic_to_local() . . . 200
10.16.5 __cvta_global_to_generic() . . . 200
10.16.6 __cvta_shared_to_generic() . . . 200
10.16.7 __cvta_constant_to_generic() . . . 200
10.16.8 __cvta_local_to_generic() . . . 201
10.17 Alloca Function . . . 201
10.17.1 Synopsis . . . 201
10.17.2 Description . . . 201
10.17.3 Example . . . 201
10.18 Compiler Optimization Hint Functions . . . 202
10.18.1 __builtin_assume_aligned() . . . 202
10.18.2 __builtin_assume() . . . 202
10.18.3 __assume() . . . 203
10.18.4 __builtin_expect() . . . 203
10.18.5 __builtin_unreachable() . . . 203
10.18.6 Restrictions . . . 204
10.19 Warp Vote Functions . . . 204
10.20 Warp Match Functions . . . 205
10.20.1 Synopsis . . . 205
10.20.2 Description . . . 205
10.21 Warp Reduce Functions . . . 206
10.21.1 Synopsis . . . 206
10.21.2 Description . . . 206
10.22 Warp Shuffle Functions . . . 207

10.22.1 Synopsis . . . 207
10.22.2 Description . . . 207
10.22.3 Examples . . . 208
10.22.3.1 Broadcast of a single value across a warp . . . 208
10.22.3.2 Inclusive plus-scan across sub-partitions of 8 threads . . . 209
10.22.3.3 Reduction across a warp . . . 209
10.23 Nanosleep Function . . . 210
10.23.1 Synopsis . . . 210
10.23.2 Description . . . 210
10.23.3 Example . . . 210
10.24 Warp Matrix Functions . . . 210
10.24.1 Description . . . 211
10.24.2 Alternate Floating Point . . . 213
10.24.3 Double Precision . . . 213
10.24.4 Sub-byte Operations . . . 213
10.24.5 Restrictions . . . 215
10.24.6 Element Types and Matrix Sizes . . . 215
10.24.7 Example . . . 217
10.25 DPX . . . 217
10.25.1 Examples . . . 218
10.26 Asynchronous Barrier . . . 219
10.26.1 Simple Synchronization Pattern . . . 219
10.26.2 Temporal Splitting and Five Stages of Synchronization . . . 219
10.26.3 Bootstrap Initialization, Expected Arrival Count, and Participation . . . 220
10.26.4 A Barrier's Phase: Arrival, Countdown, Completion, and Reset . . . 221
10.26.5 Spatial Partitioning (also known as Warp Specialization) . . . 222
10.26.6 Early Exit (Dropping out of Participation) . . . 224
10.26.7 Completion Function . . . 224
10.26.8 Memory Barrier Primitives Interface . . . 226
10.26.8.1 Data Types . . . 226
10.26.8.2 Memory Barrier Primitives API . . . 226
10.27 Asynchronous Data Copies . . . 227
10.27.1 `memcpy_async` API . . . 227
10.27.2 Copy and Compute Pattern - Staging Data Through Shared Memory . . . 227
10.27.3 Without `memcpy_async` . . . 228
10.27.4 With `memcpy_async` . . . 229
10.27.5 Asynchronous Data Copies using `cuda::barrier` . . . 230
10.27.6 Performance Guidance for `memcpy_async` . . . 231
10.27.6.1 Alignment . . . 231
10.27.6.2 Trivially copyable . . . 231
10.27.6.3 Warp Entanglement - Commit . . . 231
10.27.6.4 Warp Entanglement - Wait . . . 232
10.27.6.5 Warp Entanglement - Arrive-On . . . 232
10.27.6.6 Keep Commit and Arrive-On Operations Converged . . . 233
10.28 Asynchronous Data Copies using `cuda::pipeline` . . . 233
10.28.1 Single-Stage Asynchronous Data Copies using `cuda::pipeline` . . . 233
10.28.2 Multi-Stage Asynchronous Data Copies using `cuda::pipeline` . . . 235
10.28.3 Pipeline Interface . . . 240
10.28.4 Pipeline Primitives Interface . . . 240
10.28.4.1 `memcpy_async` Primitive . . . 241
10.28.4.2 Commit Primitive . . . 241
10.28.4.3 Wait Primitive . . . 241
10.28.4.4 Arrive On Barrier Primitive . . . 242
10.29 Asynchronous Data Copies using the Tensor Memory Accelerator (TMA) . . . 242

10.29.1 Using TMA to transfer one-dimensional arrays . . . 243
10.29.2 Using TMA to transfer multi-dimensional arrays . . . 246
10.29.2.1 Multi-dimensional TMA PTX wrappers . . . 250
10.29.3 TMA Swizzle . . . 252
10.29.3.1 Example 'Matrix Transpose' . . . 252
10.29.3.2 The Swizzle Modes . . . 256
10.30 Encoding a Tensor Map on Device . . . 258
10.30.1 Device-side Encoding and Modification of a Tensor Map . . . 259
10.30.2 Usage of a Modified Tensor Map . . . 261
10.30.3 Creating a Template Tensor Map Value Using the Driver API . . . 262
10.31 Profiler Counter Function . . . 263
10.32 Assertion . . . 263
10.33 Trap function . . . 264
10.34 Breakpoint Function . . . 265
10.35 Formatted Output . . . 265
10.35.1 Format Specifiers . . . 265
10.35.2 Limitations . . . 266
10.35.3 Associated Host-Side API . . . 267
10.35.4 Examples . . . 267
10.36 Dynamic Global Memory Allocation and Operations . . . 268
10.36.1 Heap Memory Allocation . . . 269
10.36.2 Interoperability with Host Memory API . . . 269
10.36.3 Examples . . . 269
10.36.3.1 Per Thread Allocation . . . 269
10.36.3.2 Per Thread Block Allocation . . . 270
10.36.3.3 Allocation Persisting Between Kernel Launches . . . 271
10.37 Execution Configuration . . . 272
10.38 Launch Bounds . . . 274
10.39 Maximum Number of Registers per Thread . . . 276
10.40 #pragma unroll . . . 276
10.41 SIMD Video Instructions . . . 277
10.42 Diagnostic Pragmas . . . 278
10.43 Custom ABI Pragmas . . . 279
10.44 CUDA C++ Memory Model . . . 280
10.45 CUDA C++ Execution Model . . . 280

11 Cooperative Groups **281**
11.1 Introduction . . . 281
11.2 What's New in Cooperative Groups . . . 281
11.2.1 CUDA 13.0 . . . 281
11.2.2 CUDA 12.2 . . . 282
11.2.3 CUDA 12.1 . . . 282
11.2.4 CUDA 12.0 . . . 282
11.3 Programming Model Concept . . . 282
11.3.1 Composition Example . . . 283
11.4 Group Types . . . 284
11.4.1 Implicit Groups . . . 284
11.4.1.1 Thread Block Group . . . 284
11.4.1.2 Cluster Group . . . 285
11.4.1.3 Grid Group . . . 286
11.4.2 Explicit Groups . . . 287
11.4.2.1 Thread Block Tile . . . 287
11.4.2.2 Coalesced Groups . . . 290
11.5 Group Partitioning . . . 292

11.5.1 `tiled_partition` . . . 292
11.5.2 `labeled_partition` . . . 292
11.5.3 `binary_partition` . . . 293
11.6 Group Collectives . . . 293
11.6.1 Synchronization . . . 294
11.6.1.1 `barrier_arrive` and `barrier_wait` . . . 294
11.6.1.2 `sync` . . . 295
11.6.2 Data Transfer . . . 295
11.6.2.1 `memcpy_async` . . . 295
11.6.2.2 `wait and wait_prior` . . . 297
11.6.3 Data Manipulation . . . 298
11.6.3.1 `reduce` . . . 298
11.6.3.2 `Reduce` Operators . . . 300
11.6.3.3 `inclusive_scan` and `exclusive_scan` . . . 301
11.6.4 Execution control . . . 305
11.6.4.1 `invoke_one` and `invoke_one_broadcast` . . . 305
11.7 Grid Synchronization . . . 306

12 Cluster Launch Control **309**
12.1 Introduction . . . 309
12.2 Cluster Launch Control API Details . . . 310
12.2.1 Thread block cancellation steps . . . 311
12.2.2 Thread block cancellation constraints . . . 312
12.2.3 Kernel Example: Vector-Scalar Multiplication . . . 313
12.2.4 Cluster Launch Control for Thread Block Clusters . . . 315

13 CUDA Dynamic Parallelism **317**
13.1 Introduction . . . 317
13.1.1 Overview . . . 317
13.1.2 Glossary . . . 317
13.2 Execution Environment and Memory Model . . . 318
13.2.1 Execution Environment . . . 318
13.2.1.1 Parent and Child Grids . . . 318
13.2.1.2 Scope of CUDA Primitives . . . 319
13.2.1.3 Synchronization . . . 319
13.2.1.4 Streams and Events . . . 320
13.2.1.5 Ordering and Concurrency . . . 320
13.2.1.6 Device Management . . . 320
13.2.2 Memory Model . . . 321
13.2.2.1 Coherence and Consistency . . . 321
13.3 Programming Interface . . . 323
13.3.1 CUDA C++ Reference . . . 323
13.3.1.1 Device-Side Kernel Launch . . . 323
13.3.1.2 Streams . . . 324
13.3.1.3 Events . . . 326
13.3.1.4 Synchronization . . . 327
13.3.1.5 Device Management . . . 327
13.3.1.6 Memory Declarations . . . 327
13.3.1.7 API Errors and Launch Failures . . . 328
13.3.1.8 API Reference . . . 330
13.3.2 Device-side Launch from PTX . . . 332
13.3.2.1 Kernel Launch APIs . . . 332
13.3.2.2 Parameter Buffer Layout . . . 333
13.3.3 Toolkit Support for Dynamic Parallelism . . . 333

13.3.3.1 Including Device Runtime API in CUDA Code . . . 333
13.3.3.2 Compiling and Linking . . . 334
13.4 Programming Guidelines . . . 334
13.4.1 Basics . . . 334
13.4.2 Performance . . . 335
13.4.2.1 Dynamic-parallelism-enabled Kernel Overhead . . . 335
13.4.3 Implementation Restrictions and Limitations . . . 335
13.4.3.1 Runtime . . . 336
13.5 CDP2 vs CDP1 . . . 338
13.5.1 Differences Between CDP1 and CDP2 . . . 338
13.5.2 Compatibility and Interoperability . . . 339
13.6 Legacy CUDA Dynamic Parallelism (CDP1) . . . 339
13.6.1 Execution Environment and Memory Model (CDP1) . . . 339
13.6.1.1 Execution Environment (CDP1) . . . 340
13.6.1.2 Memory Model (CDP1) . . . 343
13.6.2 Programming Interface (CDP1) . . . 346
13.6.2.1 CUDA C++ Reference (CDP1) . . . 346
13.6.2.2 Device-side Launch from PTX (CDP1) . . . 354
13.6.2.3 Toolkit Support for Dynamic Parallelism (CDP1) . . . 356
13.6.3 Programming Guidelines (CDP1) . . . 357
13.6.3.1 Basics (CDP1) . . . 357
13.6.3.2 Performance (CDP1) . . . 358
13.6.3.3 Implementation Restrictions and Limitations (CDP1) . . . 359

14 Virtual Memory Management **363**
14.1 Introduction . . . 363
14.2 Query for Support . . . 364
14.3 Allocating Physical Memory . . . 364
14.3.1 Shareable Memory Allocations . . . 365
14.3.2 Memory Type . . . 366
14.3.2.1 Compressible Memory . . . 366
14.4 Reserving a Virtual Address Range . . . 367
14.5 Virtual Aliasing Support . . . 367
14.6 Mapping Memory . . . 368
14.7 Controlling Access Rights . . . 369
14.8 Fabric Memory . . . 369
14.8.1 Query for Support . . . 369
14.9 Multicast Support . . . 370
14.9.1 Query for Support . . . 371
14.9.2 Allocating Multicast Objects . . . 371
14.9.3 Add Devices to Multicast Objects . . . 371
14.9.4 Bind Memory to Multicast Objects . . . 372
14.9.5 Use Multicast Mappings . . . 372

15 Stream Ordered Memory Allocator **375**
15.1 Introduction . . . 375
15.2 Query for Support . . . 376
15.3 API Fundamentals (cudaMallocAsync and cudaFreeAsync) . . . 376
15.4 Memory Pools and the cudaMemPool_t . . . 378
15.5 Default/Implicit Pools . . . 378
15.6 Explicit Pools . . . 378
15.7 Physical Page Caching Behavior . . . 379
15.8 Resource Usage Statistics . . . 380
15.9 Memory Reuse Policies . . . 381

15.9.1 cudaMemPoolReuseFollowEventDependencies . . . 381
15.9.2 cudaMemPoolReuseAllowOpportunistic . . . 381
15.9.3 cudaMemPoolReuseAllowInternalDependencies . . . 382
15.9.4 Disabling Reuse Policies . . . 382
15.10 Device Accessibility for Multi-GPU Support . . . 382
15.11 IPC Memory Pools . . . 383
15.11.1 Creating and Sharing IPC Memory Pools . . . 383
15.11.2 Set Access in the Importing Process . . . 384
15.11.3 Creating and Sharing Allocations from an Exported Pool . . . 384
15.11.4 IPC Export Pool Limitations . . . 386
15.11.5 IPC Import Pool Limitations . . . 386
15.12 Synchronization API Actions . . . 386
15.13 Addendums . . . 387
15.13.1 cudaMemcpyAsync Current Context/Device Sensitivity . . . 387
15.13.2 cuPointerGetAttribute Query . . . 387
15.13.3 cuGraphAddMemsetNode . . . 387
15.13.4 Pointer Attributes . . . 387
15.13.5 CPU Virtual Memory . . . 387

16 Graph Memory Nodes **389**
16.1 Introduction . . . 389
16.2 Support and Compatibility . . . 389
16.3 API Fundamentals . . . 390
16.3.1 Graph Node APIs . . . 391
16.3.2 Stream Capture . . . 392
16.3.3 Accessing and Freeing Graph Memory Outside of the Allocating Graph . . . 393
16.3.4 cudaGraphInstantiateFlagAutoFreeOnLaunch . . . 395
16.4 Optimized Memory Reuse . . . 396
16.4.1 Address Reuse within a Graph . . . 397
16.4.2 Physical Memory Management and Sharing . . . 397
16.5 Performance Considerations . . . 400
16.5.1 First Launch / cudaGraphUpload . . . 400
16.6 Physical Memory Footprint . . . 400
16.7 Peer Access . . . 401
16.7.1 Peer Access with Graph Node APIs . . . 401
16.7.2 Peer Access with Stream Capture . . . 402
16.8 Memory Nodes in Child Graphs . . . 402

17 Mathematical Functions **405**
17.1 Standard Functions . . . 405
17.2 Intrinsic Functions . . . 414

18 C++ Language Support **419**
18.1 C++11 Language Features . . . 419
18.2 C++14 Language Features . . . 422
18.3 C++17 Language Features . . . 422
18.4 C++20 Language Features . . . 423
18.5 Restrictions . . . 423
18.5.1 Host Compiler Extensions . . . 423
18.5.2 Preprocessor Symbols . . . 423
18.5.2.1 __CUDA_ARCH__ . . . 423
18.5.3 Qualifiers . . . 425
18.5.3.1 Device Memory Space Specifiers . . . 425
18.5.3.2 __managed__ Memory Space Specifier . . . 426

18.5.3.3 Volatile Qualifier . . . 427
18.5.4 Pointers . . . 429
18.5.5 Operators . . . 430
18.5.5.1 Assignment Operator . . . 430
18.5.5.2 Address Operator . . . 430
18.5.6 Run Time Type Information (RTTI) . . . 430
18.5.7 Exception Handling . . . 430
18.5.8 Standard Library . . . 430
18.5.9 Namespace Reservations . . . 430
18.5.10 Functions . . . 431
18.5.10.1 External Linkage . . . 431
18.5.10.2 Implicitly-declared and non-virtual explicitly-defaulted functions . . . 431
18.5.10.3 Function Parameters . . . 432
18.5.10.4 Static Variables within Function . . . 435
18.5.10.5 Function Pointers . . . 437
18.5.10.6 Function Recursion . . . 437
18.5.10.7 Friend Functions . . . 437
18.5.10.8 Operator Function . . . 437
18.5.10.9 Allocation and Deallocation Functions . . . 437
18.5.11 Classes . . . 438
18.5.11.1 Data Members . . . 438
18.5.11.2 Function Members . . . 438
18.5.11.3 Virtual Functions . . . 438
18.5.11.4 Virtual Base Classes . . . 439
18.5.11.5 Anonymous Unions . . . 439
18.5.11.6 Windows-Specific . . . 439
18.5.12 Templates . . . 439
18.5.13 Trigraphs and Digraphs . . . 440
18.5.14 Const-qualified variables . . . 440
18.5.15 Long Double . . . 441
18.5.16 Deprecation Annotation . . . 441
18.5.17 Noreturn Annotation . . . 442
18.5.18 [[likely]] / [[unlikely]] Standard Attributes . . . 442
18.5.19 const and pure GNU Attributes . . . 442
18.5.20 __nv_pure__ Attribute . . . 443
18.5.21 Intel Host Compiler Specific . . . 443
18.5.22 C++11 Features . . . 443
18.5.22.1 Lambda Expressions . . . 443
18.5.22.2 std::initializer_list . . . 445
18.5.22.3 Rvalue references . . . 445
18.5.22.4 Constexpr functions and function templates . . . 445
18.5.22.5 Constexpr variables . . . 446
18.5.22.6 Inline namespaces . . . 446
18.5.22.7 thread_local . . . 448
18.5.22.8 __global__ functions and function templates . . . 448
18.5.22.9 __managed__ and __shared__ variables . . . 449
18.5.22.10 Defaulted functions . . . 449
18.5.23 C++14 Features . . . 450
18.5.23.1 Functions with deduced return type . . . 451
18.5.23.2 Variable templates . . . 452
18.5.24 C++17 Features . . . 452
18.5.24.1 Inline Variable . . . 452
18.5.24.2 Structured Binding . . . 453
18.5.25 C++20 Features . . . 453

18.5.25.1 Module support . . . 453
18.5.25.2 Coroutine support . . . 453
18.5.25.3 Three-way comparison operator . . . 453
18.5.25.4 Consteval functions . . . 454
18.6 Polymorphic Function Wrappers . . . 454
18.7 Extended Lambdas . . . 457
18.7.1 Extended Lambda Type Traits . . . 458
18.7.2 Extended Lambda Restrictions . . . 459
18.7.3 Notes on __host__ __device__ lambdas . . . 469
18.7.4 *this Capture By Value . . . 469
18.7.5 Additional Notes . . . 472
18.8 Relaxed Constexpr (-expt-relaxed-constexpr) . . . 472
18.9 Code Samples . . . 475
18.9.1 Data Aggregation Class . . . 475
18.9.2 Derived Class . . . 476
18.9.3 Class Template . . . 476
18.9.4 Function Template . . . 477
18.9.5 Functor Class . . . 477

19 Texture Fetching **479**
19.1 Nearest-Point Sampling . . . 479
19.2 Linear Filtering . . . 480
19.3 Table Lookup . . . 481

20 Compute Capabilities **483**
20.1 Feature Availability . . . 483
20.1.1 Architecture-Specific Features . . . 483
20.1.2 Family-Specific Features . . . 484
20.1.3 Feature Set Compiler Targets . . . 484
20.2 Features and Technical Specifications . . . 485
20.3 Floating-Point Standard . . . 490
20.4 Compute Capability 5.x . . . 490
20.4.1 Architecture . . . 490
20.4.2 Global Memory . . . 491
20.4.3 Shared Memory . . . 492
20.5 Compute Capability 6.x . . . 492
20.5.1 Architecture . . . 492
20.5.2 Global Memory . . . 495
20.5.3 Shared Memory . . . 495
20.6 Compute Capability 7.x . . . 495
20.6.1 Architecture . . . 495
20.6.2 Independent Thread Scheduling . . . 496
20.6.3 Global Memory . . . 498
20.6.4 Shared Memory . . . 498
20.7 Compute Capability 8.x . . . 499
20.7.1 Architecture . . . 499
20.7.2 Global Memory . . . 500
20.7.3 Shared Memory . . . 500
20.8 Compute Capability 9.0 . . . 501
20.8.1 Architecture . . . 501
20.8.2 Global Memory . . . 501
20.8.3 Shared Memory . . . 502
20.8.4 Features Accelerating Specialized Computations . . . 502
20.9 Compute Capability 10.0 . . . 502

20.9.1 Architecture . . . 502
20.9.2 Global Memory . . . 503
20.9.3 Shared Memory . . . 503
20.9.4 Features Accelerating Specialized Computations . . . 504
20.10 Compute Capability 12.0 . . . 504
20.10.1 Architecture . . . 504
20.10.2 Global Memory . . . 505
20.10.3 Shared Memory . . . 505
20.10.4 Features Accelerating Specialized Computations . . . 505

21 Driver API **507**
21.1 Context . . . 509
21.2 Module . . . 510
21.3 Kernel Execution . . . 511
21.4 Interoperability between Runtime and Driver APIs . . . 513
21.5 Driver Entry Point Access . . . 513
21.5.1 Introduction . . . 513
21.5.2 Driver Function Typedefs . . . 514
21.5.3 Driver Function Retrieval . . . 515
21.5.3.1 Using the Driver API . . . 515
21.5.3.2 Using the Runtime API . . . 516
21.5.3.3 Retrieve Per-thread Default Stream Versions . . . 516
21.5.3.4 Access New CUDA features . . . 517
21.5.4 Guidelines for cuGetProcAddress . . . 518
21.5.4.1 Guidelines for Runtime API Usage . . . 518
21.5.5 Determining cuGetProcAddress Failure Reasons . . . 518

22 CUDA Environment Variables **521**

23 Error Log Management **531**
23.1 Background . . . 531
23.2 Activation . . . 531
23.3 Output . . . 531
23.4 API Description . . . 532
23.5 Limitations and Known Issues . . . 533

24 Unified Memory Programming **535**
24.1 Unified Memory Introduction . . . 535
24.1.1 System Requirements for Unified Memory . . . 536
24.1.2 Programming Model . . . 538
24.1.2.1 Allocation APIs for System-Allocated Memory . . . 540
24.1.2.2 Allocation API for CUDA Managed Memory: `cudaMallocManaged()` . . . 541
24.1.2.3 Global-Scope Managed Variables Using `__managed__` . . . 542
24.1.2.4 Difference between Unified Memory and Mapped Memory . . . 543
24.1.2.5 Pointer Attributes . . . 543
24.1.2.6 Runtime detection of Unified Memory Support Level . . . 544
24.1.2.7 GPU Memory Oversubscription . . . 545
24.1.2.8 Performance Hints . . . 545
24.2 Unified memory on devices with full CUDA Unified Memory support . . . 550
24.2.1 System-Allocated Memory: in-depth examples . . . 550
24.2.1.1 File-backed Unified Memory . . . 552
24.2.1.2 Inter-Process Communication (IPC) with Unified Memory . . . 553
24.2.2 Performance Tuning . . . 553
24.2.2.1 Memory Paging and Page Sizes . . . 554
24.2.2.2 Direct Unified Memory Access from host . . . 556

24.2.2.3 Host Native Atomics . . . 558
24.2.2.4 Atomic accesses & synchronization primitives . . . 558
24.2.2.5 Memcpy()/Memset() Behavior With Unified Memory . . . 559
24.3 Unified memory on devices without full CUDA Unified Memory support . . . 559
24.3.1 Unified memory on devices with only CUDA Managed Memory support . . . 559
24.3.2 Unified memory on Windows or devices with compute capability 5.x . . . 560
24.3.2.1 Data Migration and Coherency . . . 560
24.3.2.2 GPU Memory Oversubscription . . . 560
24.3.2.3 Multi-GPU . . . 560
24.3.2.4 Coherency and Concurrency . . . 561

25 Lazy Loading **569**
25.1 What is Lazy Loading? . . . 569
25.2 Lazy Loading version support . . . 570
25.2.1 Driver . . . 570
25.2.2 Toolkit . . . 570
25.2.3 Compiler . . . 570
25.3 Triggering loading of kernels in lazy mode . . . 570
25.3.1 CUDA Driver API . . . 571
25.3.2 CUDA Runtime API . . . 571
25.4 Querying whether Lazy Loading is Turned On . . . 571
25.5 Possible Issues when Adopting Lazy Loading . . . 572
25.5.1 Concurrent Execution . . . 572
25.5.2 Allocators . . . 572
25.5.3 Autotuning . . . 573

26 Extended GPU Memory **575**
26.1 Preliminaries . . . 575
26.1.1 EGM Platforms: System topology . . . 576
26.1.2 Socket Identifiers: What are they? How to access them? . . . 576
26.1.3 Allocators and EGM support . . . 576
26.1.4 Memory management extensions to current APIs . . . 576
26.2 Using the EGM Interface . . . 577
26.2.1 Single-Node, Single-GPU . . . 577
26.2.2 Single-Node, Multi-GPU . . . 577
26.2.2.1 Using VMM APIs . . . 578
26.2.2.2 Using CUDA Memory Pool . . . 578
26.2.3 Multi-Node, Single-GPU . . . 579

27 Notices **581**
27.1 Notice . . . 581
27.2 OpenCL . . . 582
27.3 Trademarks . . . 582

Chapter 1. Overview

CUDA C++ Programming Guide (Legacy)

> **Warning:** This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

CUDA is a parallel computing platform and programming model developed by NVIDIA that enables dramatic increases in computing performance by harnessing the power of the GPU. It allows developers to accelerate compute-intensive applications using C, C++, and Fortran, and is widely adopted in fields such as deep learning, scientific computing, and high-performance computing (HPC).

Chapter 2. What Is the CUDA C Programming Guide?

The CUDA C Programming Guide is the official, comprehensive resource that explains how to write programs using the CUDA platform. It provides detailed documentation of the CUDA architecture, programming model, language extensions, and performance guidelines. Whether you're just getting started or optimizing complex GPU kernels, this guide is an essential reference for effectively leveraging CUDA's full capabilities.

Chapter 3. Introduction

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

3.1. The Benefits of Using GPUs

The Graphics Processing Unit (GPU)[1] provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope. Many applications leverage these higher capabilities to run faster on the GPU than on the CPU (see GPU Applications). Other computing devices, like FPGAs, are also very energy efficient, but offer much less programming flexibility than GPUs.

This difference in capabilities between the GPU and the CPU exists because they are designed with different goals in mind. While the CPU is designed to excel at executing a sequence of operations, called a *thread*, as fast as possible and can execute a few tens of these threads in parallel, the GPU is designed to excel at executing thousands of them in parallel (amortizing the slower single-thread performance to achieve greater throughput).

The GPU is specialized for highly parallel computations and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. The schematic Figure 1 shows an example distribution of chip resources for a CPU versus a GPU.

Devoting more transistors to data processing, for example, floating-point computations, is beneficial for highly parallel computations; the GPU can hide memory access latencies with computation, instead of relying on large data caches and complex flow control to avoid long memory access latencies, both of which are expensive in terms of transistors.

In general, an application has a mix of parallel parts and sequential parts, so systems are designed with a mix of GPUs and CPUs in order to maximize overall performance. Applications with a high degree of parallelism can exploit this massively parallel nature of the GPU to achieve higher performance than on the CPU.

[1] The *graphics* qualifier comes from the fact that when the GPU was originally created, two decades ago, it was designed as a specialized processor to accelerate graphics rendering. Driven by the insatiable market demand for real-time, high-definition, 3D graphics, it has evolved into a general processor used for many more workloads than just graphics rendering.

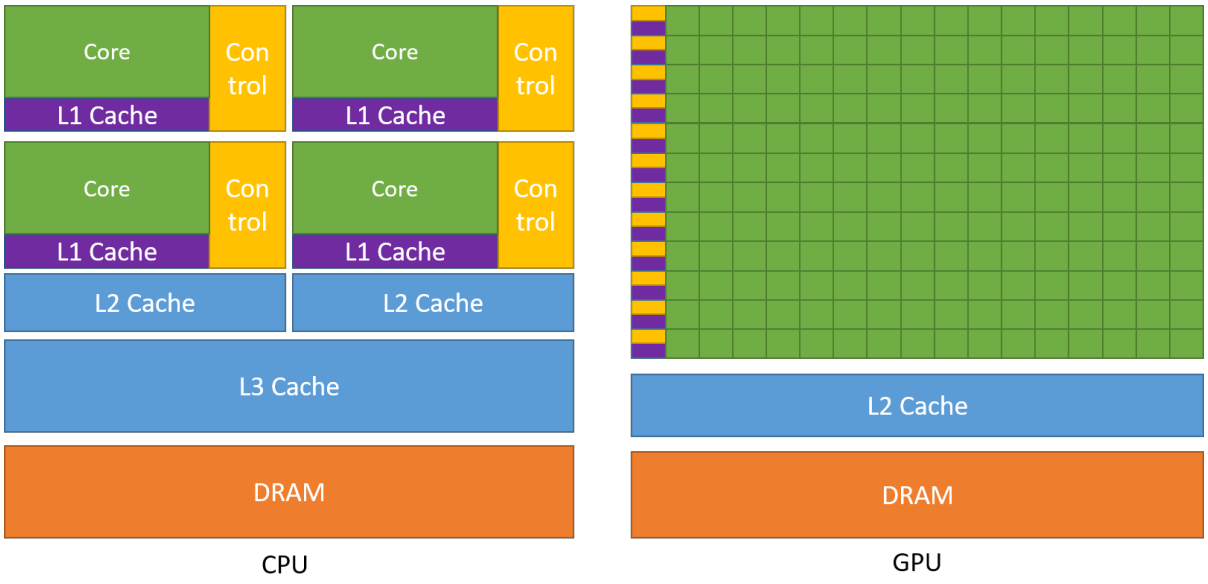


Figure 1: The GPU Devotes More Transistors to Data Processing

3.2. CUDA®: A General-Purpose Parallel Computing Platform and Programming Model

In November 2006, NVIDIA® introduced CUDA®, a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.

CUDA comes with a software environment that allows developers to use C++ as a high-level programming language. As illustrated by Figure 2, other languages, application programming interfaces, or directives-based approaches are supported, such as FORTRAN, DirectCompute, OpenACC.

3.3. A Scalable Programming Model

The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to manycore GPUs with widely varying numbers of cores.

The CUDA parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C.

At its core are three key abstractions — a hierarchy of thread groups, shared memories, and barrier synchronization — that are simply exposed to the programmer as a minimal set of language extensions.

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem

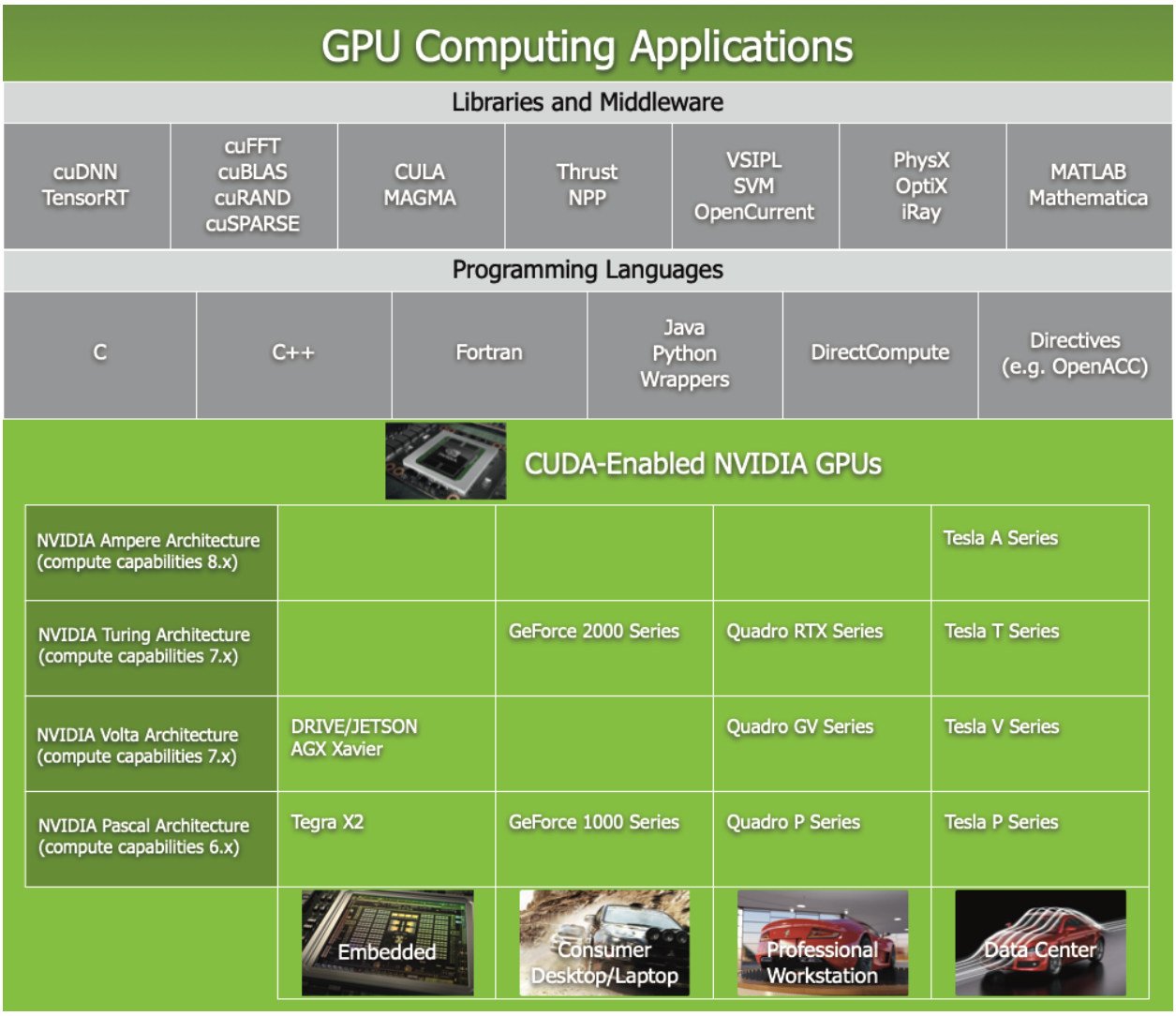


Figure 2: GPU Computing Applications. CUDA is designed to support various languages and application programming interfaces.

into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors as illustrated by Figure 3, and only the runtime system needs to know the physical multiprocessor count.

This scalable programming model allows the GPU architecture to span a wide market range by simply scaling the number of multiprocessors and memory partitions: from the high-performance enthusiast GeForce GPUs and professional Quadro and Tesla computing products to a variety of inexpensive, mainstream GeForce GPUs (see *CUDA-Enabled GPUs* for a list of all CUDA-enabled GPUs).

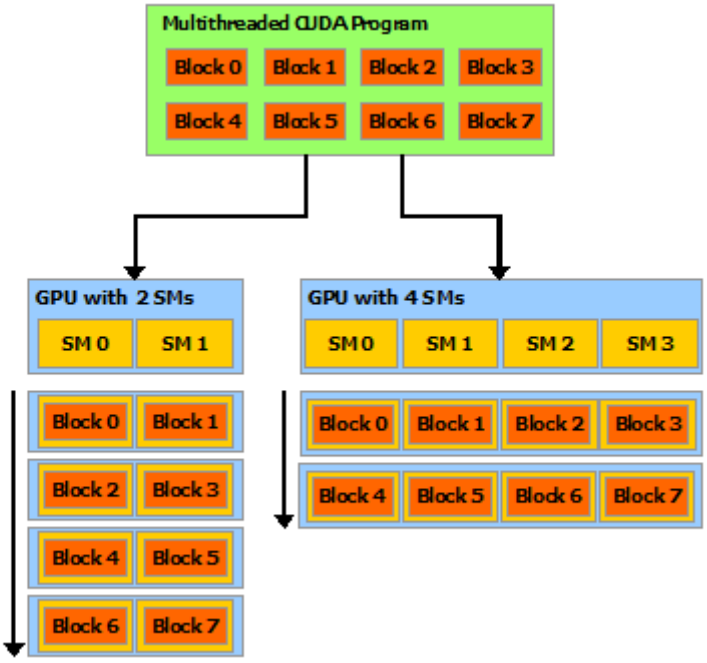


Figure 3: Automatic Scalability

Note: A GPU is built around an array of Streaming Multiprocessors (SMs) (see *Hardware Implementation* for more details). A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

Chapter 4. Changelog

Table 1: Change Log

Version	Changes
13.0	Moved the instruction throughput table from the *Performance Guidelines* section of the CUDA C++ Programming Guide to the Instruction-optimization section of the CUDA C++ Best Practices Guide. Removed unsupported architectures and corrected entries for integer arithmetic and type conversion.
12.9	Added section *Error Log Management* and CUDA_LOG_FILE to *CUDA Environment Variables*
12.8	Added section *TMA Swizzle*

Chapter 5. Programming Model

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

This chapter introduces the main concepts behind the CUDA programming model by outlining how they are exposed in C++.

An extensive description of CUDA C++ is given in *Programming Interface*.

Full code for the vector addition example used in this chapter and the next can be found in the vectorAdd CUDA sample.

5.1. Kernels

CUDA C++ extends C++ by allowing the programmer to define C++ functions, called *kernels*, that, when called, are executed N times in parallel by N different *CUDA threads*, as opposed to only once like regular C++ functions.

A kernel is defined using the `__global__` declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new `<<<...>>>`*execution configuration* syntax (see Execution Configuration). Each thread that executes the kernel is given a unique *thread ID* that is accessible within the kernel through built-in variables.

As an illustration, the following sample code, using the built-in variable `threadIdx`, adds two vectors *A* and *B* of size *N* and stores the result into vector *C*.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Here, each of the *N* threads that execute `VecAdd()` performs one pair-wise addition.

5.2. Thread Hierarchy

For convenience, `threadIdx` is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional *thread index*, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a *thread block*. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.

The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size *(Dx, Dy)*, the thread ID of a thread of index *(x, y)* is *(x + y Dx)*; for a three-dimensional block of size *(Dx, Dy, Dz)*, the thread ID of a thread of index *(x, y, z)* is *(x + y Dx + z Dx Dy)*.

As an example, the following code adds two matrices *A* and *B* of size *NxN* and stores the result into matrix *C*.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same streaming multiprocessor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads.

However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.

Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional *grid* of thread blocks as illustrated by Figure 4. The number of thread blocks in a grid is usually dictated by the size of the data being processed, which typically exceeds the number of processors in the system.

The number of threads per block and the number of blocks per grid specified in the `<<<...>>>` syntax can be of type `int` or `dim3`. Two-dimensional blocks or grids can be specified as in the example above.

Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional unique index accessible within the kernel through the built-in `blockIdx` variable. The dimension of the thread block is accessible within the kernel through the built-in `blockDim` variable.

Extending the previous `MatAdd()` example to handle multiple blocks, the code becomes as follows.

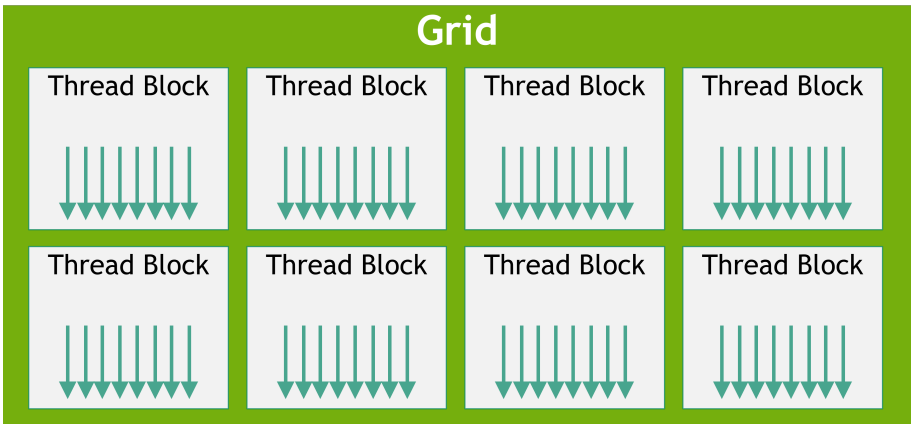


Figure 4: Grid of Thread Blocks

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

A thread block size of 16x16 (256 threads), although arbitrary in this case, is a common choice. The grid is created with enough blocks to have one thread per matrix element as before. For simplicity, this example assumes that the number of threads per grid in each dimension is evenly divisible by the number of threads per block in that dimension, although that need not be the case.

Thread blocks are required to execute independently. It must be possible to execute blocks in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order and across any number of cores as illustrated by Figure 3, enabling programmers to write code that scales with the number of cores.

Threads within a block can cooperate by sharing data through some *shared memory* and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the `__syncthreads()` intrinsic function; `__syncthreads()` acts as a barrier at which all threads in the block must wait before any is allowed to proceed. *Shared Memory* gives an example of using shared memory. In addition to `__syncthreads()`, the *Cooperative Groups API* provides a rich set of thread-synchronization primitives.

For efficient cooperation, shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and `__syncthreads()` is expected to be lightweight.

5.2.1. Thread Block Clusters

With the introduction of NVIDIA *Compute Capability 9.0*, the CUDA programming model introduces an optional level of hierarchy called Thread Block Clusters that are made up of thread blocks. Similar to how threads in a thread block are guaranteed to be co-scheduled on a streaming multiprocessor, thread blocks in a cluster are also guaranteed to be co-scheduled on a GPU Processing Cluster (GPC) in the GPU.

Similar to thread blocks, clusters are also organized into a one-dimension, two-dimension, or three-dimension grid of thread block clusters as illustrated by Figure 5. The number of thread blocks in a cluster can be user-defined, and a maximum of 8 thread blocks in a cluster is supported as a portable cluster size in CUDA. Note that on GPU hardware or MIG configurations which are too small to support 8 multiprocessors the maximum cluster size will be reduced accordingly. Identification of these smaller configurations, as well as of larger configurations supporting a thread block cluster size beyond 8, is architecture-specific and can be queried using the `cudaOccupancyMaxPotentialClusterSize` API.

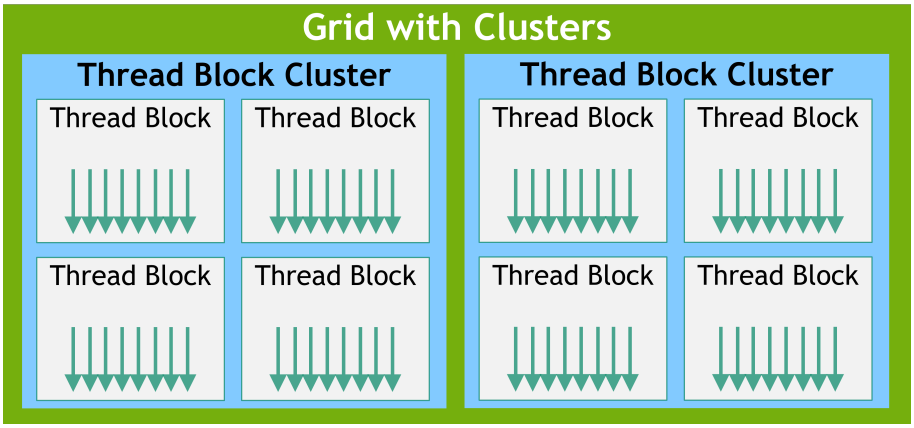


Figure 5: Grid of Thread Block Clusters

Note: In a kernel launched using cluster support, the gridDim variable still denotes the size in terms of number of thread blocks, for compatibility purposes. The rank of a block in a cluster can be found using the *Cluster Group* API.

A thread block cluster can be enabled in a kernel either using a compile-time kernel attribute using `__cluster_dims__(X,Y,Z)` or using the CUDA kernel launch API `cudaLaunchKernelEx`. The example below shows how to launch a cluster using a compile-time kernel attribute. The cluster size using kernel attribute is fixed at compile time and then the kernel can be launched using the classical `<<< , >>>`. If a kernel uses compile-time cluster size, the cluster size cannot be modified when launching the kernel.

```
// Kernel definition
// Compile time cluster size 2 in X-dimension and 1 in Y and Z dimension
__global__ void __cluster_dims__(2, 1, 1) cluster_kernel(float *input, float* output)
{

}

int main()
{
```

(continues on next page)

(continued from previous page)

```
    float *input, *output;
    // Kernel invocation with compile time cluster size
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);

    // The grid dimension is not affected by cluster launch, and is still enumerated
    // using number of blocks.
    // The grid dimension must be a multiple of cluster size.
    cluster_kernel<<<numBlocks, threadsPerBlock>>>(input, output);
}
```

A thread block cluster size can also be set at runtime and the kernel can be launched using the CUDA kernel launch API `cudaLaunchKernelEx`. The code example below shows how to launch a cluster kernel using the extensible API.

```
// Kernel definition
// No compile time attribute attached to the kernel
__global__ void cluster_kernel(float *input, float* output)
{

}

int main()
{
    float *input, *output;
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);

    // Kernel invocation with runtime cluster size
    {
        cudaLaunchConfig_t config = {0};
        // The grid dimension is not affected by cluster launch, and is still enumerated
        // using number of blocks.
        // The grid dimension should be a multiple of cluster size.
        config.gridDim = numBlocks;
        config.blockDim = threadsPerBlock;

        cudaLaunchAttribute attribute[1];
        attribute[0].id = cudaLaunchAttributeClusterDimension;
        attribute[0].val.clusterDim.x = 2; // Cluster size in X-dimension
        attribute[0].val.clusterDim.y = 1;
        attribute[0].val.clusterDim.z = 1;
        config.attrs = attribute;
        config.numAttrs = 1;

        cudaLaunchKernelEx(&config, cluster_kernel, input, output);
    }
}
```

In GPUs with compute capability 9.0, all the thread blocks in the cluster are guaranteed to be co-scheduled on a single GPU Processing Cluster (GPC) and allow thread blocks in the cluster to perform hardware-supported synchronization using the *Cluster Group* API `cluster.sync()`. Cluster group also provides member functions to query cluster group size in terms of number of threads or number of blocks using `num_threads()` and `num_blocks()` API respectively. The rank of a thread or block in the cluster group can be queried using `dim_threads()` and `dim_blocks()` API respectively.

Thread blocks that belong to a cluster have access to the Distributed Shared Memory. Thread blocks

in a cluster have the ability to read, write, and perform atomics to any address in the distributed shared memory. *Distributed Shared Memory* gives an example of performing histograms in distributed shared memory.

5.2.2. Blocks as Clusters

With `__cluster_dims__`, the number of launched clusters is kept implicit and can only be calculated manually.

```
__cluster_dims__((2, 2, 2)) __global__ void foo();

// 8x8x8 clusters each with 2x2x2 thread blocks.
foo<<<dim3(16, 16, 16), dim3(1024, 1, 1)>>>();
```

In the above example, the kernel is launched as a grid of 16x16x16 thread blocks, or in fact a grid of 8x8x8 clusters. Alternatively, with another compile-time kernel attribute `__block_size__`, one is allowed to launch a grid explicitly configured with the number of thread block clusters.

```
// Implementation detail of how many threads per block and blocks per cluster
// is handled as an attribute of the kernel.
__block_size__((1024, 1, 1), (2, 2, 2)) __global__ void foo();

// 8x8x8 clusters.
foo<<<dim3(8, 8, 8)>>>();
```

`__block_size__` requires two fields each being a tuple of 3 elements. The first tuple denotes block dimension and second cluster size. The second tuple is assumed to be `(1,1,1)` if it's not passed. To specify the stream, one must pass 1 and `0` as the second and third arguments within `<<<>>>` and lastly the stream. Passing other values would lead to undefined behavior.

Note that it is illegal for the second tuple of `__block_size__` and `__cluster_dims__` to be specified at the same time. It's also illegal to use `__block_size__` with an empty `__cluster_dims__`. When the second tuple of `__block_size__` is specified, it implies the "Blocks as Clusters" being enabled and the compiler would recognize the first argument inside `<<<>>>` as the number of clusters instead of thread blocks.

5.3. Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 6. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. Thread blocks in a thread block cluster can perform read, write, and atomics operations on each other's shared memory. All threads have access to the same global memory.

There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages (see *Device Memory Accesses*). Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats (see *Texture and Surface Memory*).

The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

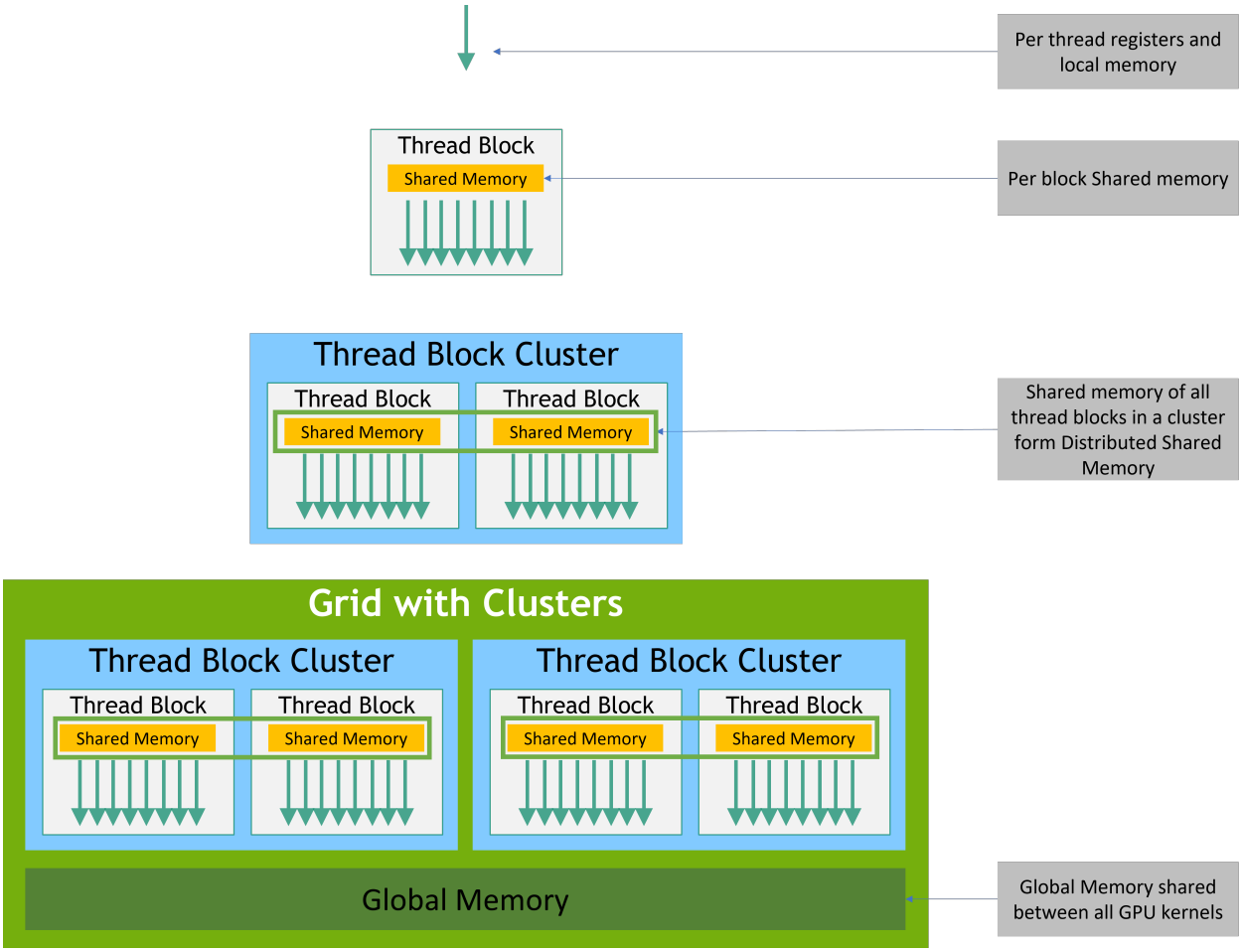


Figure 6: Memory Hierarchy

5.4. Heterogeneous Programming

As illustrated by Figure 7, the CUDA programming model assumes that the CUDA threads execute on a physically separate *device* that operates as a coprocessor to the *host* running the C++ program. This is the case, for example, when the kernels execute on a GPU and the rest of the C++ program executes on a CPU.

The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as *host memory* and *device memory*, respectively. Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime (described in *Programming Interface*). This includes device memory allocation and deallocation as well as data transfer between host and device memory.

Unified Memory provides *managed memory* to bridge the host and device memory spaces. Managed memory is accessible from all CPUs and GPUs in the system as a single, coherent memory image with a common address space. This capability enables oversubscription of device memory and can greatly simplify the task of porting applications by eliminating the need to explicitly mirror data on host and device. See *Unified Memory Programming* for an introduction to Unified Memory.

5.5. Asynchronous SIMT Programming Model

In the CUDA programming model a thread is the lowest level of abstraction for doing a computation or a memory operation. Starting with devices based on the **NVIDIA Ampere GPU Architecture**, the CUDA programming model provides acceleration to memory operations via the asynchronous programming model. The asynchronous programming model defines the behavior of asynchronous operations with respect to CUDA threads.

The asynchronous programming model defines the behavior of *Asynchronous Barrier* for synchronization between CUDA threads. The model also explains and defines how *cuda::memcpy_async* can be used to move data asynchronously from global memory while computing in the GPU.

5.5.1. Asynchronous Operations

An asynchronous operation is defined as an operation that is initiated by a CUDA thread and is executed asynchronously as-if by another thread. In a well formed program one or more CUDA threads synchronize with the asynchronous operation. The CUDA thread that initiated the asynchronous operation is not required to be among the synchronizing threads.

Such an asynchronous thread (an as-if thread) is always associated with the CUDA thread that initiated the asynchronous operation. An asynchronous operation uses a synchronization object to synchronize the completion of the operation. Such a synchronization object can be explicitly managed by a user (e.g., `cuda::memcpy_async`) or implicitly managed within a library (e.g., `cooperative_groups::memcpy_async`).

A synchronization object could be a `cuda::barrier` or a `cuda::pipeline`. These objects are explained in detail in *Asynchronous Barrier* and *Asynchronous Data Copies using cuda::pipeline*. These synchronization objects can be used at different thread scopes. A scope defines the set of threads that may use the synchronization object to synchronize with the asynchronous operation. The following table defines the thread scopes available in CUDA C++ and the threads that can be synchronized with each.

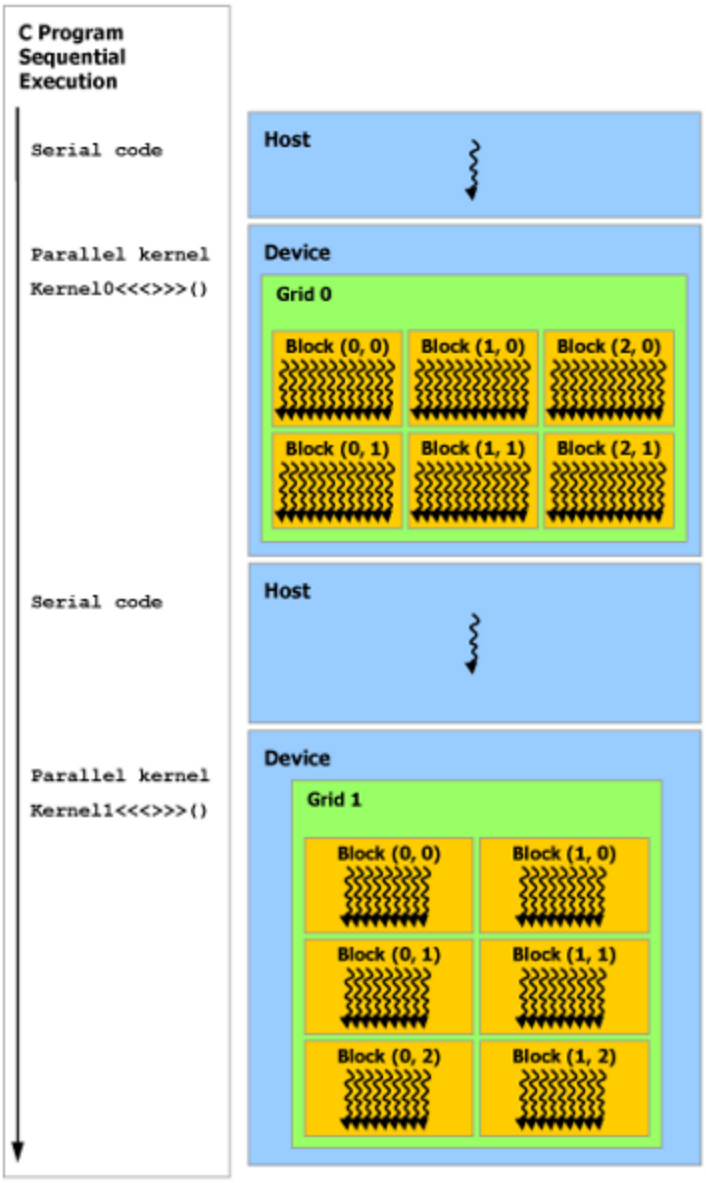


Figure 7: Heterogeneous Programming

Note: Serial code executes on the host while parallel code executes on the device.

Thread Scope	Description
`cuda::thread_scope::thread_scope_thread`	Only the CUDA thread which initiated asynchronous operations synchronizes.
`cuda::thread_scope::thread_scope_block`	All or any CUDA threads within the same thread block as the initiating thread synchronizes.
`cuda::thread_scope::thread_scope_device`	All or any CUDA threads in the same GPU device as the initiating thread synchronizes.
`cuda::thread_scope::thread_scope_system`	All or any CUDA or CPU threads in the same system as the initiating thread synchronizes.

These thread scopes are implemented as extensions to standard C++ in the CUDA Standard C++ library.

5.6. Compute Capability

The *compute capability* of a device is represented by a version number, also sometimes called its "SM version". This version number identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU.

The compute capability comprises a major revision number *X* and a minor revision number *Y* and is denoted by *X.Y*.

The major revision number indicates the core GPU architecture of a device. Devices with the same major revision number share the same fundamental architecture. The table below lists the major revision numbers corresponding to each NVIDIA GPU architecture.

Table 2: GPU Architecture and Major Revision Numbers

Major Revision Number	NVIDIA GPU Architecture
9	NVIDIA Hopper GPU Architecture
8	NVIDIA Ampere GPU Architecture
7	NVIDIA Volta GPU Architecture
6	NVIDIA Pascal GPU Architecture
5	NVIDIA Maxwell GPU Architecture
3	NVIDIA Kepler GPU Architecture

The minor revision number corresponds to an incremental improvement to the core architecture, pos-

sibly including new features.

Table 3: Incremental Updates in GPU Architectures

Compute Capability	NVIDIA GPU Architecture	Based On
7.5	NVIDIA Turing GPU Architecture	NVIDIA Volta GPU Architecture

CUDA-Enabled GPUs lists of all CUDA-enabled devices along with their compute capability. *Compute Capabilities* gives the technical specifications of each compute capability.

Note: The compute capability version of a particular GPU should not be confused with the CUDA version (for example, CUDA 7.5, CUDA 8, CUDA 9), which is the version of the CUDA *software platform*. The CUDA platform is used by application developers to create applications that run on many generations of GPU architectures, including future GPU architectures yet to be invented. While new versions of the CUDA platform often add native support for a new GPU architecture by supporting the compute capability version of that architecture, new versions of the CUDA platform typically also include software features that are independent of hardware generation.

The *Tesla* and *Fermi* architectures are no longer supported starting with CUDA 7.0 and CUDA 9.0, respectively.

Chapter 6. Programming Interface

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

CUDA C++ provides a simple path for users familiar with the C++ programming language to easily write programs for execution by the device.

It consists of a minimal set of extensions to the C++ language and a runtime library.

The core language extensions have been introduced in *Programming Model*. They allow programmers to define a kernel as a C++ function and use some new syntax to specify the grid and block dimension each time the function is called. A complete description of all extensions can be found in *C++ Language Extensions*. Any source file that contains some of these extensions must be compiled with nvcc as outlined in *Compilation with NVCC*.

The runtime is introduced in *CUDA Runtime*. It provides C and C++ functions that execute on the host to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems with multiple devices, etc. A complete description of the runtime can be found in the CUDA reference manual.

The runtime is built on top of a lower-level C API, the CUDA driver API, which is also accessible by the application. The driver API provides an additional level of control by exposing lower-level concepts such as CUDA contexts - the analogue of host processes for the device - and CUDA modules - the analogue of dynamically loaded libraries for the device. Most applications do not use the driver API as they do not need this additional level of control and when using the runtime, context and module management are implicit, resulting in more concise code. As the runtime is interoperable with the driver API, most applications that need some driver API features can default to use the runtime API and only use the driver API where needed. The driver API is introduced in *Driver API* and fully described in the reference manual.

6.1. Compilation with NVCC

Kernels can be written using the CUDA instruction set architecture, called *PTX*, which is described in the PTX reference manual. It is however usually more effective to use a high-level programming language such as C++. In both cases, kernels must be compiled into binary code by nvcc to execute on the device.

nvcc is a compiler driver that simplifies the process of compiling *C++* or *PTX* code: It provides simple and familiar command line options and executes them by invoking the collection of tools that imple-

ment the different compilation stages. This section gives an overview of nvcc workflow and command options. A complete description can be found in the nvcc user manual.

6.1.1. Compilation Workflow

6.1.1.1 Offline Compilation

Source files compiled with nvcc can include a mix of host code (i.e., code that executes on the host) and device code (i.e., code that executes on the device). nvcc's basic workflow consists in separating device code from host code and then:

- compiling the device code into an assembly form (*PTX* code) and/or binary form (*cubin* object),
- and modifying the host code by replacing the `<<<...>>>` syntax introduced in *Kernels* (and described in more details in *Execution Configuration*) by the necessary CUDA runtime function calls to load and launch each compiled kernel from the *PTX* code and/or *cubin* object.

The modified host code is output either as C++ code that is left to be compiled using another tool or as object code directly by letting nvcc invoke the host compiler during the last compilation stage.

Applications can then:

- Either link to the compiled host code (this is the most common case),
- Or ignore the modified host code (if any) and use the CUDA driver API (see *Driver API*) to load and execute the *PTX* code or *cubin* object.

6.1.1.2 Just-in-Time Compilation

Any *PTX* code loaded by an application at runtime is compiled further to binary code by the device driver. This is called *just-in-time compilation*. Just-in-time compilation increases application load time, but allows the application to benefit from any new compiler improvements coming with each new device driver. It is also the only way for applications to run on devices that did not exist at the time the application was compiled, as detailed in *Application Compatibility*.

When the device driver just-in-time compiles some *PTX* code for some application, it automatically caches a copy of the generated binary code in order to avoid repeating the compilation in subsequent invocations of the application. The cache - referred to as *compute cache* - is automatically invalidated when the device driver is upgraded, so that applications can benefit from the improvements in the new just-in-time compiler built into the device driver.

Environment variables are available to control just-in-time compilation as described in *CUDA Environment Variables*

As an alternative to using nvcc to compile CUDA C++ device code, NVRTC can be used to compile CUDA C++ device code to PTX at runtime. NVRTC is a runtime compilation library for CUDA C++; more information can be found in the NVRTC User guide.

6.1.2. Binary Compatibility

Binary code is architecture-specific. A *cubin* object is generated using the compiler option `-code` that specifies the targeted architecture: For example, compiling with `-code=sm_80` produces binary code for devices of *compute capability* 8.0. Binary compatibility is guaranteed from one minor revision to the next one, but not from one minor revision to the previous one or across major revisions. In other words, a *cubin* object generated for compute capability *X.y* will only execute on devices of compute capability *X.z* where *z≥y*.

Note: Binary compatibility is supported only for the desktop. It is not supported for Tegra. Also, the binary compatibility between desktop and Tegra is not supported.

6.1.3. PTX Compatibility

Some *PTX* instructions are only supported on devices of higher compute capabilities. For example, *Warp Shuffle Functions* are only supported on devices of compute capability 5.0 and above. The `-arch` compiler option specifies the compute capability that is assumed when compiling C++ to *PTX* code. So, code that contains warp shuffle, for example, must be compiled with `-arch=compute_50` (or higher).

PTX code produced for some specific compute capability can always be compiled to binary code of greater or equal compute capability. Note that a binary compiled from an earlier PTX version may not make use of some hardware features. For example, a binary targeting devices of compute capability 7.0 (Volta) compiled from PTX generated for compute capability 6.0 (Pascal) will not make use of Tensor Core instructions, since these were not available on Pascal. As a result, the final binary may perform worse than would be possible if the binary were generated using the latest version of PTX.

PTX code compiled to target *Architecture-Specific Features* only runs on the exact same physical architecture and nowhere else. Architecture-specific *PTX* code is not forward and backward compatible. Example code compiled with `sm_90a` or `compute_90a` only runs on devices with compute capability 9.0 and is not backward or forward compatible.

PTX code compiled to target *Family-Specific Features* only runs on the exact same physical architecture and other architectures in the same family. Family-specific *PTX* code is forward compatible with other devices in the same family, and is not backward compatible. Example code compiled with `sm_100f` or `compute_100f` only runs on devices with compute capability 10.0 and 10.3. Table 25 shows the compatibility of family-specific targets with compute capability.

6.1.4. Application Compatibility

To execute code on devices of specific compute capability, an application must load binary or *PTX* code that is compatible with this compute capability as described in *Binary Compatibility* and *PTX Compatibility*. In particular, to be able to execute code on future architectures with higher compute capability (for which no binary code can be generated yet), an application must load *PTX* code that will be just-in-time compiled for these devices (see *Just-in-Time Compilation*).

Which *PTX* and binary code gets embedded in a CUDA C++ application is controlled by the `-arch` and `-code` compiler options or the `-gencode` compiler option as detailed in the `nvcc` user manual. For example,

```
nvcc x.cu
        -gencode arch=compute_50,code=sm_50
        -gencode arch=compute_60,code=sm_60
        -gencode arch=compute_70,code=\"compute_70,sm_70\"
```

embeds binary code compatible with compute capability 5.0 and 6.0 (first and second `-gencode` options) and *PTX* and binary code compatible with compute capability 7.0 (third `-gencode` option).

Host code is generated to automatically select at runtime the most appropriate code to load and execute, which, in the above example, will be:

- 5.0 binary code for devices with compute capability 5.0 and 5.2,
- 6.0 binary code for devices with compute capability 6.0 and 6.1,
- 7.0 binary code for devices with compute capability 7.0 and 7.5,
- *PTX* code which is compiled to binary code at runtime for devices with compute capability later than 7.5

`x.cu` can have an optimized code path that uses warp reduction operations, for example, which are only supported in devices of compute capability 8.0 and higher. The `__CUDA_ARCH__` macro can be used to differentiate various code paths based on compute capability. It is only defined for device code. When compiling with `-arch=compute_80` for example, `__CUDA_ARCH__` is equal to `800`.

If `x.cu` is compiled for *Family-Specific Features* with `sm_100f` or `compute_100f`, the code can only run on devices in that specific family, which are devices with compute capability 10.0 and 10.3. For family-specific code targets an additional macro `__CUDA_ARCH_FAMILY_SPECIFIC__` is defined. In this example, `__CUDA_ARCH_FAMILY_SPECIFIC__` is equal to `1000`.

If `x.cu` is compiled for *Architecture-Specific Features* with `sm_100a` or `compute_100a`, the code can only run on devices with compute capability 10.0. For architecture-specific code targets an additional macro `__CUDA_ARCH_SPECIFIC__` is defined. In this example, `__CUDA_ARCH_SPECIFIC__` is equal to `1000`. Because architecture-specific features are a superset of family-specific features, the family-specific macro `__CUDA_ARCH_FAMILY_SPECIFIC__` is also defined and is equal to `1000`.

Applications using the driver API must compile code to separate files and explicitly load and execute the most appropriate file at runtime.

The Volta architecture introduces *Independent Thread Scheduling* which changes the way threads are scheduled on the GPU. For code relying on specific behavior of *SIMT scheduling* in previous architectures, Independent Thread Scheduling may alter the set of participating threads, leading to incorrect results. To aid migration while implementing the corrective actions detailed in *Independent Thread Scheduling*, Volta developers can opt-in to Pascal's thread scheduling with the compiler option combination `-arch=compute_60 -code=sm_70`.

The `nvcc` user manual lists various shorthands for the `-arch`, `-code`, and `-gencode` compiler options. For example, `-arch=sm_70` is a shorthand for `-arch=compute_70 -code=compute_70,sm_70` (which is the same as `-gencode arch=compute_70,code=\"compute_70,sm_70\"`).

6.1.5. C++ Compatibility

The front end of the compiler processes CUDA source files according to C++ syntax rules. Full C++ is supported for the host code. However, only a subset of C++ is fully supported for the device code as described in *C++ Language Support*.

6.1.6. 64-Bit Compatibility

The 64-bit version of `nvcc` compiles device code in 64-bit mode (i.e., pointers are 64-bit). Device code compiled in 64-bit mode is only supported with host code compiled in 64-bit mode.

6.2. CUDA Runtime

The runtime is implemented in the `cudart` library, which is linked to the application, either statically via `cudart.lib` or `libcudart.a`, or dynamically via `cudart.dll` or `libcudart.so`. Applications that require `cudart.dll` and/or `cudart.so` for dynamic linking typically include them as part of the application installation package. It is only safe to pass the address of CUDA runtime symbols between components that link to the same instance of the CUDA runtime.

All its entry points are prefixed with `cuda`.

As mentioned in *Heterogeneous Programming*, the CUDA programming model assumes a system composed of a host and a device, each with their own separate memory. *Device Memory* gives an overview of the runtime functions used to manage device memory.

Shared Memory illustrates the use of shared memory, introduced in *Thread Hierarchy*, to maximize performance.

Page-Locked Host Memory introduces page-locked host memory that is required to overlap kernel execution with data transfers between host and device memory.

Asynchronous Concurrent Execution describes the concepts and API used to enable asynchronous concurrent execution at various levels in the system.

Multi-Device System shows how the programming model extends to a system with multiple devices attached to the same host.

Error Checking describes how to properly check the errors generated by the runtime.

Call Stack mentions the runtime functions used to manage the CUDA C++ call stack.

Texture and Surface Memory presents the texture and surface memory spaces that provide another way to access device memory; they also expose a subset of the GPU texturing hardware.

Graphics Interoperability introduces the various functions the runtime provides to interoperate with the two main graphics APIs, OpenGL and Direct3D.

6.2.1. Initialization

As of CUDA 12.0, the `cudaInitDevice()` and `cudaSetDevice()` calls initialize the runtime and the primary context associated with the specified device. Absent these calls, the runtime will implicitly use device 0 and self-initialize as needed to process other runtime API requests. One needs to keep this in mind when timing runtime function calls and when interpreting the error code from the first call into the runtime. Before 12.0, `cudaSetDevice()` would not initialize the runtime and applications would often use the no-op runtime call `cudaFree(0)` to isolate the runtime initialization from other api activity (both for the sake of timing and error handling).

The runtime creates a CUDA context for each device in the system (see *Context* for more details on CUDA contexts). This context is the *primary context* for this device and is initialized at the first runtime function which requires an active context on this device. It is shared among all the host threads of the application. As part of this context creation, the device code is just-in-time compiled if necessary (see *Just-in-Time Compilation*) and loaded into device memory. This all happens transparently. If needed, for example, for driver API interoperability, the primary context of a device can be accessed from the driver API as described in *Interoperability between Runtime and Driver APIs*.

When a host thread calls `cudaDeviceReset()`, this destroys the primary context of the device the host thread currently operates on (that is, the current device as defined in *Device Selection*). The next runtime function call made by any host thread that has this device as current will create a new primary context for this device.

Note: The CUDA interfaces use global state that is initialized during host program initiation and destroyed during host program termination. The CUDA runtime and driver cannot detect if this state is invalid, so using any of these interfaces (implicitly or explicitly) during program initiation or termination after main) will result in undefined behavior.

As of CUDA 12.0, `cudaSetDevice()` will now explicitly initialize the runtime after changing the current device for the host thread. Previous versions of CUDA delayed runtime initialization on the new device until the first runtime call was made after `cudaSetDevice()`. This change means that it is now very important to check the return value of `cudaSetDevice()` for initialization errors.

The runtime functions from the error handling and version management sections of the reference manual do not initialize the runtime.

6.2.2. Device Memory

As mentioned in *Heterogeneous Programming*, the CUDA programming model assumes a system composed of a host and a device, each with their own separate memory. Kernels operate out of device memory, so the runtime provides functions to allocate, deallocate, and copy device memory, as well as transfer data between host memory and device memory.

Device memory can be allocated either as *linear memory* or as *CUDA arrays*.

CUDA arrays are opaque memory layouts optimized for texture fetching. They are described in *Texture and Surface Memory*.

Linear memory is allocated in a single unified address space, which means that separately allocated entities can reference one another via pointers, for example, in a binary tree or linked list. The size of the address space depends on the host system (CPU) and the compute capability of the used GPU:

Table 4: Linear Memory Address Space

	x86_64 (AMD64)	POWER (ppc64le)	ARM64
up to compute capability 5.3 (Maxwell)	40bit	40bit	40bit
compute capability 6.0 (Pascal) or newer	up to 47bit	up to 49bit	up to 48bit

Note: On devices of compute capability 5.3 (Maxwell) and earlier, the CUDA driver creates an uncommitted 40bit virtual address reservation to ensure that memory allocations (pointers) fall into the supported range. This reservation appears as reserved virtual memory, but does not occupy any physical memory until the program actually allocates memory.

Linear memory is typically allocated using `cudaMalloc()` and freed using `cudaFree()` and data transfer between host memory and device memory are typically done using `cudaMemcpy()`. In the vector addition code sample of *Kernels*, the vectors need to be copied from host memory to device memory:

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    float* h_C = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
```

(continues on next page)

(continued from previous page)

```
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

Linear memory can also be allocated through `cudaMallocPitch()` and `cudaMalloc3D()`. These functions are recommended for allocations of 2D or 3D arrays as it makes sure that the allocation is appropriately padded to meet the alignment requirements described in *Device Memory Accesses*, therefore ensuring best performance when accessing the row addresses or performing copies between 2D arrays and other regions of device memory (using the `cudaMemcpy2D()` and `cudaMemcpy3D()` functions). The returned pitch (or stride) must be used to access array elements. The following code sample allocates a `width` x `height` 2D array of floating-point values and shows how to loop over the array elements in device code:

```
// Host code
int width = 64, height = 64;
float* devPtr;
size_t pitch;
cudaMallocPitch(&devPtr, &pitch,
                width * sizeof(float), height);
MyKernel<<<100, 512>>>(devPtr, pitch, width, height);

// Device code
__global__ void MyKernel(float* devPtr,
                         size_t pitch, int width, int height)
{
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c) {
            float element = row[c];
        }
    }
}
```

The following code sample allocates a `width` x `height` x `depth` 3D array of floating-point values and shows how to loop over the array elements in device code:

```
// Host code
int width = 64, height = 64, depth = 64;
cudaExtent extent = make_cudaExtent(width * sizeof(float),
                                    height, depth);
cudaPitchedPtr devPitchedPtr;
cudaMalloc3D(&devPitchedPtr, extent);
MyKernel<<<100, 512>>>(devPitchedPtr, width, height, depth);

// Device code
```

(continues on next page)

(continued from previous page)

```
__global__ void MyKernel(cudaPitchedPtr devPitchedPtr,
                         int width, int height, int depth)
{
    char* devPtr = devPitchedPtr.ptr;
    size_t pitch = devPitchedPtr.pitch;
    size_t slicePitch = pitch * height;
    for (int z = 0; z < depth; ++z) {
        char* slice = devPtr + z * slicePitch;
        for (int y = 0; y < height; ++y) {
            float* row = (float*)(slice + y * pitch);
            for (int x = 0; x < width; ++x) {
                float element = row[x];
            }
        }
    }
}
```

Note: To avoid allocating too much memory and thus impacting system-wide performance, request the allocation parameters from the user based on the problem size. If the allocation fails, you can fallback to other slower memory types (`cudaMallocHost()`, `cudaHostRegister()`, etc.), or return an error telling the user how much memory was needed that was denied. If your application cannot request the allocation parameters for some reason, we recommend using `cudaMallocManaged()` for platforms that support it.

The reference manual lists all the various functions used to copy memory between linear memory allocated with `cudaMalloc()`, linear memory allocated with `cudaMallocPitch()` or `cudaMalloc3D()`, CUDA arrays, and memory allocated for variables declared in global or constant memory space.

The following code sample illustrates various ways of accessing global variables via the runtime API:

```
__constant__ float constData[256];
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data));
cudaMemcpyFromSymbol(data, constData, sizeof(data));

__device__ float devData;
float value = 3.14f;
cudaMemcpyToSymbol(devData, &value, sizeof(float));

__device__ float* devPointer;
float* ptr;
cudaMalloc(&ptr, 256 * sizeof(float));
cudaMemcpyToSymbol(devPointer, &ptr, sizeof(ptr));
```

`cudaGetSymbolAddress()` is used to retrieve the address pointing to the memory allocated for a variable declared in global memory space. The size of the allocated memory is obtained through `cudaGetSymbolSize()`.

6.2.3. Device Memory L2 Access Management

When a CUDA kernel accesses a data region in the global memory repeatedly, such data accesses can be considered to be *persisting*. On the other hand, if the data is only accessed once, such data accesses can be considered to be *streaming*.

Starting with CUDA 11.0, devices of compute capability 8.0 and above have the capability to influence persistence of data in the L2 cache, potentially providing higher bandwidth and lower latency accesses to global memory.

6.2.3.1 L2 Cache Set-Aside for Persisting Accesses

A portion of the L2 cache can be set aside to be used for persisting data accesses to global memory. Persisting accesses have prioritized use of this set-aside portion of L2 cache, whereas normal or streaming, accesses to global memory can only utilize this portion of L2 when it is unused by persisting accesses.

The L2 cache set-aside size for persisting accesses may be adjusted, within limits:

```
cudaGetDeviceProperties(&prop, device_id);
size_t size = min(int(prop.l2CacheSize * 0.75), prop.persistingL2CacheMaxSize);
cudaDeviceSetLimit(cudaLimitPersistingL2CacheSize, size); /* set-aside 3/4 of L2 cache
↪for persisting accesses or the max allowed*/
```

When the GPU is configured in Multi-Instance GPU (MIG) mode, the L2 cache set-aside functionality is disabled.

When using the Multi-Process Service (MPS), the L2 cache set-aside size cannot be changed by `cudaDeviceSetLimit`. Instead, the set-aside size can only be specified at start up of MPS server through the environment variable `CUDA_DEVICE_DEFAULT_PERSISTING_L2_CACHE_PERCENTAGE_LIMIT`.

6.2.3.2 L2 Policy for Persisting Accesses

An access policy window specifies a contiguous region of global memory and a persistence property in the L2 cache for accesses within that region.

The code example below shows how to set an L2 persisting access window using a CUDA Stream.

CUDA Stream Example

```
cudaStreamAttrValue stream_attribute;                                         //
↪Stream level attributes data structure
stream_attribute.accessPolicyWindow.base_ptr  = reinterpret_cast<void*>(ptr); //
↪Global Memory data pointer
stream_attribute.accessPolicyWindow.num_bytes = num_bytes;                    //
↪Number of bytes for persistence access.
                                                                              // (Must
↪be less than cudaDeviceProp::accessPolicyMaxWindowSize)
stream_attribute.accessPolicyWindow.hitRatio  = 0.6;                          // Hint
↪for cache hit ratio
stream_attribute.accessPolicyWindow.hitProp   = cudaAccessPropertyPersisting; // Type
↪of access property on cache hit
stream_attribute.accessPolicyWindow.missProp  = cudaAccessPropertyStreaming;  // Type
↪of access property on cache miss.
```

(continues on next page)

(continued from previous page)

```
//Set the attributes to a CUDA stream of type cudaStream_t
cudaStreamSetAttribute(stream, cudaStreamAttributeAccessPolicyWindow, &stream_
↪attribute);
```

When a kernel subsequently executes in CUDA `stream`, memory accesses within the global memory extent `[ptr..ptr+num_bytes)` are more likely to persist in the L2 cache than accesses to other global memory locations.

L2 persistence can also be set for a CUDA Graph Kernel Node as shown in the example below:

CUDA GraphKernelNode Example

```
cudaKernelNodeAttrValue node_attribute;                                 // Kernel
↪level attributes data structure
node_attribute.accessPolicyWindow.base_ptr  = reinterpret_cast<void*>(ptr); // Global
↪Memory data pointer
node_attribute.accessPolicyWindow.num_bytes = num_bytes;                // Number
↪of bytes for persistence access.
                                                                        // (Must
↪be less than cudaDeviceProp::accessPolicyMaxWindowSize)
node_attribute.accessPolicyWindow.hitRatio  = 0.6;                      // Hint
↪for cache hit ratio
node_attribute.accessPolicyWindow.hitProp   = cudaAccessPropertyPersisting; // Type of
↪access property on cache hit
node_attribute.accessPolicyWindow.missProp  = cudaAccessPropertyStreaming;  // Type of
↪access property on cache miss.

//Set the attributes to a CUDA Graph Kernel node of type cudaGraphNode_t
cudaGraphKernelNodeSetAttribute(node, cudaKernelNodeAttributeAccessPolicyWindow, &
↪node_attribute);
```

The `hitRatio` parameter can be used to specify the fraction of accesses that receive the `hitProp` property. In both of the examples above, 60% of the memory accesses in the global memory region `[ptr..ptr+num_bytes)` have the persisting property and 40% of the memory accesses have the streaming property. Which specific memory accesses are classified as persisting (the `hitProp`) is random with a probability of approximately `hitRatio`; the probability distribution depends upon the hardware architecture and the memory extent.

For example, if the L2 set-aside cache size is 16KB and the `num_bytes` in the `accessPolicyWindow` is 32KB:

- With a `hitRatio` of 0.5, the hardware will select, at random, 16KB of the 32KB window to be designated as persisting and cached in the set-aside L2 cache area.
- With a `hitRatio` of 1.0, the hardware will attempt to cache the whole 32KB window in the set-aside L2 cache area. Since the set-aside area is smaller than the window, cache lines will be evicted to keep the most recently used 16KB of the 32KB data in the set-aside portion of the L2 cache.

The `hitRatio` can therefore be used to avoid thrashing of cache lines and overall reduce the amount of data moved into and out of the L2 cache.

A `hitRatio` value below 1.0 can be used to manually control the amount of data different `accessPolicyWindows` from concurrent CUDA streams can cache in L2. For example, let the L2 set-aside cache size be 16KB; two concurrent kernels in two different CUDA streams, each with a 16KB `accessPolicyWindow`, and both with `hitRatio` value 1.0, might evict each others' cache lines when competing for the shared L2 resource. However, if both `accessPolicyWindows` have a hitRatio value of 0.5, they will be less likely to evict their own or each others' persisting cache lines.

6.2.3.3 L2 Access Properties

Three types of access properties are defined for different global memory data accesses:

1. `cudaAccessPropertyStreaming`: Memory accesses that occur with the streaming property are less likely to persist in the L2 cache because these accesses are preferentially evicted.
2. `cudaAccessPropertyPersisting`: Memory accesses that occur with the persisting property are more likely to persist in the L2 cache because these accesses are preferentially retained in the set-aside portion of L2 cache.
3. `cudaAccessPropertyNormal`: This access property forcibly resets previously applied persisting access property to a normal status. Memory accesses with the persisting property from previous CUDA kernels may be retained in L2 cache long after their intended use. This persistence-after-use reduces the amount of L2 cache available to subsequent kernels that do not use the persisting property. Resetting an access property window with the `cudaAccessPropertyNormal` property removes the persisting (preferential retention) status of the prior access, as if the prior access had been without an access property.

6.2.3.4 L2 Persistence Example

The following example shows how to set-aside L2 cache for persistent accesses, use the set-aside L2 cache in CUDA kernels via CUDA Stream and then reset the L2 cache.

```
cudaStream_t stream;
cudaStreamCreate(&stream);
↪       // Create CUDA stream

cudaDeviceProp prop;
↪       // CUDA device properties variable
cudaGetDeviceProperties( &prop, device_id);
↪       // Query GPU properties
size_t size = min( int(prop.l2CacheSize * 0.75) , prop.persistingL2CacheMaxSize );
cudaDeviceSetLimit( cudaLimitPersistingL2CacheSize, size);
↪       // set-aside 3/4 of L2 cache for persisting accesses or the max allowed

size_t window_size = min(prop.accessPolicyMaxWindowSize, num_bytes);
↪       // Select minimum of user defined num_bytes and max window size.

cudaStreamAttrValue stream_attribute;
↪       // Stream level attributes data structure
stream_attribute.accessPolicyWindow.base_ptr  = reinterpret_cast<void*>(data1);
↪         // Global Memory data pointer
stream_attribute.accessPolicyWindow.num_bytes = window_size;
↪       // Number of bytes for persistence access
stream_attribute.accessPolicyWindow.hitRatio  = 0.6;
↪       // Hint for cache hit ratio
stream_attribute.accessPolicyWindow.hitProp   = cudaAccessPropertyPersisting;
↪       // Persistence Property
stream_attribute.accessPolicyWindow.missProp  = cudaAccessPropertyStreaming;
↪       // Type of access property on cache miss

cudaStreamSetAttribute(stream, cudaStreamAttributeAccessPolicyWindow, &stream_
↪attribute);   // Set the attributes to a CUDA Stream

for(int i = 0; i < 10; i++) {
```

(continues on next page)

(continued from previous page)

```
    cuda_kernelA<<<grid_size,block_size,0,stream>>>(data1);
↪       // This data1 is used by a kernel multiple times
}
↪       // [data1 + num_bytes) benefits from L2 persistence
cuda_kernelB<<<grid_size,block_size,0,stream>>>(data1);
↪       // A different kernel in the same stream can also benefit

↪       // from the persistence of data1

stream_attribute.accessPolicyWindow.num_bytes = 0;
↪       // Setting the window size to 0 disable it
cudaStreamSetAttribute(stream, cudaStreamAttributeAccessPolicyWindow, &stream_
↪attribute);   // Overwrite the access policy attribute to a CUDA Stream
cudaCtxResetPersistingL2Cache();
↪       // Remove any persistent lines in L2

cuda_kernelC<<<grid_size,block_size,0,stream>>>(data2);
↪       // data2 can now benefit from full L2 in normal mode
```

6.2.3.5 Reset L2 Access to Normal

A persisting L2 cache line from a previous CUDA kernel may persist in L2 long after it has been used. Hence, a reset to normal for L2 cache is important for streaming or normal memory accesses to utilize the L2 cache with normal priority. There are three ways a persisting access can be reset to normal status.

1. Reset a previous persisting memory region with the access property, `cudaAccessPropertyNormal`.
2. Reset all persisting L2 cache lines to normal by calling `cudaCtxResetPersistingL2Cache()`.
3. **Eventually** untouched lines are automatically reset to normal. Reliance on automatic reset is strongly discouraged because of the undetermined length of time required for automatic reset to occur.

6.2.3.6 Manage Utilization of L2 set-aside cache

Multiple CUDA kernels executing concurrently in different CUDA streams may have a different access policy window assigned to their streams. However, the L2 set-aside cache portion is shared among all these concurrent CUDA kernels. As a result, the net utilization of this set-aside cache portion is the sum of all the concurrent kernels' individual use. The benefits of designating memory accesses as persisting diminish as the volume of persisting accesses exceeds the set-aside L2 cache capacity.

To manage utilization of the set-aside L2 cache portion, an application must consider the following:

- Size of L2 set-aside cache.
- CUDA kernels that may concurrently execute.
- The access policy window for all the CUDA kernels that may concurrently execute.
- When and how L2 reset is required to allow normal or streaming accesses to utilize the previously set-aside L2 cache with equal priority.

6.2.3.7 Query L2 cache Properties

Properties related to L2 cache are a part of `cudaDeviceProp` struct and can be queried using CUDA runtime API `cudaGetDeviceProperties`

CUDA Device Properties include:

- `l2CacheSize`: The amount of available L2 cache on the GPU.
- `persistingL2CacheMaxSize`: The maximum amount of L2 cache that can be set-aside for persisting memory accesses.
- `accessPolicyMaxWindowSize`: The maximum size of the access policy window.

6.2.3.8 Control L2 Cache Set-Aside Size for Persisting Memory Access

The L2 set-aside cache size for persisting memory accesses is queried using CUDA runtime API `cudaDeviceGetLimit` and set using CUDA runtime API `cudaDeviceSetLimit` as a `cudaLimit`. The maximum value for setting this limit is `cudaDeviceProp::persistingL2CacheMaxSize`.

```
enum cudaLimit {
    /* other fields not shown */
    cudaLimitPersistingL2CacheSize
};
```

6.2.4. Shared Memory

As detailed in *Variable Memory Space Specifiers* shared memory is allocated using the `__shared__` memory space specifier.

Shared memory is expected to be much faster than global memory as mentioned in *Thread Hierarchy* and detailed in *Shared Memory*. It can be used as scratchpad memory (or software managed cache) to minimize global memory accesses from a CUDA block as illustrated by the following matrix multiplication example.

The following code sample is a straightforward implementation of matrix multiplication that does not take advantage of shared memory. Each thread reads one row of *A* and one column of *B* and computes the corresponding element of *C* as illustrated in Figure 8. *A* is therefore read *B.width* times from global memory and *B* is read *A.height* times.

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;

// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);
```

(continues on next page)

(continued from previous page)

```
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);

    // Allocate C in device memory
    Matrix d_C;
    d_C.width = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);

    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Read C from device memory
    cudaMemcpy(C.elements, d_C.elements, size,
               cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}

// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```

The following code sample is an implementation of matrix multiplication that does take advantage of shared memory. In this implementation, each thread block is responsible for computing one square sub-matrix *Csub* of *C* and each thread within the block is responsible for computing one element of *Csub*. As illustrated in Figure 9, *Csub* is equal to the product of two rectangular matrices: the sub-

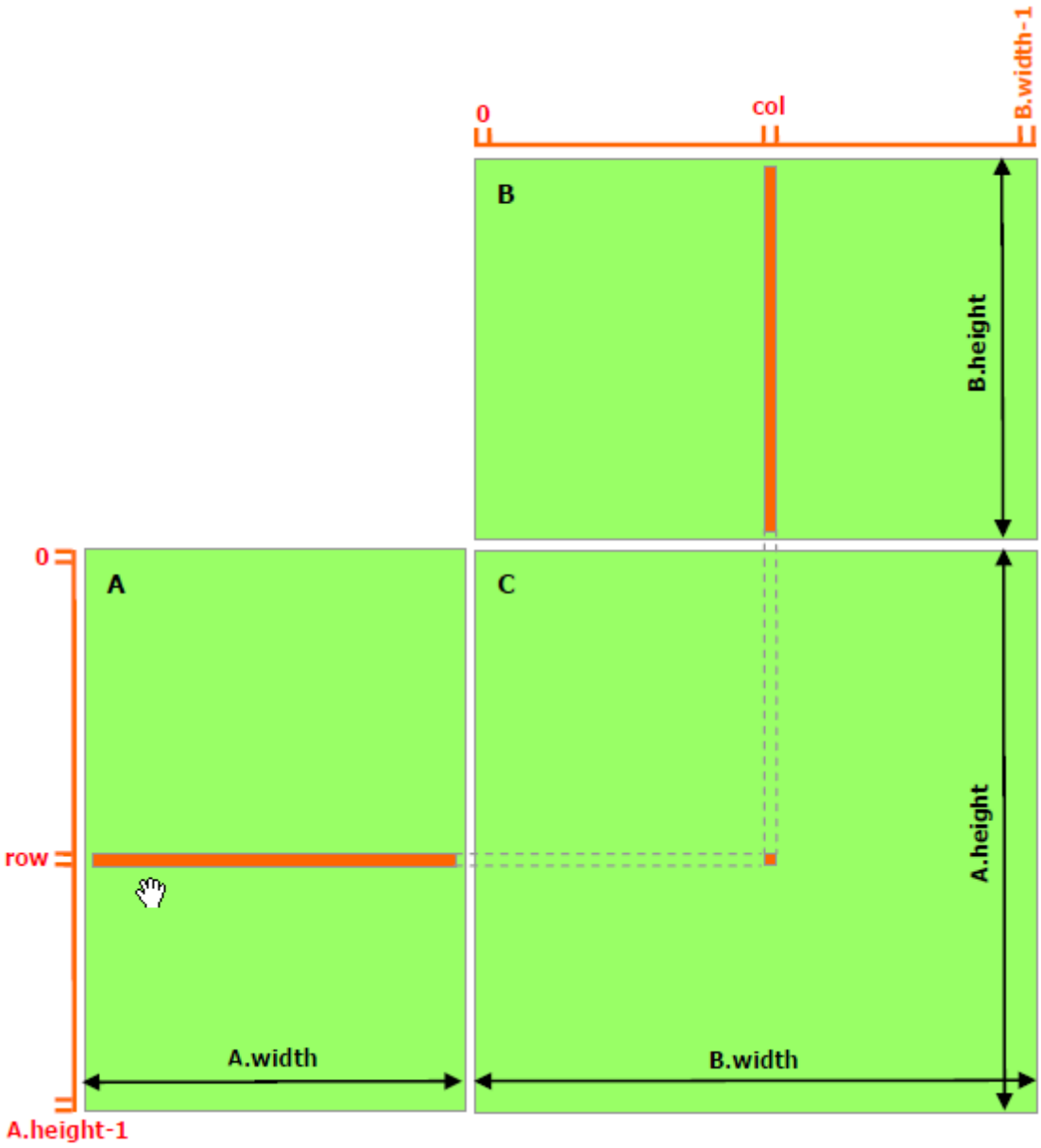


Figure 8: Matrix Multiplication without Shared Memory

matrix of *A* of dimension (*A.width, block_size*) that has the same row indices as *Csub*, and the sub-matrix of *B* of dimension (*block_size, A.width*)that has the same column indices as *Csub*. In order to fit into the device's resources, these two rectangular matrices are divided into as many square matrices of dimension *block_size* as necessary and *Csub* is computed as the sum of the products of these square matrices. Each of these products is performed by first loading the two corresponding square matrices from global memory to shared memory with one thread loading one element of each matrix, and then by having each thread compute one element of the product. Each thread accumulates the result of each of these products into a register and once done writes the result to global memory.

By blocking the computation this way, we take advantage of fast shared memory and save a lot of global memory bandwidth since *A* is only read (*B.width / block_size*) times from global memory and *B* is read (*A.height / block_size*) times.

The *Matrix* type from the previous code sample is augmented with a *stride* field, so that sub-matrices can be efficiently represented with the same type. *__device__* functions are used to get and set elements and build any sub-matrix from a matrix.

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
    int width;
    int height;
    int stride;
    float* elements;
} Matrix;
// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col)
{
    return A.elements[row * A.stride + col];
}
// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col,
                           float value)
{
    A.elements[row * A.stride + col] = value;
}
// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A
 __device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
    Matrix Asub;
    Asub.width    = BLOCK_SIZE;
    Asub.height   = BLOCK_SIZE;
    Asub.stride   = A.stride;
    Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
                                         + BLOCK_SIZE * col];
    return Asub;
}
// Thread block size
#define BLOCK_SIZE 16
// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
```

(continues on next page)

(continued from previous page)

```
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
    cudaMemcpyHostToDevice);
    // Allocate C in device memory
    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);
    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
    // Read C from device memory
    cudaMemcpy(C.elements, d_C.elements, size,
               cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
// Matrix multiplication kernel called by MatMul()
 __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Block row and column
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;
    // Each thread block computes one sub-matrix Csub of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
    // Each thread computes one element of Csub
    // by accumulating results into Cvalue
    float Cvalue = 0;
    // Thread row and column within Csub
    int row = threadIdx.y;
    int col = threadIdx.x;
    // Loop over all the sub-matrices of A and B that are
    // required to compute Csub
    // Multiply each pair of sub-matrices together
    // and accumulate the results
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
        // Get sub-matrix Asub of A
        Matrix Asub = GetSubMatrix(A, blockRow, m);
        // Get sub-matrix Bsub of B
        Matrix Bsub = GetSubMatrix(B, m, blockCol);
        // Shared memory used to store Asub and Bsub respectively
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        // Load Asub and Bsub from device memory to shared memory
```

(continues on next page)

(continued from previous page)

```
        // Each thread loads one element of each sub-matrix
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);
        // Synchronize to make sure the sub-matrices are loaded
        // before starting the computation
        __syncthreads();
        // Multiply Asub and Bsub together
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];
        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }
    // Write Csub to device memory
    // Each thread writes one element
    SetElement(Csub, row, col, Cvalue);
}
```

6.2.5. Distributed Shared Memory

Thread block clusters introduced in compute capability 9.0 provide the ability for threads in a thread block cluster to access shared memory of all the participating thread blocks in a cluster. This partitioned shared memory is called *Distributed Shared Memory*, and the corresponding address space is called Distributed shared memory address space. Threads that belong to a thread block cluster, can read, write or perform atomics in the distributed address space, regardless whether the address belongs to the local thread block or a remote thread block. Whether a kernel uses distributed shared memory or not, the shared memory size specifications, static or dynamic is still per thread block. The size of distributed shared memory is just the number of thread blocks per cluster multiplied by the size of shared memory per thread block.

Accessing data in distributed shared memory requires all the thread blocks to exist. A user can guarantee that all thread blocks have started executing using `cluster.sync()` from *Cluster Group* API. The user also needs to ensure that all distributed shared memory operations happen before the exit of a thread block, e.g., if a remote thread block is trying to read a given thread block's shared memory, user needs to ensure that the shared memory read by remote thread block is completed before it can exit.

CUDA provides a mechanism to access to distributed shared memory, and applications can benefit from leveraging its capabilities. Lets look at a simple histogram computation and how to optimize it on the GPU using thread block cluster. A standard way of computing histograms is do the computation in the shared memory of each thread block and then perform global memory atomics. A limitation of this approach is the shared memory capacity. Once the histogram bins no longer fit in the shared memory, a user needs to directly compute histograms and hence the atomics in the global memory. With distributed shared memory, CUDA provides an intermediate step, where a depending on the histogram bins size, histogram can be computed in shared memory, distributed shared memory or global memory directly.

The CUDA kernel example below shows how to compute histograms in shared memory or distributed shared memory, depending on the number of histogram bins.

```
#include <cooperative_groups.h>
```

(continues on next page)

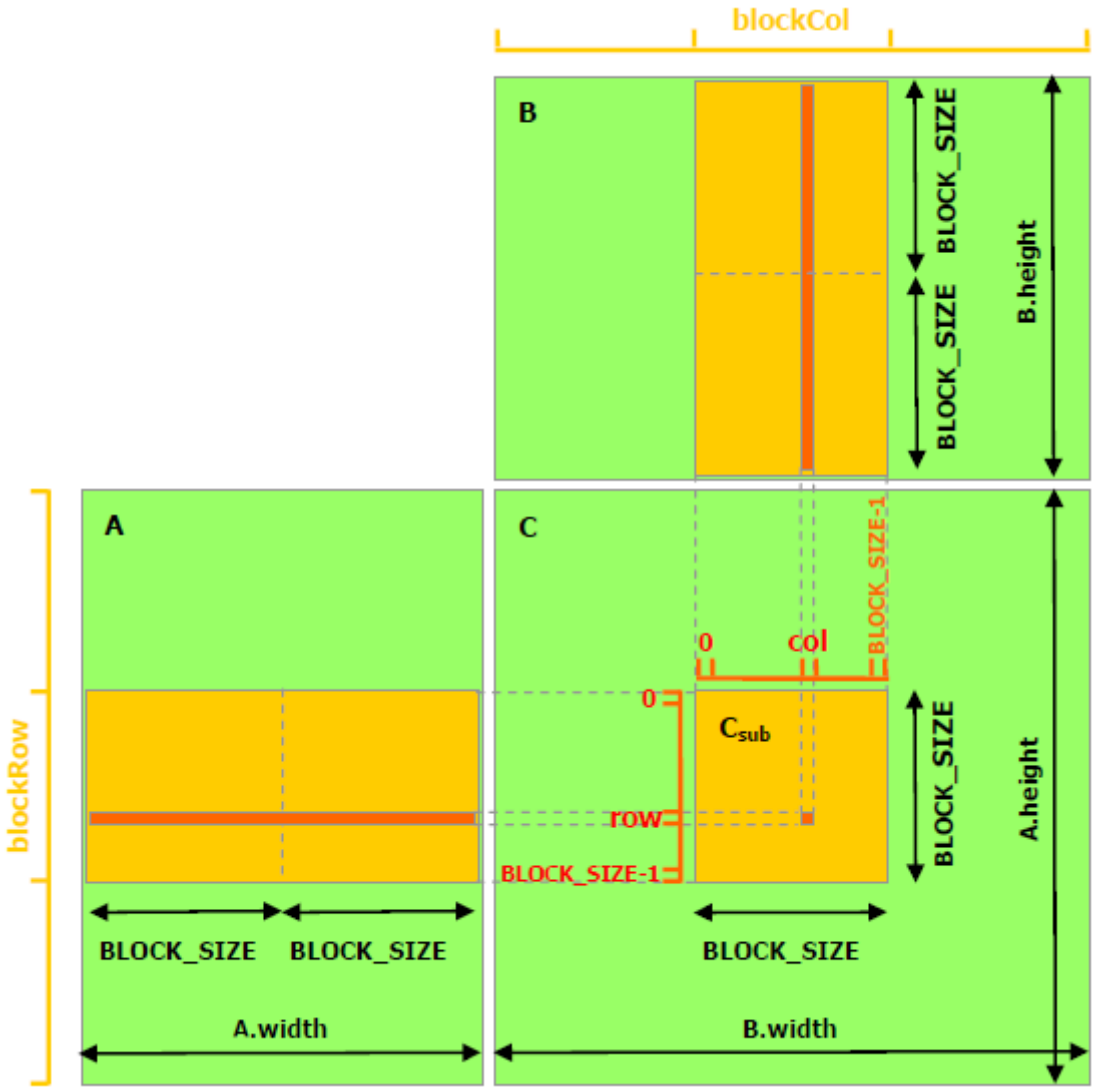


Figure 9: Matrix Multiplication with Shared Memory

(continued from previous page)

```
// Distributed Shared memory histogram kernel
__global__ void clusterHist_kernel(int *bins, const int nbins, const int bins_per_
↪block, const int *__restrict__ input,
                                   size_t array_size)
{
  extern __shared__ int smem[];
  namespace cg = cooperative_groups;
  int tid = cg::this_grid().thread_rank();

  // Cluster initialization, size and calculating local bin offsets.
  cg::cluster_group cluster = cg::this_cluster();
  unsigned int clusterBlockRank = cluster.block_rank();
  int cluster_size = cluster.dim_blocks().x;

  for (int i = threadIdx.x; i < bins_per_block; i += blockDim.x)
  {
    smem[i] = 0; //Initialize shared memory histogram to zeros
  }

  // cluster synchronization ensures that shared memory is initialized to zero in
  // all thread blocks in the cluster. It also ensures that all thread blocks
  // have started executing and they exist concurrently.
  cluster.sync();

  for (int i = tid; i < array_size; i += blockDim.x * gridDim.x)
  {
    int ldata = input[i];

    //Find the right histogram bin.
    int binid = ldata;
    if (ldata < 0)
      binid = 0;
    else if (ldata >= nbins)
      binid = nbins - 1;

    //Find destination block rank and offset for computing
    //distributed shared memory histogram
    int dst_block_rank = (int)(binid / bins_per_block);
    int dst_offset = binid % bins_per_block;

    //Pointer to target block shared memory
    int *dst_smem = cluster.map_shared_rank(smem, dst_block_rank);

    //Perform atomic update of the histogram bin
    atomicAdd(dst_smem + dst_offset, 1);
  }

  // cluster synchronization is required to ensure all distributed shared
  // memory operations are completed and no thread block exits while
  // other thread blocks are still accessing distributed shared memory
  cluster.sync();

  // Perform global memory histogram, using the local distributed memory histogram
  int *lbins = bins + cluster.block_rank() * bins_per_block;
  for (int i = threadIdx.x; i < bins_per_block; i += blockDim.x)
  {
```

(continues on next page)

(continued from previous page)

```
    atomicAdd(&lbins[i], smem[i]);
  }
}
```

The above kernel can be launched at runtime with a cluster size depending on the amount of distributed shared memory required. If histogram is small enough to fit in shared memory of just one block, user can launch kernel with cluster size 1. The code snippet below shows how to launch a cluster kernel dynamically based depending on shared memory requirements.

```
// Launch via extensible launch
{
  cudaLaunchConfig_t config = {0};
  config.gridDim = array_size / threads_per_block;
  config.blockDim = threads_per_block;

  // cluster_size depends on the histogram size.
  // ( cluster_size == 1 ) implies no distributed shared memory, just thread block
↪local shared memory
  int cluster_size = 2; // size 2 is an example here
  int nbins_per_block = nbins / cluster_size;

  //dynamic shared memory size is per block.
  //Distributed shared memory size =  cluster_size * nbins_per_block * sizeof(int)
  config.dynamicSmemBytes = nbins_per_block * sizeof(int);

  CUDA_CHECK(::cudaFuncSetAttribute((void *)clusterHist_kernel,
↪cudaFuncAttributeMaxDynamicSharedMemorySize, config.dynamicSmemBytes));

  cudaLaunchAttribute attribute[1];
  attribute[0].id = cudaLaunchAttributeClusterDimension;
  attribute[0].val.clusterDim.x = cluster_size;
  attribute[0].val.clusterDim.y = 1;
  attribute[0].val.clusterDim.z = 1;

  config.numAttrs = 1;
  config.attrs = attribute;

  cudaLaunchKernelEx(&config, clusterHist_kernel, bins, nbins, nbins_per_block, input,
↪ array_size);
}
```

6.2.6. Page-Locked Host Memory

The runtime provides functions to allow the use of *page-locked* (also known as *pinned*) host memory (as opposed to regular pageable host memory allocated by `malloc()`):

- `cudaHostAlloc()` and `cudaFreeHost()` allocate and free page-locked host memory;
- `cudaHostRegister()` page-locks a range of memory allocated by `malloc()` (see reference manual for limitations).

Using page-locked host memory has several benefits:

- Copies between page-locked host memory and device memory can be performed concurrently with kernel execution for some devices as mentioned in *Asynchronous Concurrent Execution*.

- On some devices, page-locked host memory can be mapped into the address space of the device, eliminating the need to copy it to or from device memory as detailed in *Mapped Memory*.
- On systems with a front-side bus, bandwidth between host memory and device memory is higher if host memory is allocated as page-locked and even higher if in addition it is allocated as write-combining as described in *Write-Combining Memory*.

Note: Page-locked host memory is not cached on non I/O coherent Tegra devices. Also, `cudaHostRegister()` is not supported on non I/O coherent Tegra devices.

The simple zero-copy CUDA sample comes with a detailed document on the page-locked memory APIs.

6.2.6.1 Portable Memory

A block of page-locked memory can be used in conjunction with any device in the system (see *Multi-Device System* for more details on multi-device systems), but by default, the benefits of using page-locked memory described above are only available in conjunction with the device that was current when the block was allocated (and with all devices sharing the same unified address space, if any, as described in *Unified Virtual Address Space*). To make these advantages available to all devices, the block needs to be allocated by passing the flag `cudaHostAllocPortable` to `cudaHostAlloc()` or page-locked by passing the flag `cudaHostRegisterPortable` to `cudaHostRegister()`.

6.2.6.2 Write-Combining Memory

By default page-locked host memory is allocated as cacheable. It can optionally be allocated as *write-combining* instead by passing flag `cudaHostAllocWriteCombined` to `cudaHostAlloc()`. Write-combining memory frees up the host's L1 and L2 cache resources, making more cache available to the rest of the application. In addition, write-combining memory is not snooped during transfers across the PCI Express bus, which can improve transfer performance by up to 40%.

Reading from write-combining memory from the host is prohibitively slow, so write-combining memory should in general be used for memory that the host only writes to.

Using CPU atomic instructions on WC memory should be avoided because not all CPU implementations guarantee that functionality.

6.2.6.3 Mapped Memory

A block of page-locked host memory can also be mapped into the address space of the device by passing flag `cudaHostAllocMapped` to `cudaHostAlloc()` or by passing flag `cudaHostRegisterMapped` to `cudaHostRegister()`. Such a block has therefore in general two addresses: one in host memory that is returned by `cudaHostAlloc()` or `malloc()`, and one in device memory that can be retrieved using `cudaHostGetDevicePointer()` and then used to access the block from within a kernel. The only exception is for pointers allocated with `cudaHostAlloc()` and when a unified address space is used for the host and the device as mentioned in *Unified Virtual Address Space*.

Accessing host memory directly from within a kernel does not provide the same bandwidth as device memory, but does have some advantages:

- There is no need to allocate a block in device memory and copy data between this block and the block in host memory; data transfers are implicitly performed as needed by the kernel;

- There is no need to use streams (see *Concurrent Data Transfers*) to overlap data transfers with kernel execution; the kernel-originated data transfers automatically overlap with kernel execution.

Since mapped page-locked memory is shared between host and device however, the application must synchronize memory accesses using streams or events (see *Asynchronous Concurrent Execution*) to avoid any potential read-after-write, write-after-read, or write-after-write hazards.

To be able to retrieve the device pointer to any mapped page-locked memory, page-locked memory mapping must be enabled by calling `cudaSetDeviceFlags()` with the `cudaDeviceMapHost` flag before any other CUDA call is performed. Otherwise, `cudaHostGetDevicePointer()` will return an error.

`cudaHostGetDevicePointer()` also returns an error if the device does not support mapped page-locked host memory. Applications may query this capability by checking the `canMapHostMemory` device property (see *Device Enumeration*), which is equal to 1 for devices that support mapped page-locked host memory.

Note that atomic functions (see *Atomic Functions*) operating on mapped page-locked memory are not atomic from the point of view of the host or other devices.

Also note that CUDA runtime requires that 1-byte, 2-byte, 4-byte, 8-byte, and 16-byte naturally aligned loads and stores to host memory initiated from the device are preserved as single accesses from the point of view of the host and other devices. On some platforms, atomics to memory may be broken by the hardware into separate load and store operations. These component load and store operations have the same requirements on preservation of naturally aligned accesses. The CUDA runtime does not support a PCI Express bus topology where a PCI Express bridge splits 8-byte naturally aligned operations and NVIDIA is not aware of any topology that splits 16-byte naturally aligned operations.

6.2.7. Memory Synchronization Domains

6.2.7.1 Memory Fence Interference

Some CUDA applications may see degraded performance due to memory fence/flush operations waiting on more transactions than those necessitated by the CUDA memory consistency model.

`__managed__ int x = 0;`
`__device__ cuda::atomic<int, cuda::thread_scope_device> a(0);`
`__managed__ cuda::atomic<int, cuda::thread_scope_system> b(0);`		

Thread 1 (SM)	Thread 2 (SM)	Thread 3 (CPU)

`x = 1;`
`a = 1;`
	`while (a != 1) ;`
	`assert(x == 1);`
	`b = 1;`
		`while (b != 1) ;`
		`assert(x == 1);`

Consider the example above. The CUDA memory consistency model guarantees that the asserted condition will be true, so the write to x from thread 1 must be visible to thread 3, before the write to b from thread 2.

The memory ordering provided by the release and acquire of a is only sufficient to make x visible to thread 2, not thread 3, as it is a device-scope operation. The system-scope ordering provided by release and acquire of b, therefore, needs to ensure not only writes issued from thread 2 itself are visible to thread 3, but also writes from other threads that are visible to thread 2. This is known as cumulativity. As the GPU cannot know at the time of execution which writes have been guaranteed at the source level to be visible and which are visible only by chance timing, it must cast a conservatively wide net for in-flight memory operations.

This sometimes leads to interference: because the GPU is waiting on memory operations it is not required to at the source level, the fence/flush may take longer than necessary.

Note that fences may occur explicitly as intrinsics or atomics in code, like in the example, or implicitly to implement *synchronizes-with* relationships at task boundaries.

A common example is when a kernel is performing computation in local GPU memory, and a parallel kernel (e.g. from NCCL) is performing communications with a peer. Upon completion, the local kernel will implicitly flush its writes to satisfy any *synchronizes-with* relationships to downstream work. This may unnecessarily wait, fully or partially, on slower nvlink or PCIe writes from the communication kernel.

6.2.7.2 Isolating Traffic with Domains

Beginning with Hopper architecture GPUs and CUDA 12.0, the memory synchronization domains feature provides a way to alleviate such interference. In exchange for explicit assistance from code, the GPU can reduce the net cast by a fence operation. Each kernel launch is given a domain ID. Writes and fences are tagged with the ID, and a fence will only order writes matching the fence's domain. In the concurrent compute vs communication example, the communication kernels can be placed in a different domain.

When using domains, code must abide by the rule that **ordering or synchronization between distinct domains on the same GPU requires system-scope fencing**. Within a domain, device-scope fencing remains sufficient. This is necessary for cumulativity as one kernel's writes will not be encompassed by a fence issued from a kernel in another domain. In essence, cumulativity is satisfied by ensuring that cross-domain traffic is flushed to the system scope ahead of time.

Note that this modifies the definition of `thread_scope_device`. However, because kernels will default to domain 0 as described below, backward compatibility is maintained.

6.2.7.3 Using Domains in CUDA

Domains are accessible via the new launch attributes `cudaLaunchAttributeMemSyncDomain` and `cudaLaunchAttributeMemSyncDomainMap`. The former selects between logical domains `cudaLaunchMemSyncDomainDefault` and `cudaLaunchMemSyncDomainRemote`, and the latter provides a mapping from logical to physical domains. The remote domain is intended for kernels performing remote memory access in order to isolate their memory traffic from local kernels. Note, however, the selection of a particular domain does not affect what memory access a kernel may legally perform.

The domain count can be queried via device attribute `cudaDevAttrMemSyncDomainCount`. Hopper has 4 domains. To facilitate portable code, domains functionality can be used on all devices and CUDA will report a count of 1 prior to Hopper.

Having logical domains eases application composition. An individual kernel launch at a low level in the stack, such as from NCCL, can select a semantic logical domain without concern for the surrounding application architecture. Higher levels can steer logical domains using the mapping. The default value for the logical domain if it is not set is the default domain, and the default mapping is to map the

default domain to 0 and the remote domain to 1 (on GPUs with more than 1 domain). Specific libraries may tag launches with the remote domain in CUDA 12.0 and later; for example, NCCL 2.16 will do so. Together, this provides a beneficial use pattern for common applications out of the box, with no code changes needed in other components, frameworks, or at application level. An alternative use pattern, for example in an application using nvshmem or with no clear separation of kernel types, could be to partition parallel streams. Stream A may map both logical domains to physical domain 0, stream B to 1, and so on.

```
// Example of launching a kernel with the remote logical domain
cudaLaunchAttribute domainAttr;
domainAttr.id = cudaLaunchAttrMemSyncDomain;
domainAttr.val = cudaLaunchMemSyncDomainRemote;
cudaLaunchConfig_t config;
// Fill out other config fields
config.attrs = &domainAttr;
config.numAttrs = 1;
cudaLaunchKernelEx(&config, myKernel, kernelArg1, kernelArg2...);
```

```
// Example of setting a mapping for a stream
// (This mapping is the default for streams starting on Hopper if not
// explicitly set, and provided for illustration)
cudaLaunchAttributeValue mapAttr;
mapAttr.memSyncDomainMap.default_ = 0;
mapAttr.memSyncDomainMap.remote = 1;
cudaStreamSetAttribute(stream, cudaLaunchAttributeMemSyncDomainMap, &mapAttr);
```

```
// Example of mapping different streams to different physical domains, ignoring
// logical domain settings
cudaLaunchAttributeValue mapAttr;
mapAttr.memSyncDomainMap.default_ = 0;
mapAttr.memSyncDomainMap.remote = 0;
cudaStreamSetAttribute(streamA, cudaLaunchAttributeMemSyncDomainMap, &mapAttr);
mapAttr.memSyncDomainMap.default_ = 1;
mapAttr.memSyncDomainMap.remote = 1;
cudaStreamSetAttribute(streamB, cudaLaunchAttributeMemSyncDomainMap, &mapAttr);
```

As with other launch attributes, these are exposed uniformly on CUDA streams, individual launches using `cudaLaunchKernelEx`, and kernel nodes in CUDA graphs. A typical use would set the mapping at stream level and the logical domain at launch level (or bracketing a section of stream use) as described above.

Both attributes are copied to graph nodes during stream capture. Graphs take both attributes from the node itself, essentially an indirect way of specifying a physical domain. Domain-related attributes set on the stream a graph is launched into are not used in execution of the graph.

6.2.8. Asynchronous Concurrent Execution

CUDA exposes the following operations as independent tasks that can operate concurrently with one another:

- Computation on the host;
- Computation on the device;
- Memory transfers from the host to the device;

- Memory transfers from the device to the host;
- Memory transfers within the memory of a given device;
- Memory transfers among devices.

The level of concurrency achieved between these operations will depend on the feature set and compute capability of the device as described below.

6.2.8.1 Concurrent Execution between Host and Device

Concurrent host execution is facilitated through asynchronous library functions that return control to the host thread before the device completes the requested task. Using asynchronous calls, many device operations can be queued up together to be executed by the CUDA driver when appropriate device resources are available. This relieves the host thread of much of the responsibility to manage the device, leaving it free for other tasks. The following device operations are asynchronous with respect to the host:

- Kernel launches;
- Memory copies within a single device's memory;
- Memory copies from host to device of a memory block of 64 KB or less;
- Memory copies performed by functions that are suffixed with `Async`;
- Memory set function calls.

Programmers can globally disable asynchronicity of kernel launches for all CUDA applications running on a system by setting the `CUDA_LAUNCH_BLOCKING` environment variable to 1. This feature is provided for debugging purposes only and should not be used as a way to make production software run reliably.

Kernel launches are synchronous if hardware counters are collected via a profiler (Nsight Compute) unless concurrent kernel profiling is enabled. `Async` memory copies might also be synchronous if they involve host memory that is not page-locked.

6.2.8.2 Concurrent Kernel Execution

Some devices of compute capability 2.x and higher can execute multiple kernels concurrently. Applications may query this capability by checking the `concurrentKernels` device property (see *Device Enumeration*), which is equal to 1 for devices that support it.

The maximum number of kernel launches that a device can execute concurrently depends on its compute capability and is listed in Table 27.

A kernel from one CUDA context cannot execute concurrently with a kernel from another CUDA context. The GPU may time slice to provide forward progress to each context. If a user wants to run kernels from multiple process simultaneously on the SM, one must enable MPS.

Kernels that use many textures or a large amount of local memory are less likely to execute concurrently with other kernels.

6.2.8.3 Overlap of Data Transfer and Kernel Execution

Some devices can perform an asynchronous memory copy to or from the GPU concurrently with kernel execution. Applications may query this capability by checking the `asyncEngineCount` device property (see *Device Enumeration*), which is greater than zero for devices that support it. If host memory is involved in the copy, it must be page-locked.

It is also possible to perform an intra-device copy simultaneously with kernel execution (on devices that support the `concurrentKernels` device property) and/or with copies to or from the device (for devices that support the `asyncEngineCount` property). Intra-device copies are initiated using the standard memory copy functions with destination and source addresses residing on the same device.

6.2.8.4 Concurrent Data Transfers

Some devices of compute capability 2.x and higher can overlap copies to and from the device. Applications may query this capability by checking the `asyncEngineCount` device property (see *Device Enumeration*), which is equal to 2 for devices that support it. In order to be overlapped, any host memory involved in the transfers must be page-locked.

6.2.8.5 Streams

Applications manage the concurrent operations described above through *streams*. A stream is a sequence of commands (possibly issued by different host threads) that execute in order. Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently; this behavior is not guaranteed and should therefore not be relied upon for correctness (for example, inter-kernel communication is undefined). The commands issued on a stream may execute when all the dependencies of the command are met. The dependencies could be previously launched commands on same stream or dependencies from other streams. The successful completion of synchronize call guarantees that all the commands launched are completed.

6.2.8.5.1 Creation and Destruction of Streams

A stream is defined by creating a stream object and specifying it as the stream parameter to a sequence of kernel launches and host <-> device memory copies. The following code sample creates two streams and allocates an array `hostPtr` of `float` in page-locked memory.

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

Each of these streams is defined by the following code sample as a sequence of one memory copy from host to device, one kernel launch, and one memory copy from device to host:

```
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel <<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
```

(continues on next page)

(continued from previous page)

```
                    size, cudaMemcpyDeviceToHost, stream[i]);
}
```

Each stream copies its portion of input array `hostPtr` to array `inputDevPtr` in device memory, processes `inputDevPtr` on the device by calling `MyKernel()`, and copies the result `outputDevPtr` back to the same portion of `hostPtr`. *Overlapping Behavior* describes how the streams overlap in this example depending on the capability of the device. Note that `hostPtr` must point to page-locked host memory for any overlap to occur.

Streams are released by calling `cudaStreamDestroy()`.

```
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

In case the device is still doing work in the stream when `cudaStreamDestroy()` is called, the function will return immediately and the resources associated with the stream will be released automatically once the device has completed all work in the stream.

6.2.8.5.2 Default Stream

Kernel launches and host <-> device memory copies that do not specify any stream parameter, or equivalently that set the stream parameter to zero, are issued to the default stream. They are therefore executed in order.

For code that is compiled using the `--default-stream per-thread` compilation flag (or that defines the `CUDA_API_PER_THREAD_DEFAULT_STREAM` macro before including CUDA headers (`cuda.h` and `cuda_runtime.h`)), the default stream is a regular stream and each host thread has its own default stream.

Note: `#define CUDA_API_PER_THREAD_DEFAULT_STREAM 1` cannot be used to enable this behavior when the code is compiled by `nvcc` as `nvcc` implicitly includes `cuda_runtime.h` at the top of the translation unit. In this case the `--default-stream per-thread` compilation flag needs to be used or the `CUDA_API_PER_THREAD_DEFAULT_STREAM` macro needs to be defined with the `-DCUDA_API_PER_THREAD_DEFAULT_STREAM=1` compiler flag.

For code that is compiled using the `--default-stream legacy` compilation flag, the default stream is a special stream called the *NULL stream* and each device has a single NULL stream used for all host threads. The NULL stream is special as it causes implicit synchronization as described in *Implicit Synchronization*.

For code that is compiled without specifying a `--default-stream` compilation flag, `--default-stream legacy` is assumed as the default.

6.2.8.5.3 Explicit Synchronization

There are various ways to explicitly synchronize streams with each other.

`cudaDeviceSynchronize()` waits until all preceding commands in all streams of all host threads have completed.

`cudaStreamSynchronize()`takes a stream as a parameter and waits until all preceding commands in the given stream have completed. It can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device.

`cudaStreamWaitEvent()`takes a stream and an event as parameters (see *Events* for a description of events)and makes all the commands added to the given stream after the call to `cudaStreamWaitEvent()`delay their execution until the given event has completed.

`cudaStreamQuery()`provides applications with a way to know if all preceding commands in a stream have completed.

6.2.8.5.4 Implicit Synchronization

Two operations from different streams cannot run concurrently if any CUDA operation on the NULL stream is submitted in-between them, unless the streams are non-blocking streams (created with the `cudaStreamNonBlocking` flag).

Applications should follow these guidelines to improve their potential for concurrent kernel execution:

- All independent operations should be issued before dependent operations,
- Synchronization of any kind should be delayed as long as possible.

6.2.8.5.5 Overlapping Behavior

The amount of execution overlap between two streams depends on the order in which the commands are issued to each stream and whether or not the device supports overlap of data transfer and kernel execution (see *Overlap of Data Transfer and Kernel Execution*), concurrent kernel execution (see *Concurrent Kernel Execution*), and/or concurrent data transfers (see *Concurrent Data Transfers*).

For example, on devices that do not support concurrent data transfers, the two streams of the code sample of *Creation and Destruction of Streams* do not overlap at all because the memory copy from host to device is issued to stream[1] after the memory copy from device to host is issued to stream[0], so it can only start once the memory copy from device to host issued to stream[0] has completed. If the code is rewritten the following way (and assuming the device supports overlap of data transfer and kernel execution)

```
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    MyKernel<<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
```

then the memory copy from host to device issued to stream[1] overlaps with the kernel launch issued to stream[0].

On devices that do support concurrent data transfers, the two streams of the code sample of *Creation and Destruction of Streams* do overlap: The memory copy from host to device issued to stream[1] overlaps with the memory copy from device to host issued to stream[0] and even with the kernel launch issued to stream[0] (assuming the device supports overlap of data transfer and kernel execution).

6.2.8.5.6 Host Functions (Callbacks)

The runtime provides a way to insert a CPU function call at any point into a stream via `cudaLaunchHostFunc()`. The provided function is executed on the host once all commands issued to the stream before the callback have completed.

The following code sample adds the host function `MyCallback` to each of two streams after issuing a host-to-device memory copy, a kernel launch and a device-to-host memory copy into each stream. The function will begin execution on the host after each of the device-to-host memory copies completes.

```
void CUDART_CB MyCallback(void *data){
    printf("Inside callback %d\n", (size_t)data);
}
...
for (size_t i = 0; i < 2; ++i) {
    cudaMemcpyAsync(devPtrIn[i], hostPtr[i], size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel<<<100, 512, 0, stream[i]>>>(devPtrOut[i], devPtrIn[i], size);
    cudaMemcpyAsync(hostPtr[i], devPtrOut[i], size, cudaMemcpyDeviceToHost,
↪stream[i]);
    cudaLaunchHostFunc(stream[i], MyCallback, (void*)i);
}
```

The commands that are issued in a stream after a host function do not start executing before the function has completed.

A host function enqueued into a stream must not make CUDA API calls (directly or indirectly), as it might end up waiting on itself if it makes such a call leading to a deadlock.

6.2.8.5.7 Stream Priorities

The relative priorities of streams can be specified at creation using `cudaStreamCreateWithPriority()`. The range of allowable priorities, ordered as [greatest priority, least priority] can be obtained using the `cudaDeviceGetStreamPriorityRange()` function. At runtime, the GPU scheduler utilizes stream priorities to determine task execution order, but these priorities serve as hints rather than guarantees. When selecting work to launch, pending tasks in higher-priority streams take precedence over those in lower-priority streams. Higher-priority tasks do not preempt already running lower-priority tasks. The GPU does not reassess work queues during task execution, and increasing a stream's priority will not interrupt ongoing work. Stream priorities influence task execution without enforcing strict ordering, so users can leverage stream priorities to influence task execution without relying on strict ordering guarantees.

The following code sample obtains the allowable range of priorities for the current device, and creates streams with the highest and lowest available priorities.

```
// get the range of stream priorities for this device
int leastPriority, greatestPriority;
cudaDeviceGetStreamPriorityRange(&leastPriority, &greatestPriority);
// create streams with highest and lowest available priorities
cudaStream_t st_high, st_low;
```

(continues on next page)

(continued from previous page)

```
cudaStreamCreateWithPriority(&st_high, cudaStreamNonBlocking, greatestPriority));
cudaStreamCreateWithPriority(&st_low, cudaStreamNonBlocking, leastPriority);
```

6.2.8.6 Programmatic Dependent Launch and Synchronization

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

The *Programmatic Dependent Launch* mechanism allows for a dependent *secondary* kernel to launch before the *primary* kernel it depends on in the same CUDA stream has finished executing. Available starting with devices of compute capability 9.0, this technique can provide performance benefits when the *secondary* kernel can complete significant work that does not depend on the results of the *primary* kernel.

6.2.8.6.1 Background

A CUDA application utilizes the GPU by launching and executing multiple kernels on it. A typical GPU activity timeline is shown in Figure 10.



Figure 10: GPU activity timeline

Here, `secondary_kernel` is launched after `primary_kernel` finishes its execution. Serialized execution is usually necessary because `secondary_kernel` depends on result data produced by `primary_kernel`. If `secondary_kernel` has no dependency on `primary_kernel`, both of them can be launched concurrently by using *Streams*. Even if `secondary_kernel` is dependent on `primary_kernel`, there is some potential for concurrent execution. For example, almost all the kernels have some sort of *preamble* section during which tasks such as zeroing buffers or loading constant values are performed.

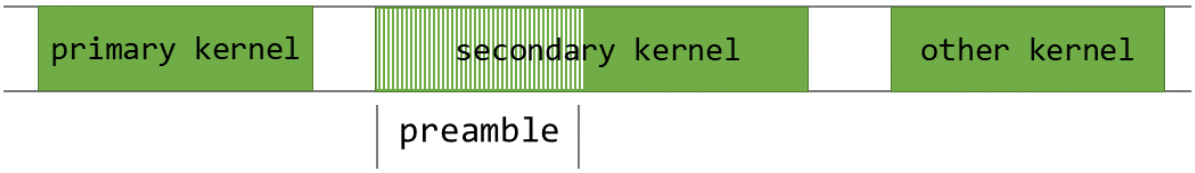


Figure 11: Preamble section of `secondary_kernel`

Figure 11 demonstrates the portion of `secondary_kernel` that could be executed concurrently without impacting the application. Note that concurrent launch also allows us to hide the launch latency of `secondary_kernel` behind the execution of `primary_kernel`.

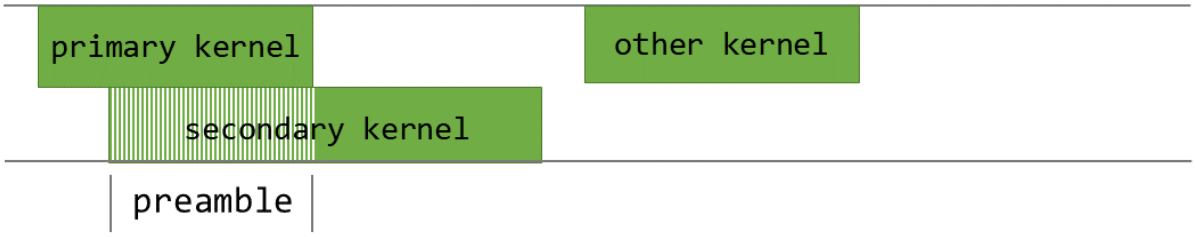


Figure 12: Concurrent execution of `primary_kernel` and `secondary_kernel`

The concurrent launch and execution of `secondary_kernel` shown in Figure 12 is achievable using *Programmatic Dependent Launch*.

Programmatic Dependent Launch introduces changes to the CUDA kernel launch APIs as explained in following section. These APIs require at least compute capability 9.0 to provide overlapping execution.

6.2.8.6.2 API Description

In Programmatic Dependent Launch, a primary and a secondary kernel are launched in the same CUDA stream. The primary kernel should execute `cudaTriggerProgrammaticLaunchCompletion` with all thread blocks when it's ready for the secondary kernel to launch. The secondary kernel must be launched using the extensible launch API as shown.

```
__global__ void primary_kernel() {
   // Initial work that should finish before starting secondary kernel

   // Trigger the secondary kernel
   cudaTriggerProgrammaticLaunchCompletion();

   // Work that can coincide with the secondary kernel
}

__global__ void secondary_kernel()
{
   // Independent work

   // Will block until all primary kernels the secondary kernel is dependent on have
↪completed and flushed results to global memory
   cudaGridDependencySynchronize();

   // Dependent work
}

cudaLaunchAttribute attribute[1];
attribute[0].id = cudaLaunchAttributeProgrammaticStreamSerialization;
attribute[0].val.programmaticStreamSerializationAllowed = 1;
configSecondary.attrs = attribute;
```

(continues on next page)

(continued from previous page)

```
configSecondary.numAttrs = 1;

primary_kernel<<<grid_dim, block_dim, 0, stream>>>();
cudaLaunchKernelEx(&configSecondary, secondary_kernel);
```

When the secondary kernel is launched using the `cudaLaunchAttributeProgrammaticStreamSerialization` attribute, the CUDA driver is safe to launch the secondary kernel early and not wait on the completion and memory flush of the primary before launching the secondary.

The CUDA driver can launch the secondary kernel when all primary thread blocks have launched and executed `cudaTriggerProgrammaticLaunchCompletion`. If the primary kernel doesn't execute the trigger, it implicitly occurs after all thread blocks in the primary kernel exit.

In either case, the secondary thread blocks might launch before data written by the primary kernel is visible. As such, when the secondary kernel is configured with *Programmatic Dependent Launch*, it must always use `cudaGridDependencySynchronize` or other means to verify that the result data from the primary is available.

Please note that these methods provide the opportunity for the primary and secondary kernels to execute concurrently, however this behavior is opportunistic and not guaranteed to lead to concurrent kernel execution. Reliance on concurrent execution in this manner is unsafe and can lead to deadlock.

6.2.8.6.3 Use in CUDA Graphs

Programmatic Dependent Launch can be used in *CUDA Graphs* via *stream capture* or directly via *edge data*. To program this feature in a CUDA Graph with edge data, use a `cudaGraphDependencyType` value of `cudaGraphDependencyTypeProgrammatic` on an edge connecting two kernel nodes. This edge type makes the upstream kernel visible to a `cudaGridDependencySynchronize()` in the downstream kernel. This type must be used with an outgoing port of either `cudaGraphKernelNodePortLaunchCompletion` or `cudaGraphKernelNodePortProgrammatic`.

The resulting graph equivalents for stream capture are as follows:

Stream code (abbreviated)	Resulting graph edge

`cudaLaunchAttribute attribute;`
`attribute.id = cudaLaunchAttributeProgrammaticStreamSeri`
`attribute.val.programmaticStreamSerializationAllowed = 1;`
	`cudaGraphEdgeData edgeData;`
	`edgeData.type = cudaGraphDependencyTypeProgrammatic;`
	`edgeData.from_port = cudaGraphKernelNodePortProgrammatic;`

`cudaLaunchAttribute attribute;`
`attribute.id = cudaLaunchAttributeProgrammaticEvent;`
`attribute.val.programmaticEvent.triggerAtBlockStart = 0;`
	`cudaGraphEdgeData edgeData;`
	`edgeData.type = cudaGraphDependencyTypeProgrammatic;`
	`edgeData.from_port = cudaGraphKernelNodePortProgrammatic;`

`cudaLaunchAttribute attribute;`
`attribute.id = cudaLaunchAttributeProgrammaticEvent;`
`attribute.val.programmaticEvent.triggerAtBlockStart = 1;`
	`cudaGraphEdgeData edgeData;`
	`edgeData.type = cudaGraphDependencyTypeProgrammatic;`
	`edgeData.from_port = cudaGraphKernelNodePortLaunchCompletion;`

6.2.8.7 CUDA Graphs

CUDA Graphs present a new model for work submission in CUDA. A graph is a series of operations, such as kernel launches, connected by dependencies, which is defined separately from its execution. This allows a graph to be defined once and then launched repeatedly. Separating out the definition of a graph from its execution enables a number of optimizations: first, CPU launch costs are reduced compared to streams, because much of the setup is done in advance; second, presenting the whole workflow to CUDA enables optimizations which might not be possible with the piecewise work submission mechanism of streams.

To see the optimizations possible with graphs, consider what happens in a stream: when you place a kernel into a stream, the host driver performs a sequence of operations in preparation for the execution of the kernel on the GPU. These operations, necessary for setting up and launching the kernel, are an overhead cost which must be paid for each kernel that is issued. For a GPU kernel with a short execution time, this overhead cost can be a significant fraction of the overall end-to-end execution time.

Work submission using graphs is separated into three distinct stages: definition, instantiation, and execution.

- During the definition phase, a program creates a description of the operations in the graph along with the dependencies between them.
- Instantiation takes a snapshot of the graph template, validates it, and performs much of the setup and initialization of work with the aim of minimizing what needs to be done at launch. The resulting instance is known as an *executable graph.*

- An executable graph may be launched into a stream, similar to any other CUDA work. It may be launched any number of times without repeating the instantiation.

6.2.8.7.1 Graph Structure

An operation forms a node in a graph. The dependencies between the operations are the edges. These dependencies constrain the execution sequence of the operations.

An operation may be scheduled at any time once the nodes on which it depends are complete. Scheduling is left up to the CUDA system.

6.2.8.7.1.1 Node Types

A graph node can be one of:

- kernel
- CPU function call
- memory copy
- memset
- empty node
- waiting on an *event*
- recording an *event*
- signalling an *external semaphore*
- waiting on an *external semaphore*
- *conditional node*
- *Graph Memory Nodes*
- child graph: To execute a separate nested graph, as shown in the following figure.

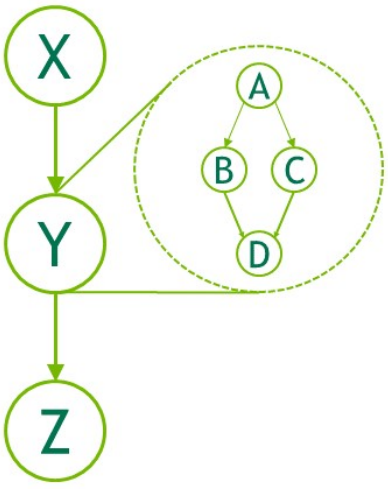


Figure 13: Child Graph Example

6.2.8.7.1.2 Edge Data

CUDA 12.3 introduced edge data on CUDA Graphs. Edge data modifies a dependency specified by an edge and consists of three parts: an outgoing port, an incoming port, and a type. An outgoing port specifies when an associated edge is triggered. An incoming port specifies what portion of a node is dependent on an associated edge. A type modifies the relation between the endpoints.

Port values are specific to node type and direction, and edge types may be restricted to specific node types. In all cases, zero-initialized edge data represents default behavior. Outgoing port 0 waits on an entire task, incoming port 0 blocks an entire task, and edge type 0 is associated with a full dependency with memory synchronizing behavior.

Edge data is optionally specified in various graph APIs via a parallel array to the associated nodes. If it is omitted as an input parameter, zero-initialized data is used. If it is omitted as an output (query) parameter, the API accepts this if the edge data being ignored is all zero-initialized, and returns `cudaErrorLossyQuery` if the call would discard information.

Edge data is also available in some stream capture APIs: `cudaStreamBeginCaptureToGraph()`, `cudaStreamGetCaptureInfo()`, and `cudaStreamUpdateCaptureDependencies()`. In these cases, there is not yet a downstream node. The data is associated with a dangling edge (half edge) which will either be connected to a future captured node or discarded at termination of stream capture. Note that some edge types do not wait on full completion of the upstream node. These edges are ignored when considering if a stream capture has been fully rejoined to the origin stream, and cannot be discarded at the end of capture. See *Creating a Graph Using Stream Capture*.

Currently, no node types define additional incoming ports, and only kernel nodes define additional outgoing ports. There is one non-default dependency type, `cudaGraphDependencyTypeProgrammatic`, which enables *Programmatic Dependent Launch* between two kernel nodes.

6.2.8.7.2 Creating a Graph Using Graph APIs

Graphs can be created via two mechanisms: explicit API and stream capture. The following is an example of creating and executing the below graph.

```
// Create the graph - it starts out empty
cudaGraphCreate(&graph, 0);

// For the purpose of this example, we'll create
// the nodes separately from the dependencies to
// demonstrate that it can be done in two stages.
// Note that dependencies can also be specified
// at node creation.
cudaGraphAddKernelNode(&a, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&b, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&c, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&d, graph, NULL, 0, &nodeParams);

// Now set up dependencies on each node
cudaGraphAddDependencies(graph, &a, &b, NULL, 1);     // A->B
cudaGraphAddDependencies(graph, &a, &c, NULL, 1);     // A->C
cudaGraphAddDependencies(graph, &b, &d, NULL, 1);     // B->D
cudaGraphAddDependencies(graph, &c, &d, NULL, 1);     // C->D
```

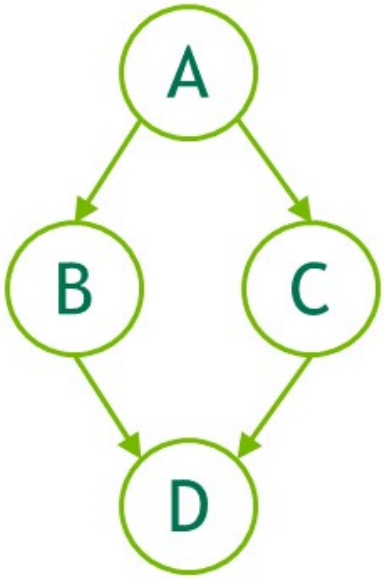


Figure 14: Creating a Graph Using Graph APIs Example

6.2.8.7.3 Creating a Graph Using Stream Capture

Stream capture provides a mechanism to create a graph from existing stream-based APIs. A section of code which launches work into streams, including existing code, can be bracketed with calls to `cudaStreamBeginCapture()` and `cudaStreamEndCapture()`. See below.

```
cudaGraph_t graph;

cudaStreamBeginCapture(stream);

kernel_A<<< ..., stream >>>(...);
kernel_B<<< ..., stream >>>(...);
libraryCall(stream);
kernel_C<<< ..., stream >>>(...);

cudaStreamEndCapture(stream, &graph);
```

A call to `cudaStreamBeginCapture()` places a stream in capture mode. When a stream is being captured, work launched into the stream is not enqueued for execution. It is instead appended to an internal graph that is progressively being built up. This graph is then returned by calling `cudaStreamEndCapture()`, which also ends capture mode for the stream. A graph which is actively being constructed by stream capture is referred to as a *capture graph.*

Stream capture can be used on any CUDA stream except `cudaStreamLegacy` (the "NULL stream"). Note that it *can* be used on `cudaStreamPerThread`. If a program is using the legacy stream, it may be possible to redefine stream 0 to be the per-thread stream with no functional change. See *Default Stream*.

Whether a stream is being captured can be queried with `cudaStreamIsCapturing()`.

Work can be captured to an existing graph using `cudaStreamBeginCaptureToGraph()`. Instead of capturing to an internal graph, work is captured to a graph provided by the user.

6.2.8.7.3.1 Cross-stream Dependencies and Events

Stream capture can handle cross-stream dependencies expressed with `cudaEventRecord()` and `cudaStreamWaitEvent()`, provided the event being waited upon was recorded into the same capture graph.

When an event is recorded in a stream that is in capture mode, it results in a *captured event.* A captured event represents a set of nodes in a capture graph.

When a captured event is waited on by a stream, it places the stream in capture mode if it is not already, and the next item in the stream will have additional dependencies on the nodes in the captured event. The two streams are then being captured to the same capture graph.

When cross-stream dependencies are present in stream capture, `cudaStreamEndCapture()` must still be called in the same stream where `cudaStreamBeginCapture()` was called; this is the *origin stream*. Any other streams which are being captured to the same capture graph, due to event-based dependencies, must also be joined back to the origin stream. This is illustrated below. All streams being captured to the same capture graph are taken out of capture mode upon `cudaStreamEndCapture()`. Failure to rejoin to the origin stream will result in failure of the overall capture operation.

```
// stream1 is the origin stream
cudaStreamBeginCapture(stream1);

kernel_A<<< ..., stream1 >>>(...);

// Fork into stream2
cudaEventRecord(event1, stream1);
cudaStreamWaitEvent(stream2, event1);

kernel_B<<< ..., stream1 >>>(...);
kernel_C<<< ..., stream2 >>>(...);

// Join stream2 back to origin stream (stream1)
cudaEventRecord(event2, stream2);
cudaStreamWaitEvent(stream1, event2);

kernel_D<<< ..., stream1 >>>(...);

// End capture in the origin stream
cudaStreamEndCapture(stream1, &graph);

// stream1 and stream2 no longer in capture mode
```

Graph returned by the above code is shown in Figure 14.

Note: When a stream is taken out of capture mode, the next non-captured item in the stream (if any) will still have a dependency on the most recent prior non-captured item, despite intermediate items having been removed.

6.2.8.7.3.2 Prohibited and Unhandled Operations

It is invalid to synchronize or query the execution status of a stream which is being captured or a captured event, because they do not represent items scheduled for execution. It is also invalid to query the execution status of or synchronize a broader handle which encompasses an active stream capture, such as a device or context handle when any associated stream is in capture mode.

When any stream in the same context is being captured, and it was not created with `cudaStreamNonBlocking`, any attempted use of the legacy stream is invalid. This is because the legacy stream handle at all times encompasses these other streams; enqueueing to the legacy stream would create a dependency on the streams being captured, and querying it or synchronizing it would query or synchronize the streams being captured.

It is therefore also invalid to call synchronous APIs in this case. Synchronous APIs, such as `cudaMemcpy()`, enqueue work to the legacy stream and synchronize it before returning.

Note: As a general rule, when a dependency relation would connect something that is captured with something that was not captured and instead enqueued for execution, CUDA prefers to return an error rather than ignore the dependency. An exception is made for placing a stream into or out of capture mode; this severs a dependency relation between items added to the stream immediately before and after the mode transition.

It is invalid to merge two separate capture graphs by waiting on a captured event from a stream which is being captured and is associated with a different capture graph than the event. It is invalid to wait on a non-captured event from a stream which is being captured without specifying the cudaEventWaitExternal flag.

A small number of APIs that enqueue asynchronous operations into streams are not currently supported in graphs and will return an error if called with a stream which is being captured, such as `cudaStreamAttachMemAsync()`.

6.2.8.7.3.3 Invalidation

When an invalid operation is attempted during stream capture, any associated capture graphs are *invalidated*. When a capture graph is invalidated, further use of any streams which are being captured or captured events associated with the graph is invalid and will return an error, until stream capture is ended with `cudaStreamEndCapture()`. This call will take the associated streams out of capture mode, but will also return an error value and a NULL graph.

6.2.8.7.4 CUDA User Objects

CUDA User Objects can be used to help manage the lifetime of resources used by asynchronous work in CUDA. In particular, this feature is useful for *CUDA Graphs* and *stream capture*.

Various resource management schemes are not compatible with CUDA graphs. Consider for example an event-based pool or a synchronous-create, asynchronous-destroy scheme.

```
// Library API with pool allocation
void libraryWork(cudaStream_t stream) {
    auto &resource = pool.claimTemporaryResource();
    resource.waitOnReadyEventInStream(stream);
    launchWork(stream, resource);
```

(continues on next page)

(continued from previous page)

```
    resource.recordReadyEvent(stream);
}
```

```
// Library API with asynchronous resource deletion
void libraryWork(cudaStream_t stream) {
    Resource *resource = new Resource(...);
    launchWork(stream, resource);
    cudaLaunchHostFunc(
        stream,
        [](void *resource) {
            delete static_cast<Resource *>(resource);
        },
        resource,
        0);
    // Error handling considerations not shown
}
```

These schemes are difficult with CUDA graphs because of the non-fixed pointer or handle for the resource which requires indirection or graph update, and the synchronous CPU code needed each time the work is submitted. They also do not work with stream capture if these considerations are hidden from the caller of the library, and because of use of disallowed APIs during capture. Various solutions exist such as exposing the resource to the caller. CUDA user objects present another approach.

A CUDA user object associates a user-specified destructor callback with an internal refcount, similar to C++ `shared_ptr`. References may be owned by user code on the CPU and by CUDA graphs. Note that for user-owned references, unlike C++ smart pointers, there is no object representing the reference; users must track user-owned references manually. A typical use case would be to immediately move the sole user-owned reference to a CUDA graph after the user object is created.

When a reference is associated to a CUDA graph, CUDA will manage the graph operations automatically. A cloned `cudaGraph_t` retains a copy of every reference owned by the source `cudaGraph_t`, with the same multiplicity. An instantiated `cudaGraphExec_t` retains a copy of every reference in the source `cudaGraph_t`. When a `cudaGraphExec_t` is destroyed without being synchronized, the references are retained until the execution is completed.

Here is an example use.

```
cudaGraph_t graph;  // Preexisting graph

Object *object = new Object;  // C++ object with possibly nontrivial destructor
cudaUserObject_t cuObject;
cudaUserObjectCreate(
    &cuObject,
    object,  // Here we use a CUDA-provided template wrapper for this API,
             // which supplies a callback to delete the C++ object pointer
    1,  // Initial refcount
    cudaUserObjectNoDestructorSync  // Acknowledge that the callback cannot be
                                    // waited on via CUDA
);
cudaGraphRetainUserObject(
    graph,
    cuObject,
    1,  // Number of references
    cudaGraphUserObjectMove  // Transfer a reference owned by the caller (do
                             // not modify the total reference count)
);
```

(continues on next page)

(continued from previous page)

```
// No more references owned by this thread; no need to call release API
cudaGraphExec_t graphExec;
cudaGraphInstantiate(&graphExec, graph, nullptr, nullptr, 0);  // Will retain a
                                                               // new reference
cudaGraphDestroy(graph);  // graphExec still owns a reference
cudaGraphLaunch(graphExec, 0);  // Async launch has access to the user objects
cudaGraphExecDestroy(graphExec);  // Launch is not synchronized; the release
                                  // will be deferred if needed
cudaStreamSynchronize(0);  // After the launch is synchronized, the remaining
                           // reference is released and the destructor will
                           // execute. Note this happens asynchronously.
// If the destructor callback had signaled a synchronization object, it would
// be safe to wait on it at this point.
```

References owned by graphs in child graph nodes are associated to the child graphs, not the parents. If a child graph is updated or deleted, the references change accordingly. If an executable graph or child graph is updated with `cudaGraphExecUpdate` or `cudaGraphExecChildGraphNodeSetParams`, the references in the new source graph are cloned and replace the references in the target graph. In either case, if previous launches are not synchronized, any references which would be released are held until the launches have finished executing.

There is not currently a mechanism to wait on user object destructors via a CUDA API. Users may signal a synchronization object manually from the destructor code. In addition, it is not legal to call CUDA APIs from the destructor, similar to the restriction on `cudaLaunchHostFunc`. This is to avoid blocking a CUDA internal shared thread and preventing forward progress. It is legal to signal another thread to perform an API call, if the dependency is one way and the thread doing the call cannot block forward progress of CUDA work.

User objects are created with `cudaUserObjectCreate`, which is a good starting point to browse related APIs.

6.2.8.7.5 Updating Instantiated Graphs

Work submission using graphs is separated into three distinct stages: definition, instantiation, and execution. In situations where the workflow is not changing, the overhead of definition and instantiation can be amortized over many executions, and graphs provide a clear advantage over streams.

A graph is a snapshot of a workflow, including kernels, parameters, and dependencies, in order to replay it as rapidly and efficiently as possible. In situations where the workflow changes the graph becomes out of date and must be modified. Major changes to graph structure such as topology or types of nodes will require re-instantiation of the source graph because various topology-related optimization techniques must be re-applied.

The cost of repeated instantiation can reduce the overall performance benefit from graph execution, but it is common for only node parameters, such as kernel parameters and `cudaMemcpy` addresses, to change while graph topology remains the same. For this case, CUDA provides a lightweight mechanism known as “Graph Update,” which allows certain node parameters to be modified in-place without having to rebuild the entire graph. This is much more efficient than re-instantiation.

Updates will take effect the next time the graph is launched, so they will not impact previous graph launches, even if they are running at the time of the update. A graph may be updated and relaunched repeatedly, so multiple updates/launches can be queued on a stream.

CUDA provides two mechanisms for updating instantiated graph parameters, whole graph update and individual node update. Whole graph update allows the user to supply a topologically identical `cudaGraph_t` object whose nodes contain updated parameters. Individual node update allows the user

to explicitly update the parameters of individual nodes. Using an updated `cudaGraph_t` is more convenient when a large number of nodes are being updated, or when the graph topology is unknown to the caller (i.e., The graph resulted from stream capture of a library call). Using individual node update is preferred when the number of changes is small and the user has the handles to the nodes requiring updates. Individual node update skips the topology checks and comparisons for unchanged nodes, so it can be more efficient in many cases.

CUDA also provides a mechanism for enabling and disabling individual nodes without affecting their current parameters.

The following sections explain each approach in more detail.

6.2.8.7.5.1 Graph Update Limitations

Kernel nodes:

- The owning context of the function cannot change.
- A node whose function originally did not use CUDA dynamic parallelism cannot be updated to a function which uses CUDA dynamic parallelism.

`cudaMemset` and `cudaMemcpy` nodes:

- The CUDA device(s) to which the operand(s) was allocated/mapped cannot change.
- The source/destination memory must be allocated from the same context as the original source/destination memory.
- Only 1D `cudaMemset`/`cudaMemcpy` nodes can be changed.

Additional memcpy node restrictions:

- Changing either the source or destination memory type (i.e., `cudaPitchedPtr`, `cudaArray_t`, etc.), or the type of transfer (i.e., `cudaMemcpyKind`) is not supported.

External semaphore wait nodes and record nodes:

- Changing the number of semaphores is not supported.

Conditional nodes:

- The order of handle creation and assignment must match between the graphs.
- Changing node parameters is not supported (i.e. number of graphs in the conditional, node context, etc).
- Changing parameters of nodes within the conditional body graph is subject to the rules above.

Memory nodes:

- It is not possible to update a `cudaGraphExec_t` with a `cudaGraph_t` if the `cudaGraph_t` is currently instantiated as a different `cudaGraphExec_t`.

There are no restrictions on updates to host nodes, event record nodes, or event wait nodes.

6.2.8.7.5.2 Whole Graph Update

`cudaGraphExecUpdate()` allows an instantiated graph (the "original graph") to be updated with the parameters from a topologically identical graph (the "updating" graph). The topology of the updating graph must be identical to the original graph used to instantiate the `cudaGraphExec_t`. In addition, the order in which the dependencies are specified must match. Finally, CUDA needs to consistently order the sink nodes (nodes with no dependencies). CUDA relies on the order of specific api calls to achieve consistent sink node ordering.

More explicitly, following the following rules will cause `cudaGraphExecUpdate()` to pair the nodes in the original graph and the updating graph deterministically:

1. For any capturing stream, the API calls operating on that stream must be made in the same order, including event wait and other api calls not directly corresponding to node creation.
2. The API calls which directly manipulate a given graph node's incoming edges (including captured stream APIs, node add APIs, and edge addition / removal APIs) must be made in the same order. Moreover, when dependencies are specified in arrays to these APIs, the order in which the dependencies are specified inside those arrays must match.
3. Sink nodes must be consistently ordered. Sink nodes are nodes without dependent nodes / outgoing edges in the final graph at the time of the `cudaGraphExecUpdate()` invocation. The following operations affect sink node ordering (if present) and must (as a combined set) be made in the same order:
   - Node add APIs resulting in a sink node.
   - Edge removal resulting in a node becoming a sink node.
   - `cudaStreamUpdateCaptureDependencies()`, if it removes a sink node from a capturing stream's dependency set.
   - `cudaStreamEndCapture()`.

The following example shows how the API could be used to update an instantiated graph:

```
cudaGraphExec_t graphExec = NULL;

for (int i = 0; i < 10; i++) {
    cudaGraph_t graph;
    cudaGraphExecUpdateResult updateResult;
    cudaGraphNode_t errorNode;

    // In this example we use stream capture to create the graph.
    // You can also use the Graph API to produce a graph.
    cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);

    // Call a user-defined, stream based workload, for example
    do_cuda_work(stream);

    cudaStreamEndCapture(stream, &graph);

    // If we've already instantiated the graph, try to update it directly
    // and avoid the instantiation overhead
    if (graphExec != NULL) {
        // If the graph fails to update, errorNode will be set to the
        // node causing the failure and updateResult will be set to a
        // reason code.
        cudaGraphExecUpdate(graphExec, graph, &errorNode, &updateResult);
```

(continues on next page)

(continued from previous page)

```
    }

    // Instantiate during the first iteration or whenever the update
    // fails for any reason
    if (graphExec == NULL || updateResult != cudaGraphExecUpdateSuccess) {

        // If a previous update failed, destroy the cudaGraphExec_t
        // before re-instantiating it
        if (graphExec != NULL) {
            cudaGraphExecDestroy(graphExec);
        }
        // Instantiate graphExec from graph. The error node and
        // error message parameters are unused here.
        cudaGraphInstantiate(&graphExec, graph, NULL, NULL, 0);
    }

    cudaGraphDestroy(graph);
    cudaGraphLaunch(graphExec, stream);
    cudaStreamSynchronize(stream);
}
```

A typical workflow is to create the initial `cudaGraph_t` using either the stream capture or graph API. The `cudaGraph_t` is then instantiated and launched as normal. After the initial launch, a new `cudaGraph_t` is created using the same method as the initial graph and `cudaGraphExecUpdate()` is called. If the graph update is successful, indicated by the `updateResult` parameter in the above example, the updated `cudaGraphExec_t` is launched. If the update fails for any reason, the `cudaGraphExecDestroy()` and `cudaGraphInstantiate()` are called to destroy the original `cudaGraphExec_t` and instantiate a new one.

It is also possible to update the `cudaGraph_t` nodes directly (i.e., Using `cudaGraphKernelNodeSetParams()`) and subsequently update the `cudaGraphExec_t`, however it is more efficient to use the explicit node update APIs covered in the next section.

Conditional handle flags and default values are updated as part of the graph update.

Please see the Graph API for more information on usage and current limitations.

6.2.8.7.5.3 Individual Node Update

Instantiated graph node parameters can be updated directly. This eliminates the overhead of instantiation as well as the overhead of creating a new `cudaGraph_t`. If the number of nodes requiring update is small relative to the total number of nodes in the graph, it is better to update the nodes individually. The following methods are available for updating `cudaGraphExec_t` nodes:

- `cudaGraphExecKernelNodeSetParams()`
- `cudaGraphExecMemcpyNodeSetParams()`
- `cudaGraphExecMemsetNodeSetParams()`
- `cudaGraphExecHostNodeSetParams()`
- `cudaGraphExecChildGraphNodeSetParams()`
- `cudaGraphExecEventRecordNodeSetEvent()`
- `cudaGraphExecEventWaitNodeSetEvent()`
- `cudaGraphExecExternalSemaphoresSignalNodeSetParams()`

- `cudaGraphExecExternalSemaphoresWaitNodeSetParams()`

Please see the Graph API for more information on usage and current limitations.

6.2.8.7.5.4 Individual Node Enable

Kernel, memset and memcpy nodes in an instantiated graph can be enabled or disabled using the `cudaGraphNodeSetEnabled()` API. This allows the creation of a graph which contains a superset of the desired functionality which can be customized for each launch. The enable state of a node can be queried using the `cudaGraphNodeGetEnabled()` API.

A disabled node is functionally equivalent to empty node until it is reenabled. Node parameters are not affected by enabling/disabling a node. Enable state is unaffected by individual node update or whole graph update with `cudaGraphExecUpdate()`. Parameter updates while the node is disabled will take effect when the node is reenabled.

The following methods are available for enabling/disabling `cudaGraphExec_t` nodes, as well as querying their status:

- `cudaGraphNodeSetEnabled()`
- `cudaGraphNodeGetEnabled()`

Refer to the Graph API for more information on usage and current limitations.

6.2.8.7.6 Using Graph APIs

`cudaGraph_t` objects are not thread-safe. It is the responsibility of the user to ensure that multiple threads do not concurrently access the same `cudaGraph_t`.

A `cudaGraphExec_t` cannot run concurrently with itself. A launch of a `cudaGraphExec_t` will be ordered after previous launches of the same executable graph.

Graph execution is done in streams for ordering with other asynchronous work. However, the stream is for ordering only; it does not constrain the internal parallelism of the graph, nor does it affect where graph nodes execute.

See Graph API.

6.2.8.7.7 Device Graph Launch

There are many workflows which need to make data-dependent decisions during runtime and execute different operations depending on those decisions. Rather than offloading this decision-making process to the host, which may require a round-trip from the device, users may prefer to perform it on the device. To that end, CUDA provides a mechanism to launch graphs from the device.

Device graph launch provides a convenient way to perform dynamic control flow from the device, be it something as simple as a loop or as complex as a device-side work scheduler. This functionality is only available on systems which support *unified addressing*.

Graphs which can be launched from the device will henceforth be referred to as device graphs, and graphs which cannot be launched from the device will be referred to as host graphs.

Device graphs can be launched from both the host and device, whereas host graphs can only be launched from the host. Unlike host launches, launching a device graph from the device while a previous launch of the graph is running will result in an error, returning `cudaErrorInvalidValue`; there-

fore, a device graph cannot be launched twice from the device at the same time. Launching a device graph from the host and device simultaneously will result in undefined behavior.

6.2.8.7.7.1 Device Graph Creation

In order for a graph to be launched from the device, it must be instantiated explicitly for device launch. This is achieved by passing the `cudaGraphInstantiateFlagDeviceLaunch` flag to the `cudaGraphInstantiate()` call. As is the case for host graphs, device graph structure is fixed at time of instantiation and cannot be updated without re-instantiation, and instantiation can only be performed on the host. In order for a graph to be able to be instantiated for device launch, it must adhere to various requirements.

6.2.8.7.7.2 Device Graph Requirements

General requirements:

- The graph's nodes must all reside on a single device.
- The graph can only contain kernel nodes, memcpy nodes, memset nodes, and child graph nodes.

Kernel nodes:

- Use of CUDA Dynamic Parallelism by kernels in the graph is not permitted.
- Cooperative launches are permitted so long as MPS is not in use.

Memcpy nodes:

- Only copies involving device memory and/or pinned device-mapped host memory are permitted.
- Copies involving CUDA arrays are not permitted.
- Both operands must be accessible from the current device at time of instantiation. Note that the copy operation will be performed from the device on which the graph resides, even if it is targeting memory on another device.

6.2.8.7.7.3 Device Graph Upload

In order to launch a graph on the device, it must first be uploaded to the device to populate the necessary device resources. This can be achieved in one of two ways.

Firstly, the graph can be uploaded explicitly, either via `cudaGraphUpload()` or by requesting an upload as part of instantiation via `cudaGraphInstantiateWithParams()`.

Alternatively, the graph can first be launched from the host, which will perform this upload step implicitly as part of the launch.

Examples of all three methods can be seen below:

```
// Explicit upload after instantiation
cudaGraphInstantiate(&deviceGraphExec1, deviceGraph1,
↪cudaGraphInstantiateFlagDeviceLaunch);
cudaGraphUpload(deviceGraphExec1, stream);

// Explicit upload as part of instantiation
cudaGraphInstantiateParams instantiateParams = {0};
```

(continues on next page)

(continued from previous page)

```
instantiateParams.flags = cudaGraphInstantiateFlagDeviceLaunch |
↪cudaGraphInstantiateFlagUpload;
instantiateParams.uploadStream = stream;
cudaGraphInstantiateWithParams(&deviceGraphExec2, deviceGraph2, &instantiateParams);

// Implicit upload via host launch
cudaGraphInstantiate(&deviceGraphExec3, deviceGraph3,
↪cudaGraphInstantiateFlagDeviceLaunch);
cudaGraphLaunch(deviceGraphExec3, stream);
```

6.2.8.7.7.4 Device Graph Update

Device graphs can only be updated from the host, and must be re-uploaded to the device upon executable graph update in order for the changes to take effect. This can be achieved using the same methods outlined in the previous section. Unlike host graphs, launching a device graph from the device while an update is being applied will result in undefined behavior.

6.2.8.7.7.5 Device Launch

Device graphs can be launched from both the host and the device via `cudaGraphLaunch()`, which has the same signature on the device as on the host. Device graphs are launched via the same handle on the host and the device. Device graphs must be launched from another graph when launched from the device.

Device-side graph launch is per-thread and multiple launches may occur from different threads at the same time, so the user will need to select a single thread from which to launch a given graph.

6.2.8.7.7.6 Device Launch Modes

Unlike host launch, device graphs cannot be launched into regular CUDA streams, and can only be launched into distinct named streams, which each denote a specific launch mode:

Table 5: Device-only Graph Launch Streams

Stream	Launch Mode
`cudaStreamGraphFireAndForget`	Fire and forget launch
`cudaStreamGraphTailLaunch`	Tail launch
`cudaStreamGraphFireAndForgetAsSibling`	Sibling launch

6.2.8.7.7.7 Fire and Forget Launch

As the name suggests, a fire and forget launch is submitted to the GPU immediately, and it runs independently of the launching graph. In a fire-and-forget scenario, the launching graph is the parent, and the launched graph is the child.

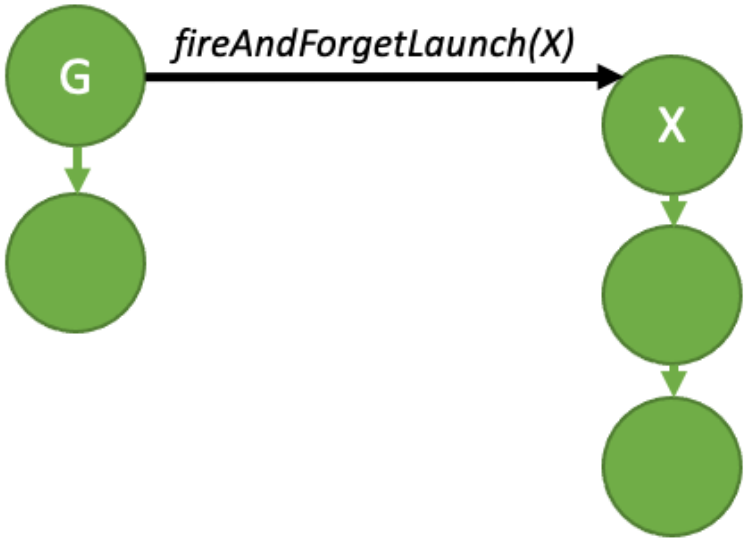


Figure 15: Fire and forget launch

The above diagram can be generated by the sample code below:

```
__global__ void launchFireAndForgetGraph(cudaGraphExec_t graph) {
    cudaGraphLaunch(graph, cudaStreamGraphFireAndForget);
}

void graphSetup() {
    cudaGraphExec_t gExec1, gExec2;
    cudaGraph_t g1, g2;

    // Create, instantiate, and upload the device graph.
    create_graph(&g2);
    cudaGraphInstantiate(&gExec2, g2, cudaGraphInstantiateFlagDeviceLaunch);
    cudaGraphUpload(gExec2, stream);

    // Create and instantiate the launching graph.
    cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
    launchFireAndForgetGraph<<<1, 1, 0, stream>>>(gExec2);
    cudaStreamEndCapture(stream, &g1);
    cudaGraphInstantiate(&gExec1, g1);

    // Launch the host graph, which will in turn launch the device graph.
    cudaGraphLaunch(gExec1, stream);
}
```

A graph can have up to 120 total fire-and-forget graphs during the course of its execution. This total resets between launches of the same parent graph.

6.2.8.7.7.8 Graph Execution Environments

In order to fully understand the device-side synchronization model, it is first necessary to understand the concept of an execution environment.

When a graph is launched from the device, it is launched into its own execution environment. The execution environment of a given graph encapsulates all work in the graph as well as all generated fire and forget work. The graph can be considered complete when it has completed execution and when all generated child work is complete.

The below diagram shows the environment encapsulation that would be generated by the fire-and-forget sample code in the previous section.

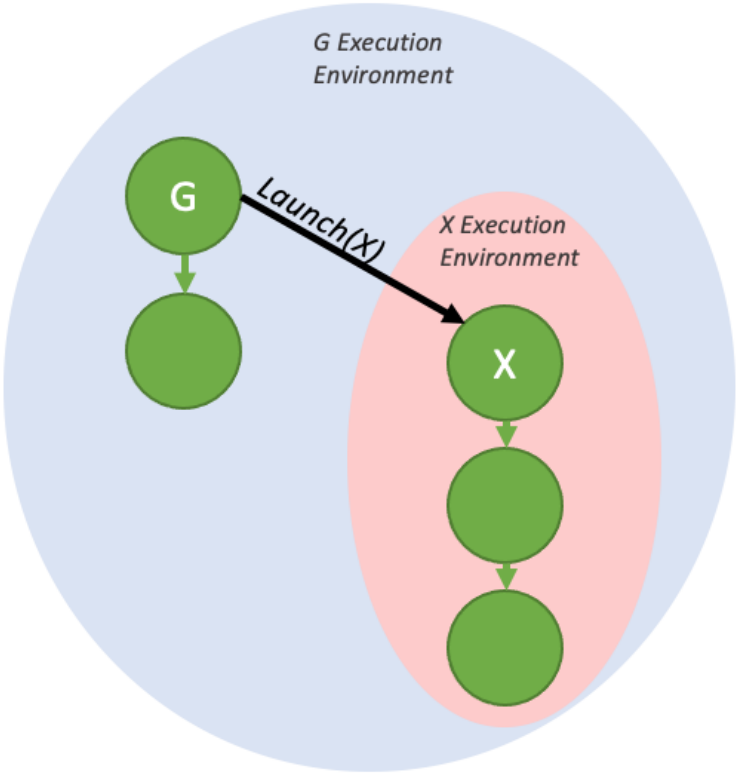


Figure 16: Fire and forget launch, with execution environments

These environments are also hierarchical, so a graph environment can include multiple levels of child-environments from fire and forget launches.

When a graph is launched from the host, there exists a stream environment that parents the execution environment of the launched graph. The stream environment encapsulates all work generated as part of the overall launch. The stream launch is complete (i.e. downstream dependent work may now run) when the overall stream environment is marked as complete.

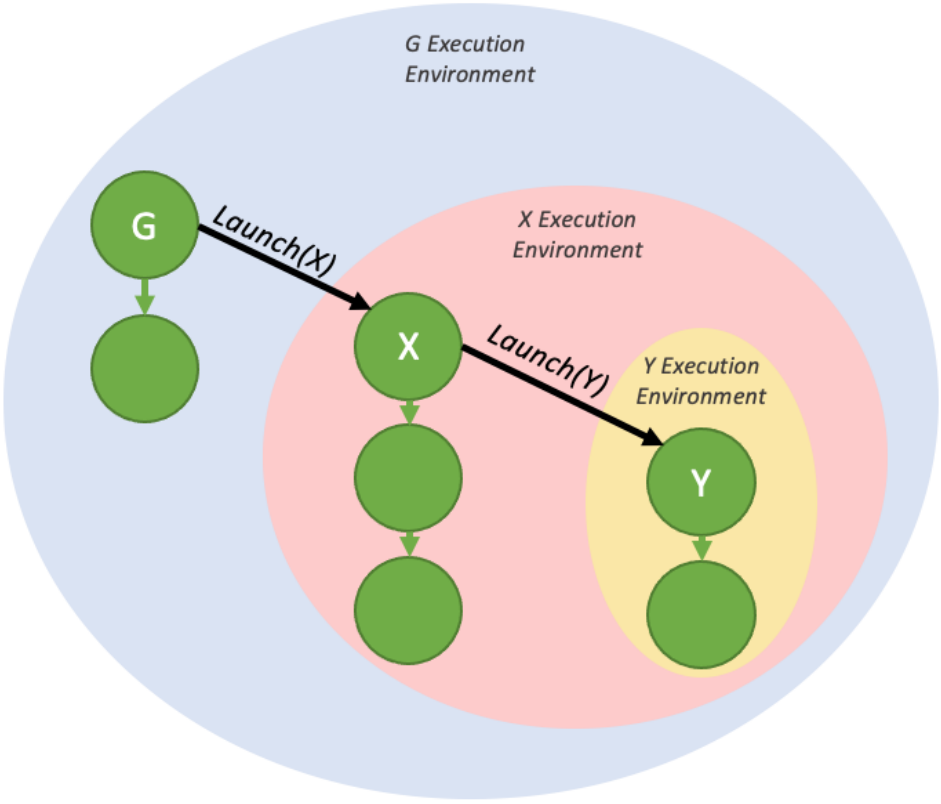


Figure 17: Nested fire and forget environments

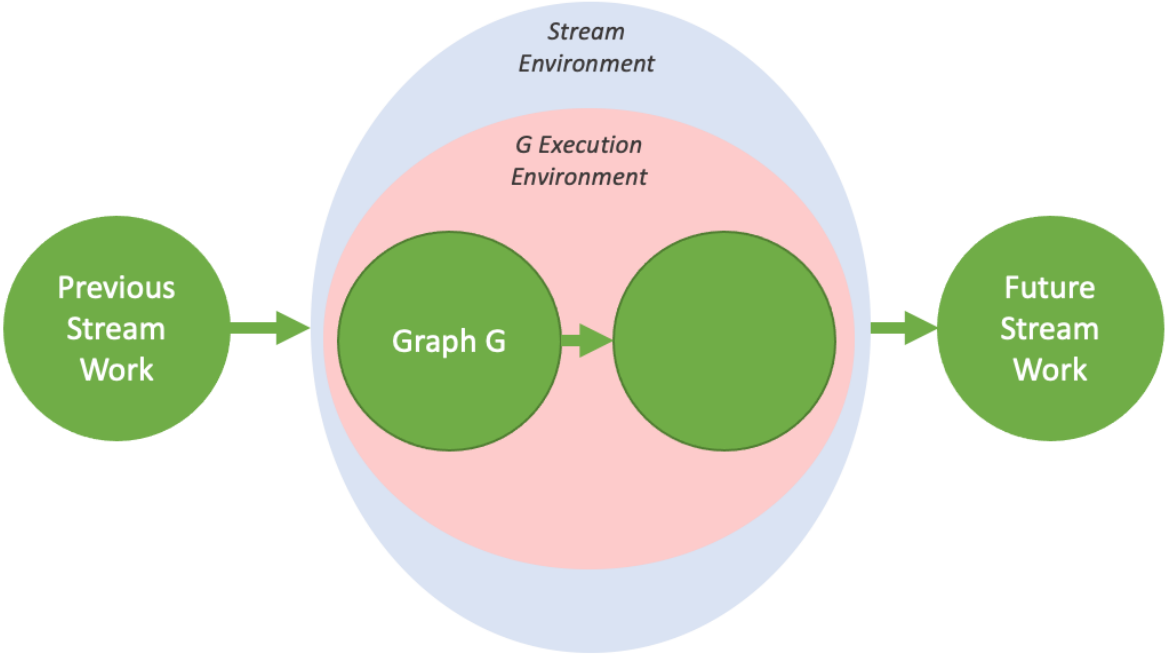


Figure 18: The stream environment, visualized

6.2.8.7.7.9 Tail Launch

Unlike on the host, it is not possible to synchronize with device graphs from the GPU via traditional methods such as `cudaDeviceSynchronize()` or `cudaStreamSynchronize()`. Rather, in order to enable serial work dependencies, a different launch mode - tail launch - is offered, to provide similar functionality.

A tail launch executes when a graph's environment is considered complete - ie, when the graph and all its children are complete. When a graph completes, the environment of the next graph in the tail launch list will replace the completed environment as a child of the parent environment. Like fire-and-forget launches, a graph can have multiple graphs enqueued for tail launch.

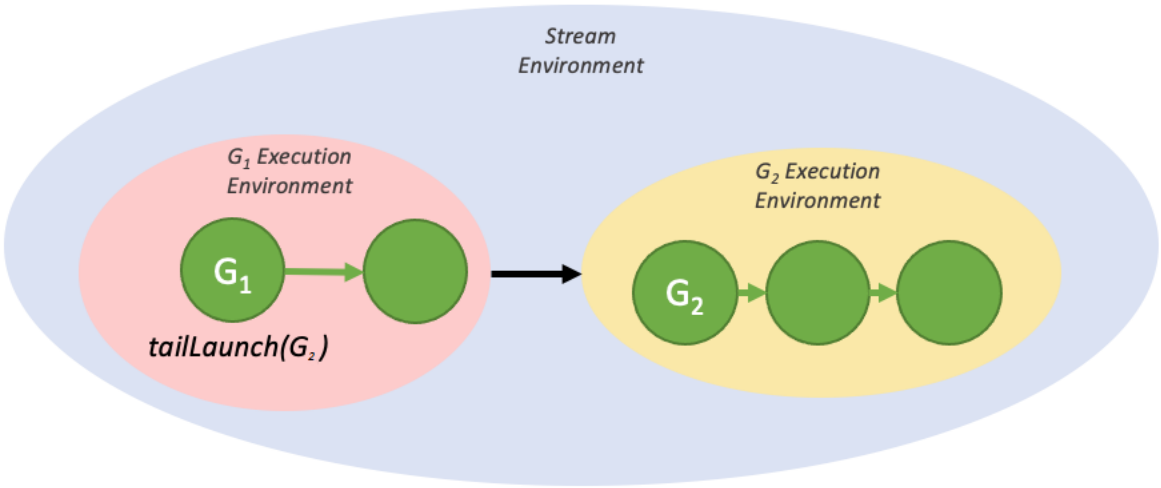


Figure 19: A simple tail launch

The above execution flow can be generated by the code below:

```
__global__ void launchTailGraph(cudaGraphExec_t graph) {
    cudaGraphLaunch(graph, cudaStreamGraphTailLaunch);
}

void graphSetup() {
    cudaGraphExec_t gExec1, gExec2;
    cudaGraph_t g1, g2;

    // Create, instantiate, and upload the device graph.
    create_graph(&g2);
    cudaGraphInstantiate(&gExec2, g2, cudaGraphInstantiateFlagDeviceLaunch);
    cudaGraphUpload(gExec2, stream);

    // Create and instantiate the launching graph.
    cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
    launchTailGraph<<<1, 1, 0, stream>>>(gExec2);
    cudaStreamEndCapture(stream, &g1);
    cudaGraphInstantiate(&gExec1, g1);

    // Launch the host graph, which will in turn launch the device graph.
    cudaGraphLaunch(gExec1, stream);
}
```

Tail launches enqueued by a given graph will execute one at a time, in order of when they were enqueued. So the first enqueued graph will run first, and then the second, and so on.

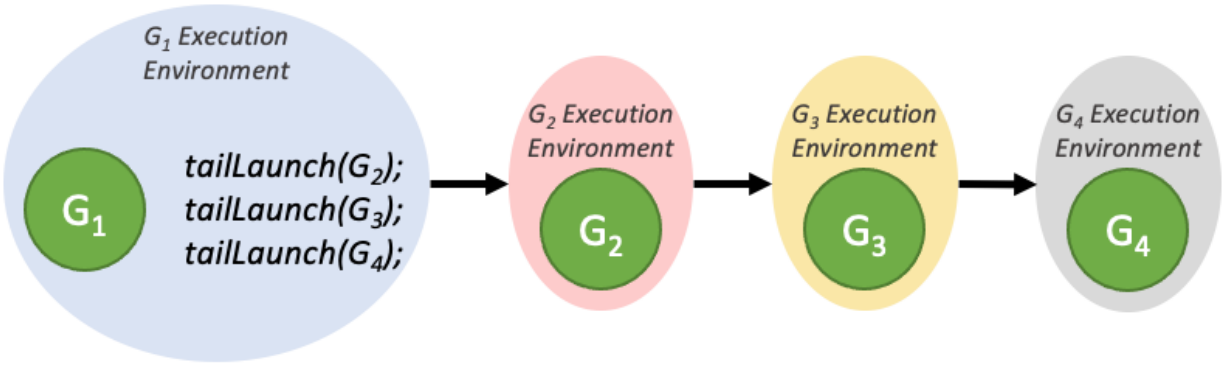


Figure 20: Tail launch ordering

Tail launches enqueued by a tail graph will execute before tail launches enqueued by previous graphs in the tail launch list. These new tail launches will execute in the order they are enqueued.

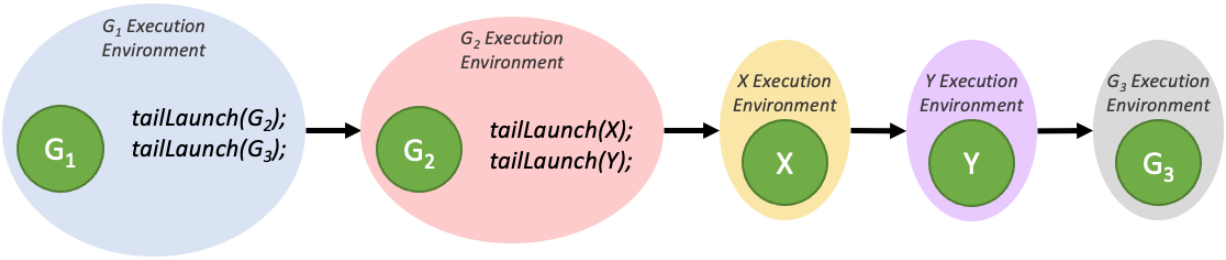


Figure 21: Tail launch ordering when enqueued from multiple graphs

A graph can have up to 255 pending tail launches.

6.2.8.7.7.10 Tail Self-launch

It is possible for a device graph to enqueue itself for a tail launch, although a given graph can only have one self-launch enqueued at a time. In order to query the currently running device graph so that it can be relaunched, a new device-side function is added:

```
cudaGraphExec_t cudaGetCurrentGraphExec();
```

This function returns the handle of the currently running graph if it is a device graph. If the currently executing kernel is not a node within a device graph, this function will return NULL.

Below is sample code showing usage of this function for a relaunch loop:

```
__device__ int relaunchCount = 0;

__global__ void relaunchSelf() {
    int relaunchMax = 100;

    if (threadIdx.x == 0) {
```

(continues on next page)

(continued from previous page)

```
        if (relaunchCount < relaunchMax) {
            cudaGraphLaunch(cudaGetCurrentGraphExec(), cudaStreamGraphTailLaunch);
        }

        relaunchCount++;
    }
}
```

6.2.8.7.7.11 Sibling Launch

Sibling launch is a variation of fire-and-forget launch in which the graph is launched not as a child of the launching graph's execution environment, but rather as a child of the launching graph's parent environment. Sibling launch is equivalent to a fire-and-forget launch from the launching graph's parent environment.

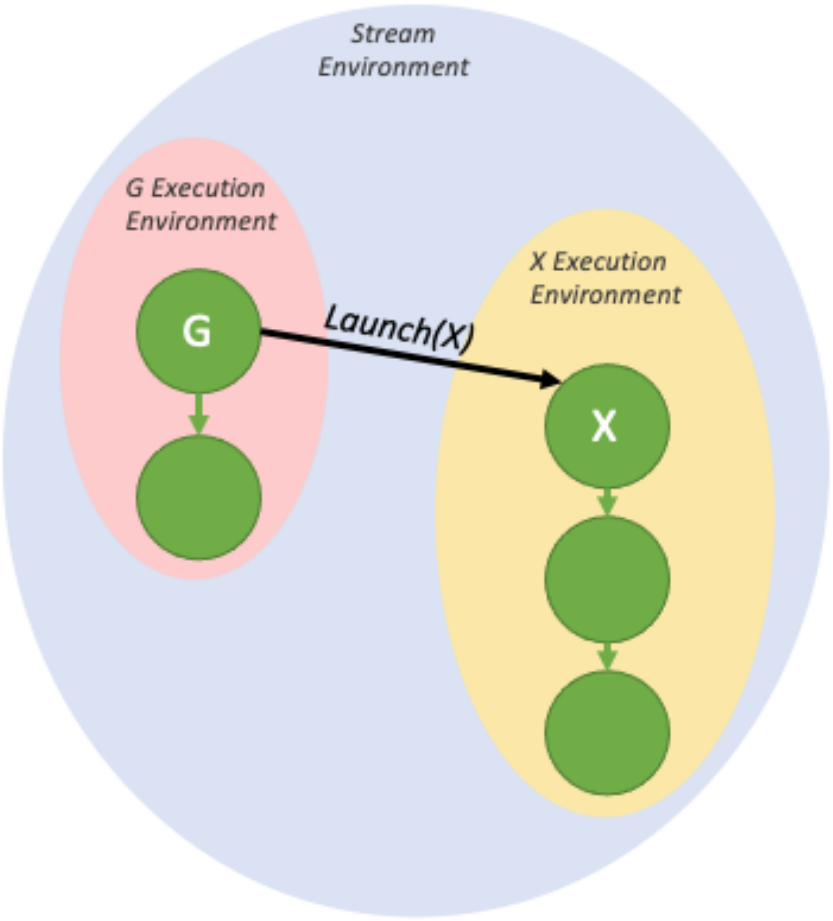


Figure 22: A simple sibling launch

The above diagram can be generated by the sample code below:

```
__global__ void launchSiblingGraph(cudaGraphExec_t graph) {
    cudaGraphLaunch(graph, cudaStreamGraphFireAndForgetAsSibling);
}

void graphSetup() {
    cudaGraphExec_t gExec1, gExec2;
    cudaGraph_t g1, g2;

    // Create, instantiate, and upload the device graph.
    create_graph(&g2);
    cudaGraphInstantiate(&gExec2, g2, cudaGraphInstantiateFlagDeviceLaunch);
    cudaGraphUpload(gExec2, stream);

    // Create and instantiate the launching graph.
    cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
    launchSiblingGraph<<<1, 1, 0, stream>>>(gExec2);
    cudaStreamEndCapture(stream, &g1);
    cudaGraphInstantiate(&gExec1, g1);

    // Launch the host graph, which will in turn launch the device graph.
    cudaGraphLaunch(gExec1, stream);
}
```

Since sibling launches are not launched into the launching graph's execution environment, they will not gate tail launches enqueued by the launching graph.

6.2.8.7.8 Conditional Graph Nodes

Conditional nodes allow conditional execution and looping of a graph contained within the conditional node. This allows dynamic and iterative workflows to be represented completely within a graph and frees up the host CPU to perform other work in parallel.

Evaluation of the condition value is performed on the device when the dependencies of the conditional node have been met. Conditional nodes can be one of the following types:

- Conditional *IF nodes* execute their body graph once if the condition value is non-zero when the node is executed. An optional second body graph can be provided and this will be executed once if the condition value is zero when the node is executed.
- Conditional *WHILE nodes* execute their body graph if the condition value is non-zero when the node is executed and will continue to execute their body graph until the condition value is zero.
- Conditional *SWITCH nodes* execute the nth body graph once if the condition value is equal to n. If the condition value does not correspond to a body graph, no body graph is launched.

A condition value is accessed by a *conditional handle* , which must be created before the node. The condition value can be set by device code using `cudaGraphSetConditional()`. A default value, applied on each graph launch, can also be specified when the handle is created.

When the conditional node is created, an empty graph is created and the handle is returned to the user so that the graph can be populated. This conditional body graph can be populated using either the *graph APIs* or *cudaStreamBeginCaptureToGraph()*.

Conditional nodes can be nested.

6.2.8.7.8.1 Conditional Handles

A condition value is represented by `cudaGraphConditionalHandle` and is created by `cudaGraphConditionalHandleCreate()`.

The handle must be associated with a single conditional node. Handles cannot be destroyed.

If `cudaGraphCondAssignDefault` is specified when the handle is created, the condition value will be initialized to the specified default at the beginning of each graph execution. If this flag is not provided, the condition value is undefined at the start of each graph execution and code should not assume that the condition value persists across executions.

The default value and flags associated with a handle will be updated during *whole graph update*.

6.2.8.7.8.2 Conditional Node Body Graph Requirements

General requirements:

- The graph's nodes must all reside on a single device.
- The graph can only contain kernel nodes, empty nodes, memcpy nodes, memset nodes, child graph nodes, and conditional nodes.

Kernel nodes:

- Use of CUDA Dynamic Parallelism or Device Graph Launch by kernels in the graph is not permitted.
- Cooperative launches are permitted so long as MPS is not in use.

Memcpy/Memset nodes:

- Only copies/memsets involving device memory and/or pinned device-mapped host memory are permitted.
- Copies/memsets involving CUDA arrays are not permitted.
- Both operands must be accessible from the current device at time of instantiation. Note that the copy operation will be performed from the device on which the graph resides, even if it is targeting memory on another device.

6.2.8.7.8.3 Conditional IF Nodes

The body graph of an IF node will be executed once if the condition is non-zero when the node is executed. The following diagram depicts a 3 node graph where the middle node, B, is a conditional node:

The following code illustrates the creation of a graph containing an IF conditional node. The default value of the condition is set using an upstream kernel. The body of the conditional is populated using the *graph API*.

```
__global__ void setHandle(cudaGraphConditionalHandle handle)
{
    ...
    cudaGraphSetConditional(handle, value);
    ...
}

void graphSetup() {
```

(continues on next page)

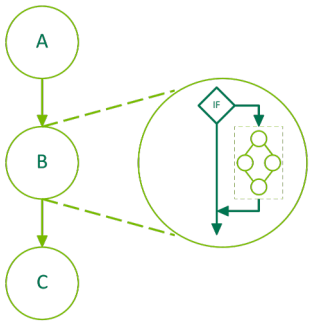


Figure 23: Conditional IF Node

(continued from previous page)

```
    cudaGraph_t graph;
    cudaGraphExec_t graphExec;
    cudaGraphNode_t node;
    void *kernelArgs[1];
    int value = 1;

    cudaGraphCreate(&graph, 0);

    cudaGraphConditionalHandle handle;
    cudaGraphConditionalHandleCreate(&handle, graph);

    // Use a kernel upstream of the conditional to set the handle value
    cudaGraphNodeParams params = { cudaGraphNodeTypeKernel };
    params.kernel.func = (void *)setHandle;
    params.kernel.gridDim.x = params.kernel.gridDim.y = params.kernel.gridDim.z = 1;
    params.kernel.blockDim.x = params.kernel.blockDim.y = params.kernel.blockDim.z =
↪1;
    params.kernel.kernelParams = kernelArgs;
    kernelArgs[0] = &handle;
    cudaGraphAddNode(&node, graph, NULL, NULL, 0, &params);

    cudaGraphNodeParams cParams = { cudaGraphNodeTypeConditional };
    cParams.conditional.handle = handle;
    cParams.conditional.type   = cudaGraphCondTypeIf;
    cParams.conditional.size   = 1;
    cudaGraphAddNode(&node, graph, &node, NULL, 1, &cParams);

    cudaGraph_t bodyGraph = cParams.conditional.phGraph_out[0];

    // Populate the body of the conditional node
    ...
    cudaGraphAddNode(&node, bodyGraph, NULL, NULL, 0, &params);

    cudaGraphInstantiate(&graphExec, graph, NULL, NULL, 0);
    cudaGraphLaunch(graphExec, 0);
    cudaDeviceSynchronize();

    cudaGraphExecDestroy(graphExec);
    cudaGraphDestroy(graph);
}
```

Starting in CUDA 12.8, IF nodes can also have an optional second body graph which is executed once

when the node is executed if the condition value is zero.

```
void graphSetup() {
    cudaGraph_t graph;
    cudaGraphExec_t graphExec;
    cudaGraphNode_t node;
    void *kernelArgs[1];
    int value = 1;

    cudaGraphCreate(&graph, 0);

    cudaGraphConditionalHandle handle;
    cudaGraphConditionalHandleCreate(&handle, graph);

    // Use a kernel upstream of the conditional to set the handle value
    cudaGraphNodeParams params = { cudaGraphNodeTypeKernel };
    params.kernel.func = (void *)setHandle;
    params.kernel.gridDim.x = params.kernel.gridDim.y = params.kernel.gridDim.z = 1;
    params.kernel.blockDim.x = params.kernel.blockDim.y = params.kernel.blockDim.z =
↪1;
    params.kernel.kernelParams = kernelArgs;
    kernelArgs[0] = &handle;
    cudaGraphAddNode(&node, graph, NULL, NULL, 0, &params);

    cudaGraphNodeParams cParams = { cudaGraphNodeTypeConditional };
    cParams.conditional.handle = handle;
    cParams.conditional.type   = cudaGraphCondTypeIf;
    cParams.conditional.size   = 2; // Note that size is now set to '2'
    cudaGraphAddNode(&node, graph, &node, NULL, 1, &cParams);

    cudaGraph_t ifBodyGraph = cParams.conditional.phGraph_out[0];
    cudaGraph_t elseBodyGraph = cParams.conditional.phGraph_out[1];

    // Populate the body graphs of the conditional node
    ...
    cudaGraphAddNode(&node, ifBodyGraph, NULL, NULL, 0, &params);
    ...
    cudaGraphAddNode(&node, elseBodyGraph, NULL, NULL, 0, &params);

    cudaGraphInstantiate(&graphExec, graph, NULL, NULL, 0);
    cudaGraphLaunch(graphExec, 0);
    cudaDeviceSynchronize();

    cudaGraphExecDestroy(graphExec);
    cudaGraphDestroy(graph);
}
```

6.2.8.7.8.4 Conditional WHILE Nodes

The body graph of a WHILE node will be executed as long as the condition is non-zero. The condition will be evaluated when the node is executed and after completion of the body graph. The following diagram depicts a 3 node graph where the middle node, B, is a conditional node:

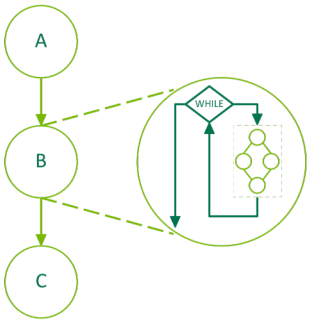


Figure 24: Conditional WHILE Node

The following code illustrates the creation of a graph containing a WHILE conditional node. The handle is created using *cudaGraphCondAssignDefault* to avoid the need for an upstream kernel. The body of the conditional is populated using the *graph API*.

```
__global__ void loopKernel(cudaGraphConditionalHandle handle)
{
    static int count = 10;
    cudaGraphSetConditional(handle, --count ? 1 : 0);
}

void graphSetup() {
    cudaGraph_t graph;
    cudaGraphExec_t graphExec;
    cudaGraphNode_t node;
    void *kernelArgs[1];

    cuGraphCreate(&graph, 0);

    cudaGraphConditionalHandle handle;
    cudaGraphConditionalHandleCreate(&handle, graph, 1, cudaGraphCondAssignDefault);

    cudaGraphNodeParams cParams = { cudaGraphNodeTypeConditional };
    cParams.conditional.handle = handle;
    cParams.conditional.type   = cudaGraphCondTypeWhile;
    cParams.conditional.size   = 1;
    cudaGraphAddNode(&node, graph, NULL, NULL, 0, &cParams);

    cudaGraph_t bodyGraph = cParams.conditional.phGraph_out[0];

    cudaGraphNodeParams params = { cudaGraphNodeTypeKernel };
    params.kernel.func = (void *)loopKernel;
    params.kernel.gridDim.x = params.kernel.gridDim.y = params.kernel.gridDim.z = 1;
    params.kernel.blockDim.x = params.kernel.blockDim.y = params.kernel.blockDim.z =
↪1;
    params.kernel.kernelParams = kernelArgs;
    kernelArgs[0] = &handle;
```

(continues on next page)

(continued from previous page)

```
    cudaGraphAddNode(&node, bodyGraph, NULL, NULL, 0, &params);

    cudaGraphInstantiate(&graphExec, graph, NULL, NULL, 0);
    cudaGraphLaunch(graphExec, 0);
    cudaDeviceSynchronize();

    cudaGraphExecDestroy(graphExec);
    cudaGraphDestroy(graph);
}
```

6.2.8.7.8.5 Conditional SWITCH Nodes

SWITCH nodes, added in CUDA 12.8, execute 1 of n different graphs within the conditional node. The nth graph will be executed when the SWITCH node is evaluated if the condition value is n. If the condition value is greater than or equal to n, no graph will be executed. The following diagram depicts a 3 node graph where the middle node, B, is a conditional node:

The following code illustrates the creation of a graph containing a SWITCH conditional node. The value of the condition is set using an upstream kernel. The bodies of the conditional are populated using the *graph API*.

```
__global__ void setHandle(cudaGraphConditionalHandle handle)
{
    ...
    cudaGraphSetConditional(handle, value);
    ...
}

void graphSetup() {
    cudaGraph_t graph;
    cudaGraphExec_t graphExec;
    cudaGraphNode_t node;
    void *kernelArgs[1];
    int value = 1;

    cudaGraphCreate(&graph, 0);

    cudaGraphConditionalHandle handle;
    cudaGraphConditionalHandleCreate(&handle, graph);

    // Use a kernel upstream of the conditional to set the handle value
    cudaGraphNodeParams params = { cudaGraphNodeTypeKernel };
    params.kernel.func = (void *)setHandle;
    params.kernel.gridDim.x = params.kernel.gridDim.y = params.kernel.gridDim.z = 1;
    params.kernel.blockDim.x = params.kernel.blockDim.y = params.kernel.blockDim.z =
↪1;
    params.kernel.kernelParams = kernelArgs;
    kernelArgs[0] = &handle;
    cudaGraphAddNode(&node, graph, NULL, NULL, 0, &params);

    cudaGraphNodeParams cParams = { cudaGraphNodeTypeConditional };
    cParams.conditional.handle = handle;
    cParams.conditional.type   = cudaGraphCondTypeSwitch;
    cParams.conditional.size   = 5;
```

(continues on next page)

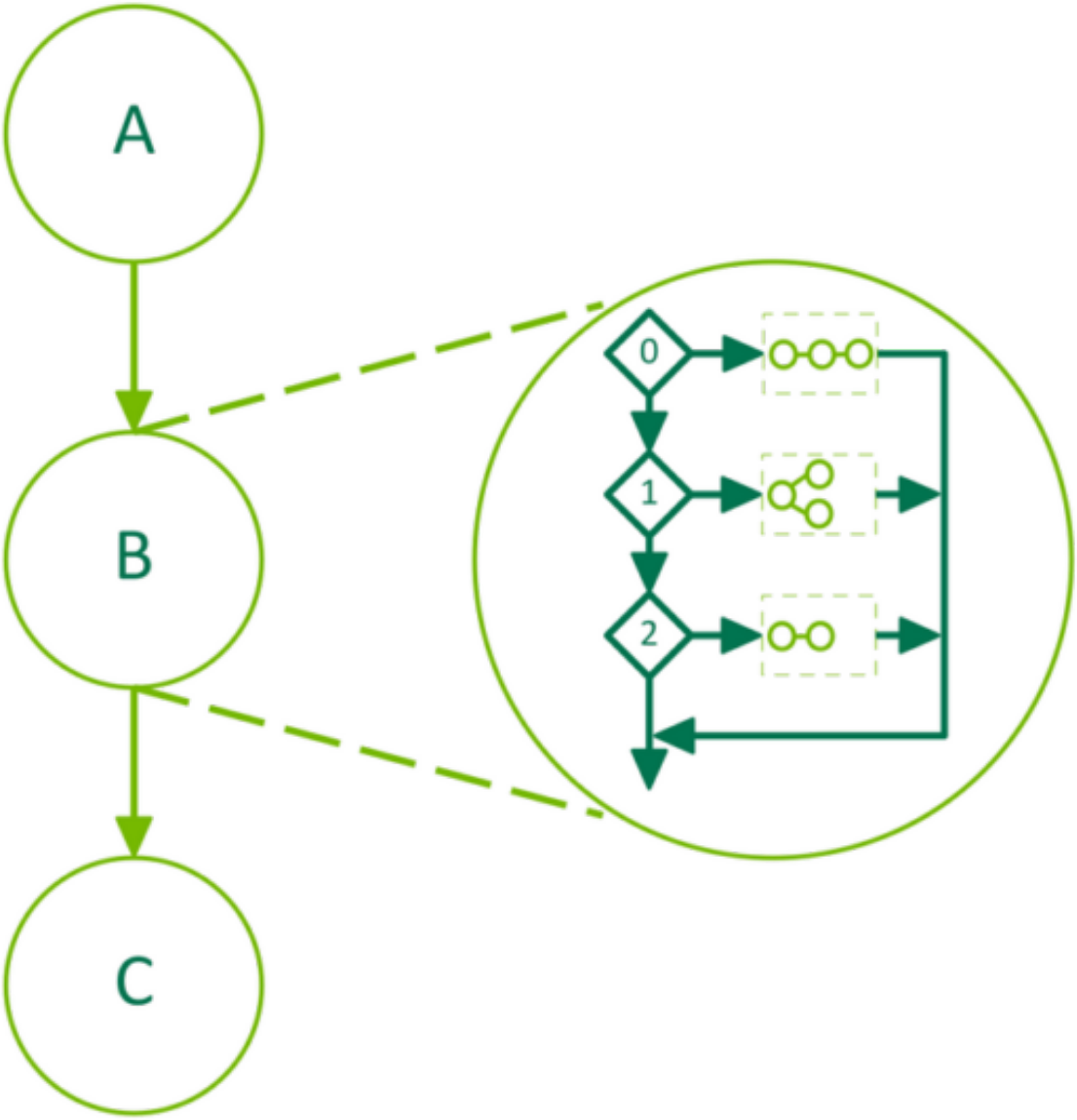


Figure 25: Conditional SWITCH Node

(continued from previous page)

```
    cudaGraphAddNode(&node, graph, &node, NULL, 1, &cParams);

    cudaGraph_t *bodyGraphs = cParams.conditional.phGraph_out;

    // Populate the first body of the conditional node
    ...
    cudaGraphAddNode(&node, bodyGraphs[0], NULL, NULL, 0, &params);
    ...
    // Populate the last body of the conditional node
    cudaGraphAddNode(&node, bodyGraphs[4], NULL, NULL, 0, &params);

    cudaGraphInstantiate(&graphExec, graph, NULL, NULL, 0);
    cudaGraphLaunch(graphExec, 0);
    cudaDeviceSynchronize();

    cudaGraphExecDestroy(graphExec);
    cudaGraphDestroy(graph);
}
```

6.2.8.8 Events

The runtime also provides a way to closely monitor the device's progress, as well as perform accurate timing, by letting the application asynchronously record *events* at any point in the program, and query when these events are completed. An event has completed when all tasks - or optionally, all commands in a given stream - preceding the event have completed. Events in stream zero are completed after all preceding tasks and commands in all streams are completed.

6.2.8.8.1 Creation and Destruction of Events

The following code sample creates two events:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
```

They are destroyed this way:

```
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

6.2.8.8.2 Elapsed Time

The events created in *Creation and Destruction of Events* can be used to time the code sample of *Creation and Destruction of Streams* the following way:

```
cudaEventRecord(start, 0);
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDev + i * size, inputHost + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel<<<100, 512, 0, stream[i]>>>
               (outputDev + i * size, inputDev + i * size, size);
```

(continues on next page)

(continued from previous page)

```
    cudaMemcpyAsync(outputHost + i * size, outputDev + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

6.2.8.9 Synchronous Calls

When a synchronous function is called, control is not returned to the host thread before the device has completed the requested task. Whether the host thread will then yield, block, or spin can be specified by calling `cudaSetDeviceFlags()` with some specific flags (see reference manual for details) before any other CUDA call is performed by the host thread.

6.2.9. Multi-Device System

6.2.9.1 Device Enumeration

A host system can have multiple devices. The following code sample shows how to enumerate these devices, query their properties, and determine the number of CUDA-enabled devices.

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, device);
    printf("Device %d has compute capability %d.%d.\n",
           device, deviceProp.major, deviceProp.minor);
}
```

6.2.9.2 Device Selection

A host thread can set the device it operates on at any time by calling `cudaSetDevice()`. Device memory allocations and kernel launches are made on the currently set device; streams and events are created in association with the currently set device. If no call to `cudaSetDevice()` is made, the current device is device 0.

The following code sample illustrates how setting the current device affects memory allocation and kernel execution.

```
size_t size = 1024 * sizeof(float);
cudaSetDevice(0);            // Set device 0 as current
float* p0;
cudaMalloc(&p0, size);       // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0); // Launch kernel on device 0
cudaSetDevice(1);            // Set device 1 as current
float* p1;
```

(continues on next page)

(continued from previous page)

```
cudaMalloc(&p1, size);       // Allocate memory on device 1
MyKernel<<<1000, 128>>>(p1); // Launch kernel on device 1
```

6.2.9.3 Stream and Event Behavior

A kernel launch will fail if it is issued to a stream that is not associated to the current device as illustrated in the following code sample.

```
cudaSetDevice(0);               // Set device 0 as current
cudaStream_t s0;
cudaStreamCreate(&s0);          // Create stream s0 on device 0
MyKernel<<<100, 64, 0, s0>>>(); // Launch kernel on device 0 in s0
cudaSetDevice(1);               // Set device 1 as current
cudaStream_t s1;
cudaStreamCreate(&s1);          // Create stream s1 on device 1
MyKernel<<<100, 64, 0, s1>>>(); // Launch kernel on device 1 in s1

// This kernel launch will fail:
MyKernel<<<100, 64, 0, s0>>>(); // Launch kernel on device 1 in s0
```

A memory copy will succeed even if it is issued to a stream that is not associated to the current device.

`cudaEventRecord()` will fail if the input event and input stream are associated to different devices.

`cudaEventElapsedTime()` will fail if the two input events are associated to different devices.

`cudaEventSynchronize()` and `cudaEventQuery()` will succeed even if the input event is associated to a device that is different from the current device.

`cudaStreamWaitEvent()` will succeed even if the input stream and input event are associated to different devices. `cudaStreamWaitEvent()` can therefore be used to synchronize multiple devices with each other.

Each device has its own default stream (see *Default Stream*), so commands issued to the default stream of a device may execute out of order or concurrently with respect to commands issued to the default stream of any other device.

6.2.9.4 Peer-to-Peer Memory Access

Depending on the system properties, specifically the PCIe and/or NVLINK topology, devices are able to address each other's memory (i.e., a kernel executing on one device can dereference a pointer to the memory of the other device). This peer-to-peer memory access feature is supported between two devices if `cudaDeviceCanAccessPeer()` returns true for these two devices.

Peer-to-peer memory access is only supported in 64-bit applications and must be enabled between two devices by calling `cudaDeviceEnablePeerAccess()` as illustrated in the following code sample. On non-NVSwitch enabled systems, each device can support a system-wide maximum of eight peer connections.

A unified address space is used for both devices (see *Unified Virtual Address Space*), so the same pointer can be used to address memory from both devices as shown in the code sample below.

```
cudaSetDevice(0);                   // Set device 0 as current
float* p0;
size_t size = 1024 * sizeof(float);
```

(continues on next page)

(continued from previous page)

```
cudaMalloc(&p0, size);                // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0);          // Launch kernel on device 0
cudaSetDevice(1);                     // Set device 1 as current
cudaDeviceEnablePeerAccess(0, 0);     // Enable peer-to-peer access
                                      // with device 0

// Launch kernel on device 1
// This kernel launch can access memory on device 0 at address p0
MyKernel<<<1000, 128>>>(p0);
```

6.2.9.4.1 IOMMU on Linux

On Linux only, CUDA and the display driver does not support IOMMU-enabled bare-metal PCIe peer to peer memory copy. However, CUDA and the display driver does support IOMMU via VM pass through. As a consequence, users on Linux, when running on a native bare metal system, should disable the IOMMU. The IOMMU should be enabled and the VFIO driver be used as a PCIe pass through for virtual machines.

On Windows the above limitation does not exist.

See also Allocating DMA Buffers on 64-bit Platforms.

6.2.9.5 Peer-to-Peer Memory Copy

Memory copies can be performed between the memories of two different devices.

When a unified address space is used for both devices (see *Unified Virtual Address Space*), this is done using the regular memory copy functions mentioned in *Device Memory*.

Otherwise, this is done using `cudaMemcpyPeer()`, `cudaMemcpyPeerAsync()`, `cudaMemcpy3DPeer()`, or `cudaMemcpy3DPeerAsync()` as illustrated in the following code sample.

```
cudaSetDevice(0);                     // Set device 0 as current
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);                // Allocate memory on device 0
cudaSetDevice(1);                     // Set device 1 as current
float* p1;
cudaMalloc(&p1, size);                // Allocate memory on device 1
cudaSetDevice(0);                     // Set device 0 as current
MyKernel<<<1000, 128>>>(p0);          // Launch kernel on device 0
cudaSetDevice(1);                     // Set device 1 as current
cudaMemcpyPeer(p1, 1, p0, 0, size); // Copy p0 to p1
MyKernel<<<1000, 128>>>(p1);          // Launch kernel on device 1
```

A copy (in the implicit *NULL* stream) between the memories of two different devices:

- does not start until all commands previously issued to either device have completed and
- runs to completion before any commands (see *Asynchronous Concurrent Execution*) issued after the copy to either device can start.

Consistent with the normal behavior of streams, an asynchronous copy between the memories of two devices may overlap with copies or kernels in another stream.

Note that if peer-to-peer access is enabled between two devices via `cudaDeviceEnablePeerAccess()` as described in *Peer-to-Peer Memory Access*, peer-to-peer memory copy between these two devices no longer needs to be staged through the host and is therefore faster.

6.2.10. Unified Virtual Address Space

When the application is run as a 64-bit process, a single address space is used for the host and all the devices of compute capability 2.0 and higher. All host memory allocations made via CUDA API calls and all device memory allocations on supported devices are within this virtual address range. As a consequence:

- The location of any memory on the host allocated through CUDA, or on any of the devices which use the unified address space, can be determined from the value of the pointer using `cudaPointerGetAttributes()`.
- When copying to or from the memory of any device which uses the unified address space, the `cudaMemcpyKind` parameter of `cudaMemcpy*()` can be set to `cudaMemcpyDefault` to determine locations from the pointers. This also works for host pointers not allocated through CUDA, as long as the current device uses unified addressing.
- Allocations via `cudaHostAlloc()` are automatically portable (see *Portable Memory*) across all the devices for which the unified address space is used, and pointers returned by `cudaHostAlloc()` can be used directly from within kernels running on these devices (i.e., there is no need to obtain a device pointer via `cudaHostGetDevicePointer()` as described in *Mapped Memory*.

Applications may query if the unified address space is used for a particular device by checking that the `unifiedAddressing` device property (see *Device Enumeration*) is equal to 1.

6.2.11. Interprocess Communication

Any device memory pointer or event handle created by a host thread can be directly referenced by any other thread within the same process. It is not valid outside this process however, and therefore cannot be directly referenced by threads belonging to a different process.

To share device memory pointers and events across processes, an application must use the Inter Process Communication API, which is described in detail in the reference manual. The IPC API is only supported for 64-bit processes on Linux and for devices of compute capability 2.0 and higher. Note that the IPC API is not supported for `cudaMallocManaged` allocations.

Using this API, an application can get the IPC handle for a given device memory pointer using `cudaIpcGetMemHandle()`, pass it to another process using standard IPC mechanisms (for example, interprocess shared memory or files), and use `cudaIpcOpenMemHandle()` to retrieve a device pointer from the IPC handle that is a valid pointer within this other process. Event handles can be shared using similar entry points.

Note that allocations made by `cudaMalloc()` may be sub-allocated from a larger block of memory for performance reasons. In such case, CUDA IPC APIs will share the entire underlying memory block which may cause other sub-allocations to be shared, which can potentially lead to information disclosure between processes. To prevent this behavior, it is recommended to only share allocations with a 2MiB aligned size.

An example of using the IPC API is where a single primary process generates a batch of input data, making the data available to multiple secondary processes without requiring regeneration or copying.

Applications using CUDA IPC to communicate with each other should be compiled, linked, and run with the same CUDA driver and runtime.

Note: Since CUDA 11.5, only events-sharing IPC APIs are supported on L4T and embedded Linux Tegra devices with compute capability 7.x and higher. The memory-sharing IPC APIs are still not supported on Tegra platforms.

6.2.12. Error Checking

All runtime functions return an error code, but for an asynchronous function (see *Asynchronous Concurrent Execution*), this error code cannot possibly report any of the asynchronous errors that could occur on the device since the function returns before the device has completed the task; the error code only reports errors that occur on the host prior to executing the task, typically related to parameter validation; if an asynchronous error occurs, it will be reported by some subsequent unrelated runtime function call.

The only way to check for asynchronous errors just after some asynchronous function call is therefore to synchronize just after the call by calling `cudaDeviceSynchronize()` (or by using any other synchronization mechanisms described in *Asynchronous Concurrent Execution*) and checking the error code returned by `cudaDeviceSynchronize()`.

The runtime maintains an error variable for each host thread that is initialized to `cudaSuccess` and is overwritten by the error code every time an error occurs (be it a parameter validation error or an asynchronous error). `cudaPeekAtLastError()` returns this variable. `cudaGetLastError()` returns this variable and resets it to `cudaSuccess`.

Kernel launches do not return any error code, so `cudaPeekAtLastError()` or `cudaGetLastError()` must be called just after the kernel launch to retrieve any pre-launch errors. To ensure that any error returned by `cudaPeekAtLastError()` or `cudaGetLastError()` does not originate from calls prior to the kernel launch, one has to make sure that the runtime error variable is set to `cudaSuccess` just before the kernel launch, for example, by calling `cudaGetLastError()` just before the kernel launch. Kernel launches are asynchronous, so to check for asynchronous errors, the application must synchronize in-between the kernel launch and the call to `cudaPeekAtLastError()` or `cudaGetLastError()`.

Note that `cudaErrorNotReady` that may be returned by `cudaStreamQuery()` and `cudaEventQuery()` is not considered an error and is therefore not reported by `cudaPeekAtLastError()` or `cudaGetLastError()`.

6.2.13. Call Stack

On devices of compute capability 2.x and higher, the size of the call stack can be queried using`cudaDeviceGetLimit()` and set using `cudaDeviceSetLimit()`.

When the call stack overflows, the kernel call fails with a stack overflow error if the application is run via a CUDA debugger (CUDA-GDB, Nsight) or an unspecified launch error, otherwise. When the compiler cannot determine the stack size, it issues a warning saying Stack size cannot be statically determined. This is usually the case with recursive functions. Once this warning is issued, user will need to set stack size manually if default stack size is not sufficient.

6.2.14. Texture and Surface Memory

CUDA supports a subset of the texturing hardware that the GPU uses for graphics to access texture and surface memory. Reading data from texture or surface memory instead of global memory can have several performance benefits as described in *Device Memory Accesses*.

6.2.14.1 Texture Memory

Texture memory is read from kernels using the device functions described in *Texture Functions*. The process of reading a texture calling one of these functions is called a *texture fetch*. Each texture fetch specifies a parameter called a *texture object* for the texture object API.

The texture object specifies:

- The *texture*, which is the piece of texture memory that is fetched. Texture objects are created at runtime and the texture is specified when creating the texture object as described in *Texture Object API*.
- Its *dimensionality* that specifies whether the texture is addressed as a one dimensional array using one texture coordinate, a two-dimensional array using two texture coordinates, or a three-dimensional array using three texture coordinates. Elements of the array are called *texels*, short for *texture elements*. The *texture width*, *height*, and *depth* refer to the size of the array in each dimension. Table 27 lists the maximum texture width, height, and depth depending on the compute capability of the device.
- The type of a texel, which is restricted to the basic integer and single-precision floating-point types and any of the 1-, 2-, and 4-component vector types defined in *Built-in Vector Types* that are derived from the basic integer and single-precision floating-point types.
- The *read mode*, which is equal to `cudaReadModeNormalizedFloat` or `cudaReadModeElementType`. If it is `cudaReadModeNormalizedFloat` and the type of the texel is a 16-bit or 8-bit integer type, the value returned by the texture fetch is actually returned as floating-point type and the full range of the integer type is mapped to [0.0, 1.0] for unsigned integer type and [-1.0, 1.0] for signed integer type; for example, an unsigned 8-bit texture element with the value 0xff reads as 1. If it is `cudaReadModeElementType`, no conversion is performed.
- Whether texture coordinates are normalized or not. By default, textures are referenced (by the functions of *Texture Functions*) using floating-point coordinates in the range [0, N-1] where N is the size of the texture in the dimension corresponding to the coordinate. For example, a texture that is 64x32 in size will be referenced with coordinates in the range [0, 63] and [0, 31] for the x and y dimensions, respectively. Normalized texture coordinates cause the coordinates to be specified in the range [0.0, 1.0-1/N] instead of [0, N-1], so the same 64x32 texture would be addressed by normalized coordinates in the range [0, 1-1/N] in both the x and y dimensions. Normalized texture coordinates are a natural fit to some applications' requirements, if it is preferable for the texture coordinates to be independent of the texture size.
- The *addressing mode*. It is valid to call the device functions of Section B.8 with coordinates that are out of range. The addressing mode defines what happens in that case. The default addressing mode is to clamp the coordinates to the valid range: [0, N) for non-normalized coordinates and [0.0, 1.0) for normalized coordinates. If the border mode is specified instead, texture fetches with out-of-range texture coordinates return zero. For normalized coordinates, the wrap mode and the mirror mode are also available. When using the wrap mode, each coordinate x is converted to *frac(x)=x - floor(x)* where *floor(x)* is the largest integer not greater than *x*. When using the mirror mode, each coordinate *x* is converted to *frac(x)* if *floor(x)* is even and *1-frac(x)* if *floor(x)* is odd. The addressing mode is specified as an array of size three whose first, second,

and third elements specify the addressing mode for the first, second, and third texture coordinates, respectively; the addressing mode are `cudaAddressModeBorder`, `cudaAddressModeClamp`, `cudaAddressModeWrap`, and `cudaAddressModeMirror`; `cudaAddressModeWrap` and `cudaAddressModeMirror` are only supported for normalized texture coordinates

- The *filtering* mode which specifies how the value returned when fetching the texture is computed based on the input texture coordinates. Linear texture filtering may be done only for textures that are configured to return floating-point data. It performs low-precision interpolation between neighboring texels. When enabled, the texels surrounding a texture fetch location are read and the return value of the texture fetch is interpolated based on where the texture coordinates fell between the texels. Simple linear interpolation is performed for one-dimensional textures, bilinear interpolation for two-dimensional textures, and trilinear interpolation for three-dimensional textures. *Texture Fetching* gives more details on texture fetching. The filtering mode is equal to `cudaFilterModePoint` or `cudaFilterModeLinear`. If it is `cudaFilterModePoint`, the returned value is the texel whose texture coordinates are the closest to the input texture coordinates. If it is `cudaFilterModeLinear`, the returned value is the linear interpolation of the two (for a one-dimensional texture), four (for a two dimensional texture), or eight (for a three dimensional texture) texels whose texture coordinates are the closest to the input texture coordinates. `cudaFilterModeLinear` is only valid for returned values of floating-point type.

Texture Object API introduces the texture object API.

16-Bit Floating-Point Textures explains how to deal with 16-bit floating-point textures.

Textures can also be layered as described in *Layered Textures*.

Cubemap Textures and *Cubemap Layered Textures* describe a special type of texture, the cubemap texture.

Texture Gather describes a special texture fetch, texture gather.

6.2.14.1.1 Texture Object API

A texture object is created using `cudaCreateTextureObject()` from a resource description of type `struct cudaResourceDesc`, which specifies the texture, and from a texture description defined as such:

```
struct cudaTextureDesc
{
    enum cudaTextureAddressMode addressMode[3];
    enum cudaTextureFilterMode  filterMode;
    enum cudaTextureReadMode    readMode;
    int                         sRGB;
    int                         normalizedCoords;
    unsigned int                maxAnisotropy;
    enum cudaTextureFilterMode  mipmapFilterMode;
    float                       mipmapLevelBias;
    float                       minMipmapLevelClamp;
    float                       maxMipmapLevelClamp;
};
```

- `addressMode` specifies the addressing mode;
- `filterMode` specifies the filter mode;
- `readMode` specifies the read mode;
- `normalizedCoords` specifies whether texture coordinates are normalized or not;

- See reference manual for `sRGB`, `maxAnisotropy`, `mipmapFilterMode`, `mipmapLevelBias`, `minMipmapLevelClamp`, and `maxMipmapLevelClamp`.

The following code sample applies some simple transformation kernel to a texture.

```
// Simple transformation kernel
__global__ void transformKernel(float* output,
                                cudaTextureObject_t texObj,
                                int width, int height,
                                float theta)
{
    // Calculate normalized texture coordinates
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

    float u = x / (float)width;
    float v = y / (float)height;

    // Transform coordinates
    u -= 0.5f;
    v -= 0.5f;
    float tu = u * cosf(theta) - v * sinf(theta) + 0.5f;
    float tv = v * cosf(theta) + u * sinf(theta) + 0.5f;

    // Read from texture and write to global memory
    output[y * width + x] = tex2D<float>(texObj, tu, tv);
}
```

```
// Host code
int main()
{
    const int height = 1024;
    const int width = 1024;
    float angle = 0.5;

    // Allocate and set some host data
    float *h_data = (float *)std::malloc(sizeof(float) * width * height);
    for (int i = 0; i < height * width; ++i)
        h_data[i] = i;

    // Allocate CUDA array in device memory
    cudaChannelFormatDesc channelDesc =
        cudaCreateChannelDesc(32, 0, 0, 0, cudaChannelFormatKindFloat);
    cudaArray_t cuArray;
    cudaMallocArray(&cuArray, &channelDesc, width, height);

    // Set pitch of the source (the width in memory in bytes of the 2D array pointed
    // to by src, including padding), we dont have any padding
    const size_t spitch = width * sizeof(float);
    // Copy data located at address h_data in host memory to device memory
    cudaMemcpy2DToArray(cuArray, 0, 0, h_data, spitch, width * sizeof(float),
                        height, cudaMemcpyHostToDevice);

    // Specify texture
    struct cudaResourceDesc resDesc;
    memset(&resDesc, 0, sizeof(resDesc));
    resDesc.resType = cudaResourceTypeArray;
    resDesc.res.array.array = cuArray;
```

(continues on next page)

(continued from previous page)

```
    // Specify texture object parameters
    struct cudaTextureDesc texDesc;
    memset(&texDesc, 0, sizeof(texDesc));
    texDesc.addressMode[0] = cudaAddressModeWrap;
    texDesc.addressMode[1] = cudaAddressModeWrap;
    texDesc.filterMode = cudaFilterModeLinear;
    texDesc.readMode = cudaReadModeElementType;
    texDesc.normalizedCoords = 1;

    // Create texture object
    cudaTextureObject_t texObj = 0;
    cudaCreateTextureObject(&texObj, &resDesc, &texDesc, NULL);

    // Allocate result of transformation in device memory
    float *output;
    cudaMalloc(&output, width * height * sizeof(float));

    // Invoke kernel
    dim3 threadsperBlock(16, 16);
    dim3 numBlocks((width + threadsperBlock.x - 1) / threadsperBlock.x,
                   (height + threadsperBlock.y - 1) / threadsperBlock.y);
    transformKernel<<<numBlocks, threadsperBlock>>>(output, texObj, width, height,
                                                    angle);
    // Copy data from device back to host
    cudaMemcpy(h_data, output, width * height * sizeof(float),
               cudaMemcpyDeviceToHost);

    // Destroy texture object
    cudaDestroyTextureObject(texObj);

    // Free device memory
    cudaFreeArray(cuArray);
    cudaFree(output);

    // Free host memory
    free(h_data);

    return 0;
}
```

6.2.14.1.2 16-Bit Floating-Point Textures

The 16-bit floating-point or *half* format supported by CUDA arrays is the same as the IEEE 754-2008 binary2 format.

CUDA C++ does not support a matching data type, but provides intrinsic functions to convert to and from the 32-bit floating-point format via the `unsigned short` type: `__float2half_rn(float)` and `__half2float(unsigned short)`. These functions are only supported in device code. Equivalent functions for the host code can be found in the OpenEXR library, for example.

16-bit floating-point components are promoted to 32 bit float during texture fetching before any filtering is performed.

A channel description for the 16-bit floating-point format can be created by calling one of the `cudaCreateChannelDescHalf*()` functions.

6.2.14.1.3 Layered Textures

A one-dimensional or two-dimensional layered texture (also known as *texture array* in Direct3D and *array texture* in OpenGL) is a texture made up of a sequence of layers, all of which are regular textures of same dimensionality, size, and data type.

A one-dimensional layered texture is addressed using an integer index and a floating-point texture coordinate; the index denotes a layer within the sequence and the coordinate addresses a texel within that layer. A two-dimensional layered texture is addressed using an integer index and two floating-point texture coordinates; the index denotes a layer within the sequence and the coordinates address a texel within that layer.

A layered texture can only be a CUDA array by calling `cudaMalloc3DArray()` with the `cudaArrayLayered` flag (and a height of zero for one-dimensional layered texture).

Layered textures are fetched using the device functions described in *tex1DLayered()* and *tex2DLayered()*. Texture filtering (see *Texture Fetching*) is done only within a layer, not across layers.

Layered textures are only supported on devices of compute capability 2.0 and higher.

6.2.14.1.4 Cubemap Textures

A *cubemap* texture is a special type of two-dimensional layered texture that has six layers representing the faces of a cube:

- The width of a layer is equal to its height.
- The cubemap is addressed using three texture coordinates *x*, *y*, and *z* that are interpreted as a direction vector emanating from the center of the cube and pointing to one face of the cube and a texel within the layer corresponding to that face. More specifically, the face is selected by the coordinate with largest magnitude *m* and the corresponding layer is addressed using coordinates *(s/m+1)/2* and *(t/m+1)/2* where *s* and *t* are defined in Table 6.

Table 6: Cubemap Fetch

		face	m	s	t
|x| > |y| and |x| > |z|	x ≥ 0	0	x	-z	-y
	x < 0	1	-x	z	-y
|y| > |x| and |y| > |z|	y ≥ 0	2	y	x	z
	y < 0	3	-y	x	-z
|z| > |x| and |z| > |y|	z ≥ 0	4	z	x	-y
	z < 0	5	-z	-x	-y

A cubemap texture can only be a CUDA array by calling `cudaMalloc3DArray()` with the `cudaArrayCubemap` flag.

Cubemap textures are fetched using the device function described in *texCubemap()*.

Cubemap textures are only supported on devices of compute capability 2.0 and higher.

6.2.14.1.5 Cubemap Layered Textures

A *cubemap layered* texture is a layered texture whose layers are cubemaps of same dimension.

A cubemap layered texture is addressed using an integer index and three floating-point texture coordinates; the index denotes a cubemap within the sequence and the coordinates address a texel within that cubemap.

A cubemap layered texture can only be a CUDA array by calling `cudaMalloc3DArray()` with the `cudaArrayLayered` and `cudaArrayCubemap` flags.

Cubemap layered textures are fetched using the device function described in *texCubemapLayered()*. Texture filtering (see *Texture Fetching*) is done only within a layer, not across layers.

Cubemap layered textures are only supported on devices of compute capability 2.0 and higher.

6.2.14.1.6 Texture Gather

Texture gather is a special texture fetch that is available for two-dimensional textures only. It is performed by the `tex2Dgather()` function, which has the same parameters as `tex2D()`, plus an additional `comp` parameter equal to 0, 1, 2, or 3 (see *tex2Dgather()*). It returns four 32-bit numbers that correspond to the value of the component `comp` of each of the four texels that would have been used for bilinear filtering during a regular texture fetch. For example, if these texels are of values (253, 20, 31, 255), (250, 25, 29, 254), (249, 16, 37, 253), (251, 22, 30, 250), and `comp` is 2, `tex2Dgather()` returns (31, 29, 37, 30).

Note that texture coordinates are computed with only 8 bits of fractional precision. `tex2Dgather()` may therefore return unexpected results for cases where `tex2D()` would use 1.0 for one of its weights (or , see *Linear Filtering*). For example, with an *x* texture coordinate of 2.49805: *xB=x-0.5=1.99805*, however the fractional part of *xB* is stored in an 8-bit fixed-point format. Since 0.99805 is closer to 256.f/256.f than it is to 255.f/256.f, *xB* has the value 2. A `tex2Dgather()` in this case would therefore return indices 2 and 3 in *x*, instead of indices 1 and 2.

Texture gather is only supported for CUDA arrays created with the `cudaArrayTextureGather` flag and of width and height less than the maximum specified in Table 27 for texture gather, which is smaller than for regular texture fetch.

Texture gather is only supported on devices of compute capability 2.0 and higher.

6.2.14.2 Surface Memory

For devices of compute capability 2.0 and higher, a CUDA array (described in *Cubemap Surfaces*), created with the `cudaArraySurfaceLoadStore` flag, can be read and written via a *surface object* using the functions described in *Surface Functions*.

Table 27 lists the maximum surface width, height, and depth depending on the compute capability of the device.

6.2.14.2.1 Surface Object API

A surface object is created using `cudaCreateSurfaceObject()` from a resource description of type `struct cudaResourceDesc`. Unlike texture memory, surface memory uses byte addressing. This means that the x-coordinate used to access a texture element via texture functions needs to be multiplied by the byte size of the element to access the same element via a surface function. For example, the element at texture coordinate x of a one-dimensional floating-point CUDA array bound to a texture object `texObj` and a surface object `surfObj` is read using `tex1d(texObj, x)` via `texObj`, but `surf1Dread(surfObj, 4*x)` via `surfObj`. Similarly, the element at texture coordinate x and y of a two-dimensional floating-point CUDA array bound to a texture object `texObj` and a surface object `surfObj` is accessed using `tex2d(texObj, x, y)` via `texObj`, but `surf2Dread(surfObj, 4*x, y)` via `surObj` (the byte offset of the y-coordinate is internally calculated from the underlying line pitch of the CUDA array).

The following code sample applies some simple transformation kernel to a surface.

```
// Simple copy kernel
__global__ void copyKernel(cudaSurfaceObject_t inputSurfObj,
                           cudaSurfaceObject_t outputSurfObj,
                           int width, int height)
{
    // Calculate surface coordinates
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < width && y < height) {
        uchar4 data;
        // Read from input surface
        surf2Dread(&data,  inputSurfObj, x * 4, y);
        // Write to output surface
        surf2Dwrite(data, outputSurfObj, x * 4, y);
    }
}

// Host code
int main()
{
    const int height = 1024;
    const int width = 1024;

    // Allocate and set some host data
    unsigned char *h_data =
        (unsigned char *)std::malloc(sizeof(unsigned char) * width * height * 4);
    for (int i = 0; i < height * width * 4; ++i)
        h_data[i] = i;

    // Allocate CUDA arrays in device memory
    cudaChannelFormatDesc channelDesc =
        cudaCreateChannelDesc(8, 8, 8, 8, cudaChannelFormatKindUnsigned);
    cudaArray_t cuInputArray;
    cudaMallocArray(&cuInputArray, &channelDesc, width, height,
                    cudaArraySurfaceLoadStore);
    cudaArray_t cuOutputArray;
    cudaMallocArray(&cuOutputArray, &channelDesc, width, height,
                    cudaArraySurfaceLoadStore);

    // Set pitch of the source (the width in memory in bytes of the 2D array
    // pointed to by src, including padding), we dont have any padding
```

(continues on next page)

(continued from previous page)

```
    const size_t spitch = 4 * width * sizeof(unsigned char);
    // Copy data located at address h_data in host memory to device memory
    cudaMemcpy2DToArray(cuInputArray, 0, 0, h_data, spitch,
                        4 * width * sizeof(unsigned char), height,
                        cudaMemcpyHostToDevice);

    // Specify surface
    struct cudaResourceDesc resDesc;
    memset(&resDesc, 0, sizeof(resDesc));
    resDesc.resType = cudaResourceTypeArray;

    // Create the surface objects
    resDesc.res.array.array = cuInputArray;
    cudaSurfaceObject_t inputSurfObj = 0;
    cudaCreateSurfaceObject(&inputSurfObj, &resDesc);
    resDesc.res.array.array = cuOutputArray;
    cudaSurfaceObject_t outputSurfObj = 0;
    cudaCreateSurfaceObject(&outputSurfObj, &resDesc);

    // Invoke kernel
    dim3 threadsperBlock(16, 16);
    dim3 numBlocks((width + threadsperBlock.x - 1) / threadsperBlock.x,
                   (height + threadsperBlock.y - 1) / threadsperBlock.y);
    copyKernel<<<numBlocks, threadsperBlock>>>(inputSurfObj, outputSurfObj, width,
                                               height);

    // Copy data from device back to host
    cudaMemcpy2DFromArray(h_data, spitch, cuOutputArray, 0, 0,
                          4 * width * sizeof(unsigned char), height,
                          cudaMemcpyDeviceToHost);

    // Destroy surface objects
    cudaDestroySurfaceObject(inputSurfObj);
    cudaDestroySurfaceObject(outputSurfObj);

    // Free device memory
    cudaFreeArray(cuInputArray);
    cudaFreeArray(cuOutputArray);

    // Free host memory
    free(h_data);

  return 0;
}
```

6.2.14.2.2 Cubemap Surfaces

Cubemap surfaces are accessed using`surfCubemapread()` and `surfCubemapwrite()` (*surfCubemapread()* and *surfCubemapwrite()*) as a two-dimensional layered surface, i.e., using an integer index denoting a face and two floating-point texture coordinates addressing a texel within the layer corresponding to this face. Faces are ordered as indicated in Table 6.

6.2.14.2.3 Cubemap Layered Surfaces

Cubemap layered surfaces are accessed using `surfCubemapLayeredread()` and `surfCubemapLayeredwrite()` (*surfCubemapLayeredread()* and *surfCubemapLayeredwrite()*) as a two-dimensional layered surface, i.e., using an integer index denoting a face of one of the cubemaps and two floating-point texture coordinates addressing a texel within the layer corresponding to this face. Faces are ordered as indicated in Table 6, so index ((2 * 6) + 3), for example, accesses the fourth face of the third cubemap.

6.2.14.3 CUDA Arrays

CUDA arrays are opaque memory layouts optimized for texture fetching. They are one dimensional, two dimensional, or three-dimensional and composed of elements, each of which has 1, 2 or 4 components that may be signed or unsigned 8-, 16-, or 32-bit integers, 16-bit floats, or 32-bit floats. CUDA arrays are only accessible by kernels through texture fetching as described in *Texture Memory* or surface reading and writing as described in *Surface Memory*.

6.2.14.4 Read/Write Coherency

The texture and surface memory is cached (see *Device Memory Accesses*) and within the same kernel call, the cache is not kept coherent with respect to global memory writes and surface memory writes, so any texture fetch or surface read to an address that has been written to via a global write or a surface write in the same kernel call returns undefined data. In other words, a thread can safely read some texture or surface memory location only if this memory location has been updated by a previous kernel call or memory copy, but not if it has been previously updated by the same thread or another thread from the same kernel call.

6.2.15. Graphics Interoperability

Some resources from OpenGL and Direct3D may be mapped into the address space of CUDA, either to enable CUDA to read data written by OpenGL or Direct3D, or to enable CUDA to write data for consumption by OpenGL or Direct3D.

A resource must be registered to CUDA before it can be mapped using the functions mentioned in *OpenGL Interoperability* and *Direct3D Interoperability*. These functions return a pointer to a CUDA graphics resource of type `struct cudaGraphicsResource`. Registering a resource is potentially high-overhead and therefore typically called only once per resource. A CUDA graphics resource is unregistered using `cudaGraphicsUnregisterResource()`. Each CUDA context which intends to use the resource is required to register it separately.

Once a resource is registered to CUDA, it can be mapped and unmapped as many times as necessary using `cudaGraphicsMapResources()` and `cudaGraphicsUnmapResources()`. `cudaGraphicsResourceSetMapFlags()` can be called to specify usage hints (write-only, read-only) that the CUDA driver can use to optimize resource management.

A mapped resource can be read from or written to by kernels using the device memory address returned by `cudaGraphicsResourceGetMappedPointer()` for buffers and`cudaGraphicsSubResourceGetMappedArray()` for CUDA arrays.

Accessing a resource through OpenGL, Direct3D, or another CUDA context while it is mapped produces undefined results. *OpenGL Interoperability* and *Direct3D Interoperability* give specifics for each graphics API and some code samples. *SLI Interoperability* gives specifics for when the system is in SLI mode.

6.2.15.1 OpenGL Interoperability

The OpenGL resources that may be mapped into the address space of CUDA are OpenGL buffer, texture, and renderbuffer objects.

A buffer object is registered using `cudaGraphicsGLRegisterBuffer()`. In CUDA, it appears as a device pointer and can therefore be read and written by kernels or via `cudaMemcpy()` calls.

A texture or renderbuffer object is registered using `cudaGraphicsGLRegisterImage()`. In CUDA, it appears as a CUDA array. Kernels can read from the array by binding it to a texture or surface reference. They can also write to it via the surface write functions if the resource has been registered with the `cudaGraphicsRegisterFlagsSurfaceLoadStore` flag. The array can also be read and written via `cudaMemcpy2D()` calls. `cudaGraphicsGLRegisterImage()` supports all texture formats with 1, 2, or 4 components and an internal type of float (for example, `GL_RGBA_FLOAT32`), normalized integer (for example, `GL_RGBA8`, `GL_INTENSITY16`), and unnormalized integer (for example, `GL_RGBA8UI`) (please note that since unnormalized integer formats require OpenGL 3.0, they can only be written by shaders, not the fixed function pipeline).

The OpenGL context whose resources are being shared has to be current to the host thread making any OpenGL interoperability API calls.

Please note: When an OpenGL texture is made bindless (say for example by requesting an image or texture handle using the `glGetTextureHandle*`/`glGetImageHandle*` APIs) it cannot be registered with CUDA. The application needs to register the texture for interop before requesting an image or texture handle.

The following code sample uses a kernel to dynamically modify a 2D `width` x `height` grid of vertices stored in a vertex buffer object:

```
GLuint positionsVBO;
struct cudaGraphicsResource* positionsVBO_CUDA;

int main()
{
    // Initialize OpenGL and GLUT for device 0
    // and make the OpenGL context current
    ...
    glutDisplayFunc(display);

    // Explicitly set device 0
    cudaSetDevice(0);

    // Create buffer object and register it with CUDA
    glGenBuffers(1, &positionsVBO);
    glBindBuffer(GL_ARRAY_BUFFER, positionsVBO);
    unsigned int size = width * height * 4 * sizeof(float);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
```

(continues on next page)

(continued from previous page)

```
    cudaGraphicsGLRegisterBuffer(&positionsVBO_CUDA,
                                 positionsVBO,
                                 cudaGraphicsMapFlagsWriteDiscard);

    // Launch rendering loop
    glutMainLoop();

    ...
}

void display()
{
    // Map buffer object for writing from CUDA
    float4* positions;
    cudaGraphicsMapResources(1, &positionsVBO_CUDA, 0);
    size_t num_bytes;
    cudaGraphicsResourceGetMappedPointer((void**)&positions,
                                         &num_bytes,
                                         positionsVBO_CUDA));

    // Execute kernel
    dim3 dimBlock(16, 16, 1);
    dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);
    createVertices<<<dimGrid, dimBlock>>>(positions, time,
                                          width, height);

    // Unmap buffer object
    cudaGraphicsUnmapResources(1, &positionsVBO_CUDA, 0);

    // Render from buffer object
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBindBuffer(GL_ARRAY_BUFFER, positionsVBO);
    glVertexPointer(4, GL_FLOAT, 0, 0);
    glEnableClientState(GL_VERTEX_ARRAY);
    glDrawArrays(GL_POINTS, 0, width * height);
    glDisableClientState(GL_VERTEX_ARRAY);

    // Swap buffers
    glutSwapBuffers();
    glutPostRedisplay();
}
```

```
void deleteVBO()
{
    cudaGraphicsUnregisterResource(positionsVBO_CUDA);
    glDeleteBuffers(1, &positionsVBO);
}

__global__ void createVertices(float4* positions, float time,
                               unsigned int width, unsigned int height)
{
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

    // Calculate uv coordinates
    float u = x / (float)width;
```

(continues on next page)

(continued from previous page)

```
    float v = y / (float)height;
    u = u * 2.0f - 1.0f;
    v = v * 2.0f - 1.0f;

    // calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u * freq + time)
            * cosf(v * freq + time) * 0.5f;

    // Write positions
    positions[y * width + x] = make_float4(u, w, v, 1.0f);
}
```

On Windows and for Quadro GPUs, `cudaWGLGetDevice()` can be used to retrieve the CUDA device associated to the handle returned by `wglEnumGpusNV()`. Quadro GPUs offer higher performance OpenGL interoperability than GeForce and Tesla GPUs in a multi-GPU configuration where OpenGL rendering is performed on the Quadro GPU and CUDA computations are performed on other GPUs in the system.

6.2.15.2 Direct3D Interoperability

Direct3D interoperability is supported for Direct3D 9Ex, Direct3D 10, and Direct3D 11.

A CUDA context may interoperate only with Direct3D devices that fulfill the following criteria: Direct3D 9Ex devices must be created with `DeviceType` set to `D3DDEVTYPE_HAL` and `BehaviorFlags` with the `D3DCREATE_HARDWARE_VERTEXPROCESSING` flag; Direct3D 10 and Direct3D 11 devices must be created with `DriverType` set to `D3D_DRIVER_TYPE_HARDWARE`.

The Direct3D resources that may be mapped into the address space of CUDA are Direct3D buffers, textures, and surfaces. These resources are registered using `cudaGraphicsD3D9RegisterResource()`, `cudaGraphicsD3D10RegisterResource()`, and `cudaGraphicsD3D11RegisterResource()`.

The following code sample uses a kernel to dynamically modify a 2D `width` x `height` grid of vertices stored in a vertex buffer object.

6.2.15.2.1 Direct3D 9 Version

```
IDirect3D9* D3D;
IDirect3DDevice9* device;
struct CUSTOMVERTEX {
    FLOAT x, y, z;
    DWORD color;
};
IDirect3DVertexBuffer9* positionsVB;
struct cudaGraphicsResource* positionsVB_CUDA;

int main()
{
    int dev;
    // Initialize Direct3D
    D3D = Direct3DCreate9Ex(D3D_SDK_VERSION);

    // Get a CUDA-enabled adapter
```

(continues on next page)

(continued from previous page)

```
    unsigned int adapter = 0;
    for (; adapter < g_pD3D->GetAdapterCount(); adapter++) {
        D3DADAPTER_IDENTIFIER9 adapterId;
        g_pD3D->GetAdapterIdentifier(adapter, 0, &adapterId);
        if (cudaD3D9GetDevice(&dev, adapterId.DeviceName)
            == cudaSuccess)
            break;
    }

     // Create device
    ...
    D3D->CreateDeviceEx(adapter, D3DDEVTYPE_HAL, hWnd,
                        D3DCREATE_HARDWARE_VERTEXPROCESSING,
                        &params, NULL, &device);

    // Use the same device
    cudaSetDevice(dev);

    // Create vertex buffer and register it with CUDA
    unsigned int size = width * height * sizeof(CUSTOMVERTEX);
    device->CreateVertexBuffer(size, 0, D3DFVF_CUSTOMVERTEX,
                               D3DPOOL_DEFAULT, &positionsVB, 0);
    cudaGraphicsD3D9RegisterResource(&positionsVB_CUDA,
                                     positionsVB,
                                     cudaGraphicsRegisterFlagsNone);
    cudaGraphicsResourceSetMapFlags(positionsVB_CUDA,
                                    cudaGraphicsMapFlagsWriteDiscard);

    // Launch rendering loop
    while (...) {
        ...
        Render();
        ...
    }
    ...
}
```

```
void Render()
{
    // Map vertex buffer for writing from CUDA
    float4* positions;
    cudaGraphicsMapResources(1, &positionsVB_CUDA, 0);
    size_t num_bytes;
    cudaGraphicsResourceGetMappedPointer((void**)&positions,
                                         &num_bytes,
                                         positionsVB_CUDA));

    // Execute kernel
    dim3 dimBlock(16, 16, 1);
    dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);
    createVertices<<<dimGrid, dimBlock>>>(positions, time,
                                          width, height);

    // Unmap vertex buffer
    cudaGraphicsUnmapResources(1, &positionsVB_CUDA, 0);
```

(continues on next page)

(continued from previous page)

```
    // Draw and present
    ...
}

void releaseVB()
{
    cudaGraphicsUnregisterResource(positionsVB_CUDA);
    positionsVB->Release();
}

__global__ void createVertices(float4* positions, float time,
                               unsigned int width, unsigned int height)
{
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

    // Calculate uv coordinates
    float u = x / (float)width;
    float v = y / (float)height;
    u = u * 2.0f - 1.0f;
    v = v * 2.0f - 1.0f;

    // Calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u * freq + time)
            * cosf(v * freq + time) * 0.5f;

    // Write positions
    positions[y * width + x] =
                make_float4(u, w, v, __int_as_float(0xff00ff00));
}
```

6.2.15.2.2 Direct3D 10 Version

```
ID3D10Device* device;
struct CUSTOMVERTEX {
    FLOAT x, y, z;
    DWORD color;
};
ID3D10Buffer* positionsVB;
struct cudaGraphicsResource* positionsVB_CUDA;

int main()
{
    int dev;
    // Get a CUDA-enabled adapter
    IDXGIFactory* factory;
    CreateDXGIFactory(__uuidof(IDXGIFactory), (void**)&factory);
    IDXGIAdapter* adapter = 0;
    for (unsigned int i = 0; !adapter; ++i) {
        if (FAILED(factory->EnumAdapters(i, &adapter))
            break;
        if (cudaD3D10GetDevice(&dev, adapter) == cudaSuccess)
            break;
```

(continues on next page)

(continued from previous page)

```
        adapter->Release();
    }
    factory->Release();

    // Create swap chain and device
    ...
    D3D10CreateDeviceAndSwapChain(adapter,
                                  D3D10_DRIVER_TYPE_HARDWARE, 0,
                                  D3D10_CREATE_DEVICE_DEBUG,
                                  D3D10_SDK_VERSION,
                                  &swapChainDesc, &swapChain,
                                  &device);
    adapter->Release();

    // Use the same device
    cudaSetDevice(dev);

    // Create vertex buffer and register it with CUDA
    unsigned int size = width * height * sizeof(CUSTOMVERTEX);
    D3D10_BUFFER_DESC bufferDesc;
    bufferDesc.Usage          = D3D10_USAGE_DEFAULT;
    bufferDesc.ByteWidth      = size;
    bufferDesc.BindFlags      = D3D10_BIND_VERTEX_BUFFER;
    bufferDesc.CPUAccessFlags = 0;
    bufferDesc.MiscFlags      = 0;
    device->CreateBuffer(&bufferDesc, 0, &positionsVB);
    cudaGraphicsD3D10RegisterResource(&positionsVB_CUDA,
                                      positionsVB,
                                      cudaGraphicsRegisterFlagsNone);
                                      cudaGraphicsResourceSetMapFlags(positionsVB_
↪CUDA,
                                      cudaGraphicsMapFlagsWriteDiscard);

    // Launch rendering loop
    while (...) {
        ...
        Render();
        ...
    }
    ...
}
```

```
void Render()
{
    // Map vertex buffer for writing from CUDA
    float4* positions;
    cudaGraphicsMapResources(1, &positionsVB_CUDA, 0);
    size_t num_bytes;
    cudaGraphicsResourceGetMappedPointer((void**)&positions,
                                         &num_bytes,
                                         positionsVB_CUDA));

    // Execute kernel
    dim3 dimBlock(16, 16, 1);
    dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);
    createVertices<<<dimGrid, dimBlock>>>(positions, time,
```

(continues on next page)

(continued from previous page)

```
                                           width, height);

    // Unmap vertex buffer
    cudaGraphicsUnmapResources(1, &positionsVB_CUDA, 0);

    // Draw and present
    ...
}

void releaseVB()
{
    cudaGraphicsUnregisterResource(positionsVB_CUDA);
    positionsVB->Release();
}

__global__ void createVertices(float4* positions, float time,
                               unsigned int width, unsigned int height)
{
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

    // Calculate uv coordinates
    float u = x / (float)width;
    float v = y / (float)height;
    u = u * 2.0f - 1.0f;
    v = v * 2.0f - 1.0f;

    // Calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u * freq + time)
            * cosf(v * freq + time) * 0.5f;

    // Write positions
    positions[y * width + x] =
                make_float4(u, w, v, __int_as_float(0xff00ff00));
}
```

6.2.15.2.3 Direct3D 11 Version

```
ID3D11Device* device;
struct CUSTOMVERTEX {
    FLOAT x, y, z;
    DWORD color;
};
ID3D11Buffer* positionsVB;
struct cudaGraphicsResource* positionsVB_CUDA;

int main()
{
    int dev;
    // Get a CUDA-enabled adapter
    IDXGIFactory* factory;
    CreateDXGIFactory(__uuidof(IDXGIFactory), (void**)&factory);
    IDXGIAdapter* adapter = 0;
```

(continues on next page)

(continued from previous page)

```
    for (unsigned int i = 0; !adapter; ++i) {
        if (FAILED(factory->EnumAdapters(i, &adapter))
            break;
        if (cudaD3D11GetDevice(&dev, adapter) == cudaSuccess)
            break;
        adapter->Release();
    }
    factory->Release();

    // Create swap chain and device
    ...
    sFnPtr_D3D11CreateDeviceAndSwapChain(adapter,
                                         D3D11_DRIVER_TYPE_HARDWARE,
                                         0,
                                         D3D11_CREATE_DEVICE_DEBUG,
                                         featureLevels, 3,
                                         D3D11_SDK_VERSION,
                                         &swapChainDesc, &swapChain,
                                         &device,
                                         &featureLevel,
                                         &deviceContext);
    adapter->Release();

    // Use the same device
    cudaSetDevice(dev);

    // Create vertex buffer and register it with CUDA
    unsigned int size = width * height * sizeof(CUSTOMVERTEX);
    D3D11_BUFFER_DESC bufferDesc;
    bufferDesc.Usage          = D3D11_USAGE_DEFAULT;
    bufferDesc.ByteWidth      = size;
    bufferDesc.BindFlags      = D3D11_BIND_VERTEX_BUFFER;
    bufferDesc.CPUAccessFlags = 0;
    bufferDesc.MiscFlags      = 0;
    device->CreateBuffer(&bufferDesc, 0, &positionsVB);
    cudaGraphicsD3D11RegisterResource(&positionsVB_CUDA,
                                      positionsVB,
                                      cudaGraphicsRegisterFlagsNone);
    cudaGraphicsResourceSetMapFlags(positionsVB_CUDA,
                                    cudaGraphicsMapFlagsWriteDiscard);

    // Launch rendering loop
    while (...) {
        ...
        Render();
        ...
    }
    ...
}
```

```
void Render()
{
    // Map vertex buffer for writing from CUDA
    float4* positions;
    cudaGraphicsMapResources(1, &positionsVB_CUDA, 0);
    size_t num_bytes;
```

(continues on next page)

(continued from previous page)

```
    cudaGraphicsResourceGetMappedPointer((void**)&positions,
                                         &num_bytes,
                                         positionsVB_CUDA));

    // Execute kernel
    dim3 dimBlock(16, 16, 1);
    dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);
    createVertices<<<dimGrid, dimBlock>>>(positions, time,
                                          width, height);

    // Unmap vertex buffer
    cudaGraphicsUnmapResources(1, &positionsVB_CUDA, 0);

    // Draw and present
    ...
}

void releaseVB()
{
    cudaGraphicsUnregisterResource(positionsVB_CUDA);
    positionsVB->Release();
}

    __global__ void createVertices(float4* positions, float time,
                           unsigned int width, unsigned int height)
{
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

// Calculate uv coordinates
    float u = x / (float)width;
    float v = y / (float)height;
    u = u * 2.0f - 1.0f;
    v = v * 2.0f - 1.0f;

    // Calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u * freq + time)
            * cosf(v * freq + time) * 0.5f;

    // Write positions
    positions[y * width + x] =
                make_float4(u, w, v, __int_as_float(0xff00ff00));
}
```

6.2.15.3 SLI Interoperability

In a system with multiple GPUs, all CUDA-enabled GPUs are accessible via the CUDA driver and runtime as separate devices. There are however special considerations as described below when the system is in SLI mode.

First, an allocation in one CUDA device on one GPU will consume memory on other GPUs that are part of the SLI configuration of the Direct3D or OpenGL device. Because of this, allocations may fail earlier than otherwise expected.

Second, applications should create multiple CUDA contexts, one for each GPU in the SLI configura-

tion. While this is not a strict requirement, it avoids unnecessary data transfers between devices. The application can use the `cudaD3D[9|10|11]GetDevices()` for Direct3D and `cudaGLGetDevices()` for OpenGL set of calls to identify the CUDA device handle(s) for the device(s) that are performing the rendering in the current and next frame. Given this information the application will typically choose the appropriate device and map Direct3D or OpenGL resources to the CUDA device returned by `cudaD3D[9|10|11]GetDevices()` or `cudaGLGetDevices()` when the `deviceList` parameter is set to `cudaD3D[9|10|11]DeviceListCurrentFrame` or `cudaGLDeviceListCurrentFrame`.

Please note that resource returned from `cudaGraphicsD9D[9|10|11]RegisterResource` and `cudaGraphicsGLRegister[Buffer|Image]` must be only used on device the registration happened. Therefore on SLI configurations when data for different frames is computed on different CUDA devices it is necessary to register the resources for each separately.

See *Direct3D Interoperability* and *OpenGL Interoperability* for details on how the CUDA runtime interoperate with Direct3D and OpenGL, respectively.

6.2.16. External Resource Interoperability

External resource interoperability allows CUDA to import certain resources that are explicitly exported by other APIs. These objects are typically exported by other APIs using handles native to the Operating System, like file descriptors on Linux or NT handles on Windows. They could also be exported using other unified interfaces such as the NVIDIA Software Communication Interface. There are two types of resources that can be imported: memory objects and synchronization objects.

Memory objects can be imported into CUDA using `cudaImportExternalMemory()`. An imported memory object can be accessed from within kernels using device pointers mapped onto the memory object via `cudaExternalMemoryGetMappedBuffer()`or CUDA mipmapped arrays mapped via `cudaExternalMemoryGetMappedMipmappedArray()`. Depending on the type of memory object, it may be possible for more than one mapping to be setup on a single memory object. The mappings must match the mappings setup in the exporting API. Any mismatched mappings result in undefined behavior. Imported memory objects must be freed using `cudaDestroyExternalMemory()`. Freeing a memory object does not free any mappings to that object. Therefore, any device pointers mapped onto that object must be explicitly freed using `cudaFree()` and any CUDA mipmapped arrays mapped onto that object must be explicitly freed using `cudaFreeMipmappedArray()`. It is illegal to access mappings to an object after it has been destroyed.

Synchronization objects can be imported into CUDA using `cudaImportExternalSemaphore()`. An imported synchronization object can then be signaled using `cudaSignalExternalSemaphoresAsync()` and waited on using `cudaWaitExternalSemaphoresAsync()`. It is illegal to issue a wait before the corresponding signal has been issued. Also, depending on the type of the imported synchronization object, there may be additional constraints imposed on how they can be signaled and waited on, as described in subsequent sections. Imported semaphore objects must be freed using `cudaDestroyExternalSemaphore()`. All outstanding signals and waits must have completed before the semaphore object is destroyed.

6.2.16.1 Vulkan Interoperability

6.2.16.1.1 Matching device UUIDs

When importing memory and synchronization objects exported by Vulkan, they must be imported and mapped on the same device as they were created on. The CUDA device that corresponds to the Vulkan physical device on which the objects were created can be determined by comparing the UUID of a CUDA device with that of the Vulkan physical device, as shown in the following code sample. Note that the Vulkan physical device should not be part of a device group that contains more than one Vulkan physical device. The device group as returned by `vkEnumeratePhysicalDeviceGroups` that contains the given Vulkan physical device must have a physical device count of 1.

```
int getCudaDeviceForVulkanPhysicalDevice(VkPhysicalDevice vkPhysicalDevice) {
    VkPhysicalDeviceIDProperties vkPhysicalDeviceIDProperties = {};
    vkPhysicalDeviceIDProperties.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_
↪PROPERTIES;
    vkPhysicalDeviceIDProperties.pNext = NULL;

    VkPhysicalDeviceProperties2 vkPhysicalDeviceProperties2 = {};
    vkPhysicalDeviceProperties2.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_
↪2;
    vkPhysicalDeviceProperties2.pNext = &vkPhysicalDeviceIDProperties;

    vkGetPhysicalDeviceProperties2(vkPhysicalDevice, &vkPhysicalDeviceProperties2);

    int cudaDeviceCount;
    cudaGetDeviceCount(&cudaDeviceCount);

    for (int cudaDevice = 0; cudaDevice < cudaDeviceCount; cudaDevice++) {
        cudaDeviceProp deviceProp;
        cudaGetDeviceProperties(&deviceProp, cudaDevice);
        if (!memcmp(&deviceProp.uuid, vkPhysicalDeviceIDProperties.deviceUUID, VK_
↪UUID_SIZE)) {
            return cudaDevice;
        }
    }
    return cudaInvalidDeviceId;
}
```

6.2.16.1.2 Importing Memory Objects

On Linux and Windows 10, both dedicated and non-dedicated memory objects exported by Vulkan can be imported into CUDA. On Windows 7, only dedicated memory objects can be imported. When importing a Vulkan dedicated memory object, the flag `cudaExternalMemoryDedicated` must be set.

A Vulkan memory object exported using `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT` can be imported into CUDA using the file descriptor associated with that object as shown below. Note that CUDA assumes ownership of the file descriptor once it is imported. Using the file descriptor after a successful import results in undefined behavior.

```
cudaExternalMemory_t importVulkanMemoryObjectFromFileDescriptor(int fd, unsigned long
↪long size, bool isDedicated) {
    cudaExternalMemory_t extMem = NULL;
    cudaExternalMemoryHandleDesc desc = {};
```

(continues on next page)

(continued from previous page)

```
    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalMemoryHandleTypeOpaqueFd;
    desc.handle.fd = fd;
    desc.size = size;
    if (isDedicated) {
        desc.flags |= cudaExternalMemoryDedicated;
    }

    cudaImportExternalMemory(&extMem, &desc);

    // Input parameter 'fd' should not be used beyond this point as CUDA has assumed
↪ownership of it

    return extMem;
}
```

A Vulkan memory object exported using VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT can be imported into CUDA using the NT handle associated with that object as shown below. Note that CUDA does not assume ownership of the NT handle and it is the application's responsibility to close the handle when it is not required anymore. The NT handle holds a reference to the resource, so it must be explicitly freed before the underlying memory can be freed.

```
cudaExternalMemory_t importVulkanMemoryObjectFromNTHandle(HANDLE handle, unsigned
↪long long size, bool isDedicated) {
    cudaExternalMemory_t extMem = NULL;
    cudaExternalMemoryHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalMemoryHandleTypeOpaqueWin32;
    desc.handle.win32.handle = handle;
    desc.size = size;
    if (isDedicated) {
        desc.flags |= cudaExternalMemoryDedicated;
    }

    cudaImportExternalMemory(&extMem, &desc);

    // Input parameter 'handle' should be closed if it's not needed anymore
    CloseHandle(handle);

    return extMem;
}
```

A Vulkan memory object exported using VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT can also be imported using a named handle if one exists as shown below.

```
cudaExternalMemory_t importVulkanMemoryObjectFromNamedNTHandle(LPCWSTR name, unsigned
↪long long size, bool isDedicated) {
    cudaExternalMemory_t extMem = NULL;
    cudaExternalMemoryHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));
```

(continues on next page)

(continued from previous page)

```
    desc.type = cudaExternalMemoryHandleTypeOpaqueWin32;
    desc.handle.win32.name = (void *)name;
    desc.size = size;
    if (isDedicated) {
        desc.flags |= cudaExternalMemoryDedicated;
    }

    cudaImportExternalMemory(&extMem, &desc);

    return extMem;
}
```

A Vulkan memory object exported using VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT can be imported into CUDA using the globally shared D3DKMT handle associated with that object as shown below. Since a globally shared D3DKMT handle does not hold a reference to the underlying memory it is automatically destroyed when all other references to the resource are destroyed.

```
cudaExternalMemory_t importVulkanMemoryObjectFromKMTHandle(HANDLE handle, unsigned
↪long long size, bool isDedicated) {
    cudaExternalMemory_t extMem = NULL;
    cudaExternalMemoryHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalMemoryHandleTypeOpaqueWin32Kmt;
    desc.handle.win32.handle = (void *)handle;
    desc.size = size;
    if (isDedicated) {
        desc.flags |= cudaExternalMemoryDedicated;
    }

    cudaImportExternalMemory(&extMem, &desc);

    return extMem;
}
```

6.2.16.1.3 Mapping Buffers onto Imported Memory Objects

A device pointer can be mapped onto an imported memory object as shown below. The offset and size of the mapping must match that specified when creating the mapping using the corresponding Vulkan API. All mapped device pointers must be freed using `cudaFree()`.

```
void * mapBufferOntoExternalMemory(cudaExternalMemory_t extMem, unsigned long long
↪offset, unsigned long long size) {

    void *ptr = NULL;

    cudaExternalMemoryBufferDesc desc = {};



    memset(&desc, 0, sizeof(desc));
```

(continues on next page)

(continued from previous page)

```
    desc.offset = offset;

    desc.size = size;



    cudaExternalMemoryGetMappedBuffer(&ptr, extMem, &desc);



    // Note: 'ptr' must eventually be freed using cudaFree()

    return ptr;

}
```

6.2.16.1.4 Mapping Mipmapped Arrays onto Imported Memory Objects

A CUDA mipmapped array can be mapped onto an imported memory object as shown below. The offset, dimensions, format and number of mip levels must match that specified when creating the mapping using the corresponding Vulkan API. Additionally, if the mipmapped array is bound as a color target in Vulkan, the flag`cudaArrayColorAttachment` must be set. All mapped mipmapped arrays must be freed using `cudaFreeMipmappedArray()`. The following code sample shows how to convert Vulkan parameters into the corresponding CUDA parameters when mapping mipmapped arrays onto imported memory objects.

```
cudaMipmappedArray_t mapMipmappedArrayOntoExternalMemory(cudaExternalMemory_t extMem,
↪unsigned long long offset, cudaChannelFormatDesc *formatDesc, cudaExtent *extent,
↪unsigned int flags, unsigned int numLevels) {
    cudaMipmappedArray_t mipmap = NULL;
    cudaExternalMemoryMipmappedArrayDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.offset = offset;
    desc.formatDesc = *formatDesc;
    desc.extent = *extent;
    desc.flags = flags;
    desc.numLevels = numLevels;

    // Note: 'mipmap' must eventually be freed using cudaFreeMipmappedArray()
    cudaExternalMemoryGetMappedMipmappedArray(&mipmap, extMem, &desc);

    return mipmap;
}

cudaChannelFormatDesc getCudaChannelFormatDescForVulkanFormat(VkFormat format)
{
    cudaChannelFormatDesc d;

    memset(&d, 0, sizeof(d));

    switch (format) {
```

(continues on next page)

(continued from previous page)

```
    case VK_FORMAT_R8_UINT:             d.x = 8;  d.y = 0;  d.z = 0;  d.w = 0;  d.f =
↪cudaChannelFormatKindUnsigned; break;
    case VK_FORMAT_R8_SINT:             d.x = 8;  d.y = 0;  d.z = 0;  d.w = 0;  d.f =
↪cudaChannelFormatKindSigned;   break;
    case VK_FORMAT_R8G8_UINT:           d.x = 8;  d.y = 8;  d.z = 0;  d.w = 0;  d.f =
↪cudaChannelFormatKindUnsigned; break;
    case VK_FORMAT_R8G8_SINT:           d.x = 8;  d.y = 8;  d.z = 0;  d.w = 0;  d.f =
↪cudaChannelFormatKindSigned;   break;
    case VK_FORMAT_R8G8B8A8_UINT:       d.x = 8;  d.y = 8;  d.z = 8;  d.w = 8;  d.f =
↪cudaChannelFormatKindUnsigned; break;
    case VK_FORMAT_R8G8B8A8_SINT:       d.x = 8;  d.y = 8;  d.z = 8;  d.w = 8;  d.f =
↪cudaChannelFormatKindSigned;   break;
    case VK_FORMAT_R16_UINT:            d.x = 16; d.y = 0;  d.z = 0;  d.w = 0;  d.f =
↪cudaChannelFormatKindUnsigned; break;
    case VK_FORMAT_R16_SINT:            d.x = 16; d.y = 0;  d.z = 0;  d.w = 0;  d.f =
↪cudaChannelFormatKindSigned;   break;
    case VK_FORMAT_R16G16_UINT:         d.x = 16; d.y = 16; d.z = 0;  d.w = 0;  d.f =
↪cudaChannelFormatKindUnsigned; break;
    case VK_FORMAT_R16G16_SINT:         d.x = 16; d.y = 16; d.z = 0;  d.w = 0;  d.f =
↪cudaChannelFormatKindSigned;   break;
    case VK_FORMAT_R16G16B16A16_UINT:   d.x = 16; d.y = 16; d.z = 16; d.w = 16; d.f =
↪cudaChannelFormatKindUnsigned; break;
    case VK_FORMAT_R16G16B16A16_SINT:   d.x = 16; d.y = 16; d.z = 16; d.w = 16; d.f =
↪cudaChannelFormatKindSigned;   break;
    case VK_FORMAT_R32_UINT:            d.x = 32; d.y = 0;  d.z = 0;  d.w = 0;  d.f =
↪cudaChannelFormatKindUnsigned; break;
    case VK_FORMAT_R32_SINT:            d.x = 32; d.y = 0;  d.z = 0;  d.w = 0;  d.f =
↪cudaChannelFormatKindSigned;   break;
    case VK_FORMAT_R32_SFLOAT:          d.x = 32; d.y = 0;  d.z = 0;  d.w = 0;  d.f =
↪cudaChannelFormatKindFloat;    break;
    case VK_FORMAT_R32G32_UINT:         d.x = 32; d.y = 32; d.z = 0;  d.w = 0;  d.f =
↪cudaChannelFormatKindUnsigned; break;
    case VK_FORMAT_R32G32_SINT:         d.x = 32; d.y = 32; d.z = 0;  d.w = 0;  d.f =
↪cudaChannelFormatKindSigned;   break;
    case VK_FORMAT_R32G32_SFLOAT:       d.x = 32; d.y = 32; d.z = 0;  d.w = 0;  d.f =
↪cudaChannelFormatKindFloat;    break;
    case VK_FORMAT_R32G32B32A32_UINT:   d.x = 32; d.y = 32; d.z = 32; d.w = 32; d.f =
↪cudaChannelFormatKindUnsigned; break;
    case VK_FORMAT_R32G32B32A32_SINT:   d.x = 32; d.y = 32; d.z = 32; d.w = 32; d.f =
↪cudaChannelFormatKindSigned;   break;
    case VK_FORMAT_R32G32B32A32_SFLOAT: d.x = 32; d.y = 32; d.z = 32; d.w = 32; d.f =
↪cudaChannelFormatKindFloat;    break;
    default: assert(0);
    }

    return d;
}

cudaExtent getCudaExtentForVulkanExtent(VkExtent3D vkExt, uint32_t arrayLayers,
↪VkImageViewType vkImageViewType) {
    cudaExtent e = { 0, 0, 0 };

    switch (vkImageViewType) {
    case VK_IMAGE_VIEW_TYPE_1D:         e.width = vkExt.width; e.height = 0;
↪   e.depth = 0;           break;
    case VK_IMAGE_VIEW_TYPE_2D:         e.width = vkExt.width; e.height = vkExt.
↪height; e.depth = 0;           break;
```

(continues on next page)

(continued from previous page)

```
    case VK_IMAGE_VIEW_TYPE_3D:         e.width = vkExt.width; e.height = vkExt.
↪height; e.depth = vkExt.depth; break;
    case VK_IMAGE_VIEW_TYPE_CUBE:       e.width = vkExt.width; e.height = vkExt.
↪height; e.depth = arrayLayers; break;
    case VK_IMAGE_VIEW_TYPE_1D_ARRAY:   e.width = vkExt.width; e.height = 0;
↪  e.depth = arrayLayers; break;
    case VK_IMAGE_VIEW_TYPE_2D_ARRAY:   e.width = vkExt.width; e.height = vkExt.
↪height; e.depth = arrayLayers; break;
    case VK_IMAGE_VIEW_TYPE_CUBE_ARRAY: e.width = vkExt.width; e.height = vkExt.
↪height; e.depth = arrayLayers; break;
    default: assert(0);
    }

    return e;
}

unsigned int getCudaMipmappedArrayFlagsForVulkanImage(VkImageViewType vkImageViewType,
↪ VkImageUsageFlags vkImageUsageFlags, bool allowSurfaceLoadStore) {
    unsigned int flags = 0;

    switch (vkImageViewType) {
    case VK_IMAGE_VIEW_TYPE_CUBE:       flags |= cudaArrayCubemap;
↪break;
    case VK_IMAGE_VIEW_TYPE_CUBE_ARRAY: flags |= cudaArrayCubemap | cudaArrayLayered;
↪break;
    case VK_IMAGE_VIEW_TYPE_1D_ARRAY:   flags |= cudaArrayLayered;
↪break;
    case VK_IMAGE_VIEW_TYPE_2D_ARRAY:   flags |= cudaArrayLayered;
↪break;
    default: break;
    }

    if (vkImageUsageFlags & VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT) {
        flags |= cudaArrayColorAttachment;
    }

    if (allowSurfaceLoadStore) {
        flags |= cudaArraySurfaceLoadStore;
    }
    return flags;
}
```

6.2.16.1.5 Importing Synchronization Objects

A Vulkan semaphore object exported using `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT`can be imported into CUDA using the file descriptor associated with that object as shown below. Note that CUDA assumes ownership of the file descriptor once it is imported. Using the file descriptor after a successful import results in undefined behavior.

```
cudaExternalSemaphore_t importVulkanSemaphoreObjectFromFileDescriptor(int fd) {
    cudaExternalSemaphore_t extSem = NULL;
    cudaExternalSemaphoreHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));
```

(continues on next page)

(continued from previous page)

```
    desc.type = cudaExternalSemaphoreHandleTypeOpaqueFd;
    desc.handle.fd = fd;

    cudaImportExternalSemaphore(&extSem, &desc);

    // Input parameter 'fd' should not be used beyond this point as CUDA has assumed
↪ownership of it

    return extSem;
}
```

A Vulkan semaphore object exported using VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT can be imported into CUDA using the NT handle associated with that object as shown below. Note that CUDA does not assume ownership of the NT handle and it is the application's responsibility to close the handle when it is not required anymore. The NT handle holds a reference to the resource, so it must be explicitly freed before the underlying semaphore can be freed.

```
cudaExternalSemaphore_t importVulkanSemaphoreObjectFromNTHandle(HANDLE handle) {
    cudaExternalSemaphore_t extSem = NULL;
    cudaExternalSemaphoreHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalSemaphoreHandleTypeOpaqueWin32;
    desc.handle.win32.handle = handle;

    cudaImportExternalSemaphore(&extSem, &desc);

    // Input parameter 'handle' should be closed if it's not needed anymore
    CloseHandle(handle);

    return extSem;
}
```

A Vulkan semaphore object exported using VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT can also be imported using a named handle if one exists as shown below.

```
cudaExternalSemaphore_t importVulkanSemaphoreObjectFromNamedNTHandle(LPCWSTR name) {
    cudaExternalSemaphore_t extSem = NULL;
    cudaExternalSemaphoreHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalSemaphoreHandleTypeOpaqueWin32;
    desc.handle.win32.name = (void *)name;

    cudaImportExternalSemaphore(&extSem, &desc);

    return extSem;
}
```

A Vulkan semaphore object exported using VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT can be imported into CUDA using the globally shared D3DKMT handle associated with that object as shown below. Since a globally shared D3DKMT handle does not hold a reference to the underlying semaphore it is automatically destroyed when all other references to the resource are destroyed.

```
cudaExternalSemaphore_t importVulkanSemaphoreObjectFromKMTHandle(HANDLE handle) {
    cudaExternalSemaphore_t extSem = NULL;
    cudaExternalSemaphoreHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt;
    desc.handle.win32.handle = (void *)handle;

    cudaImportExternalSemaphore(&extSem, &desc);

    return extSem;
}
```

6.2.16.1.6 Signaling/Waiting on Imported Synchronization Objects

An imported Vulkan semaphore object can be signaled as shown below. Signaling such a semaphore object sets it to the signaled state. The corresponding wait that waits on this signal must be issued in Vulkan. Additionally, the wait that waits on this signal must be issued after this signal has been issued.

```
void signalExternalSemaphore(cudaExternalSemaphore_t extSem, cudaStream_t stream) {
    cudaExternalSemaphoreSignalParams params = {};

    memset(&params, 0, sizeof(params));

    cudaSignalExternalSemaphoresAsync(&extSem, &params, 1, stream);
}
```

An imported Vulkan semaphore object can be waited on as shown below. Waiting on such a semaphore object waits until it reaches the signaled state and then resets it back to the unsignaled state. The corresponding signal that this wait is waiting on must be issued in Vulkan. Additionally, the signal must be issued before this wait can be issued.

```
void waitExternalSemaphore(cudaExternalSemaphore_t extSem, cudaStream_t stream) {
    cudaExternalSemaphoreWaitParams params = {};

    memset(&params, 0, sizeof(params));

    cudaWaitExternalSemaphoresAsync(&extSem, &params, 1, stream);
}
```

6.2.16.2 OpenGL Interoperability

Traditional OpenGL-CUDA interop as outlined in *OpenGL Interoperability* works by CUDA directly consuming handles created in OpenGL. However, since OpenGL can also consume memory and synchronization objects created in Vulkan, there exists an alternative approach to doing OpenGL-CUDA interop. Essentially, memory and synchronization objects exported by Vulkan could be imported into both, OpenGL and CUDA, and then used to coordinate memory accesses between OpenGL and CUDA. Please refer to the following OpenGL extensions for further details on how to import memory and synchronization objects exported by Vulkan:

- GL_EXT_memory_object
- GL_EXT_memory_object_fd

- GL_EXT_memory_object_win32
- GL_EXT_semaphore
- GL_EXT_semaphore_fd
- GL_EXT_semaphore_win32

6.2.16.3 Direct3D 12 Interoperability

6.2.16.3.1 Matching Device LUIDs

When importing memory and synchronization objects exported by Direct3D 12, they must be imported and mapped on the same device as they were created on. The CUDA device that corresponds to the Direct3D 12 device on which the objects were created can be determined by comparing the LUID of a CUDA device with that of the Direct3D 12 device, as shown in the following code sample. Note that the Direct3D 12 device must not be created on a linked node adapter. I.e. the node count as returned by `ID3D12Device::GetNodeCount` must be 1.

```
int getCudaDeviceForD3D12Device(ID3D12Device *d3d12Device) {
    LUID d3d12Luid = d3d12Device->GetAdapterLuid();

    int cudaDeviceCount;
    cudaGetDeviceCount(&cudaDeviceCount);

    for (int cudaDevice = 0; cudaDevice < cudaDeviceCount; cudaDevice++) {
        cudaDeviceProp deviceProp;
        cudaGetDeviceProperties(&deviceProp, cudaDevice);
        char *cudaLuid = deviceProp.luid;

        if (!memcmp(&d3d12Luid.LowPart, cudaLuid, sizeof(d3d12Luid.LowPart)) &&
            !memcmp(&d3d12Luid.HighPart, cudaLuid + sizeof(d3d12Luid.LowPart),
↪sizeof(d3d12Luid.HighPart))) {
            return cudaDevice;
        }
    }
    return cudaInvalidDeviceId;
}
```

6.2.16.3.2 Importing Memory Objects

A shareable Direct3D 12 heap memory object, created by setting the flag `D3D12_HEAP_FLAG_SHARED` in the call to `ID3D12Device::CreateHeap`, can be imported into CUDA using the NT handle associated with that object as shown below. Note that it is the application's responsibility to close the NT handle when it is not required anymore. The NT handle holds a reference to the resource, so it must be explicitly freed before the underlying memory can be freed.

```
cudaExternalMemory_t importD3D12HeapFromNTHandle(HANDLE handle, unsigned long long
↪size) {
    cudaExternalMemory_t extMem = NULL;
    cudaExternalMemoryHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));
```

(continues on next page)

(continued from previous page)

```
    desc.type = cudaExternalMemoryHandleTypeD3D12Heap;
    desc.handle.win32.handle = (void *)handle;
    desc.size = size;

    cudaImportExternalMemory(&extMem, &desc);

    // Input parameter 'handle' should be closed if it's not needed anymore
    CloseHandle(handle);

    return extMem;
}
```

A shareable Direct3D 12 heap memory object can also be imported using a named handle if one exists as shown below.

```
cudaExternalMemory_t importD3D12HeapFromNamedNTHandle(LPCWSTR name, unsigned long
↪long size) {
    cudaExternalMemory_t extMem = NULL;
    cudaExternalMemoryHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalMemoryHandleTypeD3D12Heap;
    desc.handle.win32.name = (void *)name;
    desc.size = size;

    cudaImportExternalMemory(&extMem, &desc);

    return extMem;
}
```

A shareable Direct3D 12 committed resource, created by setting the flag `D3D12_HEAP_FLAG_SHARED` in the call to `D3D12Device::CreateCommittedResource`, can be imported into CUDA using the NT handle associated with that object as shown below. When importing a Direct3D 12 committed resource, the flag `cudaExternalMemoryDedicated` must be set. Note that it is the application's responsibility to close the NT handle when it is not required anymore. The NT handle holds a reference to the resource, so it must be explicitly freed before the underlying memory can be freed.

```
cudaExternalMemory_t importD3D12CommittedResourceFromNTHandle(HANDLE handle, unsigned
↪long long size) {
    cudaExternalMemory_t extMem = NULL;
    cudaExternalMemoryHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalMemoryHandleTypeD3D12Resource;
    desc.handle.win32.handle = (void *)handle;
    desc.size = size;
    desc.flags |= cudaExternalMemoryDedicated;

    cudaImportExternalMemory(&extMem, &desc);

    // Input parameter 'handle' should be closed if it's not needed anymore
    CloseHandle(handle);

    return extMem;
}
```

A shareable Direct3D 12 committed resource can also be imported using a named handle if one exists as shown below.

```
cudaExternalMemory_t importD3D12CommittedResourceFromNamedNTHandle(LPCWSTR name,
↪unsigned long long size) {
    cudaExternalMemory_t extMem = NULL;
    cudaExternalMemoryHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalMemoryHandleTypeD3D12Resource;
    desc.handle.win32.name = (void *)name;
    desc.size = size;
    desc.flags |= cudaExternalMemoryDedicated;

    cudaImportExternalMemory(&extMem, &desc);

    return extMem;
}
```

6.2.16.3.3 Mapping Buffers onto Imported Memory Objects

A device pointer can be mapped onto an imported memory object as shown below. The offset and size of the mapping must match that specified when creating the mapping using the corresponding Direct3D 12 API. All mapped device pointers must be freed using `cudaFree()`.

```
void * mapBufferOntoExternalMemory(cudaExternalMemory_t extMem, unsigned long long
↪offset, unsigned long long size) {
    void *ptr = NULL;
    cudaExternalMemoryBufferDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.offset = offset;
    desc.size = size;

    cudaExternalMemoryGetMappedBuffer(&ptr, extMem, &desc);

    // Note: 'ptr' must eventually be freed using cudaFree()
    return ptr;
}
```

6.2.16.3.4 Mapping Mipmapped Arrays onto Imported Memory Objects

A CUDA mipmapped array can be mapped onto an imported memory object as shown below. The offset, dimensions, format and number of mip levels must match that specified when creating the mapping using the corresponding Direct3D 12 API. Additionally, if the mipmapped array can be bound as a render target in Direct3D 12, the flag `cudaArrayColorAttachment` must be set. All mapped mipmapped arrays must be freed using `cudaFreeMipmappedArray()`. The following code sample shows how to convert Vulkan parameters into the corresponding CUDA parameters when mapping mipmapped arrays onto imported memory objects.

```
cudaMipmappedArray_t mapMipmappedArrayOntoExternalMemory(cudaExternalMemory_t extMem,
↪unsigned long long offset, cudaChannelFormatDesc *formatDesc, cudaExtent *extent,
↪unsigned int flags, unsigned int numLevels) {
    cudaMipmappedArray_t mipmap = NULL;
    cudaExternalMemoryMipmappedArrayDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.offset = offset;
    desc.formatDesc = *formatDesc;
    desc.extent = *extent;
    desc.flags = flags;
    desc.numLevels = numLevels;

    // Note: 'mipmap' must eventually be freed using cudaFreeMipmappedArray()
    cudaExternalMemoryGetMappedMipmappedArray(&mipmap, extMem, &desc);

    return mipmap;
}

cudaChannelFormatDesc getCudaChannelFormatDescForDxgiFormat(DXGI_FORMAT dxgiFormat)
{
    cudaChannelFormatDesc d;

    memset(&d, 0, sizeof(d));

    switch (dxgiFormat) {
    case DXGI_FORMAT_R8_UINT:           d.x = 8;  d.y = 0;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R8_SINT:           d.x = 8;  d.y = 0;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R8G8_UINT:         d.x = 8;  d.y = 8;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R8G8_SINT:         d.x = 8;  d.y = 8;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R8G8B8A8_UINT:     d.x = 8;  d.y = 8;  d.z = 8;  d.w = 8;  d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R8G8B8A8_SINT:     d.x = 8;  d.y = 8;  d.z = 8;  d.w = 8;  d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R16_UINT:          d.x = 16; d.y = 0;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R16_SINT:          d.x = 16; d.y = 0;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R16G16_UINT:       d.x = 16; d.y = 16; d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R16G16_SINT:       d.x = 16; d.y = 16; d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R16G16B16A16_UINT: d.x = 16; d.y = 16; d.z = 16; d.w = 16; d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R16G16B16A16_SINT: d.x = 16; d.y = 16; d.z = 16; d.w = 16; d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R32_UINT:          d.x = 32; d.y = 0;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R32_SINT:          d.x = 32; d.y = 0;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R32_FLOAT:         d.x = 32; d.y = 0;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindFloat;    break;
```

(continues on next page)

(continued from previous page)

```
    case DXGI_FORMAT_R32G32_UINT:        d.x = 32; d.y = 32; d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R32G32_SINT:        d.x = 32; d.y = 32; d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R32G32_FLOAT:       d.x = 32; d.y = 32; d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindFloat;    break;
    case DXGI_FORMAT_R32G32B32A32_UINT:  d.x = 32; d.y = 32; d.z = 32; d.w = 32; d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R32G32B32A32_SINT:  d.x = 32; d.y = 32; d.z = 32; d.w = 32; d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R32G32B32A32_FLOAT: d.x = 32; d.y = 32; d.z = 32; d.w = 32; d.f
↪= cudaChannelFormatKindFloat;    break;
    default: assert(0);
```

```
    }

    return d;
}

cudaExtent getCudaExtentForD3D12Extent(UINT64 width, UINT height, UINT16
↪depthOrArraySize, D3D12_SRV_DIMENSION d3d12SRVDimension) {
    cudaExtent e = { 0, 0, 0 };

    switch (d3d12SRVDimension) {
    case D3D12_SRV_DIMENSION_TEXTURE1D:        e.width = width; e.height = 0;      e.
↪depth = 0;                break;
    case D3D12_SRV_DIMENSION_TEXTURE2D:        e.width = width; e.height = height; e.
↪depth = 0;                break;
    case D3D12_SRV_DIMENSION_TEXTURE3D:        e.width = width; e.height = height; e.
↪depth = depthOrArraySize; break;
    case D3D12_SRV_DIMENSION_TEXTURECUBE:      e.width = width; e.height = height; e.
↪depth = depthOrArraySize; break;
    case D3D12_SRV_DIMENSION_TEXTURE1DARRAY:   e.width = width; e.height = 0;      e.
↪depth = depthOrArraySize; break;
    case D3D12_SRV_DIMENSION_TEXTURE2DARRAY:   e.width = width; e.height = height; e.
↪depth = depthOrArraySize; break;
    case D3D12_SRV_DIMENSION_TEXTURECUBEARRAY: e.width = width; e.height = height; e.
↪depth = depthOrArraySize; break;
    default: assert(0);
    }

    return e;
}

unsigned int getCudaMipmappedArrayFlagsForD3D12Resource(D3D12_SRV_DIMENSION
↪d3d12SRVDimension, D3D12_RESOURCE_FLAGS d3d12ResourceFlags, bool
↪allowSurfaceLoadStore) {
    unsigned int flags = 0;

    switch (d3d12SRVDimension) {
    case D3D12_SRV_DIMENSION_TEXTURECUBE:      flags |= cudaArrayCubemap;
↪       break;
    case D3D12_SRV_DIMENSION_TEXTURECUBEARRAY: flags |= cudaArrayCubemap |
↪cudaArrayLayered; break;
    case D3D12_SRV_DIMENSION_TEXTURE1DARRAY:   flags |= cudaArrayLayered;
↪       break;
```

(continues on next page)

(continued from previous page)

```
    case D3D12_SRV_DIMENSION_TEXTURE2DARRAY:   flags |= cudaArrayLayered;
↪       break;
    default: break;
    }

    if (d3d12ResourceFlags & D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET) {
        flags |= cudaArrayColorAttachment;
    }
    if (allowSurfaceLoadStore) {
        flags |= cudaArraySurfaceLoadStore;
    }

    return flags;
}
```

6.2.16.3.5 Importing Synchronization Objects

A shareable Direct3D 12 fence object, created by setting the flag `D3D12_FENCE_FLAG_SHARED` in the call to `ID3D12Device::CreateFence`, can be imported into CUDA using the NT handle associated with that object as shown below. Note that it is the application's responsibility to close the handle when it is not required anymore. The NT handle holds a reference to the resource, so it must be explicitly freed before the underlying semaphore can be freed.

```
cudaExternalSemaphore_t importD3D12FenceFromNTHandle(HANDLE handle) {
    cudaExternalSemaphore_t extSem = NULL;
    cudaExternalSemaphoreHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalSemaphoreHandleTypeD3D12Fence;
    desc.handle.win32.handle = handle;

    cudaImportExternalSemaphore(&extSem, &desc);

    // Input parameter 'handle' should be closed if it's not needed anymore
    CloseHandle(handle);

    return extSem;
}
```

A shareable Direct3D 12 fence object can also be imported using a named handle if one exists as shown below.

```
cudaExternalSemaphore_t importD3D12FenceFromNamedNTHandle(LPCWSTR name) {
    cudaExternalSemaphore_t extSem = NULL;
    cudaExternalSemaphoreHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalSemaphoreHandleTypeD3D12Fence;
    desc.handle.win32.name = (void *)name;

    cudaImportExternalSemaphore(&extSem, &desc);
```

(continues on next page)

(continued from previous page)

```
    return extSem;
}
```

6.2.16.3.6 Signaling/Waiting on Imported Synchronization Objects

An imported Direct3D 12 fence object can be signaled as shown below. Signaling such a fence object sets its value to the one specified. The corresponding wait that waits on this signal must be issued in Direct3D 12. Additionally, the wait that waits on this signal must be issued after this signal has been issued.

```
void signalExternalSemaphore(cudaExternalSemaphore_t extSem, unsigned long long value,
↪ cudaStream_t stream) {
    cudaExternalSemaphoreSignalParams params = {};

    memset(&params, 0, sizeof(params));

    params.params.fence.value = value;

    cudaSignalExternalSemaphoresAsync(&extSem, &params, 1, stream);
}
```

An imported Direct3D 12 fence object can be waited on as shown below. Waiting on such a fence object waits until its value becomes greater than or equal to the specified value. The corresponding signal that this wait is waiting on must be issued in Direct3D 12. Additionally, the signal must be issued before this wait can be issued.

```
void waitExternalSemaphore(cudaExternalSemaphore_t extSem, unsigned long long value,
↪cudaStream_t stream) {
    cudaExternalSemaphoreWaitParams params = {};

    memset(&params, 0, sizeof(params));

    params.params.fence.value = value;

    cudaWaitExternalSemaphoresAsync(&extSem, &params, 1, stream);
}
```

6.2.16.4 Direct3D 11 Interoperability

6.2.16.4.1 Matching Device LUIDs

When importing memory and synchronization objects exported by Direct3D 11, they must be imported and mapped on the same device as they were created on. The CUDA device that corresponds to the Direct3D 11 device on which the objects were created can be determined by comparing the LUID of a CUDA device with that of the Direct3D 11 device, as shown in the following code sample.

```
int getCudaDeviceForD3D11Device(ID3D11Device *d3d11Device) {
    IDXGIDevice *dxgiDevice;
    d3d11Device->QueryInterface(__uuidof(IDXGIDevice), (void **)&dxgiDevice);

    IDXGIAdapter *dxgiAdapter;
```

(continues on next page)

(continued from previous page)

```
    dxgiDevice->GetAdapter(&dxgiAdapter);

    DXGI_ADAPTER_DESC dxgiAdapterDesc;
    dxgiAdapter->GetDesc(&dxgiAdapterDesc);

    LUID d3d11Luid = dxgiAdapterDesc.AdapterLuid;

    int cudaDeviceCount;
    cudaGetDeviceCount(&cudaDeviceCount);

    for (int cudaDevice = 0; cudaDevice < cudaDeviceCount; cudaDevice++) {
        cudaDeviceProp deviceProp;
        cudaGetDeviceProperties(&deviceProp, cudaDevice);
        char *cudaLuid = deviceProp.luid;

        if (!memcmp(&d3d11Luid.LowPart, cudaLuid, sizeof(d3d11Luid.LowPart)) &&
            !memcmp(&d3d11Luid.HighPart, cudaLuid + sizeof(d3d11Luid.LowPart),
↪sizeof(d3d11Luid.HighPart))) {
            return cudaDevice;
        }
    }
    return cudaInvalidDeviceId;
}
```

6.2.16.4.2 Importing Memory Objects

A shareable Direct3D 11 texture resource, viz, `ID3D11Texture1D`, `ID3D11Texture2D` or `ID3D11Texture3D`, can be created by setting either the `D3D11_RESOURCE_MISC_SHARED` or `D3D11_RESOURCE_MISC_SHARED_KEYEDMUTEX` (on Windows 7) or `D3D11_RESOURCE_MISC_SHARED_NTHANDLE` (on Windows 10) when calling `ID3D11Device:CreateTexture1D`, `ID3D11Device:CreateTexture2D` or `ID3D11Device:CreateTexture3D` respectively. A shareable Direct3D 11 buffer resource, `ID3D11Buffer`, can be created by specifying either of the above flags when calling `ID3D11Device::CreateBuffer`. A shareable resource created by specifying the `D3D11_RESOURCE_MISC_SHARED_NTHANDLE` can be imported into CUDA using the NT handle associated with that object as shown below. Note that it is the application's responsibility to close the NT handle when it is not required anymore. The NT handle holds a reference to the resource, so it must be explicitly freed before the underlying memory can be freed. When importing a Direct3D 11 resource, the flag `cudaExternalMemoryDedicated` must be set.

```
cudaExternalMemory_t importD3D11ResourceFromNTHandle(HANDLE handle, unsigned long
↪long size) {
    cudaExternalMemory_t extMem = NULL;
    cudaExternalMemoryHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalMemoryHandleTypeD3D11Resource;
    desc.handle.win32.handle = (void *)handle;
    desc.size = size;
    desc.flags |= cudaExternalMemoryDedicated;

    cudaImportExternalMemory(&extMem, &desc);
```

(continues on next page)

(continued from previous page)

```
    // Input parameter 'handle' should be closed if it's not needed anymore
    CloseHandle(handle);

    return extMem;
}
```

A shareable Direct3D 11 resource can also be imported using a named handle if one exists as shown below.

```
cudaExternalMemory_t importD3D11ResourceFromNamedNTHandle(LPCWSTR name, unsigned long
↪long size) {
    cudaExternalMemory_t extMem = NULL;
    cudaExternalMemoryHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalMemoryHandleTypeD3D11Resource;
    desc.handle.win32.name = (void *)name;
    desc.size = size;
    desc.flags |= cudaExternalMemoryDedicated;

    cudaImportExternalMemory(&extMem, &desc);

    return extMem;
}
```

A shareable Direct3D 11 resource, created by specifying the `D3D11_RESOURCE_MISC_SHARED` or `D3D11_RESOURCE_MISC_SHARED_KEYEDMUTEX`, can be imported into CUDA using the globally shared `D3DKMT` handle associated with that object as shown below. Since a globally shared `D3DKMT` handle does not hold a reference to the underlying memory it is automatically destroyed when all other references to the resource are destroyed.

```
cudaExternalMemory_t importD3D11ResourceFromKMTHandle(HANDLE handle, unsigned long
↪long size) {
    cudaExternalMemory_t extMem = NULL;
    cudaExternalMemoryHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalMemoryHandleTypeD3D11ResourceKmt;
    desc.handle.win32.handle = (void *)handle;
    desc.size = size;
    desc.flags |= cudaExternalMemoryDedicated;

    cudaImportExternalMemory(&extMem, &desc);

    return extMem;
}
```

6.2.16.4.3 Mapping Buffers onto Imported Memory Objects

A device pointer can be mapped onto an imported memory object as shown below. The offset and size of the mapping must match that specified when creating the mapping using the corresponding Direct3D 11 API. All mapped device pointers must be freed using `cudaFree()`.

```
void * mapBufferOntoExternalMemory(cudaExternalMemory_t extMem, unsigned long long
↪offset, unsigned long long size) {
    void *ptr = NULL;
    cudaExternalMemoryBufferDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.offset = offset;
    desc.size = size;

    cudaExternalMemoryGetMappedBuffer(&ptr, extMem, &desc);

    // Note: 'ptr' must eventually be freed using cudaFree()
    return ptr;
}
```

6.2.16.4.4 Mapping Mipmapped Arrays onto Imported Memory Objects

A CUDA mipmapped array can be mapped onto an imported memory object as shown below. The offset, dimensions, format and number of mip levels must match that specified when creating the mapping using the corresponding Direct3D 11 API. Additionally, if the mipmapped array can be bound as a render target in Direct3D 12, the flag `cudaArrayColorAttachment` must be set. All mapped mipmapped arrays must be freed using `cudaFreeMipmappedArray()`. The following code sample shows how to convert Direct3D 11 parameters into the corresponding CUDA parameters when mapping mipmapped arrays onto imported memory objects.

```
cudaMipmappedArray_t mapMipmappedArrayOntoExternalMemory(cudaExternalMemory_t extMem,
↪unsigned long long offset, cudaChannelFormatDesc *formatDesc, cudaExtent *extent,
↪unsigned int flags, unsigned int numLevels) {
    cudaMipmappedArray_t mipmap = NULL;
    cudaExternalMemoryMipmappedArrayDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.offset = offset;
    desc.formatDesc = *formatDesc;
    desc.extent = *extent;
    desc.flags = flags;
    desc.numLevels = numLevels;

    // Note: 'mipmap' must eventually be freed using cudaFreeMipmappedArray()
    cudaExternalMemoryGetMappedMipmappedArray(&mipmap, extMem, &desc);

    return mipmap;
}

cudaChannelFormatDesc getCudaChannelFormatDescForDxgiFormat(DXGI_FORMAT dxgiFormat)
{
    cudaChannelFormatDesc d;
```

(continues on next page)

(continued from previous page)

```
    memset(&d, 0, sizeof(d));
    switch (dxgiFormat) {
    case DXGI_FORMAT_R8_UINT:           d.x = 8;  d.y = 0;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R8_SINT:           d.x = 8;  d.y = 0;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R8G8_UINT:         d.x = 8;  d.y = 8;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R8G8_SINT:         d.x = 8;  d.y = 8;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R8G8B8A8_UINT:     d.x = 8;  d.y = 8;  d.z = 8;  d.w = 8;  d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R8G8B8A8_SINT:     d.x = 8;  d.y = 8;  d.z = 8;  d.w = 8;  d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R16_UINT:          d.x = 16; d.y = 0;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R16_SINT:          d.x = 16; d.y = 0;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R16G16_UINT:       d.x = 16; d.y = 16; d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R16G16_SINT:       d.x = 16; d.y = 16; d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R16G16B16A16_UINT: d.x = 16; d.y = 16; d.z = 16; d.w = 16; d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R16G16B16A16_SINT: d.x = 16; d.y = 16; d.z = 16; d.w = 16; d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R32_UINT:          d.x = 32; d.y = 0;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R32_SINT:          d.x = 32; d.y = 0;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R32_FLOAT:         d.x = 32; d.y = 0;  d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindFloat;    break;
    case DXGI_FORMAT_R32G32_UINT:       d.x = 32; d.y = 32; d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R32G32_SINT:       d.x = 32; d.y = 32; d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R32G32_FLOAT:      d.x = 32; d.y = 32; d.z = 0;  d.w = 0;  d.f
↪= cudaChannelFormatKindFloat;    break;
    case DXGI_FORMAT_R32G32B32A32_UINT: d.x = 32; d.y = 32; d.z = 32; d.w = 32; d.f
↪= cudaChannelFormatKindUnsigned; break;
    case DXGI_FORMAT_R32G32B32A32_SINT: d.x = 32; d.y = 32; d.z = 32; d.w = 32; d.f
↪= cudaChannelFormatKindSigned;   break;
    case DXGI_FORMAT_R32G32B32A32_FLOAT: d.x = 32; d.y = 32; d.z = 32; d.w = 32; d.f
↪= cudaChannelFormatKindFloat;    break;
    default: assert(0);
    }
```

```
    return d;
}

cudaExtent getCudaExtentForD3D11Extent(UINT64 width, UINT height, UINT16
↪depthOrArraySize, D3D12_SRV_DIMENSION d3d11SRVDimension) {
    cudaExtent e = { 0, 0, 0 };

    switch (d3d11SRVDimension) {
    case D3D11_SRV_DIMENSION_TEXTURE1D:        e.width = width; e.height = 0;      e.
↪depth = 0;                 break;
```

(continues on next page)

(continued from previous page)

```
    case D3D11_SRV_DIMENSION_TEXTURE2D:        e.width = width; e.height = height; e.
↪depth = 0;                 break;
    case D3D11_SRV_DIMENSION_TEXTURE3D:        e.width = width; e.height = height; e.
↪depth = depthOrArraySize; break;
    case D3D11_SRV_DIMENSION_TEXTURECUBE:      e.width = width; e.height = height; e.
↪depth = depthOrArraySize; break;
    case D3D11_SRV_DIMENSION_TEXTURE1DARRAY:   e.width = width; e.height = 0;      e.
↪depth = depthOrArraySize; break;
    case D3D11_SRV_DIMENSION_TEXTURE2DARRAY:   e.width = width; e.height = height; e.
↪depth = depthOrArraySize; break;
    case D3D11_SRV_DIMENSION_TEXTURECUBEARRAY: e.width = width; e.height = height; e.
↪depth = depthOrArraySize; break;
    default: assert(0);
    }
    return e;
}

unsigned int getCudaMipmappedArrayFlagsForD3D12Resource(D3D11_SRV_DIMENSION
↪d3d11SRVDimension, D3D11_BIND_FLAG d3d11BindFlags, bool allowSurfaceLoadStore) {
    unsigned int flags = 0;

    switch (d3d11SRVDimension) {
    case D3D11_SRV_DIMENSION_TEXTURECUBE:      flags |= cudaArrayCubemap;
↪       break;
    case D3D11_SRV_DIMENSION_TEXTURECUBEARRAY: flags |= cudaArrayCubemap |
↪cudaArrayLayered; break;
    case D3D11_SRV_DIMENSION_TEXTURE1DARRAY:   flags |= cudaArrayLayered;
↪       break;
    case D3D11_SRV_DIMENSION_TEXTURE2DARRAY:   flags |= cudaArrayLayered;
↪       break;
    default: break;
    }

    if (d3d11BindFlags & D3D11_BIND_RENDER_TARGET) {
        flags |= cudaArrayColorAttachment;
    }

    if (allowSurfaceLoadStore) {
        flags |= cudaArraySurfaceLoadStore;
    }

    return flags;
}
```

6.2.16.4.5 Importing Synchronization Objects

A shareable Direct3D 11 fence object, created by setting the flag `D3D11_FENCE_FLAG_SHARED` in the call to `ID3D11Device5::CreateFence`, can be imported into CUDA using the NT handle associated with that object as shown below. Note that it is the application's responsibility to close the handle when it is not required anymore. The NT handle holds a reference to the resource, so it must be explicitly freed before the underlying semaphore can be freed.

```
cudaExternalSemaphore_t importD3D11FenceFromNTHandle(HANDLE handle) {
    cudaExternalSemaphore_t extSem = NULL;
```

(continues on next page)

(continued from previous page)

```
    cudaExternalSemaphoreHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalSemaphoreHandleTypeD3D11Fence;
    desc.handle.win32.handle = handle;

    cudaImportExternalSemaphore(&extSem, &desc);

    // Input parameter 'handle' should be closed if it's not needed anymore
    CloseHandle(handle);

    return extSem;
}
```

A shareable Direct3D 11 fence object can also be imported using a named handle if one exists as shown below.

```
cudaExternalSemaphore_t importD3D11FenceFromNamedNTHandle(LPCWSTR name) {
    cudaExternalSemaphore_t extSem = NULL;
    cudaExternalSemaphoreHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalSemaphoreHandleTypeD3D11Fence;
    desc.handle.win32.name = (void *)name;

    cudaImportExternalSemaphore(&extSem, &desc);

    return extSem;
}
```

A shareable Direct3D 11 keyed mutex object associated with a shareable Direct3D 11 resource, viz, `IDXGIKeyedMutex`, created by setting the flag `D3D11_RESOURCE_MISC_SHARED_KEYEDMUTEX`, can be imported into CUDA using the NT handle associated with that object as shown below. Note that it is the application's responsibility to close the handle when it is not required anymore. The NT handle holds a reference to the resource, so it must be explicitly freed before the underlying semaphore can be freed.

```
cudaExternalSemaphore_t importD3D11KeyedMutexFromNTHandle(HANDLE handle) {
    cudaExternalSemaphore_t extSem = NULL;
    cudaExternalSemaphoreHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalSemaphoreHandleTypeKeyedMutex;
    desc.handle.win32.handle = handle;

    cudaImportExternalSemaphore(&extSem, &desc);

    // Input parameter 'handle' should be closed if it's not needed anymore
    CloseHandle(handle);

    return extSem;
}
```

A shareable Direct3D 11 keyed mutex object can also be imported using a named handle if one exists

as shown below.

```
cudaExternalSemaphore_t importD3D11KeyedMutexFromNamedNTHandle(LPCWSTR name) {
    cudaExternalSemaphore_t extSem = NULL;
    cudaExternalSemaphoreHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalSemaphoreHandleTypeKeyedMutex;
    desc.handle.win32.name = (void *)name;

    cudaImportExternalSemaphore(&extSem, &desc);

    return extSem;
}
```

A shareable Direct3D 11 keyed mutex object can be imported into CUDA using the globally shared D3DKMT handle associated with that object as shown below. Since a globally shared D3DKMT handle does not hold a reference to the underlying memory it is automatically destroyed when all other references to the resource are destroyed.

```
cudaExternalSemaphore_t importD3D11FenceFromKMTHandle(HANDLE handle) {
    cudaExternalSemaphore_t extSem = NULL;
    cudaExternalSemaphoreHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalSemaphoreHandleTypeKeyedMutexKmt;
    desc.handle.win32.handle = handle;

    cudaImportExternalSemaphore(&extSem, &desc);

    // Input parameter 'handle' should be closed if it's not needed anymore
    CloseHandle(handle);

    return extSem;
}
```

6.2.16.4.6 Signaling/Waiting on Imported Synchronization Objects

An imported Direct3D 11 fence object can be signaled as shown below. Signaling such a fence object sets its value to the one specified. The corresponding wait that waits on this signal must be issued in Direct3D 11. Additionally, the wait that waits on this signal must be issued after this signal has been issued.

```
void signalExternalSemaphore(cudaExternalSemaphore_t extSem, unsigned long long value,
↪ cudaStream_t stream) {
    cudaExternalSemaphoreSignalParams params = {};

    memset(&params, 0, sizeof(params));

    params.params.fence.value = value;

    cudaSignalExternalSemaphoresAsync(&extSem, &params, 1, stream);
}
```

An imported Direct3D 11 fence object can be waited on as shown below. Waiting on such a fence object waits until its value becomes greater than or equal to the specified value. The corresponding signal that this wait is waiting on must be issued in Direct3D 11. Additionally, the signal must be issued before this wait can be issued.

```
void waitExternalSemaphore(cudaExternalSemaphore_t extSem, unsigned long long value,
↪cudaStream_t stream) {
    cudaExternalSemaphoreWaitParams params = {};

    memset(&params, 0, sizeof(params));

    params.params.fence.value = value;

    cudaWaitExternalSemaphoresAsync(&extSem, &params, 1, stream);
}
```

An imported Direct3D 11 keyed mutex object can be signaled as shown below. Signaling such a keyed mutex object by specifying a key value releases the keyed mutex for that value. The corresponding wait that waits on this signal must be issued in Direct3D 11 with the same key value. Additionally, the Direct3D 11 wait must be issued after this signal has been issued.

```
void signalExternalSemaphore(cudaExternalSemaphore_t extSem, unsigned long long key,
↪cudaStream_t stream) {
    cudaExternalSemaphoreSignalParams params = {};

    memset(&params, 0, sizeof(params));

    params.params.keyedmutex.key = key;

    cudaSignalExternalSemaphoresAsync(&extSem, &params, 1, stream);
}
```

An imported Direct3D 11 keyed mutex object can be waited on as shown below. A timeout value in milliseconds is needed when waiting on such a keyed mutex. The wait operation waits until the keyed mutex value is equal to the specified key value or until the timeout has elapsed. The timeout interval can also be an infinite value. In case an infinite value is specified the timeout never elapses. The windows INFINITE macro must be used to specify an infinite timeout. The corresponding signal that this wait is waiting on must be issued in Direct3D 11. Additionally, the Direct3D 11 signal must be issued before this wait can be issued.

```
void waitExternalSemaphore(cudaExternalSemaphore_t extSem, unsigned long long key,
↪unsigned int timeoutMs, cudaStream_t stream) {
    cudaExternalSemaphoreWaitParams params = {};

    memset(&params, 0, sizeof(params));

    params.params.keyedmutex.key = key;
    params.params.keyedmutex.timeoutMs = timeoutMs;

    cudaWaitExternalSemaphoresAsync(&extSem, &params, 1, stream);
}
```

6.2.16.5 NVIDIA Software Communication Interface Interoperability (NVSCI)

NvSciBuf and NvSciSync are interfaces developed for serving the following purposes:

- NvSciBuf: Allows applications to allocate and exchange buffers in memory
- NvSciSync: Allows applications to manage synchronization objects at operation boundaries

More details on these interfaces are available at: https://docs.nvidia.com/drive.

6.2.16.5.1 Importing Memory Objects

For allocating an NvSciBuf object compatible with a given CUDA device, the corresponding GPU id must be set with `NvSciBufGeneralAttrKey_GpuId` in the NvSciBuf attribute list as shown below. Optionally, applications can specify the following attributes -

- `NvSciBufGeneralAttrKey_NeedCpuAccess`: Specifies if CPU access is required for the buffer
- `NvSciBufRawBufferAttrKey_Align`: Specifies the alignment requirement of `NvSciBufType_RawBuffer`
- `NvSciBufGeneralAttrKey_RequiredPerm`: Different access permissions can be configured for different UMDs per NvSciBuf memory object instance. For example, to provide the GPU with read-only access permissions to the buffer, create a duplicate NvSciBuf object using `NvSciBufObjDupWithReducePerm()` with `NvSciBufAccessPerm_Readonly` as the input parameter. Then import this newly created duplicate object with reduced permission into CUDA as shown
- `NvSciBufGeneralAttrKey_EnableGpuCache`: To control GPU L2 cacheability
- `NvSciBufGeneralAttrKey_EnableGpuCompression`: To specify GPU compression

Note: For more details on these attributes and their valid input options, refer to NvSciBuf Documentation.

The following code snippet illustrates their sample usage.

```
NvSciBufObj createNvSciBufObject() {
   // Raw Buffer Attributes for CUDA
    NvSciBufType bufType = NvSciBufType_RawBuffer;
    uint64_t rawsize = SIZE;
    uint64_t align = 0;
    bool cpuaccess_flag = true;
    NvSciBufAttrValAccessPerm perm = NvSciBufAccessPerm_ReadWrite;

    NvSciRmGpuId gpuid[] ={};
    CUuuid uuid;
    cuDeviceGetUuid(&uuid, dev));

    memcpy(&gpuid[0].bytes, &uuid.bytes, sizeof(uuid.bytes));
    // Disable cache on dev
    NvSciBufAttrValGpuCache gpuCache[] = {{gpuid[0], false}};
    NvSciBufAttrValGpuCompression gpuCompression[] = {{gpuid[0],
↪NvSciBufCompressionType_GenericCompressible}};
    // Fill in values
    NvSciBufAttrKeyValuePair rawbuffattrs[] = {
         { NvSciBufGeneralAttrKey_Types, &bufType, sizeof(bufType) },
         { NvSciBufRawBufferAttrKey_Size, &rawsize, sizeof(rawsize) },
```

(continues on next page)

(continued from previous page)

```
        { NvSciBufRawBufferAttrKey_Align, &align, sizeof(align) },
        { NvSciBufGeneralAttrKey_NeedCpuAccess, &cpuaccess_flag, sizeof(cpuaccess_
↪flag) },
        { NvSciBufGeneralAttrKey_RequiredPerm, &perm, sizeof(perm) },
        { NvSciBufGeneralAttrKey_GpuId, &gpuid, sizeof(gpuid) },
        { NvSciBufGeneralAttrKey_EnableGpuCache &gpuCache, sizeof(gpuCache) },
        { NvSciBufGeneralAttrKey_EnableGpuCompression &gpuCompression,
↪sizeof(gpuCompression) }
    };

    // Create list by setting attributes
    err = NvSciBufAttrListSetAttrs(attrListBuffer, rawbuffattrs,
            sizeof(rawbuffattrs)/sizeof(NvSciBufAttrKeyValuePair));

    NvSciBufAttrListCreate(NvSciBufModule, &attrListBuffer);

    // Reconcile And Allocate
    NvSciBufAttrListReconcile(&attrListBuffer, 1, &attrListReconciledBuffer,
                        &attrListConflictBuffer)
    NvSciBufObjAlloc(attrListReconciledBuffer, &bufferObjRaw);
    return bufferObjRaw;
}
```

```
NvSciBufObj bufferObjRo; // Readonly NvSciBuf memory obj
// Create a duplicate handle to the same memory buffer with reduced permissions
NvSciBufObjDupWithReducePerm(bufferObjRaw, NvSciBufAccessPerm_Readonly, &bufferObjRo);
return bufferObjRo;
```

The allocated NvSciBuf memory object can be imported in CUDA using the NvSciBufObj handle as shown below. Application should query the allocated NvSciBufObj for attributes required for filling CUDA External Memory Descriptor. Note that the attribute list and NvSciBuf objects should be maintained by the application. If the NvSciBuf object imported into CUDA is also mapped by other drivers, then based on `NvSciBufGeneralAttrKey_GpuSwNeedCacheCoherency` output attribute value the application must use NvSciSync objects (refer to *Importing Synchronization Objects*) as appropriate barriers to maintain coherence between CUDA and the other drivers.

Note: For more details on how to allocate and maintain NvSciBuf objects refer to NvSciBuf API Documentation.

```
cudaExternalMemory_t importNvSciBufObject (NvSciBufObj bufferObjRaw) {

    /*************** Query NvSciBuf Object **************/
    NvSciBufAttrKeyValuePair bufattrs[] = {
                { NvSciBufRawBufferAttrKey_Size, NULL, 0 },
                { NvSciBufGeneralAttrKey_GpuSwNeedCacheCoherency, NULL, 0 },
                { NvSciBufGeneralAttrKey_EnableGpuCompression, NULL, 0 }
    };
    NvSciBufAttrListGetAttrs(retList, bufattrs,
        sizeof(bufattrs)/sizeof(NvSciBufAttrKeyValuePair)));
                ret_size = *(static_cast<const uint64_t*>(bufattrs[0].value));

    // Note cache and compression are per GPU attributes, so read values for specific
↪gpu by comparing UUID
    // Read cacheability granted by NvSciBuf
```

(continues on next page)

(continued from previous page)

```
    int numGpus = bufattrs[1].len / sizeof(NvSciBufAttrValGpuCache);
    NvSciBufAttrValGpuCache[] cacheVal = (NvSciBufAttrValGpuCache *)bufattrs[1].value;
    bool ret_cacheVal;
    for (int i = 0; i < numGpus; i++) {
        if (memcmp(gpuid[0].bytes, cacheVal[i].gpuId.bytes, sizeof(CUuuid)) == 0) {
            ret_cacheVal = cacheVal[i].cacheability);
        }
    }

    // Read compression granted by NvSciBuf
    numGpus = bufattrs[2].len / sizeof(NvSciBufAttrValGpuCompression);
    NvSciBufAttrValGpuCompression[] compVal = (NvSciBufAttrValGpuCompression
↪*)bufattrs[2].value;
    NvSciBufCompressionType ret_compVal;
    for (int i = 0; i < numGpus; i++) {
        if (memcmp(gpuid[0].bytes, compVal[i].gpuId.bytes, sizeof(CUuuid)) == 0) {
            ret_compVal = compVal[i].compressionType);
        }
    }

    /*************** NvSciBuf Registration With CUDA **************/

    // Fill up CUDA_EXTERNAL_MEMORY_HANDLE_DESC
    cudaExternalMemoryHandleDesc memHandleDesc;
    memset(&memHandleDesc, 0, sizeof(memHandleDesc));
    memHandleDesc.type = cudaExternalMemoryHandleTypeNvSciBuf;
    memHandleDesc.handle.nvSciBufObject = bufferObjRaw;
    // Set the NvSciBuf object with required access permissions in this step
    memHandleDesc.handle.nvSciBufObject = bufferObjRo;
    memHandleDesc.size = ret_size;
    cudaImportExternalMemory(&extMemBuffer, &memHandleDesc);
    return extMemBuffer;
 }
```

6.2.16.5.2 Mapping Buffers onto Imported Memory Objects

A device pointer can be mapped onto an imported memory object as shown below. The offset and size of the mapping can be filled as per the attributes of the allocated `NvSciBufObj`. All mapped device pointers must be freed using `cudaFree()`.

```
void * mapBufferOntoExternalMemory(cudaExternalMemory_t extMem, unsigned long long
↪offset, unsigned long long size) {
    void *ptr = NULL;
    cudaExternalMemoryBufferDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.offset = offset;
    desc.size = size;

    cudaExternalMemoryGetMappedBuffer(&ptr, extMem, &desc);

    // Note: 'ptr' must eventually be freed using cudaFree()
    return ptr;
}
```

6.2.16.5.3 Mapping Mipmapped Arrays onto Imported Memory Objects

A CUDA mipmapped array can be mapped onto an imported memory object as shown below. The offset, dimensions and format can be filled as per the attributes of the allocated `NvSciBufObj`. All mapped mipmapped arrays must be freed using `cudaFreeMipmappedArray()`. The following code sample shows how to convert NvSciBuf attributes into the corresponding CUDA parameters when mapping mipmapped arrays onto imported memory objects.

Note: The number of mip levels must be 1.

```
cudaMipmappedArray_t mapMipmappedArrayOntoExternalMemory(cudaExternalMemory_t extMem,
↪unsigned long long offset, cudaChannelFormatDesc *formatDesc, cudaExtent *extent,
↪unsigned int flags, unsigned int numLevels) {
    cudaMipmappedArray_t mipmap = NULL;
    cudaExternalMemoryMipmappedArrayDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.offset = offset;
    desc.formatDesc = *formatDesc;
    desc.extent = *extent;
    desc.flags = flags;
    desc.numLevels = numLevels;

    // Note: 'mipmap' must eventually be freed using cudaFreeMipmappedArray()
    cudaExternalMemoryGetMappedMipmappedArray(&mipmap, extMem, &desc);

    return mipmap;
}
```

6.2.16.5.4 Importing Synchronization Objects

NvSciSync attributes that are compatible with a given CUDA device can be generated using `cudaDeviceGetNvSciSyncAttributes()`. The returned attribute list can be used to create a `NvSciSyncObj` that is guaranteed compatibility with a given CUDA device.

```
NvSciSyncObj createNvSciSyncObject() {
    NvSciSyncObj nvSciSyncObj
    int cudaDev0 = 0;
    int cudaDev1 = 1;
    NvSciSyncAttrList signalerAttrList = NULL;
    NvSciSyncAttrList waiterAttrList = NULL;
    NvSciSyncAttrList reconciledList = NULL;
    NvSciSyncAttrList newConflictList = NULL;

    NvSciSyncAttrListCreate(module, &signalerAttrList);
    NvSciSyncAttrListCreate(module, &waiterAttrList);
    NvSciSyncAttrList unreconciledList[2] = {NULL, NULL};
    unreconciledList[0] = signalerAttrList;
    unreconciledList[1] = waiterAttrList;

    cudaDeviceGetNvSciSyncAttributes(signalerAttrList, cudaDev0, CUDA_NVSCISYNC_ATTR_
↪SIGNAL);
```

(continues on next page)

(continued from previous page)

```
    cudaDeviceGetNvSciSyncAttributes(waiterAttrList, cudaDev1, CUDA_NVSCISYNC_ATTR_
↪WAIT);

    NvSciSyncAttrListReconcile(unreconciledList, 2, &reconciledList, &
↪newConflictList);

    NvSciSyncObjAlloc(reconciledList, &nvSciSyncObj);

    return nvSciSyncObj;
}
```

An NvSciSync object (created as above) can be imported into CUDA using the NvSciSyncObj handle as shown below. Note that ownership of the NvSciSyncObj handle continues to lie with the application even after it is imported.

```
cudaExternalSemaphore_t importNvSciSyncObject(void* nvSciSyncObj) {
    cudaExternalSemaphore_t extSem = NULL;
    cudaExternalSemaphoreHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalSemaphoreHandleTypeNvSciSync;
    desc.handle.nvSciSyncObj = nvSciSyncObj;

    cudaImportExternalSemaphore(&extSem, &desc);

    // Deleting/Freeing the nvSciSyncObj beyond this point will lead to undefined
↪behavior in CUDA

    return extSem;
}
```

6.2.16.5.5 Signaling/Waiting on Imported Synchronization Objects

An imported `NvSciSyncObj` object can be signaled as outlined below. Signaling NvSciSync backed semaphore object initializes the *fence* parameter passed as input. This fence parameter is waited upon by a wait operation that corresponds to the aforementioned signal. Additionally, the wait that waits on this signal must be issued after this signal has been issued. If the flags are set to `cudaExternalSemaphoreSignalSkipNvSciBufMemSync` then memory synchronization operations (over all the imported NvSciBuf in this process) that are executed as a part of the signal operation by default are skipped. When `NvsciBufGeneralAttrKey_GpuSwNeedCacheCoherency` is FALSE, this flag should be set.

```
void signalExternalSemaphore(cudaExternalSemaphore_t extSem, cudaStream_t stream,
↪void *fence) {
    cudaExternalSemaphoreSignalParams signalParams = {};

    memset(&signalParams, 0, sizeof(signalParams));

    signalParams.params.nvSciSync.fence = (void*)fence;
    signalParams.flags = 0; //OR cudaExternalSemaphoreSignalSkipNvSciBufMemSync

    cudaSignalExternalSemaphoresAsync(&extSem, &signalParams, 1, stream);
```

(continues on next page)

(continued from previous page)

```
}
```

An imported `NvSciSyncObj` object can be waited upon as outlined below. Waiting on NvSciSync backed semaphore object waits until the input *fence* parameter is signaled by the corresponding signaler. Additionally, the signal must be issued before the wait can be issued. If the flags are set to `cudaExternalSemaphoreWaitSkipNvSciBufMemSync` then memory synchronization operations (over all the imported NvSciBuf in this process) that are executed as a part of the signal operation by default are skipped. When `NvsciBufGeneralAttrKey_GpuSwNeedCacheCoherency` is FALSE, this flag should be set.

```
void waitExternalSemaphore(cudaExternalSemaphore_t extSem, cudaStream_t stream, void
↪*fence) {
     cudaExternalSemaphoreWaitParams waitParams = {};

    memset(&waitParams, 0, sizeof(waitParams));

    waitParams.params.nvSciSync.fence = (void*)fence;
    waitParams.flags = 0; //OR cudaExternalSemaphoreWaitSkipNvSciBufMemSync

    cudaWaitExternalSemaphoresAsync(&extSem, &waitParams, 1, stream);
}
```

6.3. Versioning and Compatibility

There are two version numbers that developers should care about when developing a CUDA application: The compute capability that describes the general specifications and features of the compute device (see *Compute Capability*) and the version of the CUDA driver API that describes the features supported by the driver API and runtime.

The version of the driver API is defined in the driver header file as `CUDA_VERSION`. It allows developers to check whether their application requires a newer device driver than the one currently installed. This is important, because the driver API is *backward compatible*, meaning that applications, plug-ins, and libraries (including the CUDA runtime) compiled against a particular version of the driver API will continue to work on subsequent device driver releases as illustrated in Figure 26. The driver API is not *forward compatible*, which means that applications, plug-ins, and libraries (including the CUDA runtime) compiled against a particular version of the driver API will not work on previous versions of the device driver.

It is important to note that there are limitations on the mixing and matching of versions that is supported:

- Since only one version of the CUDA Driver can be installed at a time on a system, the installed driver must be of the same or higher version than the maximum Driver API version against which any application, plug-ins, or libraries that must run on that system were built.
- All plug-ins and libraries used by an application must use the same version of the CUDA Runtime unless they statically link to the Runtime, in which case multiple versions of the runtime can coexist in the same process space. Note that if `nvcc` is used to link the application, the static version of the CUDA Runtime library will be used by default, and all CUDA Toolkit libraries are statically linked against the CUDA Runtime.

- All plug-ins and libraries used by an application must use the same version of any libraries that use the runtime (such as cuFFT, cuBLAS, ...) unless statically linking to those libraries.

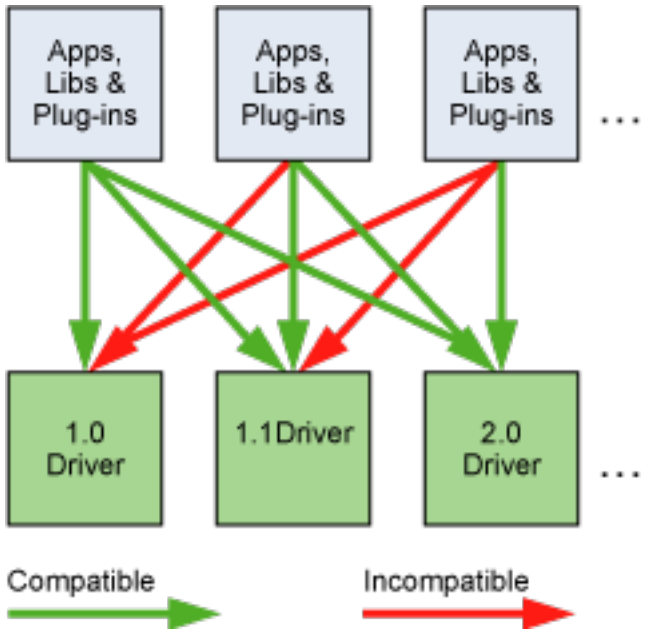


Figure 26: The Driver API Is Backward but Not Forward Compatible

For Tesla GPU products, CUDA 10 introduced a new forward-compatible upgrade path for the user-mode components of the CUDA Driver. This feature is described in CUDA Compatibility. The requirements on the CUDA Driver version described here apply to the version of the user-mode components.

6.4. Compute Modes

On Tesla solutions running Windows Server 2008 and later or Linux, one can set any device in a system in one of the three following modes using NVIDIA's System Management Interface (nvidia-smi), which is a tool distributed as part of the driver:

- *Default* compute mode: Multiple host threads can use the device (by calling `cudaSetDevice()` on this device, when using the runtime API, or by making current a context associated to the device, when using the driver API) at the same time.
- *Exclusive-process* compute mode: Only one CUDA context may be created on the device across all processes in the system. The context may be current to as many threads as desired within the process that created that context.
- *Prohibited* compute mode: No CUDA context can be created on the device.

This means, in particular, that a host thread using the runtime API without explicitly calling `cudaSetDevice()` might be associated with a device other than device 0 if device 0 turns out to be in prohibited mode or in exclusive-process mode and used by another process. `cudaSetValidDevices()` can be used to set a device from a prioritized list of devices.

Note also that, for devices featuring the Pascal architecture onwards (compute capability with major revision number 6 and higher), there exists support for Compute Preemption. This allows compute tasks to be preempted at instruction-level granularity, rather than thread block granularity as in prior

Maxwell and Kepler GPU architecture, with the benefit that applications with long-running kernels can be prevented from either monopolizing the system or timing out. However, there will be context switch overheads associated with Compute Preemption, which is automatically enabled on those devices for which support exists. The individual attribute query function `cudaDeviceGetAttribute()` with the attribute `cudaDevAttrComputePreemptionSupported` can be used to determine if the device in use supports Compute Preemption. Users wishing to avoid context switch overheads associated with different processes can ensure that only one process is active on the GPU by selecting exclusive-process mode.

Applications may query the compute mode of a device by checking the attribute `cudaDevAttrComputeMode`.

6.5. Mode Switches

GPUs that have a display output dedicate some DRAM memory to the so-called *primary surface*, which is used to refresh the display device whose output is viewed by the user. When users initiate a *mode switch* of the display by changing the resolution or bit depth of the display (using NVIDIA control panel or the Display control panel on Windows), the amount of memory needed for the primary surface changes. For example, if the user changes the display resolution from 1280x1024x32-bit to 1600x1200x32-bit, the system must dedicate 7.68 MB to the primary surface rather than 5.24 MB. (Full-screen graphics applications running with anti-aliasing enabled may require much more display memory for the primary surface.) On Windows, other events that may initiate display mode switches include launching a full-screen DirectX application, hitting Alt+Tab to task switch away from a full-screen DirectX application, or hitting Ctrl+Alt+Del to lock the computer.

If a mode switch increases the amount of memory needed for the primary surface, the system may have to cannibalize memory allocations dedicated to CUDA applications. Therefore, a mode switch results in any call to the CUDA runtime to fail and return an invalid context error.

6.6. Tesla Compute Cluster Mode for Windows

Using NVIDIA's System Management Interface (*nvidia-smi*), the Windows device driver can be put in TCC (Tesla Compute Cluster) mode for devices of the Tesla and Quadro Series.

TCC mode removes support for any graphics functionality.

Chapter 7. Hardware Implementation

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

The NVIDIA GPU architecture is built around a scalable array of multithreaded *Streaming Multiprocessors* (*SMs*). When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors.

A multiprocessor is designed to execute hundreds of threads concurrently. To manage such a large number of threads, it employs a unique architecture called *SIMT* (*Single-Instruction, Multiple-Thread*) that is described in *SIMT Architecture*. The instructions are pipelined, leveraging instruction-level parallelism within a single thread, as well as extensive thread-level parallelism through simultaneous hardware multithreading as detailed in *Hardware Multithreading*. Unlike CPU cores, they are issued in order and there is no branch prediction or speculative execution.

SIMT Architecture and *Hardware Multithreading* describe the architecture features of the streaming multiprocessor that are common to all devices. *Compute Capability 5.x*, *Compute Capability 6.x*, and *Compute Capability 7.x* provide the specifics for devices of compute capabilities 5.x, 6.x, and 7.x respectively.

The NVIDIA GPU architecture uses a little-endian representation.

7.1. SIMT Architecture

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. The term *warp* originates from weaving, the first parallel thread technology. A *half-warp* is either the first or second half of a warp. A *quarter-warp* is either the first, second, third, or fourth quarter of a warp.

When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps and each warp gets scheduled by a *warp scheduler* for execution. The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0. *Thread Hierarchy* describes how thread IDs relate to thread indices in the block.

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp executes each branch path taken, disabling threads that are not on that path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths.

The SIMT architecture is akin to SIMD (Single Instruction, Multiple Data) vector organizations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. For the purposes of correctness, the programmer can essentially ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional code: Cache line size can be safely ignored when designing for correctness but must be considered in the code structure when designing for peak performance. Vector architectures, on the other hand, require the software to coalesce loads into vectors and manage divergence manually.

Prior to NVIDIA Volta, warps used a single program counter shared amongst all 32 threads in the warp together with an active mask specifying the active threads of the warp. As a result, threads from the same warp in divergent regions or different states of execution cannot signal each other or exchange data, and algorithms requiring fine-grained sharing of data guarded by locks or mutexes can easily lead to deadlock, depending on which warp the contending threads come from.

Starting with the NVIDIA Volta architecture, *Independent Thread Scheduling* allows full concurrency between threads, regardless of warp. With Independent Thread Scheduling, the GPU maintains execution state per thread, including a program counter and call stack, and can yield execution at a per-thread granularity, either to make better use of execution resources or to allow one thread to wait for data to be produced by another. A schedule optimizer determines how to group active threads from the same warp together into SIMT units. This retains the high throughput of SIMT execution as in prior NVIDIA GPUs, but with much more flexibility: threads can now diverge and reconverge at sub-warp granularity.

Independent Thread Scheduling can lead to a rather different set of threads participating in the executed code than intended if the developer made assumptions about warp-synchronicity[2] of previous hardware architectures. In particular, any warp-synchronous code (such as synchronization-free, intra-warp reductions) should be revisited to ensure compatibility with NVIDIA Volta and beyond. See *Compute Capability 7.x* for further details.

Note: The threads of a warp that are participating in the current instruction are called the *active* threads, whereas threads not on the current instruction are *inactive* (disabled). Threads can be inactive for a variety of reasons including having exited earlier than other threads of their warp, having taken a different branch path than the branch path currently executed by the warp, or being the last threads of a block whose number of threads is not a multiple of the warp size.

If a non-atomic instruction executed by a warp writes to the same location in global or shared memory for more than one of the threads of the warp, the number of serialized writes that occur to that location varies depending on the compute capability of the device (see *Compute Capability 5.x*, *Compute Capability 6.x*, and *Compute Capability 7.x*), and which thread performs the final write is undefined.

If an *atomic* instruction executed by a warp reads, modifies, and writes to the same location in global memory for more than one of the threads of the warp, each read/modify/write to that location occurs and they are all serialized, but the order in which they occur is undefined.

---

[2] The term *warp-synchronous* refers to code that implicitly assumes threads in the same warp are synchronized at every instruction.

7.2. Hardware Multithreading

The execution context (program counters, registers, and so on) for each warp processed by a multiprocessor is maintained on-chip during the entire lifetime of the warp. Therefore, switching from one execution context to another has no cost, and at every instruction issue time, a warp scheduler selects a warp that has threads ready to execute its next instruction (the *active threads* of the warp) and issues the instruction to those threads.

In particular, each multiprocessor has a set of 32-bit registers that are partitioned among the warps, and a *parallel data cache* or *shared memory* that is partitioned among the thread blocks.

The number of blocks and warps that can reside and be processed together on the multiprocessor for a given kernel depends on the amount of registers and shared memory used by the kernel and the amount of registers and shared memory available on the multiprocessor. There are also a maximum number of resident blocks and a maximum number of resident warps per multiprocessor. These limits as well the amount of registers and shared memory available on the multiprocessor are a function of the compute capability of the device and are given in *Compute Capabilities*. If there are not enough registers or shared memory available per multiprocessor to process at least one block, the kernel will fail to launch.

The total number of warps in a block is as follows:

ceil $\left(\frac{T}{W_{size}}, 1\right)$

- T is the number of threads per block,
- *Wsize* is the warp size, which is equal to 32,
- ceil(x, y) is equal to x rounded up to the nearest multiple of y.

The total number of registers and total amount of shared memory allocated for a block are documented in the CUDA Occupancy Calculator provided in the CUDA Toolkit.

Chapter 8. Performance Guidelines

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

8.1. Overall Performance Optimization Strategies

Performance optimization revolves around four basic strategies:

- Maximize parallel execution to achieve maximum utilization;
- Optimize memory usage to achieve maximum memory throughput;
- Optimize instruction usage to achieve maximum instruction throughput;
- Minimize memory thrashing.

Which strategies will yield the best performance gain for a particular portion of an application depends on the performance limiters for that portion; optimizing instruction usage of a kernel that is mostly limited by memory accesses will not yield any significant performance gain, for example. Optimization efforts should therefore be constantly directed by measuring and monitoring the performance limiters, for example using the CUDA profiler. Also, comparing the floating-point operation throughput or memory throughput—whichever makes more sense—of a particular kernel to the corresponding peak theoretical throughput of the device indicates how much room for improvement there is for the kernel.

8.2. Maximize Utilization

To maximize utilization the application should be structured in a way that it exposes as much parallelism as possible and efficiently maps this parallelism to the various components of the system to keep them busy most of the time.

8.2.1. Application Level

At a high level, the application should maximize parallel execution between the host, the devices, and the bus connecting the host to the devices, by using asynchronous functions calls and streams as described in *Asynchronous Concurrent Execution*. It should assign to each processor the type of work it does best: serial workloads to the host; parallel workloads to the devices.

For the parallel workloads, at points in the algorithm where parallelism is broken because some threads need to synchronize in order to share data with each other, there are two cases: Either these threads belong to the same block, in which case they should use `__syncthreads()` and share data through shared memory within the same kernel invocation, or they belong to different blocks, in which case they must share data through global memory using two separate kernel invocations, one for writing to and one for reading from global memory. The second case is much less optimal since it adds the overhead of extra kernel invocations and global memory traffic. Its occurrence should therefore be minimized by mapping the algorithm to the CUDA programming model in such a way that the computations that require inter-thread communication are performed within a single thread block as much as possible.

8.2.2. Device Level

At a lower level, the application should maximize parallel execution between the multiprocessors of a device.

Multiple kernels can execute concurrently on a device, so maximum utilization can also be achieved by using streams to enable enough kernels to execute concurrently as described in *Asynchronous Concurrent Execution*.

8.2.3. Multiprocessor Level

At an even lower level, the application should maximize parallel execution between the various functional units within a multiprocessor.

As described in *Hardware Multithreading*, a GPU multiprocessor primarily relies on thread-level parallelism to maximize utilization of its functional units. Utilization is therefore directly linked to the number of resident warps. At every instruction issue time, a warp scheduler selects an instruction that is ready to execute. This instruction can be another independent instruction of the same warp, exploiting instruction-level parallelism, or more commonly an instruction of another warp, exploiting thread-level parallelism. If a ready to execute instruction is selected it is issued to the *active* threads of the warp. The number of clock cycles it takes for a warp to be ready to execute its next instruction is called the *latency*, and full utilization is achieved when all warp schedulers always have some instruction to issue for some warp at every clock cycle during that latency period, or in other words, when latency is completely “hidden”. The number of instructions required to hide a latency of L clock cycles depends on the respective throughputs of these instructions (see the CUDA C++ Best Practices Guide for the throughputs of various arithmetic instructions). If we assume instructions with maximum throughput, it is equal to:

- *4L* for devices of compute capability 5.x, 6.1, 6.2, 7.x and 8.x since for these devices, a multiprocessor issues one instruction per warp over one clock cycle for four warps at a time, as mentioned in *Compute Capabilities*.
- *2L* for devices of compute capability 6.0 since for these devices, the two instructions issued every cycle are one instruction for two different warps.

The most common reason a warp is not ready to execute its next instruction is that the instruction's input operands are not available yet.

If all input operands are registers, latency is caused by register dependencies, i.e., some of the input operands are written by some previous instruction(s) whose execution has not completed yet. In this case, the latency is equal to the execution time of the previous instruction and the warp schedulers must schedule instructions of other warps during that time. Execution time varies depending on the instruction. On devices of compute capability 7.x, for most arithmetic instructions, it is typically 4 clock cycles. This means that 16 active warps per multiprocessor (4 cycles, 4 warp schedulers) are required to hide arithmetic instruction latencies (assuming that warps execute instructions with maximum throughput, otherwise fewer warps are needed). If the individual warps exhibit instruction-level parallelism, i.e. have multiple independent instructions in their instruction stream, fewer warps are needed because multiple independent instructions from a single warp can be issued back to back.

If some input operand resides in off-chip memory, the latency is much higher: typically hundreds of clock cycles. The number of warps required to keep the warp schedulers busy during such high latency periods depends on the kernel code and its degree of instruction-level parallelism. In general, more warps are required if the ratio of the number of instructions with no off-chip memory operands (i.e., arithmetic instructions most of the time) to the number of instructions with off-chip memory operands is low (this ratio is commonly called the arithmetic intensity of the program).

Another reason a warp is not ready to execute its next instruction is that it is waiting at some memory fence (*Memory Fence Functions*) or synchronization point (*Synchronization Functions*). A synchronization point can force the multiprocessor to idle as more and more warps wait for other warps in the same block to complete execution of instructions prior to the synchronization point. Having multiple resident blocks per multiprocessor can help reduce idling in this case, as warps from different blocks do not need to wait for each other at synchronization points.

The number of blocks and warps residing on each multiprocessor for a given kernel call depends on the execution configuration of the call (*Execution Configuration*), the memory resources of the multiprocessor, and the resource requirements of the kernel as described in *Hardware Multithreading*. Register and shared memory usage are reported by the compiler when compiling with the `--ptxas-options=-v` option.

The total amount of shared memory required for a block is equal to the sum of the amount of statically allocated shared memory and the amount of dynamically allocated shared memory.

The number of registers used by a kernel can have a significant impact on the number of resident warps. For example, for devices of compute capability 6.x, if a kernel uses 64 registers and each block has 512 threads and requires very little shared memory, then two blocks (i.e., 32 warps) can reside on the multiprocessor since they require 2x512x64 registers, which exactly matches the number of registers available on the multiprocessor. But as soon as the kernel uses one more register, only one block (i.e., 16 warps) can be resident since two blocks would require 2x512x65 registers, which are more registers than are available on the multiprocessor. Therefore, the compiler attempts to minimize register usage while keeping register spilling (see *Device Memory Accesses*) and the number of instructions to a minimum. Register usage can be controlled using the `maxrregcount` compiler option, the `__launch_bounds__()` qualifier as described in *Launch Bounds*, or the `__maxnreg__()` qualifier as described in *Maximum Number of Registers per Thread*.

The register file is organized as 32-bit registers. So, each variable stored in a register needs at least one 32-bit register, for example, a `double` variable uses two 32-bit registers.

The effect of execution configuration on performance for a given kernel call generally depends on the kernel code. Experimentation is therefore recommended. Applications can also parametrize execution configurations based on register file size and shared memory size, which depends on the compute capability of the device, as well as on the number of multiprocessors and memory bandwidth of the device, all of which can be queried using the runtime (see reference manual).

The number of threads per block should be chosen as a multiple of the warp size to avoid wasting computing resources with under-populated warps as much as possible.

8.2.3.1 Occupancy Calculator

Several API functions exist to assist programmers in choosing thread block size and cluster size based on register and shared memory requirements.

- The occupancy calculator API, `cudaOccupancyMaxActiveBlocksPerMultiprocessor`, can provide an occupancy prediction based on the block size and shared memory usage of a kernel. This function reports occupancy in terms of the number of concurrent thread blocks per multiprocessor.
  - Note that this value can be converted to other metrics. Multiplying by the number of warps per block yields the number of concurrent warps per multiprocessor; further dividing concurrent warps by max warps per multiprocessor gives the occupancy as a percentage.
- The occupancy-based launch configurator APIs, `cudaOccupancyMaxPotentialBlockSize` and `cudaOccupancyMaxPotentialBlockSizeVariableSMem`, heuristically calculate an execution configuration that achieves the maximum multiprocessor-level occupancy.
- The occupancy calculator API, `cudaOccupancyMaxActiveClusters`, can provided occupancy prediction based on the cluster size, block size and shared memory usage of a kernel. This function reports occupancy in terms of number of max active clusters of a given size on the GPU present in the system.

The following code sample calculates the occupancy of MyKernel. It then reports the occupancy level with the ratio between concurrent warps versus maximum warps per multiprocessor.

```
// Device code
__global__ void MyKernel(int *d, int *a, int *b)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    d[idx] = a[idx] * b[idx];
}

// Host code
int main()
{
    int numBlocks;        // Occupancy in terms of active blocks
    int blockSize = 32;

    // These variables are used to convert occupancy to warps
    int device;
    cudaDeviceProp prop;
    int activeWarps;
    int maxWarps;

    cudaGetDevice(&device);
    cudaGetDeviceProperties(&prop, device);

    cudaOccupancyMaxActiveBlocksPerMultiprocessor(
        &numBlocks,
        MyKernel,
        blockSize,
        0);
```

(continues on next page)

(continued from previous page)

```
    activeWarps = numBlocks * blockSize / prop.warpSize;
    maxWarps = prop.maxThreadsPerMultiProcessor / prop.warpSize;

    std::cout << "Occupancy: " << (double)activeWarps / maxWarps * 100 << "%" <<
↪std::endl;

    return 0;
}
```

The following code sample configures an occupancy-based kernel launch of MyKernel according to the user input.

```
// Device code
__global__ void MyKernel(int *array, int arrayCount)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < arrayCount) {
        array[idx] *= array[idx];
    }
}

// Host code
int launchMyKernel(int *array, int arrayCount)
{
    int blockSize;      // The launch configurator returned block size
    int minGridSize;    // The minimum grid size needed to achieve the
                        // maximum occupancy for a full device
                        // launch
    int gridSize;       // The actual grid size needed, based on input
                        // size

    cudaOccupancyMaxPotentialBlockSize(
        &minGridSize,
        &blockSize,
        (void*)MyKernel,
        0,
        arrayCount);

    // Round up according to array size
    gridSize = (arrayCount + blockSize - 1) / blockSize;

    MyKernel<<<gridSize, blockSize>>>(array, arrayCount);
    cudaDeviceSynchronize();

    // If interested, the occupancy can be calculated with
    // cudaOccupancyMaxActiveBlocksPerMultiprocessor

    return 0;
}
```

The following code sample shows how to use the cluster occupancy API to find the max number of active clusters of a given size. Example code below calcualtes occupancy for cluster of size 2 and 128 threads per block.

Cluster size of 8 is forward compatible starting compute capability 9.0, except on GPU hardware or MIG configurations which are too small to support 8 multiprocessors in which case the maximum cluster size will be reduced. But it is recommended that the users query the maximum cluster size before

launching a cluster kernel. Max cluster size can be queried using `cudaOccupancyMaxPotentialClusterSize` API.

```
{
  cudaLaunchConfig_t config = {0};
  config.gridDim = number_of_blocks;
  config.blockDim = 128; // threads_per_block = 128
  config.dynamicSmemBytes = dynamic_shared_memory_size;

  cudaLaunchAttribute attribute[1];
  attribute[0].id = cudaLaunchAttributeClusterDimension;
  attribute[0].val.clusterDim.x = 2; // cluster_size = 2
  attribute[0].val.clusterDim.y = 1;
  attribute[0].val.clusterDim.z = 1;
  config.attrs = attribute;
  config.numAttrs = 1;

  int max_cluster_size = 0;
  cudaOccupancyMaxPotentialClusterSize(&max_cluster_size, (void *)kernel, &config);

  int max_active_clusters = 0;
  cudaOccupancyMaxActiveClusters(&max_active_clusters, (void *)kernel, &config);

  std::cout << "Max Active Clusters of size 2: " << max_active_clusters << std::endl;
}
```

The CUDA Nsight Compute User Interface also provides a standalone occupancy calculator and launch configurator implementation in `<CUDA_Toolkit_Path>/include/cuda_occupancy.h` for any use cases that cannot depend on the CUDA software stack. The Nsight Compute version of the occupancy calculator is particularly useful as a learning tool that visualizes the impact of changes to the parameters that affect occupancy (block size, registers per thread, and shared memory per thread).

8.3. Maximize Memory Throughput

The first step in maximizing overall memory throughput for the application is to minimize data transfers with low bandwidth.

That means minimizing data transfers between the host and the device, as detailed in *Data Transfer between Host and Device*, since these have much lower bandwidth than data transfers between global memory and the device.

That also means minimizing data transfers between global memory and the device by maximizing use of on-chip memory: shared memory and caches (i.e., L1 cache and L2 cache available on devices of compute capability 2.x and higher, texture cache and constant cache available on all devices).

Shared memory is equivalent to a user-managed cache: The application explicitly allocates and accesses it. As illustrated in *CUDA Runtime*, a typical programming pattern is to stage data coming from device memory into shared memory; in other words, to have each thread of a block:

- Load data from device memory to shared memory,
- Synchronize with all the other threads of the block so that each thread can safely read shared memory locations that were populated by different threads,
- Process the data in shared memory,

- Synchronize again if necessary to make sure that shared memory has been updated with the results,
- Write the results back to device memory.

For some applications (for example, for which global memory access patterns are data-dependent), a traditional hardware-managed cache is more appropriate to exploit data locality. As mentioned in *Compute Capability 7.x*, *Compute Capability 8.x* and *Compute Capability 9.0*, for devices of compute capability 7.x, 8.x and 9.0, the same on-chip memory is used for both L1 and shared memory, and how much of it is dedicated to L1 versus shared memory is configurable for each kernel call.

The throughput of memory accesses by a kernel can vary by an order of magnitude depending on access pattern for each type of memory. The next step in maximizing memory throughput is therefore to organize memory accesses as optimally as possible based on the optimal memory access patterns described in *Device Memory Accesses*. This optimization is especially important for global memory accesses as global memory bandwidth is low compared to available on-chip bandwidths and arithmetic instruction throughput, so non-optimal global memory accesses generally have a high impact on performance.

8.3.1. Data Transfer between Host and Device

Applications should strive to minimize data transfer between the host and the device. One way to accomplish this is to move more code from the host to the device, even if that means running kernels that do not expose enough parallelism to execute on the device with full efficiency. Intermediate data structures may be created in device memory, operated on by the device, and destroyed without ever being mapped by the host or copied to host memory.

Also, because of the overhead associated with each transfer, batching many small transfers into a single large transfer always performs better than making each transfer separately.

On systems with a front-side bus, higher performance for data transfers between host and device is achieved by using page-locked host memory as described in *Page-Locked Host Memory*.

In addition, when using mapped page-locked memory (*Mapped Memory*), there is no need to allocate any device memory and explicitly copy data between device and host memory. Data transfers are implicitly performed each time the kernel accesses the mapped memory. For maximum performance, these memory accesses must be coalesced as with accesses to global memory (see *Device Memory Accesses*). Assuming that they are and that the mapped memory is read or written only once, using mapped page-locked memory instead of explicit copies between device and host memory can be a win for performance.

On integrated systems where device memory and host memory are physically the same, any copy between host and device memory is superfluous and mapped page-locked memory should be used instead. Applications may query a device is `integrated` by checking that the integrated device property (see *Device Enumeration*) is equal to 1.

8.3.2. Device Memory Accesses

An instruction that accesses addressable memory (i.e., global, local, shared, constant, or texture memory) might need to be re-issued multiple times depending on the distribution of the memory addresses across the threads within the warp. How the distribution affects the instruction throughput this way is specific to each type of memory and described in the following sections. For example, for global memory, as a general rule, the more scattered the addresses are, the more reduced the throughput is.

Global Memory

Global memory resides in device memory and device memory is accessed via 32-, 64-, or 128-byte memory transactions. These memory transactions must be naturally aligned: Only the 32-, 64-, or 128-byte segments of device memory that are aligned to their size (i.e., whose first address is a multiple of their size) can be read or written by memory transactions.

When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. In general, the more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, reducing the instruction throughput accordingly. For example, if a 32-byte memory transaction is generated for each thread's 4-byte access, throughput is divided by 8.

How many transactions are necessary and how much throughput is ultimately affected varies with the compute capability of the device. *Compute Capability 5.x*, *Compute Capability 6.x*, *Compute Capability 7.x*, *Compute Capability 8.x*, *Compute Capability 9.0*, *Compute Capability 10.0*, and *Compute Capability 12.0* give more details on how global memory accesses are handled for various compute capabilities.

To maximize global memory throughput, it is therefore important to maximize coalescing by:

- Following the most optimal access patterns based on *Compute Capability 5.x*, *Compute Capability 6.x*, *Compute Capability 7.x*, *Compute Capability 8.x*, *Compute Capability 9.0*, *Compute Capability 10.0*, and *Compute Capability 12.0*.
- Using data types that meet the size and alignment requirement detailed in the section Size and Alignment Requirement below,
- Padding data in some cases, for example, when accessing a two-dimensional array as described in the section Two-Dimensional Arrays below.

Size and Alignment Requirement

Global memory instructions support reading or writing words of size equal to 1, 2, 4, 8, or 16 bytes. Any access (via a variable or a pointer) to data residing in global memory compiles to a single global memory instruction if and only if the size of the data type is 1, 2, 4, 8, or 16 bytes and the data is naturally aligned (i.e., its address is a multiple of that size).

If this size and alignment requirement is not fulfilled, the access compiles to multiple instructions with interleaved access patterns that prevent these instructions from fully coalescing. It is therefore recommended to use types that meet this requirement for data that resides in global memory.

The alignment requirement is automatically fulfilled for the *Built-in Vector Types*.

For structures, the size and alignment requirements can be enforced by the compiler using the alignment specifiers `__align__(8)` or `__align__(16)`, such as

```
struct __align__(8) {
    float x;
    float y;
};
```

or

```
struct __align__(16) {
    float x;
    float y;
    float z;
};
```

Any address of a variable residing in global memory or returned by one of the memory allocation routines from the driver or runtime API is always aligned to at least 256 bytes.

Reading non-naturally aligned 8-byte or 16-byte words produces incorrect results (off by a few words), so special care must be taken to maintain alignment of the starting address of any value or array of values of these types. A typical case where this might be easily overlooked is when using some custom global memory allocation scheme, whereby the allocations of multiple arrays (with multiple calls to `cudaMalloc()` or `cuMemAlloc()`) is replaced by the allocation of a single large block of memory partitioned into multiple arrays, in which case the starting address of each array is offset from the block's starting address.

Two-Dimensional Arrays

A common global memory access pattern is when each thread of index `(tx,ty)` uses the following address to access one element of a 2D array of width `width`, located at address `BaseAddress` of type `type*` (where `type` meets the requirement described in *Maximize Utilization*):

```
BaseAddress + width * ty + tx
```

For these accesses to be fully coalesced, both the width of the thread block and the width of the array must be a multiple of the warp size.

In particular, this means that an array whose width is not a multiple of this size will be accessed much more efficiently if it is actually allocated with a width rounded up to the closest multiple of this size and its rows padded accordingly. The `cudaMallocPitch()` and `cuMemAllocPitch()` functions and associated memory copy functions described in the reference manual enable programmers to write non-hardware-dependent code to allocate arrays that conform to these constraints.

Local Memory

Local memory accesses only occur for some automatic variables as mentioned in *Variable Memory Space Specifiers*. Automatic variables that the compiler is likely to place in local memory are:

- Arrays for which it cannot determine that they are indexed with constant quantities,
- Large structures or arrays that would consume too much register space,
- Any variable if the kernel uses more registers than available (this is also known as *register spilling*).

Inspection of the *PTX* assembly code (obtained by compiling with the `-ptx` or `-keep` option) will tell if a variable has been placed in local memory during the first compilation phases as it will be declared using the `.local` mnemonic and accessed using the `ld.local` and `st.local` mnemonics. Even if it has not, subsequent compilation phases might still decide otherwise though if they find it consumes too much register space for the targeted architecture: Inspection of the *cubin* object using `cuobjdump` will tell if this is the case. Also, the compiler reports total local memory usage per kernel (`lmem`) when compiling with the `--ptxas-options=-v` option. Note that some mathematical functions have implementation paths that might access local memory.

The local memory space resides in device memory, so local memory accesses have the same high latency and low bandwidth as global memory accesses and are subject to the same requirements for memory coalescing as described in *Device Memory Accesses*. Local memory is however organized such that consecutive 32-bit words are accessed by consecutive thread IDs. Accesses are therefore fully

coalesced as long as all threads in a warp access the same relative address (for example, same index in an array variable, same member in a structure variable).

On devices of compute capability 5.x onwards, local memory accesses are always cached in L2 in the same way as global memory accesses (see *Compute Capability 5.x* and *Compute Capability 6.x*).

Shared Memory

Because it is on-chip, shared memory has much higher bandwidth and much lower latency than local or global memory.

To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Any memory read or write request made of *n* addresses that fall in *n* distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is *n* times as high as the bandwidth of a single module.

However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing throughput by a factor equal to the number of separate memory requests. If the number of separate memory requests is *n*, the initial memory request is said to cause *n*-way bank conflicts.

To get maximum performance, it is therefore important to understand how memory addresses map to memory banks in order to schedule the memory requests so as to minimize bank conflicts. This is described in *Compute Capability 5.x*, *Compute Capability 6.x*, *Compute Capability 7.x*, *Compute Capability 8.x*, *Compute Capability 9.0*, *Compute Capability 10.0*, and *Compute Capability 12.0* for devices of these compute capabilities respectively.

Constant Memory

The constant memory space resides in device memory and is cached in the constant cache.

A request is then split into as many separate requests as there are different memory addresses in the initial request, decreasing throughput by a factor equal to the number of separate requests.

The resulting requests are then serviced at the throughput of the constant cache in case of a cache hit, or at the throughput of device memory otherwise.

Texture and Surface Memory

The texture and surface memory spaces reside in device memory and are cached in texture cache, so a texture fetch or surface read costs one memory read from device memory only on a cache miss, otherwise it just costs one read from texture cache. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture or surface addresses that are close together in 2D will achieve best performance. Also, it is designed for streaming fetches with a constant latency; a cache hit reduces DRAM bandwidth demand but not fetch latency.

Reading device memory through texture or surface fetching present some benefits that can make it an advantageous alternative to reading device memory from global or constant memory:

- If the memory reads do not follow the access patterns that global or constant memory reads must follow to get good performance, higher bandwidth can be achieved providing that there is locality in the texture fetches or surface reads;
- Addressing calculations are performed outside the kernel by dedicated units;
- Packed data may be broadcast to separate variables in a single operation;
- 8-bit and 16-bit integer input data may be optionally converted to 32 bit floating-point values in the range [0.0, 1.0] or [-1.0, 1.0] (see *Texture Memory*).

8.4. Maximize Instruction Throughput

See the CUDA C++ Best Practices Guide for more details on optimizing instruction throughput.

8.5. Minimize Memory Thrashing

Applications that constantly allocate and free memory too often may find that the allocation calls tend to get slower over time up to a limit. This is typically expected due to the nature of releasing memory back to the operating system for its own use. For best performance in this regard, we recommend the following:

- Try to size your allocation to the problem at hand. Don't try to allocate all available memory with `cudaMalloc` / `cudaMallocHost` / `cuMemCreate`, as this forces memory to be resident immediately and prevents other applications from being able to use that memory. This can put more pressure on operating system schedulers, or just prevent other applications using the same GPU from running entirely.
- Try to allocate memory in appropriately sized allocations early in the application and allocations only when the application does not have any use for it. Reduce the number of `cudaMalloc+cudaFree` calls in the application, especially in performance-critical regions.
- If an application cannot allocate enough device memory, consider falling back on other memory types such as `cudaMallocHost` or `cudaMallocManaged`, which may not be as performant, but will enable the application to make progress.
- For platforms that support the feature, `cudaMallocManaged` allows for oversubscription, and with the correct `cudaMemAdvise` policies enabled, will allow the application to retain most if not all the performance of `cudaMalloc`. `cudaMallocManaged` also won't force an allocation to be resident until it is needed or prefetched, reducing the overall pressure on the operating system schedulers and better enabling multi-tenet use cases.

Chapter 9. CUDA-Enabled GPUs

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

https://developer.nvidia.com/cuda-gpus lists all CUDA-enabled devices with their compute capability.

The compute capability, number of multiprocessors, clock frequency, total amount of device memory, and other properties can be queried using the runtime (see reference manual).

Chapter 10. C++ Language Extensions

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

10.1. Function Execution Space Specifiers

Function execution space specifiers denote whether a function executes on the host or on the device and whether it is callable from the host or from the device.

10.1.1. __global__

The `__global__` execution space specifier declares a function as being a kernel. Such a function is:

- Executed on the device,
- Callable from the host,
- Callable from the device for devices of compute capability 5.0 or higher (see *CUDA Dynamic Parallelism* for more details).

A `__global__` function must have void return type, and cannot be a member of a class.

Any call to a `__global__` function must specify its execution configuration as described in *Execution Configuration*.

A call to a `__global__` function is asynchronous, meaning it returns before the device has completed its execution.

10.1.2. __device__

The `__device__` execution space specifier declares a function that is:

- Executed on the device,
- Callable from the device only.

The `__global__` and `__device__` execution space specifiers cannot be used together.

10.1.3. __host__

The `__host__` execution space specifier declares a function that is:

- Executed on the host,
- Callable from the host only.

It is equivalent to declare a function with only the `__host__` execution space specifier or to declare it without any of the `__host__`, `__device__`, or `__global__` execution space specifier; in either case the function is compiled for the host only.

The `__global__` and `__host__` execution space specifiers cannot be used together.

The `__device__` and `__host__` execution space specifiers can be used together however, in which case the function is compiled for both the host and the device. The `__CUDA_ARCH__` macro introduced in *Application Compatibility* can be used to differentiate code paths between host and device:

```
__host__ __device__ func()
{
#if __CUDA_ARCH__ >= 800
   // Device code path for compute capability 8.x
#elif __CUDA_ARCH__ >= 700
   // Device code path for compute capability 7.x
#elif __CUDA_ARCH__ >= 600
   // Device code path for compute capability 6.x
#elif __CUDA_ARCH__ >= 500
   // Device code path for compute capability 5.x
#elif !defined(__CUDA_ARCH__)
   // Host code path
#endif
}
```

10.1.4. Undefined behavior

A 'cross-execution space' call has undefined behavior when:

- `__CUDA_ARCH__` is defined, a call from within a `__global__`, `__device__` or `__host__ __device__` function to a `__host__` function.
- `__CUDA_ARCH__` is undefined, a call from within a `__host__` function to a `__device__` function.[4]

[4] When the enclosing __host__ function is a template, nvcc may currently fail to issue a diagnostic message in some cases; this behavior may change in the future.

10.1.5. __noinline__ and __forceinline__

The compiler inlines any `__device__` function when deemed appropriate.

The `__noinline__` function qualifier can be used as a hint for the compiler not to inline the function if possible.

The `__forceinline__` function qualifier can be used to force the compiler to inline the function.

The `__noinline__` and `__forceinline__` function qualifiers cannot be used together, and neither function qualifier can be applied to an inline function.

10.1.6. __inline_hint__

The `__inline_hint__` qualifier enables more aggressive inlining in the compiler. Unlike `__forceinline__`, it does not imply that the function is inline. It can be used to improve inlining across modules when using LTO.

Neither the `__noinline__` nor the `__forceinline__` function qualifier can be used with the `__inline_hint__` function qualifier.

10.2. Variable Memory Space Specifiers

Variable memory space specifiers denote the memory location on the device of a variable.

An automatic variable declared in device code without any of the `__device__`, `__shared__` and `__constant__` memory space specifiers described in this section generally resides in a register. However in some cases the compiler might choose to place it in local memory, which can have adverse performance consequences as detailed in *Device Memory Accesses*.

10.2.1. __device__

The `__device__` memory space specifier declares a variable that resides on the device.

At most one of the other memory space specifiers defined in the next three sections may be used together with `__device__` to further denote which memory space the variable belongs to. If none of them is present, the variable:

- Resides in global memory space,
- Has the lifetime of the CUDA context in which it is created,
- Has a distinct object per device,
- Is accessible from all the threads within the grid and from the host through the runtime library (`cudaGetSymbolAddress()` / `cudaGetSymbolSize()` / `cudaMemcpyToSymbol()` / `cudaMemcpyFromSymbol()`).

10.2.2. __constant__

The `__constant__` memory space specifier, optionally used together with `__device__`, declares a variable that:

- Resides in constant memory space,
- Has the lifetime of the CUDA context in which it is created,
- Has a distinct object per device,
- Is accessible from all the threads within the grid and from the host through the runtime library (`cudaGetSymbolAddress()` / `cudaGetSymbolSize()` / `cudaMemcpyToSymbol()` / `cudaMemcpyFromSymbol()`).

The behavior of modifying a constant from the host while there is a concurrent grid that access that constant at any point of this grid's lifetime is undefined.

10.2.3. __shared__

The `__shared__` memory space specifier, optionally used together with `__device__`, declares a variable that:

- Resides in the shared memory space of a thread block,
- Has the lifetime of the block,
- Has a distinct object per block,
- Is only accessible from all the threads within the block,
- Does not have a constant address.

When declaring a variable in shared memory as an external array such as

```
extern __shared__ float shared[];
```

the size of the array is determined at launch time (see *Execution Configuration*). All variables declared in this fashion, start at the same address in memory, so that the layout of the variables in the array must be explicitly managed through offsets. For example, if one wants the equivalent of

```
short array0[128];
float array1[64];
int   array2[256];
```

in dynamically allocated shared memory, one could declare and initialize the arrays the following way:

```
extern __shared__ float array[];
__device__ void func()      // __device__ or __global__ function
{
    short* array0 = (short*)array;
    float* array1 = (float*)&array0[128];
    int*   array2 =   (int*)&array1[64];
}
```

Note that pointers need to be aligned to the type they point to, so the following code, for example, does not work since array1 is not aligned to 4 bytes.

```
extern __shared__ float array[];
__device__ void func()      // __device__ or __global__ function
{
    short* array0 = (short*)array;
    float* array1 = (float*)&array0[127];
}
```

Alignment requirements for the built-in vector types are listed in Table 7.

10.2.4. __grid_constant__

The `__grid_constant__` annotation for compute architectures greater or equal to 7.0 annotates a `const`-qualified `__global__` function parameter of non-reference type that:

- Has the lifetime of the grid,
- Is private to the grid, i.e., the object is not accessible to host threads and threads from other grids, including sub-grids,
- Has a distinct object per grid, i.e., all threads in the grid see the same address,
- Is read-only, i.e., modifying a `__grid_constant__` object or any of its sub-objects is *undefined behavior*, including `mutable` members.

Requirements:

- Kernel parameters annotated with `__grid_constant__` must have `const`-qualified non-reference types.
- All function declarations must match with respect to any `__grid_constant_` parameters.
- A function template specialization must match the primary template declaration with respect to any `__grid_constant__` parameters.
- A function template instantiation directive must match the primary template declaration with respect to any `__grid_constant__` parameters.

If the address of a `__global__` function parameter is taken, the compiler will ordinarily make a copy of the kernel parameter in thread local memory and use the address of the copy, to partially support C++ semantics, which allow each thread to modify its own local copy of function parameters. Annotating a `__global__` function parameter with `__grid_constant__` ensures that the compiler will not create a copy of the kernel parameter in thread local memory, but will instead use the generic address of the parameter itself. Avoiding the local copy may result in improved performance.

```
__device__ void unknown_function(S const&);
__global__ void kernel(const __grid_constant__ S s) {
   s.x += threadIdx.x;  // Undefined Behavior: tried to modify read-only memory

   // Compiler will _not_ create a per-thread thread local copy of "s":
   unknown_function(s);
}
```

10.2.5. __managed__

The `__managed__` memory space specifier, optionally used together with `__device__`, declares a variable that:

- Can be referenced from both device and host code, for example, its address can be taken or it can be read or written directly from a device or host function.
- Has the lifetime of an application.

See *__managed__ Memory Space Specifier* for more details.

10.2.6. __restrict__

nvcc supports restricted pointers via the `__restrict__` keyword.

Restricted pointers were introduced in C99 to alleviate the aliasing problem that exists in C-type languages, and which inhibits all kind of optimization from code re-ordering to common sub-expression elimination.

Here is an example subject to the aliasing issue, where use of restricted pointer can help the compiler to reduce the number of instructions:

```
void foo(const float* a,
         const float* b,
         float* c)
{
    c[0] = a[0] * b[0];
    c[1] = a[0] * b[0];
    c[2] = a[0] * b[0] * a[1];
    c[3] = a[0] * a[1];
    c[4] = a[0] * b[0];
    c[5] = b[0];
    ...
}
```

In C-type languages, the pointers `a`, `b`, and `c` may be aliased, so any write through `c` could modify elements of `a` or `b`. This means that to guarantee functional correctness, the compiler cannot load `a[0]` and `b[0]` into registers, multiply them, and store the result to both `c[0]` and `c[1]`, because the results would differ from the abstract execution model if, say, `a[0]` is really the same location as `c[0]`. So the compiler cannot take advantage of the common sub-expression. Likewise, the compiler cannot just reorder the computation of `c[4]` into the proximity of the computation of `c[0]` and `c[1]` because the preceding write to `c[3]` could change the inputs to the computation of `c[4]`.

By making `a`, `b`, and `c` restricted pointers, the programmer asserts to the compiler that the pointers are in fact not aliased, which in this case means writes through `c` would never overwrite elements of `a` or `b`. This changes the function prototype as follows:

```
void foo(const float* __restrict__ a,
         const float* __restrict__ b,
         float* __restrict__ c);
```

Note that all pointer arguments need to be made restricted for the compiler optimizer to derive any benefit. With the `__restrict__` keywords added, the compiler can now reorder and do common sub-expression elimination at will, while retaining functionality identical with the abstract execution model:

```
void foo(const float* __restrict__ a,
         const float* __restrict__ b,
         float* __restrict__ c)
{
    float t0 = a[0];
    float t1 = b[0];
    float t2 = t0 * t1;
    float t3 = a[1];
    c[0] = t2;
    c[1] = t2;
    c[4] = t2;
    c[2] = t2 * t3;
    c[3] = t0 * t3;
    c[5] = t1;
    ...
}
```

The effects here are a reduced number of memory accesses and reduced number of computations. This is balanced by an increase in register pressure due to "cached" loads and common subexpressions.

Since register pressure is a critical issue in many CUDA codes, use of restricted pointers can have negative performance impact on CUDA code, due to reduced occupancy.

10.3. Built-in Vector Types

10.3.1. char, short, int, long, longlong, float, double

These are vector types derived from the basic integer and floating-point types. They are structures and the 1st, 2nd, 3rd, and 4th components are accessible through the fields `x`, `y`, `z`, and `w`, respectively. They all come with a constructor function of the form `make_<type name>`; for example,

```
int2 make_int2(int x, int y);
```

which creates a vector of type `int2` with value `(x, y)`.

The alignment requirements of the vector types are detailed in Table 7.

Table 7: Alignment Requirements

Type	Alignment
char1, uchar1	1
char2, uchar2	2
char3, uchar3	1
char4, uchar4	4
short1, ushort1	2
short2, ushort2	4

continues on next page

Table 7 – continued from previous page

Type	Alignment
short3, ushort3	2
short4, ushort4	8
int1, uint1	4
int2, uint2	8
int3, uint3	4
int4, uint4	16
long1, ulong1	4 if sizeof(long) is equal to sizeof(int) 8, otherwise
long2, ulong2	8 if sizeof(long) is equal to sizeof(int), 16, otherwise
long3, ulong3	4 if sizeof(long) is equal to sizeof(int), 8, otherwise
long4[3]	16
long4_16a	
long4_32a	32
ulong4[Page 167, 3]	16
ulong4_16a	
ulong4_32a	32
longlong1, ulonglong1	8
longlong2, ulonglong2	16
longlong3, ulonglong3	8
longlong4[Page 167, 3]	16
longlong4_16a	
longlong4_32a	32
ulonglong4[Page 167, 3]	16
ulonglong4_16a	
ulonglong4_32a	32
float1	4
float2	8
float3	4
float4	16
double1	8
double2	16
double3	8

continues on next page

Table 7 – continued from previous page

Type	Alignment
double4[Page 167, 3]	16
double4_16a	
double4_32a	32

10.3.2. dim3

This type is an integer vector type based on `uint3` that is used to specify dimensions. When defining a variable of type `dim3`, any component left unspecified is initialized to 1.

10.4. Built-in Variables

Built-in variables specify the grid and block dimensions and the block and thread indices. They are only valid within functions that are executed on the device.

10.4.1. gridDim

This variable is of type `dim3` (see *dim3*) and contains the dimensions of the grid.

10.4.2. blockIdx

This variable is of type `uint3` (see *char, short, int, long, longlong, float, double*) and contains the block index within the grid.

10.4.3. blockDim

This variable is of type `dim3` (see *dim3*) and contains the dimensions of the block.

[3] This vector type was deprecated in CUDA Toolkit 13.0. Please use the `_16a` or `_32a` variant instead, depending on your alignment requirements.

10.4.4. threadIdx

This variable is of type `uint3` (see *char, short, int, long, longlong, float, double*) and contains the thread index within the block.

10.4.5. warpSize

This variable is of type `int` and contains the warp size in threads (see *SIMT Architecture* for the definition of a warp).

10.5. Memory Fence Functions

The CUDA programming model assumes a device with a weakly-ordered memory model, that is the order in which a CUDA thread writes data to shared memory, global memory, page-locked host memory, or the memory of a peer device is not necessarily the order in which the data is observed being written by another CUDA or host thread. It is undefined behavior for two threads to read from or write to the same memory location without synchronization.

In the following example, thread 1 executes `writeXY()`, while thread 2 executes `readXY()`.

```
__device__ int X = 1, Y = 2;

__device__ void writeXY()
{
    X = 10;
    Y = 20;
}

__device__ void readXY()
{
    int B = Y;
    int A = X;
}
```

The two threads read and write from the same memory locations `X` and `Y` simultaneously. Any data-race is undefined behavior, and has no defined semantics. The resulting values for `A` and `B` can be anything.

Memory fence functions can be used to enforce a sequentially-consistent ordering on memory accesses. The memory fence functions differ in the scope in which the orderings are enforced but they are independent of the accessed memory space (shared memory, global memory, page-locked host memory, and the memory of a peer device).

```
void __threadfence_block();
```

is equivalent to cuda::atomic_thread_fence(cuda::memory_order_seq_cst, cuda::thread_scope_block) and ensures that:

- All writes to all memory made by the calling thread before the call to `__threadfence_block()` are observed by all threads in the block of the calling thread as occurring before all writes to all memory made by the calling thread after the call to `__threadfence_block()`;

- All reads from all memory made by the calling thread before the call to `__threadfence_block()` are ordered before all reads from all memory made by the calling thread after the call to `__threadfence_block()`.

```
void __threadfence();
```

is equivalent to cuda::atomic_thread_fence(cuda::memory_order_seq_cst, cuda::thread_scope_device) and ensures that no writes to all memory made by the calling thread after the call to `__threadfence()` are observed by any thread in the device as occurring before any write to all memory made by the calling thread before the call to `__threadfence()`.

```
void __threadfence_system();
```

is equivalent to cuda::atomic_thread_fence(cuda::memory_order_seq_cst, cuda::thread_scope_system) and ensures that all writes to all memory made by the calling thread before the call to `__threadfence_system()` are observed by all threads in the device, host threads, and all threads in peer devices as occurring before all writes to all memory made by the calling thread after the call to `__threadfence_system()`.

`__threadfence_system()` is only supported by devices of compute capability 2.x and higher.

In the previous code sample, we can insert fences in the codes as follows:

```
__device__ int X = 1, Y = 2;

__device__ void writeXY()
{
    X = 10;
    __threadfence();
    Y = 20;
}

__device__ void readXY()
{
    int B = Y;
    __threadfence();
    int A = X;
}
```

For this code, the following outcomes can be observed:

- A equal to 1 and B equal to 2,
- A equal to 10 and B equal to 2,
- A equal to 10 and B equal to 20.

The fourth outcome is not possible, because the first write must be visible before the second write. If thread 1 and 2 belong to the same block, it is enough to use `__threadfence_block()`. If thread 1 and 2 do not belong to the same block, `__threadfence()` must be used if they are CUDA threads from the same device and `__threadfence_system()` must be used if they are CUDA threads from two different devices.

A common use case is when threads consume some data produced by other threads as illustrated by the following code sample of a kernel that computes the sum of an array of N numbers in one call. Each block first sums a subset of the array and stores the result in global memory. When all blocks are done, the last block done reads each of these partial sums from global memory and sums them to obtain the final result. In order to determine which block is finished last, each block atomically increments a counter to signal that it is done with computing and storing its partial sum (see *Atomic Functions*

about atomic functions). The last block is the one that receives the counter value equal to `gridDim.x-1`. If no fence is placed between storing the partial sum and incrementing the counter, the counter might increment before the partial sum is stored and therefore, might reach `gridDim.x-1` and let the last block start reading partial sums before they have been actually updated in memory.

Memory fence functions only affect the ordering of memory operations by a thread; they do not, by themselves, ensure that these memory operations are visible to other threads (like `__syncthreads()` does for threads within a block; see *Synchronization Functions*). In the code sample below, the visibility of memory operations on the `result` variable is ensured by declaring it as volatile (see *Volatile Qualifier*).

```
__device__ unsigned int count = 0;
__shared__ bool isLastBlockDone;
__global__ void sum(const float* array, unsigned int N,
                    volatile float* result)
{
    // Each block sums a subset of the input array.
    float partialSum = calculatePartialSum(array, N);

    if (threadIdx.x == 0) {

        // Thread 0 of each block stores the partial sum
        // to global memory. The compiler will use
        // a store operation that bypasses the L1 cache
        // since the "result" variable is declared as
        // volatile. This ensures that the threads of
        // the last block will read the correct partial
        // sums computed by all other blocks.
        result[blockIdx.x] = partialSum;

        // Thread 0 makes sure that the incrementing
        // of the "count" variable is only performed after
        // the partial sum has been written to global memory.
        __threadfence();

        // Thread 0 signals that it is done.
        unsigned int value = atomicInc(&count, gridDim.x);

        // Thread 0 determines if its block is the last
        // block to be done.
        isLastBlockDone = (value == (gridDim.x - 1));
    }

    // Synchronize to make sure that each thread reads
    // the correct value of isLastBlockDone.
    __syncthreads();

    if (isLastBlockDone) {

        // The last block sums the partial sums
        // stored in result[0 .. gridDim.x-1]
        float totalSum = calculateTotalSum(result);

        if (threadIdx.x == 0) {

            // Thread 0 of last block stores the total sum
            // to global memory and resets the count
            // variable, so that the next kernel call
```

(continues on next page)

(continued from previous page)

```
            // works properly.
            result[0] = totalSum;
            count = 0;
        }
    }
}
```

10.6. Synchronization Functions

```
void __syncthreads();
```

waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to `__syncthreads()` are visible to all threads in the block.

`__syncthreads()` is used to coordinate communication between the threads of the same block. When some threads within a block access the same addresses in shared or global memory, there are potential read-after-write, write-after-read, or write-after-write hazards for some of these memory accesses. These data hazards can be avoided by synchronizing threads in-between these accesses.

`__syncthreads()` is allowed in conditional code but only if the conditional evaluates identically across the entire thread block, otherwise the code execution is likely to hang or produce unintended side effects.

Devices of compute capability 2.x and higher support three variations of `__syncthreads()` described below.

```
int __syncthreads_count(int predicate);
```

is identical to `__syncthreads()` with the additional feature that it evaluates predicate for all threads of the block and returns the number of threads for which predicate evaluates to non-zero.

```
int __syncthreads_and(int predicate);
```

is identical to `__syncthreads()` with the additional feature that it evaluates predicate for all threads of the block and returns non-zero if and only if predicate evaluates to non-zero for all of them.

```
int __syncthreads_or(int predicate);
```

is identical to `__syncthreads()` with the additional feature that it evaluates predicate for all threads of the block and returns non-zero if and only if predicate evaluates to non-zero for any of them.

```
void __syncwarp(unsigned mask=0xffffffff);
```

will cause the executing thread to wait until all warp lanes named in mask have executed a `__syncwarp()` (with the same mask) before resuming execution. Each calling thread must have its own bit set in the mask and all non-exited threads named in mask must execute a corresponding `__syncwarp()` with the same mask, or the result is undefined.

Executing `__syncwarp()` guarantees memory ordering among threads participating in the barrier. Thus, threads within a warp that wish to communicate via memory can store to memory, execute `__syncwarp()`, and then safely read values stored by other threads in the warp.

Note: For .target sm_6x or below, all threads in mask must execute the same `__syncwarp()` in convergence, and the union of all values in mask must be equal to the active mask. Otherwise, the behavior is undefined.

10.7. Mathematical Functions

The reference manual lists all C/C++ standard library mathematical functions that are supported in device code and all intrinsic functions that are only supported in device code.

Mathematical Functions provides accuracy information for some of these functions when relevant.

10.8. Texture Functions

Texture objects are described in *Texture Object API*.

Texture fetching is described in *Texture Fetching*.

10.8.1. Texture Object API

10.8.1.1 tex1Dfetch()

```
template<class T>
T tex1Dfetch(cudaTextureObject_t texObj, int x);
```

fetches from the region of linear memory specified by the one-dimensional texture object `texObj` using integer texture coordinate `x`. `tex1Dfetch()` only works with non-normalized coordinates, so only the border and clamp addressing modes are supported. It does not perform any texture filtering. For integer types, it may optionally promote the integer to single-precision floating point.

10.8.1.2 tex1D()

```
template<class T>
T tex1D(cudaTextureObject_t texObj, float x);
```

fetches from the CUDA array specified by the one-dimensional texture object `texObj` using texture coordinate `x`.

10.8.1.3 tex1DLod()

```
template<class T>
T tex1DLod(cudaTextureObject_t texObj, float x, float level);
```

fetches from the CUDA array specified by the one-dimensional texture object `texObj` using texture coordinate `x` at the level-of-detail `level`.

10.8.1.4 tex1DGrad()

```
template<class T>
T tex1DGrad(cudaTextureObject_t texObj, float x, float dx, float dy);
```

fetches from the CUDA array specified by the one-dimensional texture object `texObj` using texture coordinate `x`. The level-of-detail is derived from the X-gradient `dx` and Y-gradient `dy`.

10.8.1.5 tex2D()

```
template<class T>
T tex2D(cudaTextureObject_t texObj, float x, float y);
```

fetches from the CUDA array or the region of linear memory specified by the two-dimensional texture object `texObj` using texture coordinate `(x,y)`.

10.8.1.6 tex2D() for sparse CUDA arrays

```
               template<class T>
T tex2D(cudaTextureObject_t texObj, float x, float y, bool* isResident);
```

fetches from the CUDA array specified by the two-dimensional texture object `texObj` using texture coordinate `(x,y)`. Also returns whether the texel is resident in memory via `isResident` pointer. If not, the values fetched will be zeros.

10.8.1.7 tex2Dgather()

```
template<class T>
T tex2Dgather(cudaTextureObject_t texObj,
              float x, float y, int comp = 0);
```

fetches from the CUDA array specified by the 2D texture object `texObj` using texture coordinates `x` and `y` and the `comp` parameter as described in *Texture Gather*.

10.8.1.8 tex2Dgather() for sparse CUDA arrays

```
                template<class T>
T tex2Dgather(cudaTextureObject_t texObj,
              float x, float y, bool* isResident, int comp = 0);
```

fetches from the CUDA array specified by the 2D texture object `texObj` using texture coordinates `x` and `y` and the `comp` parameter as described in *Texture Gather*. Also returns whether the texel is resident in memory via `isResident` pointer. If not, the values fetched will be zeros.

10.8.1.9 tex2DGrad()

```
template<class T>
T tex2DGrad(cudaTextureObject_t texObj, float x, float y,
            float2 dx, float2 dy);
```

fetches from the CUDA array specified by the two-dimensional texture object `texObj` using texture coordinate `(x,y)`. The level-of-detail is derived from the `dx` and `dy` gradients.

10.8.1.10 tex2DGrad() for sparse CUDA arrays

```
                template<class T>
T tex2DGrad(cudaTextureObject_t texObj, float x, float y,
        float2 dx, float2 dy, bool* isResident);
```

fetches from the CUDA array specified by the two-dimensional texture object `texObj` using texture coordinate `(x,y)`. The level-of-detail is derived from the `dx` and `dy` gradients. Also returns whether the texel is resident in memory via `isResident` pointer. If not, the values fetched will be zeros.

10.8.1.11 tex2DLod()

```
template<class T>
tex2DLod(cudaTextureObject_t texObj, float x, float y, float level);
```

fetches from the CUDA array or the region of linear memory specified by the two-dimensional texture object `texObj` using texture coordinate `(x,y)` at level-of-detail `level`.

10.8.1.12 tex2DLod() for sparse CUDA arrays

```
        template<class T>
tex2DLod(cudaTextureObject_t texObj, float x, float y, float level, bool* isResident);
```

fetches from the CUDA array specified by the two-dimensional texture object `texObj` using texture coordinate `(x,y)` at level-of-detail `level`. Also returns whether the texel is resident in memory via `isResident` pointer. If not, the values fetched will be zeros.

10.8.1.13 tex3D()

```
template<class T>
T tex3D(cudaTextureObject_t texObj, float x, float y, float z);
```

fetches from the CUDA array specified by the three-dimensional texture object `texObj` using texture coordinate `(x,y,z)`.

10.8.1.14 tex3D() for sparse CUDA arrays

```
              template<class T>
T tex3D(cudaTextureObject_t texObj, float x, float y, float z, bool* isResident);
```

fetches from the CUDA array specified by the three-dimensional texture object `texObj` using texture coordinate `(x,y,z)`. Also returns whether the texel is resident in memory via `isResident` pointer. If not, the values fetched will be zeros.

10.8.1.15 tex3DLod()

```
template<class T>
T tex3DLod(cudaTextureObject_t texObj, float x, float y, float z, float level);
```

fetches from the CUDA array or the region of linear memory specified by the three-dimensional texture object `texObj` using texture coordinate `(x,y,z)` at level-of-detail `level`.

10.8.1.16 tex3DLod() for sparse CUDA arrays

```
              template<class T>
T tex3DLod(cudaTextureObject_t texObj, float x, float y, float z, float level, bool*
↪isResident);
```

fetches from the CUDA array or the region of linear memory specified by the three-dimensional texture object `texObj` using texture coordinate `(x,y,z)` at level-of-detail `level`. Also returns whether the texel is resident in memory via `isResident` pointer. If not, the values fetched will be zeros.

10.8.1.17 tex3DGrad()

```
template<class T>
T tex3DGrad(cudaTextureObject_t texObj, float x, float y, float z,
            float4 dx, float4 dy);
```

fetches from the CUDA array specified by the three-dimensional texture object `texObj` using texture coordinate `(x,y,z)` at a level-of-detail derived from the X and Y gradients `dx` and `dy`.

10.8.1.18 tex3DGrad() for sparse CUDA arrays

```
                template<class T>
T tex3DGrad(cudaTextureObject_t texObj, float x, float y, float z,
        float4 dx, float4 dy, bool* isResident);
```

fetches from the CUDA array specified by the three-dimensional texture object `texObj` using texture coordinate `(x,y,z)` at a level-of-detail derived from the X and Y gradients `dx` and `dy`. Also returns whether the texel is resident in memory via `isResident` pointer. If not, the values fetched will be zeros.

10.8.1.19 tex1DLayered()

```
template<class T>
T tex1DLayered(cudaTextureObject_t texObj, float x, int layer);
```

fetches from the CUDA array specified by the one-dimensional texture object `texObj` using texture coordinate `x` and index `layer`, as described in *Layered Textures*.

10.8.1.20 tex1DLayeredLod()

```
template<class T>
T tex1DLayeredLod(cudaTextureObject_t texObj, float x, int layer, float level);
```

fetches from the CUDA array specified by the one-dimensional *Layered Textures* at layer `layer` using texture coordinate `x` and level-of-detail `level`.

10.8.1.21 tex1DLayeredGrad()

```
template<class T>
T tex1DLayeredGrad(cudaTextureObject_t texObj, float x, int layer,
                   float dx, float dy);
```

fetches from the CUDA array specified by the one-dimensional *layered texture* at layer `layer` using texture coordinate `x` and a level-of-detail derived from the `dx` and `dy` gradients.

10.8.1.22 tex2DLayered()

```
template<class T>
T tex2DLayered(cudaTextureObject_t texObj,
               float x, float y, int layer);
```

fetches from the CUDA array specified by the two-dimensional texture object `texObj` using texture coordinate `(x,y)` and index `layer`, as described in *Layered Textures*.

10.8.1.23 tex2DLayered() for Sparse CUDA Arrays

```
                template<class T>
T tex2DLayered(cudaTextureObject_t texObj,
               float x, float y, int layer, bool* isResident);
```

fetches from the CUDA array specified by the two-dimensional texture object `texObj` using texture coordinate `(x,y)` and index `layer`, as described in *Layered Textures*. Also returns whether the texel is resident in memory via `isResident` pointer. If not, the values fetched will be zeros.

10.8.1.24 tex2DLayeredLod()

```
template<class T>
T tex2DLayeredLod(cudaTextureObject_t texObj, float x, float y, int layer,
                  float level);
```

fetches from the CUDA array specified by the two-dimensional *layered texture* at layer `layer` using texture coordinate `(x,y)`.

10.8.1.25 tex2DLayeredLod() for sparse CUDA arrays

```
                template<class T>
T tex2DLayeredLod(cudaTextureObject_t texObj, float x, float y, int layer,
                float level, bool* isResident);
```

fetches from the CUDA array specified by the two-dimensional *layered texture* at layer `layer` using texture coordinate `(x,y)`. Also returns whether the texel is resident in memory via `isResident` pointer. If not, the values fetched will be zeros.

10.8.1.26 tex2DLayeredGrad()

```
template<class T>
T tex2DLayeredGrad(cudaTextureObject_t texObj, float x, float y, int layer,
                   float2 dx, float2 dy);
```

fetches from the CUDA array specified by the two-dimensional *layered texture* at layer `layer` using texture coordinate `(x,y)` and a level-of-detail derived from the `dx` and `dy` gradients.

10.8.1.27 tex2DLayeredGrad() for sparse CUDA arrays

```
                template<class T>
T tex2DLayeredGrad(cudaTextureObject_t texObj, float x, float y, int layer,
                float2 dx, float2 dy, bool* isResident);
```

fetches from the CUDA array specified by the two-dimensional *layered texture* at layer `layer` using texture coordinate `(x,y)` and a level-of-detail derived from the `dx` and `dy` gradients. Also returns whether the texel is resident in memory via `isResident` pointer. If not, the values fetched will be zeros.

10.8.1.28 texCubemap()

```
template<class T>
T texCubemap(cudaTextureObject_t texObj, float x, float y, float z);
```

fetches the CUDA array specified by the cubemap texture object `texObj` using texture coordinate `(x,y,z)`, as described in *Cubemap Textures*.

10.8.1.29 texCubemapGrad()

```
template<class T>
T texCubemapGrad(cudaTextureObject_t texObj, float x, float, y, float z,
                 float4 dx, float4 dy);
```

fetches from the CUDA array specified by the cubemap texture object `texObj` using texture coordinate `(x,y,z)` as described in *Cubemap Textures*. The level-of-detail used is derived from the `dx` and `dy` gradients.

10.8.1.30 texCubemapLod()

```
template<class T>
T texCubemapLod(cudaTextureObject_t texObj, float x, float, y, float z,
                float level);
```

fetches from the CUDA array specified by the cubemap texture object `texObj` using texture coordinate `(x,y,z)` as described in *Cubemap Textures*. The level-of-detail used is given by `level`.

10.8.1.31 texCubemapLayered()

```
template<class T>
T texCubemapLayered(cudaTextureObject_t texObj,
                    float x, float y, float z, int layer);
```

fetches from the CUDA array specified by the cubemap layered texture object `texObj` using texture coordinates `(x,y,z)`, and index `layer`, as described in *Cubemap Layered Textures*.

10.8.1.32 texCubemapLayeredGrad()

```
template<class T>
T texCubemapLayeredGrad(cudaTextureObject_t texObj, float x, float y, float z,
                        int layer, float4 dx, float4 dy);
```

fetches from the CUDA array specified by the cubemap layered texture object `texObj` using texture coordinate `(x,y,z)` and index `layer`, as described in *Cubemap Layered Textures*, at level-of-detail derived from the `dx` and `dy` gradients.

10.8.1.33 texCubemapLayeredLod()

```
template<class T>
T texCubemapLayeredLod(cudaTextureObject_t texObj, float x, float y, float z,
                       int layer, float level);
```

fetches from the CUDA array specified by the cubemap layered texture object `texObj` using texture coordinate `(x,y,z)` and index `layer`, as described in *Cubemap Layered Textures*, at level-of-detail level `level`.

10.9. Surface Functions

Surface functions are only supported by devices of compute capability 2.0 and higher.

Surface objects are described in described in *Surface Object API*.

In the sections below, `boundaryMode` specifies the boundary mode, that is how out-of-range surface coordinates are handled; it is equal to either `cudaBoundaryModeClamp`, in which case out-of-range coordinates are clamped to the valid range, or `cudaBoundaryModeZero`, in which case out-of-range reads return zero and out-of-range writes are ignored, or `cudaBoundaryModeTrap`, in which case out-of-range accesses cause the kernel execution to fail.

10.9.1. Surface Object API

10.9.1.1 surf1Dread()

```
template<class T>
T surf1Dread(cudaSurfaceObject_t surfObj, int x,
             boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array specified by the one-dimensional surface object `surfObj` using byte coordinate x.

10.9.1.2 surf1Dwrite

```
template<class T>
void surf1Dwrite(T data,
                 cudaSurfaceObject_t surfObj,
                 int x,
                 boundaryMode = cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the one-dimensional surface object `surfObj` at byte coordinate x.

10.9.1.3 surf2Dread()

```
template<class T>
T surf2Dread(cudaSurfaceObject_t surfObj,
             int x, int y,
             boundaryMode = cudaBoundaryModeTrap);
template<class T>
void surf2Dread(T* data,
                cudaSurfaceObject_t surfObj,
                int x, int y,
                boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array specified by the two-dimensional surface object `surfObj` using byte coordinates x and y.

10.9.1.4 surf2Dwrite()

```
template<class T>
void surf2Dwrite(T data,
                 cudaSurfaceObject_t surfObj,
                 int x, int y,
                 boundaryMode = cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the two-dimensional surface object `surfObj` at byte coordinate x and y.

10.9.1.5 surf3Dread()

```
template<class T>
T surf3Dread(cudaSurfaceObject_t surfObj,
             int x, int y, int z,
             boundaryMode = cudaBoundaryModeTrap);
template<class T>
void surf3Dread(T* data,
                cudaSurfaceObject_t surfObj,
                int x, int y, int z,
                boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array specified by the three-dimensional surface object `surfObj` using byte coordinates x, y, and z.

10.9.1.6 surf3Dwrite()

```
template<class T>
void surf3Dwrite(T data,
                 cudaSurfaceObject_t surfObj,
                 int x, int y, int z,
                 boundaryMode = cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the three-dimensional object `surfObj` at byte coordinate x, y, and z.

10.9.1.7 surf1DLayeredread()

```
template<class T>
T surf1DLayeredread(
                 cudaSurfaceObject_t surfObj,
                 int x, int layer,
                 boundaryMode = cudaBoundaryModeTrap);
template<class T>
void surf1DLayeredread(T data,
                 cudaSurfaceObject_t surfObj,
                 int x, int layer,
                 boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array specified by the one-dimensional layered surface object `surfObj` using byte coordinate x and index `layer`.

10.9.1.8 surf1DLayeredwrite()

```
template<class Type>
void surf1DLayeredwrite(T data,
                 cudaSurfaceObject_t surfObj,
                 int x, int layer,
                 boundaryMode = cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the two-dimensional layered surface object `surfObj` at byte coordinate x and index `layer`.

10.9.1.9 surf2DLayeredread()

```
template<class T>
T surf2DLayeredread(
                 cudaSurfaceObject_t surfObj,
                 int x, int y, int layer,
                 boundaryMode = cudaBoundaryModeTrap);
template<class T>
void surf2DLayeredread(T data,
                        cudaSurfaceObject_t surfObj,
                        int x, int y, int layer,
                        boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array specified by the two-dimensional layered surface object `surfObj` using byte coordinate x and y, and index `layer`.

10.9.1.10 surf2DLayeredwrite()

```
template<class T>
void surf2DLayeredwrite(T data,
                        cudaSurfaceObject_t surfObj,
                        int x, int y, int layer,
                        boundaryMode = cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the one-dimensional layered surface object `surfObj` at byte coordinate x and y, and index `layer`.

10.9.1.11 surfCubemapread()

```
template<class T>
T surfCubemapread(
                 cudaSurfaceObject_t surfObj,
                 int x, int y, int face,
                 boundaryMode = cudaBoundaryModeTrap);
template<class T>
void surfCubemapread(T data,
                 cudaSurfaceObject_t surfObj,
                 int x, int y, int face,
                 boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array specified by the cubemap surface object `surfObj` using byte coordinate x and y, and face index face.

10.9.1.12 surfCubemapwrite()

```
template<class T>
void surfCubemapwrite(T data,
                 cudaSurfaceObject_t surfObj,
                 int x, int y, int face,
                 boundaryMode = cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the cubemap object `surfObj` at byte coordinate x and y, and face index face.

10.9.1.13 surfCubemapLayeredread()

```
template<class T>
T surfCubemapLayeredread(
             cudaSurfaceObject_t surfObj,
             int x, int y, int layerFace,
             boundaryMode = cudaBoundaryModeTrap);
template<class T>
void surfCubemapLayeredread(T data,
             cudaSurfaceObject_t surfObj,
             int x, int y, int layerFace,
             boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array specified by the cubemap layered surface object `surfObj` using byte coordinate x and y, and index `layerFace`.

10.9.1.14 surfCubemapLayeredwrite()

```
template<class T>
void surfCubemapLayeredwrite(T data,
             cudaSurfaceObject_t surfObj,
             int x, int y, int layerFace,
             boundaryMode = cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the cubemap layered object `surfObj` at byte coordinate `x` and `y`, and index `layerFace`.

10.10. Read-Only Data Cache Load Function

The read-only data cache load function is only supported by devices of compute capability 5.0 and higher.

```
T __ldg(const T* address);
```

returns the data of type `T` located at address `address`, where `T` is `char`, `signed char`, `short`, `int`, `long`, `long long`unsigned `char`, `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long`, `char2`, `char4`, `short2`, `short4`, `int2`, `int4`, `longlong2uchar2`, `uchar4`, `ushort2`, `ushort4`, `uint2`, `uint4`, `ulonglong2float`, `float2`, `float4`, `double`, or `double2`. With the `cuda_fp16.h` header included, `T` can be `__half` or `__half2`. Similarly, with the `cuda_bf16.h` header included, `T` can also be `__nv_bfloat16` or `__nv_bfloat162`. The operation is cached in the read-only data cache (see *Global Memory*).

10.11. Load Functions Using Cache Hints

These load functions are only supported by devices of compute capability 5.0 and higher.

```
T __ldcg(const T* address);
T __ldca(const T* address);
T __ldcs(const T* address);
T __ldlu(const T* address);
T __ldcv(const T* address);
```

returns the data of type `T` located at address `address`, where `T` is `char`, `signed char`, `short`, `int`, `long`, `long long`unsigned `char`, `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long`, `char2`, `char4`, `short2`, `short4`, `int2`, `int4`, `longlong2uchar2`, `uchar4`, `ushort2`, `ushort4`, `uint2`, `uint4`, `ulonglong2float`, `float2`, `float4`, `double`, or `double2`. With the `cuda_fp16.h` header included, `T` can be `__half` or `__half2`. Similarly, with the `cuda_bf16.h` header included, `T` can also be `__nv_bfloat16` or `__nv_bfloat162`. The operation is using the corresponding cache operator (see PTX ISA)

10.12. Store Functions Using Cache Hints

These store functions are only supported by devices of compute capability 5.0 and higher.

```
void __stwb(T* address, T value);
void __stcg(T* address, T value);
void __stcs(T* address, T value);
void __stwt(T* address, T value);
```

stores the `value` argument of type T to the location at address `address`, where T is `char`, `signed char`, `short`, `int`, `long`, `long long`unsigned `char`, `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long`, `char2`, `char4`, `short2`, `short4`, `int2`, `int4`, `longlong2uchar2`, `uchar4`, `ushort2`, `ushort4`, `uint2`, `uint4`, `ulonglong2float`, `float2`, `float4`, `double`, or `double2`. With the `cuda_fp16.h` header included, T can be `__half` or `__half2`. Similarly, with the `cuda_bf16.h` header included, T can also be `__nv_bfloat16` or `__nv_bfloat162`. The operation is using the corresponding cache operator (see PTX ISA)

10.13. Time Function

```
clock_t clock();
long long int clock64();
```

when executed in device code, returns the value of a per-multiprocessor counter that is incremented every clock cycle. Sampling this counter at the beginning and at the end of a kernel, taking the difference of the two samples, and recording the result per thread provides a measure for each thread of the number of clock cycles taken by the device to completely execute the thread, but not of the number of clock cycles the device actually spent executing thread instructions. The former number is greater than the latter since threads are time sliced.

10.14. Atomic Functions

An atomic function performs a read-modify-write atomic operation on one 32-bit, 64-bit, or 128-bit word residing in global or shared memory. In the case of `float2` or `float4`, the read-modify-write operation is performed on each element of the vector residing in global memory. For example, `atomicAdd()` reads a word at some address in global or shared memory, adds a number to it, and writes the result back to the same address. Atomic functions can only be used in device functions.

The atomic functions described in this section have ordering cuda::memory_order_relaxed and are only atomic at a particular scope:

- Atomic APIs with `_system` suffix (example: `atomicAdd_system`) are atomic at scope `cuda::thread_scope_system` if they meet particular conditions.
- Atomic APIs without a suffix (example: `atomicAdd`) are atomic at scope `cuda::thread_scope_device`.
- Atomic APIs with `_block` suffix (example: `atomicAdd_block`) are atomic at scope `cuda::thread_scope_block`.

In the following example both the CPU and the GPU atomically update an integer value at address `addr`:

```
__global__ void mykernel(int *addr) {
  atomicAdd_system(addr, 10);       // only available on devices with compute
↪capability 6.x
}

void foo() {
  int *addr;
  cudaMallocManaged(&addr, 4);
  *addr = 0;

   mykernel<<<...>>>(addr);
   __sync_fetch_and_add(addr, 10);  // CPU atomic operation
}
```

Note that any atomic operation can be implemented based on `atomicCAS()` (Compare And Swap). For example, `atomicAdd()` for double-precision floating-point numbers is not available on devices with compute capability lower than 6.0 but it can be implemented as follows:

```
#if __CUDA_ARCH__ < 600
__device__ double atomicAdd(double* address, double val)
{
    unsigned long long int* address_as_ull =
                              (unsigned long long int*)address;
    unsigned long long int old = *address_as_ull, assumed;

    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed,
                        __double_as_longlong(val +
                               __longlong_as_double(assumed)));

    // Note: uses integer comparison to avoid hang in case of NaN (since NaN != NaN)
    } while (assumed != old);

    return __longlong_as_double(old);
}
#endif
```

There are system-wide and block-wide variants of the following device-wide atomic APIs, with the following exceptions:

- Devices with compute capability less than 6.0 only support device-wide atomic operations,
- Tegra devices with compute capability less than 7.2 do not support system-wide atomic operations.

CUDA 12.8 and later support CUDA compiler builtin functions for atomic operations with memory order and thread scope. We follows the GNU's atomic built-in function signature with an extra argument of thread scope. We use the following atomic operation memory orders and thread scopes:

```
enum {
   __NV_ATOMIC_RELAXED,
   __NV_ATOMIC_CONSUME,
   __NV_ATOMIC_ACQUIRE,
   __NV_ATOMIC_RELEASE,
   __NV_ATOMIC_ACQ_REL,
```

(continues on next page)

(continued from previous page)

```
    __NV_ATOMIC_SEQ_CST
};

enum {
    __NV_THREAD_SCOPE_THREAD,
    __NV_THREAD_SCOPE_BLOCK,
    __NV_THREAD_SCOPE_CLUSTER,
    __NV_THREAD_SCOPE_DEVICE,
    __NV_THREAD_SCOPE_SYSTEM
};
```

Example:

```
__device__ T __nv_atomic_load_n(T* ptr, int order, int scope = __NV_THREAD_SCOPE_
↪SYSTEM);
```

T can be any integral type that is size of 1, 2, 4, 8 and 16 bytes.

These atomic functions cannot operate on local memory. For example:

```
__device__ void foo() {
    int a = 1; // defined in local memory
    int b;
    __nv_atomic_load(&a, &b, __NV_ATOMIC_RELAXED, __NV_THREAD_SCOPE_SYSTEM);
}
```

These functions must only be used within the block scope of a `__device__` function. For example:

```
__device__ void foo() {
    __shared__ unsigned int u1 = 1;
    __shared__ unsigned int u2 = 2;
    __nv_atomic_load(&u1, &u2, __NV_ATOMIC_RELAXED, __NV_THREAD_SCOPE_SYSTEM);
}
```

And these functions' address cannot be taken. Here are three unsupported examples:

```
// Not permitted to be used in a host function
__host__ void bar() {
    __shared__ unsigned int u1 = 1;
    __shared__ unsigned int u2 = 2;
    __nv_atomic_load(&u1, &u2, __NV_ATOMIC_RELAXED, __NV_THREAD_SCOPE_SYSTEM);
}

// Not permitted to be used as a template default argument.
// The function address cannot be taken.
template<void *F = __nv_atomic_load_n>
class X {
    void *f = F;
};

// Not permitted to be called in a constructor initialization list.
class Y {
    int a;
public:
    __device__ Y(int *b): a(__nv_atomic_load_n(b, __NV_ATOMIC_RELAXED)) {}
};
```

The memory order corresponds to C++ standard atomic operation's memory order. And for thread scope, we follows cuda::thread_scope's definition.

`__NV_ATOMIC_CONSUME` memory order is currently implemented using stronger `__NV_ATOMIC_ACQUIRE` memory order.

`__NV_THREAD_SCOPE_THREAD` thread scope is currently implemented using wider `__NV_THREAD_SCOPE_BLOCK` thread scope.

For the supported data types, please refer to the corresponding section of different atomic operations.

10.14.1. Arithmetic Functions

10.14.1.1 atomicAdd()

```
int atomicAdd(int* address, int val);
unsigned int atomicAdd(unsigned int* address,
                       unsigned int val);
unsigned long long int atomicAdd(unsigned long long int* address,
                                 unsigned long long int val);
float atomicAdd(float* address, float val);
double atomicAdd(double* address, double val);
__half2 atomicAdd(__half2 *address, __half2 val);
__half atomicAdd(__half *address, __half val);
__nv_bfloat162 atomicAdd(__nv_bfloat162 *address, __nv_bfloat162 val);
__nv_bfloat16 atomicAdd(__nv_bfloat16 *address, __nv_bfloat16 val);
float2 atomicAdd(float2* address, float2 val);
float4 atomicAdd(float4* address, float4 val);
```

reads the 16-bit, 32-bit or 64-bit `old` located at the address `address` in global or shared memory, computes `(old + val)`, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns `old`.

The 32-bit floating-point version of `atomicAdd()` is only supported by devices of compute capability 2.x and higher.

The 64-bit floating-point version of `atomicAdd()` is only supported by devices of compute capability 6.x and higher.

The 32-bit `__half2` floating-point version of `atomicAdd()` is only supported by devices of compute capability 6.x and higher. The atomicity of the `__half2` or `__nv_bfloat162` add operation is guaranteed separately for each of the two `__half` or `__nv_bfloat16` elements; the entire `__half2` or `__nv_bfloat162` is not guaranteed to be atomic as a single 32-bit access.

The `float2` and `float4` floating-point vector versions of `atomicAdd()` are only supported by devices of compute capability 9.x and higher. The atomicity of the `float2` or `float4` add operation is guaranteed separately for each of the two or four `float` elements; the entire `float2` or `float4` is not guaranteed to be atomic as a single 64-bit or 128-bit access.

The 16-bit `__half` floating-point version of `atomicAdd()` is only supported by devices of compute capability 7.x and higher.

The 16-bit `__nv_bfloat16` floating-point version of `atomicAdd()` is only supported by devices of compute capability 8.x and higher.

The `float2` and `float4` floating-point vector versions of `atomicAdd()` are only supported by devices of compute capability 9.x and higher.

The `float2` and `float4` floating-point vector versions of `atomicAdd()` are only supported for global memory addresses.

10.14.1.2 atomicSub()

```
int atomicSub(int* address, int val);
unsigned int atomicSub(unsigned int* address,
                       unsigned int val);
```

reads the 32-bit word `old` located at the address `address` in global or shared memory, computes `(old - val)`, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns `old`.

10.14.1.3 atomicExch()

```
int atomicExch(int* address, int val);
unsigned int atomicExch(unsigned int* address,
                        unsigned int val);
unsigned long long int atomicExch(unsigned long long int* address,
                                  unsigned long long int val);
float atomicExch(float* address, float val);
```

reads the 32-bit or 64-bit word `old` located at the address `address` in global or shared memory and stores `val` back to memory at the same address. These two operations are performed in one atomic transaction. The function returns `old`.

```
template<typename T> T atomicExch(T* address, T val);
```

reads the 128-bit word `old` located at the address `address` in global or shared memory and stores `val` back to memory at the same address. These two operations are performed in one atomic transaction. The function returns `old`. The type T must meet the following requirements:

```
sizeof(T) == 16
alignof(T) >= 16
std::is_trivially_copyable<T>::value == true
// for C++03 and older
std::is_default_constructible<T>::value == true
```

So, T must be 128-bit and properly aligned, be trivially copyable, and on C++03 or older, it must also be default constructible.

The 128-bit `atomicExch()` is only supported by devices of compute capability 9.x and higher.

10.14.1.4 atomicMin()

```
int atomicMin(int* address, int val);
unsigned int atomicMin(unsigned int* address,
                       unsigned int val);
unsigned long long int atomicMin(unsigned long long int* address,
                                 unsigned long long int val);
long long int atomicMin(long long int* address,
                        long long int val);
```

reads the 32-bit or 64-bit word `old` located at the address `address` in global or shared memory, computes the minimum of `old` and `val`, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns `old`.

The 64-bit version of `atomicMin()` is only supported by devices of compute capability 5.0 and higher.

10.14.1.5 atomicMax()

```
int atomicMax(int* address, int val);
unsigned int atomicMax(unsigned int* address,
                       unsigned int val);
unsigned long long int atomicMax(unsigned long long int* address,
                                 unsigned long long int val);
long long int atomicMax(long long int* address,
                                 long long int val);
```

reads the 32-bit or 64-bit word `old` located at the address `address` in global or shared memory, computes the maximum of `old` and `val`, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns `old`.

The 64-bit version of `atomicMax()` is only supported by devices of compute capability 5.0 and higher.

10.14.1.6 atomicInc()

```
unsigned int atomicInc(unsigned int* address,
                       unsigned int val);
```

reads the 32-bit word `old` located at the address `address` in global or shared memory, computes `((old >= val) ? 0 : (old+1))`, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns `old`.

10.14.1.7 atomicDec()

```
unsigned int atomicDec(unsigned int* address,
                       unsigned int val);
```

reads the 32-bit word `old` located at the address `address` in global or shared memory, computes `(((old == 0) || (old > val)) ? val : (old-1) )`, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns `old`.

10.14.1.8 atomicCAS()

```
int atomicCAS(int* address, int compare, int val);
unsigned int atomicCAS(unsigned int* address,
                       unsigned int compare,
                       unsigned int val);
unsigned long long int atomicCAS(unsigned long long int* address,
                                 unsigned long long int compare,
                                 unsigned long long int val);
unsigned short int atomicCAS(unsigned short int *address,
```

(continues on next page)

(continued from previous page)

```
                                unsigned short int compare,
                                unsigned short int val);
```

reads the 16-bit, 32-bit or 64-bit word `old` located at the address `address` in global or shared memory, computes `(old == compare ? val : old)`, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns `old` (Compare And Swap).

```
template<typename T> T atomicCAS(T* address, T compare, T val);
```

reads the 128-bit word `old` located at the address `address` in global or shared memory, computes `(old == compare ? val : old)`, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns `old` (Compare And Swap). The type T must meet the following requirements:

```
sizeof(T) == 16
alignof(T) >= 16
std::is_trivially_copyable<T>::value == true
// for C++03 and older
std::is_default_constructible<T>::value == true
```

So, T must be 128-bit and properly aligned, be trivially copyable, and on C++03 or older, it must also be default constructible.

The 128-bit `atomicCAS()` is only supported by devices of compute capability 9.x and higher.

10.14.1.9 __nv_atomic_exchange()

```
__device__ void __nv_atomic_exchange(T* ptr, T* val, T *ret, int order, int scope = __
↪NV_THREAD_SCOPE_SYSTEM);
```

This atomic function is introduced in CUDA 12.8. It reads the value where `ptr` points to and stores the value to where `ret` points to. And it reads the value where `val` points to and stores the value to where `ptr` points to.

This is a generic atomic exchange, which means that T can be any data type that is size of 4, 8 or 16 bytes.

The atomic operation with memory order and thread scope is supported on the architecture `sm_60` and higher.

16-byte data type is supported on the architecture `sm_90` and higher.

The thread scope of `cluster` is supported on the architecture `sm_90` and higher.

The arguments `order` and `scope` need to be integer literals, i.e., the arguments cannot be variables.

10.14.1.10 __nv_atomic_exchange_n()

```
__device__ T __nv_atomic_exchange_n(T* ptr, T val, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
```

This atomic function is introduced in CUDA 12.8. It reads the value where `ptr` points to and use this value as the return value. And it stores `val` to where `ptr` points to.

This is a non-generic atomic exchange, which means that T can only be an integral type that is size of 4, 8 or 16 bytes.

The atomic operation with memory order and thread scope is supported on the architecture `sm_60` and higher.

16-byte data type is supported on the architecture `sm_90` and higher.

The thread scope of `cluster` is supported on the architecture `sm_90` and higher.

The arguments `order` and `scope` need to be integer literals, i.e., the arguments cannot be variables.

10.14.1.11 __nv_atomic_compare_exchange()

```
__device__ bool __nv_atomic_compare_exchange (T* ptr, T* expected, T* desired, bool
↪weak, int success_order, int failure_order, int scope = __NV_THREAD_SCOPE_SYSTEM);
```

This atomic function is introduced in CUDA 12.8. It reads the value where `ptr` points to and compare it with the value where `expected` points to. If they are equal, the return value is `true` and the value where `desired` points to is stored to where `ptr` points to. Otherwise, it returns `false` and the value where `ptr` points to is stored to where `expected` points to. The parameter `weak` is ignored and it picks the stronger memory order between `success_order` and `failure_order` to execute the compare-and-exchange operation.

This is a generic atomic compare-and-exchange, which means that T can be any data type that is size of 2, 4, 8 or 16 bytes.

The atomic operation with memory order and thread scope is supported on the architecture `sm_60` and higher.

16-byte data type is supported on the architecture `sm_90` and higher.

2-byte data type is supported on the architecture `sm_70` and higher.

The thread scope of `cluster` is supported on the architecture `sm_90` and higher.

The arguments `order` and `scope` need to be integer literals, i.e., the arguments cannot be variables.

10.14.1.12 __nv_atomic_compare_exchange_n()

```
__device__ bool __nv_atomic_compare_exchange_n (T* ptr, T* expected, T desired, bool
↪weak, int success_order, int failure_order, int scope = __NV_THREAD_SCOPE_SYSTEM);
```

This atomic function is introduced in CUDA 12.8. It reads the value where `ptr` points to and compare it with the value where `expected` points to. If they are equal, the return value is `true` and `desired` is stored to where `ptr` points to. Otherwise, it returns `false` and the value where `ptr` points to is stored to where `expected` points to. The parameter `weak` is ignored and it picks the stronger memory order between `success_order` and `failure_order` to execute the compare-and-exchange operation.

This is a non-generic atomic compare-and-exchange, which means that T can only be an integral type that is size of 2, 4, 8 or 16 bytes.

The atomic operation with memory order and thread scope is supported on the architecture `sm_60` and higher.

16-byte data type is supported on the architecture `sm_90` and higher.

2-byte data type is supported on the architecture `sm_70` and higher.

The thread scope of `cluster` is supported on the architecture `sm_90` and higher.

The arguments `order` and `scope` need to be integer literals, i.e., the arguments cannot be variables.

10.14.1.13 __nv_atomic_fetch_add() and __nv_atomic_add()

```
__device__ T __nv_atomic_fetch_add (T* ptr, T val, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
__device__ void __nv_atomic_add (T* ptr, T val, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
```

These two atomic functions are introduced in CUDA 12.8. It reads the value where `ptr` points to, adds with `val`, and stores the result back to where `ptr` points to. `__nv_atomic_fetch_add` returns the old value where `ptr` points to. `__nv_atomic_add` does not have return value.

T can only be `unsigned int`, `int`, `unsigned long long`, `float` or `double`.

The atomic operation with memory order and thread scope is supported on the architecture `sm_60` and higher.

The thread scope of `cluster` is supported on the architecture `sm_90` and higher.

The arguments `order` and `scope` need to be integer literals, i.e., the arguments cannot be variables.

10.14.1.14 __nv_atomic_fetch_sub() and __nv_atomic_sub()

```
__device__ T __nv_atomic_fetch_sub (T* ptr, T val, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
__device__ void __nv_atomic_sub (T* ptr, T val, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
```

These two atomic functions are introduced in CUDA 12.8. It reads the value where `ptr` points to, subtracts with `val`, and stores the result back to where `ptr` points to. `__nv_atomic_fetch_sub` returns the old value where `ptr` points to. `__nv_atomic_sub` does not have return value.

T can only be `unsigned int`, `int`, `unsigned long long`, `float` or `double`.

The atomic operation with memory order and thread scope is supported on the architecture `sm_60` and higher.

The thread scope of `cluster` is supported on the architecture `sm_90` and higher.

The arguments `order` and `scope` need to be integer literals, i.e., the arguments cannot be variables.

10.14.1.15 __nv_atomic_fetch_min() and __nv_atomic_min()

```
__device__ T __nv_atomic_fetch_min (T* ptr, T val, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
__device__ void __nv_atomic_min (T* ptr, T val, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
```

These two atomic functions are introduced in CUDA 12.8. It reads the value where `ptr` points to, compares with `val`, and stores the smaller value back to where `ptr` points to. `__nv_atomic_fetch_min` returns the old value where `ptr` points to. `__nv_atomic_min` does not have return value.

T can only be `unsigned int`, `int`, `unsigned long long` or `long long`.

The atomic operation with memory order and thread scope is supported on the architecture `sm_60` and higher.

The thread scope of `cluster` is supported on the architecture `sm_90` and higher.

The arguments `order` and `scope` need to be integer literals, i.e., the arguments cannot be variables.

10.14.1.16 __nv_atomic_fetch_max() and __nv_atomic_max()

```
__device__ T __nv_atomic_fetch_max (T* ptr, T val, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
__device__ void __nv_atomic_max (T* ptr, T val, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
```

These two atomic functions are introduced in CUDA 12.8. It reads the value where `ptr` points to, compares with `val`, and stores the bigger value back to where `ptr` points to. `__nv_atomic_fetch_max` returns the old value where `ptr` points to. `__nv_atomic_max` does not have return value.

T can only be `unsigned int`, `int`, `unsigned long long` or `long long`.

The atomic operation with memory order and thread scope is supported on the architecture `sm_60` and higher.

The thread scope of `cluster` is supported on the architecture `sm_90` and higher.

The arguments `order` and `scope` need to be integer literals, i.e., the arguments cannot be variables.

10.14.2. Bitwise Functions

10.14.2.1 atomicAnd()

```
int atomicAnd(int* address, int val);
unsigned int atomicAnd(unsigned int* address,
                       unsigned int val);
unsigned long long int atomicAnd(unsigned long long int* address,
                                 unsigned long long int val);
```

reads the 32-bit or 64-bit word `old` located at the address `address` in global or shared memory, computes (`old & val`), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns `old`.

The 64-bit version of `atomicAnd()` is only supported by devices of compute capability 5.0 and higher.

10.14.2.2 atomicOr()

```
int atomicOr(int* address, int val);
unsigned int atomicOr(unsigned int* address,
                      unsigned int val);
unsigned long long int atomicOr(unsigned long long int* address,
                                unsigned long long int val);
```

reads the 32-bit or 64-bit word `old` located at the address `address` in global or shared memory, computes `(old | val)`, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns `old`.

The 64-bit version of `atomicOr()` is only supported by devices of compute capability 5.0 and higher.

10.14.2.3 atomicXor()

```
int atomicXor(int* address, int val);
unsigned int atomicXor(unsigned int* address,
                       unsigned int val);
unsigned long long int atomicXor(unsigned long long int* address,
                                 unsigned long long int val);
```

reads the 32-bit or 64-bit word `old` located at the address `address` in global or shared memory, computes `(old ^ val)`, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns `old`.

The 64-bit version of `atomicXor()` is only supported by devices of compute capability 5.0 and higher.

10.14.2.4 __nv_atomic_fetch_or() and __nv_atomic_or()

```
__device__ T __nv_atomic_fetch_or (T* ptr, T val, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
__device__ void __nv_atomic_or (T* ptr, T val, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
```

These two atomic functions are introduced in CUDA 12.8. It reads the value where `ptr` points to, `or` with `val`, and stores the result back to where `ptr` points to. `__nv_atomic_fetch_or` returns the old value where `ptr` points to. `__nv_atomic_or` does not have return value.

T can only be an integral type that is size of 4 or 8 bytes.

The atomic operation with memory order and thread scope is supported on the architecture `sm_60` and higher.

The thread scope of `cluster` is supported on the architecture `sm_90` and higher.

The arguments `order` and `scope` need to be integer literals, i.e., the arguments cannot be variables.

10.14.2.5 __nv_atomic_fetch_xor() and __nv_atomic_xor()

```
__device__ T __nv_atomic_fetch_xor (T* ptr, T val, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
__device__ void __nv_atomic_xor (T* ptr, T val, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
```

These two atomic functions are introduced in CUDA 12.8. It reads the value where `ptr` points to, `xor` with `val`, and stores the result back to where `ptr` points to. `__nv_atomic_fetch_xor` returns the old value where `ptr` points to. `__nv_atomic_xor` does not have return value.

T can only be an integral type that is size of 4 or 8 bytes.

The atomic operation with memory order and thread scope is supported on the architecture `sm_60` and higher.

The thread scope of `cluster` is supported on the architecture `sm_90` and higher.

The arguments `order` and `scope` need to be integer literals, i.e., the arguments cannot be variables.

10.14.2.6 __nv_atomic_fetch_and() and __nv_atomic_and()

```
__device__ T __nv_atomic_fetch_and (T* ptr, T val, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
__device__ void __nv_atomic_and (T* ptr, T val, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
```

These two atomic functions are introduced in CUDA 12.8. It reads the value where `ptr` points to, `and` with `val`, and stores the result back to where `ptr` points to. `__nv_atomic_fetch_and` returns the old value where `ptr` points to. `__nv_atomic_and` does not have return value.

T can only be an integral type that is size of 4 or 8 bytes.

The atomic operation with memory order and thread scope is supported on the architecture `sm_60` and higher.

The thread scope of `cluster` is supported on the architecture `sm_90` and higher.

The arguments `order` and `scope` need to be integer literals, i.e., the arguments cannot be variables.

10.14.3. Other atomic functions

10.14.3.1 __nv_atomic_load()

```
__device__ void __nv_atomic_load(T* ptr, T* ret, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
```

This atomic function is introduced in CUDA 12.8. It loads the value where `ptr` points to and writes the value to where `ret` points to.

This is a generic atomic load, which means that T can be any data type that is size of 1, 2, 4, 8 or 16 bytes.

The atomic operation with memory order and thread scope is supported on the architecture `sm_60` and higher.

16-byte data type is supported on the architecture `sm_70` and higher.

The thread scope of `cluster` is supported on the architecture `sm_90` and higher.

The arguments `order` and `scope` need to be integer literals, i.e., the arguments cannot be variables. `order` cannot be `__NV_ATOMIC_RELEASE` or `__NV_ATOMIC_ACQ_REL`.

10.14.3.2 __nv_atomic_load_n()

```
__device__ T __nv_atomic_load_n(T* ptr, int order, int scope = __NV_THREAD_SCOPE_
↪SYSTEM);
```

This atomic function is introduced in CUDA 12.8. It loads the value where `ptr` points to and returns this value.

This is a non-generic atomic load, which means that T can only be an integral type that is size of 1, 2, 4, 8 or 16 bytes.

The atomic operation with memory order and thread scope is supported on the architecture `sm_60` and higher.

16-byte data type is supported on the architecture `sm_70` and higher.

The thread scope of `cluster` is supported on the architecture `sm_90` and higher.

The arguments `order` and `scope` need to be integer literals, i.e., the arguments cannot be variables. `order` cannot be `__NV_ATOMIC_RELEASE` or `__NV_ATOMIC_ACQ_REL`.

10.14.3.3 __nv_atomic_store()

```
__device__ void __nv_atomic_store(T* ptr, T* val, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
```

This atomic function is introduced in CUDA 12.8. It reads the value where `val` points to and stores to where `ptr` points to.

This is a generic atomic load, which means that T can be any data type that is size of 1, 2, 4, 8 or 16 bytes.

The atomic operation with memory order and thread scope is supported on the architecture `sm_60` and higher.

16-byte data type is supported on the architecture `sm_70` and higher.

The thread scope of `cluster` is supported on the architecture `sm_90` and higher.

The arguments `order` and `scope` need to be integer literals, i.e., the arguments cannot be variables. `order` cannot be `__NV_ATOMIC_CONSUME`, `__NV_ATOMIC_ACQUIRE` or `__NV_ATOMIC_ACQ_REL`.

10.14.3.4 __nv_atomic_store_n()

```
__device__ void __nv_atomic_store_n(T* ptr, T val, int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
```

This atomic function is introduced in CUDA 12.8. It stores `val` to where `ptr` points to.

This is a non-generic atomic load, which means that T can only be an integral type that is size of 1, 2, 4, 8 or 16 bytes.

The atomic operation with memory order and thread scope is supported on the architecture `sm_60` and higher.

16-byte data type is supported on the architecture `sm_70` and higher.

The thread scope of `cluster` is supported on the architecture `sm_90` and higher.

The arguments `order` and `scope` need to be integer literals, i.e., the arguments cannot be variables. `order` cannot be `__NV_ATOMIC_CONSUME`, `__NV_ATOMIC_ACQUIRE` or `__NV_ATOMIC_ACQ_REL`.

10.14.3.5 __nv_atomic_thread_fence()

```
__device__ void __nv_atomic_thread_fence (int order, int scope = __NV_THREAD_SCOPE_
↪SYSTEM);
```

This atomic function establishes an ordering between memory accesses requested by this thread based on the specified memory order. And the thread scope parameter specifies the set of threads that may observe the ordering effect of this operation.

The thread scope of `cluster` is supported on the architecture `sm_90` and higher.

The arguments `order` and `scope` need to be integer literals, i.e., the arguments cannot be variables.

10.15. Address Space Predicate Functions

The functions described in this section have unspecified behavior if the argument is a null pointer.

10.15.1. __isGlobal()

```
__device__ unsigned int __isGlobal(const void *ptr);
```

Returns 1 if `ptr` contains the generic address of an object in global memory space, otherwise returns 0.

10.15.2. __isShared()

```
__device__ unsigned int __isShared(const void *ptr);
```

Returns 1 if `ptr` contains the generic address of an object in shared memory space, otherwise returns 0.

10.15.3. __isConstant()

```
__device__ unsigned int __isConstant(const void *ptr);
```

Returns 1 if `ptr` contains the generic address of an object in constant memory space, otherwise returns 0.

10.15.4. __isGridConstant()

```
__device__ unsigned int __isGridConstant(const void *ptr);
```

Returns 1 if `ptr` contains the generic address of a kernel parameter annotated with `__grid_constant__`, otherwise returns 0. Only supported for compute architectures greater than or equal to 7.x or later.

10.15.5. __isLocal()

```
__device__ unsigned int __isLocal(const void *ptr);
```

Returns 1 if `ptr` contains the generic address of an object in local memory space, otherwise returns 0.

10.16. Address Space Conversion Functions

CUDA C++ pointers (T*) can access CUDA C++ objects independently of where these objects are stored. For example, an `int*` can access `int` objects independently of whether they are stored in global or shared memory.

The Address Space Conversion Functions below enable converting CUDA C++ pointers from and to other representations. This is required, among others, to interoperate with certain PTX instructions, or to exploit properties of these other representations for performance optimizations.

As an example of interoperating with certain PTX instructions, an `ld.shared.u32 r0, [addr];` PTX instruction expects `addr` to refer to the `shared` space. A CUDA C++ program with a CUDA C++ `uint32_t*` pointer to an object in `__shared__` memory, needs to convert this pointer to the `shared` space before passing it to such a PTX instruction by calling `__cvta_generic_to_shared` as follows:

```
__shared__ uint32_t x;
x = 42;
void* p = &x;
size_t sp = __cvta_generic_to_shared(p);
uint32_t o;
asm volatile("ld.shared.u32 %0, [%1];" : "=r"(o) : "l"(sp) : "memory");
assert(o == 42);
```

A common program optimization that exploits properties of these other address representations is reducing data-structure size by leveraging that the address ranges of shared, local, and const spaces is smaller than 32-bit, which allows programs to store 32-bit addresses instead of 64-bit pointers. To obtain the 32-bit integer representation of these addresses, it suffices to truncate it by performing an unsigned 64-bit integer to unsigned 32-bit integer cast:

```
__shared__ int x;
void* p = &x;
uint32_t smem32 = __cvta_generic_to_shared(p);
```

To obtain a generic address from such a 32-bit integer representation, it suffices to zero-extend the address back to an unsigned 64-bit integer before calling the corresponding address space conversion function:

```
size_t smem64 = smem32;
void* q = __cvta_shared_to_generic(smem64);
assert(p == q);
```

A roundtrip from an input generic space pointer to its 32-bit integer representation and back to an output generic space pointer is guaranteed to return an output pointer that is equivalent to the input pointer of the roundtrip for the spaces listed above.

10.16.1. __cvta_generic_to_global()

```
__device__ size_t __cvta_generic_to_global(const void *ptr);
```

Returns the result of executing the *PTX* `cvta.to.global` instruction on the generic address denoted by `ptr`.

10.16.2. __cvta_generic_to_shared()

```
__device__ size_t __cvta_generic_to_shared(const void *ptr);
```

Returns the result of executing the *PTX* `cvta.to.shared` instruction on the generic address denoted by `ptr`.

10.16.3. __cvta_generic_to_constant()

```
__device__ size_t __cvta_generic_to_constant(const void *ptr);
```

Returns the result of executing the *PTX* `cvta.to.const` instruction on the generic address denoted by `ptr`.

10.16.4. __cvta_generic_to_local()

```
__device__ size_t __cvta_generic_to_local(const void *ptr);
```

Returns the result of executing the *PTX* `cvta.to.local` instruction on the generic address denoted by `ptr`.

10.16.5. __cvta_global_to_generic()

```
__device__ void * __cvta_global_to_generic(size_t rawbits);
```

Returns the generic pointer obtained by executing the *PTX* `cvta.global` instruction on the value provided by `rawbits`.

10.16.6. __cvta_shared_to_generic()

```
__device__ void * __cvta_shared_to_generic(size_t rawbits);
```

Returns the generic pointer obtained by executing the *PTX* `cvta.shared` instruction on the value provided by `rawbits`.

10.16.7. __cvta_constant_to_generic()

```
__device__ void * __cvta_constant_to_generic(size_t rawbits);
```

Returns the generic pointer obtained by executing the *PTX* `cvta.const` instruction on the value provided by `rawbits`.

10.16.8. __cvta_local_to_generic()

```
__device__ void * __cvta_local_to_generic(size_t rawbits);
```

Returns the generic pointer obtained by executing the *PTX* `cvta.local` instruction on the value provided by `rawbits`.

10.17. Alloca Function

10.17.1. Synopsis

```
__host__ __device__ void * alloca(size_t size);
```

10.17.2. Description

The `alloca()` function allocates `size` bytes of memory in the stack frame of the caller. The returned value is a pointer to allocated memory, the beginning of the memory is 16 bytes aligned when the function is invoked from device code. The allocated memory is automatically freed when the caller to `alloca()` is returned.

Note: On Windows platform, `<malloc.h>` must be included before using `alloca()`. Using `alloca()` may cause the stack to overflow, user needs to adjust stack size accordingly.

It is supported with compute capability 5.2 or higher.

10.17.3. Example

```
__device__ void foo(unsigned int num) {
    int4 *ptr = (int4 *)alloca(num * sizeof(int4));
    // use of ptr
    ...
}
```

10.18. Compiler Optimization Hint Functions

The functions described in this section can be used to provide additional information to the compiler optimizer.

10.18.1. __builtin_assume_aligned()

```
void * __builtin_assume_aligned (const void *exp, size_t align)
```

Allows the compiler to assume that the argument pointer is aligned to at least `align` bytes, and returns the argument pointer.

Example:

```
void *res = __builtin_assume_aligned(ptr, 32); // compiler can assume 'res' is
                                               // at least 32-byte aligned
```

Three parameter version:

```
void * __builtin_assume_aligned (const void *exp, size_t align,
                                 <integral type> offset)
```

Allows the compiler to assume that `(char *)exp - offset` is aligned to at least `align` bytes, and returns the argument pointer.

Example:

```
void *res = __builtin_assume_aligned(ptr, 32, 8); // compiler can assume
                                                  // '(char *)res - 8' is
                                                  // at least 32-byte aligned.
```

10.18.2. __builtin_assume()

```
void __builtin_assume(bool exp)
```

Allows the compiler to assume that the Boolean argument is true. If the argument is not true at run time, then the behavior is undefined. Note that if the argument has side effects, the behavior is unspecified.

Example:

```
 __device__ int get(int *ptr, int idx) {
   __builtin_assume(idx <= 2);
   return ptr[idx];
}
```

10.18.3. __assume()

```
void __assume(bool exp)
```

Allows the compiler to assume that the Boolean argument is true. If the argument is not true at run time, then the behavior is undefined. Note that if the argument has side effects, the behavior is unspecified.

Example:

```
__device__ int get(int *ptr, int idx) {
  __assume(idx <= 2);
  return ptr[idx];
}
```

10.18.4. __builtin_expect()

```
long __builtin_expect (long exp, long c)
```

Indicates to the compiler that it is expected that `exp == c`, and returns the value of `exp`. Typically used to indicate branch prediction information to the compiler.

Example:

```
// indicate to the compiler that likely "var == 0",
// so the body of the if-block is unlikely to be
// executed at run time.
if (__builtin_expect (var, 0))
  doit ();
```

10.18.5. __builtin_unreachable()

```
void __builtin_unreachable(void)
```

Indicates to the compiler that control flow never reaches the point where this function is being called from. The program has undefined behavior if the control flow does actually reach this point at run time.

Example:

```
// indicates to the compiler that the default case label is never reached.
switch (in) {
case 1: return 4;
case 2: return 10;
default: __builtin_unreachable();
}
```

10.18.6. Restrictions

`__assume()` is only supported when using `cl.exe` host compiler. The other functions are supported on all platforms, subject to the following restrictions:

- If the host compiler supports the function, the function can be invoked from anywhere in translation unit.
- Otherwise, the function must be invoked from within the body of a `__device__`/`__global__`function, or only when the `__CUDA_ARCH__` macro is defined[5].

10.19. Warp Vote Functions

```
int __all_sync(unsigned mask, int predicate);
int __any_sync(unsigned mask, int predicate);
unsigned __ballot_sync(unsigned mask, int predicate);
unsigned __activemask();
```

Deprecation notice: `__any`, `__all`, and `__ballot` have been deprecated in CUDA 9.0 for all devices.

Removal notice: When targeting devices with compute capability 7.x or higher, `__any`, `__all`, and `__ballot` are no longer available and their sync variants should be used instead.

The warp vote functions allow the threads of a given *warp* to perform a reduction-and-broadcast operation. These functions take as input an integer `predicate` from each thread in the warp and compare those values with zero. The results of the comparisons are combined (reduced) across the *active* threads of the warp in one of the following ways, broadcasting a single return value to each participating thread:

`__all_sync(unsigned mask, predicate)`:
Evaluate `predicate` for all non-exited threads in `mask` and return non-zero if and only if `predicate` evaluates to non-zero for all of them.

`__any_sync(unsigned mask, predicate)`:
Evaluate `predicate` for all non-exited threads in `mask` and return non-zero if and only if `predicate` evaluates to non-zero for any of them.

`__ballot_sync(unsigned mask, predicate)`:
Evaluate `predicate` for all non-exited threads in `mask` and return an integer whose Nth bit is set if and only if `predicate` evaluates to non-zero for the Nth thread of the warp and the Nth thread is active.

`__activemask()`:
Returns a 32-bit integer mask of all currently active threads in the calling warp. The Nth bit is set if the Nth lane in the warp is active when `__activemask()` is called. *Inactive* threads are represented by 0 bits in the returned mask. Threads which have exited the program are always marked as inactive. Note that threads that are convergent at an `__activemask()` call are not guaranteed to be convergent at subsequent instructions unless those instructions are synchronizing warp-builtin functions.

For `__all_sync`, `__any_sync`, and `__ballot_sync`, a mask must be passed that specifies the threads participating in the call. A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware.

[5] The intent is to prevent the host compiler from encountering the call to the function if the host compiler does not support it.

Each calling thread must have its own bit set in the mask and all non-exited threads named in mask must execute the same intrinsic with the same mask, or the result is undefined.

These intrinsics do not imply a memory barrier. They do not guarantee any memory ordering.

10.20. Warp Match Functions

`__match_any_sync` and `__match_all_sync` perform a broadcast-and-compare operation of a variable between threads within a *warp*.

Supported by devices of compute capability 7.x or higher.

10.20.1. Synopsis

```
unsigned int __match_any_sync(unsigned mask, T value);
unsigned int __match_all_sync(unsigned mask, T value, int *pred);
```

T can be `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float` or `double`.

10.20.2. Description

The `__match_sync()` intrinsics permit a broadcast-and-compare of a value `value` across threads in a warp after synchronizing threads named in `mask`.

`__match_any_sync`
: Returns mask of threads that have same value of `value` in `mask`

`__match_all_sync`
: Returns `mask` if all threads in `mask` have the same value for `value`; otherwise 0 is returned. Predicate `pred` is set to true if all threads in `mask` have the same value of `value`; otherwise the predicate is set to false.

The new `*_sync` match intrinsics take in a mask indicating the threads participating in the call. A bit, representing the thread's lane id, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware. Each calling thread must have its own bit set in the mask and all non-exited threads named in mask must execute the same intrinsic with the same mask, or the result is undefined.

These intrinsics do not imply a memory barrier. They do not guarantee any memory ordering.

10.21. Warp Reduce Functions

The `__reduce_sync(unsigned mask, T value)` intrinsics perform a reduction operation on the data provided in `value` after synchronizing threads named in `mask`. T can be unsigned or signed for {add, min, max} and unsigned only for {and, or, xor} operations.

Supported by devices of compute capability 8.x or higher.

10.21.1. Synopsis

```
// add/min/max
unsigned __reduce_add_sync(unsigned mask, unsigned value);
unsigned __reduce_min_sync(unsigned mask, unsigned value);
unsigned __reduce_max_sync(unsigned mask, unsigned value);
int __reduce_add_sync(unsigned mask, int value);
int __reduce_min_sync(unsigned mask, int value);
int __reduce_max_sync(unsigned mask, int value);

// and/or/xor
unsigned __reduce_and_sync(unsigned mask, unsigned value);
unsigned __reduce_or_sync(unsigned mask, unsigned value);
unsigned __reduce_xor_sync(unsigned mask, unsigned value);
```

10.21.2. Description

`__reduce_add_sync`, `__reduce_min_sync`, `__reduce_max_sync`
: Returns the result of applying an arithmetic add, min, or max reduction operation on the values provided in `value` by each thread named in `mask`.

`__reduce_and_sync`, `__reduce_or_sync`, `__reduce_xor_sync`
: Returns the result of applying a logical AND, OR, or XOR reduction operation on the values provided in `value` by each thread named in `mask`.

The `mask` indicates the threads participating in the call. A bit, representing the thread's lane id, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware. Each calling thread must have its own bit set in the mask and all non-exited threads named in mask must execute the same intrinsic with the same mask, or the result is undefined.

These intrinsics do not imply a memory barrier. They do not guarantee any memory ordering.

10.22. Warp Shuffle Functions

`__shfl_sync`, `__shfl_up_sync`, `__shfl_down_sync`, and `__shfl_xor_sync` exchange a variable between threads within a *warp*.

Supported by devices of compute capability 5.0 or higher.

Deprecation Notice: `__shfl`, `__shfl_up`, `__shfl_down`, and `__shfl_xor` have been deprecated in CUDA 9.0 for all devices.

Removal Notice: When targeting devices with compute capability 7.x or higher, `__shfl`, `__shfl_up`, `__shfl_down`, and `__shfl_xor` are no longer available and their sync variants should be used instead.

10.22.1. Synopsis

```
T __shfl_sync(unsigned mask, T var, int srcLane, int width=warpSize);
T __shfl_up_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);
T __shfl_down_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);
T __shfl_xor_sync(unsigned mask, T var, int laneMask, int width=warpSize);
```

T can be `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float` or `double`. With the `cuda_fp16.h` header included, T can also be `__half` or `__half2`. Similarly, with the `cuda_bf16.h` header included, T can also be `__nv_bfloat16` or `__nv_bfloat162`.

10.22.2. Description

The `__shfl_sync()` intrinsics permit exchanging of a variable between threads within a warp without use of shared memory. The exchange occurs simultaneously for all *active* threads within the warp (and named in `mask`), moving 4 or 8 bytes of data per thread depending on the type.

Threads within a warp are referred to as *lanes*, and may have an index between 0 and `warpSize-1` (inclusive). Four source-lane addressing modes are supported:

`__shfl_sync()`
Direct copy from indexed lane

`__shfl_up_sync()`
Copy from a lane with lower ID relative to caller

`__shfl_down_sync()`
Copy from a lane with higher ID relative to caller

`__shfl_xor_sync()`
Copy from a lane based on bitwise XOR of own lane ID

Threads may only read data from another thread which is actively participating in the `__shfl_sync()` command. If the target thread is *inactive*, the retrieved value is undefined.

All of the `__shfl_sync()` intrinsics take an optional `width` parameter which alters the behavior of the intrinsic. `width` must have a value which is a power of two in the range [1, warpSize] (i.e., 1, 2, 4, 8, 16 or 32). Results are undefined for other values.

`__shfl_sync()` returns the value of `var` held by the thread whose ID is given by `srcLane`. If width is less than `warpSize` then each subsection of the warp behaves as a separate entity with a starting

logical lane ID of 0. If `srcLane` is outside the range `[0:width-1]`, the value returned corresponds to the value of var held by the `srcLane modulo width` (i.e. within the same subsection).

`__shfl_up_sync()` calculates a source lane ID by subtracting `delta` from the caller's lane ID. The value of `var` held by the resulting lane ID is returned: in effect, `var` is shifted up the warp by `delta` lanes. If width is less than `warpSize` then each subsection of the warp behaves as a separate entity with a starting logical lane ID of 0. The source lane index will not wrap around the value of `width`, so effectively the lower `delta` lanes will be unchanged.

`__shfl_down_sync()` calculates a source lane ID by adding `delta` to the caller's lane ID. The value of `var` held by the resulting lane ID is returned: this has the effect of shifting `var` down the warp by `delta` lanes. If width is less than `warpSize` then each subsection of the warp behaves as a separate entity with a starting logical lane ID of 0. As for `__shfl_up_sync()`, the ID number of the source lane will not wrap around the value of width and so the upper `delta` lanes will remain unchanged.

`__shfl_xor_sync()` calculates a source line ID by performing a bitwise XOR of the caller's lane ID with `laneMask`: the value of `var` held by the resulting lane ID is returned. If `width` is less than `warpSize` then each group of `width` consecutive threads are able to access elements from earlier groups of threads, however if they attempt to access elements from later groups of threads their own value of `var` will be returned. This mode implements a butterfly addressing pattern such as is used in tree reduction and broadcast.

The new `*_sync` shfl intrinsics take in a mask indicating the threads participating in the call. A bit, representing the thread's lane id, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware. Each calling thread must have its own bit set in the mask and all non-exited threads named in mask must execute the same intrinsic with the same mask, or the result is undefined.

Threads may only read data from another thread which is actively participating in the `__shfl_sync()` command. If the target thread is inactive, the retrieved value is undefined.

These intrinsics do not imply a memory barrier. They do not guarantee any memory ordering.

10.22.3. Examples

10.22.3.1 Broadcast of a single value across a warp

```
#include <stdio.h>

__global__ void bcast(int arg) {
    int laneId = threadIdx.x & 0x1f;
    int value;
    if (laneId == 0)        // Note unused variable for
        value = arg;        // all threads except lane 0
    value = __shfl_sync(0xffffffff, value, 0);   // Synchronize all threads in warp,
↪and get "value" from lane 0
    if (value != arg)
        printf("Thread %d failed.\n", threadIdx.x);
}

int main() {
    bcast<<< 1, 32 >>>(1234);
    cudaDeviceSynchronize();

    return 0;
}
```

10.22.3.2 Inclusive plus-scan across sub-partitions of 8 threads

```
#include <stdio.h>

__global__ void scan4() {
    int laneId = threadIdx.x & 0x1f;
    // Seed sample starting value (inverse of lane ID)
    int value = 31 - laneId;

    // Loop to accumulate scan within my partition.
    // Scan requires log2(n) == 3 steps for 8 threads
    // It works by an accumulated sum up the warp
    // by 1, 2, 4, 8 etc. steps.
    for (int i=1; i<=4; i*=2) {
        // We do the __shfl_sync unconditionally so that we
        // can read even from threads which won't do a
        // sum, and then conditionally assign the result.
        int n = __shfl_up_sync(0xffffffff, value, i, 8);
        if ((laneId & 7) >= i)
            value += n;
    }

    printf("Thread %d final value = %d\n", threadIdx.x, value);
}

int main() {
    scan4<<< 1, 32 >>>();
    cudaDeviceSynchronize();

    return 0;
}
```

10.22.3.3 Reduction across a warp

```
#include <stdio.h>

__global__ void warpReduce() {
    int laneId = threadIdx.x & 0x1f;
    // Seed starting value as inverse lane ID
    int value = 31 - laneId;

    // Use XOR mode to perform butterfly reduction
    for (int i=16; i>=1; i/=2)
        value += __shfl_xor_sync(0xffffffff, value, i, 32);

    // "value" now contains the sum across all threads
    printf("Thread %d final value = %d\n", threadIdx.x, value);
}

int main() {
    warpReduce<<< 1, 32 >>>();
    cudaDeviceSynchronize();

    return 0;
}
```

10.23. Nanosleep Function

10.23.1. Synopsis

```
void __nanosleep(unsigned ns);
```

10.23.2. Description

`__nanosleep(ns)` suspends the thread for a sleep duration of approximately `ns` nanoseconds. The maximum sleep duration is approximately 1 millisecond.

It is supported with compute capability 7.0 or higher.

10.23.3. Example

The following code implements a mutex with exponential back-off.

```
__device__ void mutex_lock(unsigned int *mutex) {
    unsigned int ns = 8;
    while (atomicCAS(mutex, 0, 1) == 1) {
        __nanosleep(ns);
        if (ns < 256) {
            ns *= 2;
        }
    }
}

__device__ void mutex_unlock(unsigned int *mutex) {
    atomicExch(mutex, 0);
}
```

10.24. Warp Matrix Functions

C++ warp matrix operations leverage Tensor Cores to accelerate matrix problems of the form `D=A*B+C`. These operations are supported on mixed-precision floating point data for devices of compute capability 7.0 or higher. This requires co-operation from all threads in a *warp*. In addition, these operations are allowed in conditional code only if the condition evaluates identically across the entire *warp*, otherwise the code execution is likely to hang.

10.24.1. Description

All following functions and types are defined in the namespace `nvcuda::wmma`. Sub-byte operations are considered preview, i.e. the data structures and APIs for them are subject to change and may not be compatible with future releases. This extra functionality is defined in the `nvcuda::wmma::experimental` namespace.

```
template<typename Use, int m, int n, int k, typename T, typename Layout=void> class
↪fragment;

void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm);
void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm, layout_t layout);
void store_matrix_sync(T* mptr, const fragment<...> &a, unsigned ldm, layout_t
↪layout);
void fill_fragment(fragment<...> &a, const T& v);
void mma_sync(fragment<...> &d, const fragment<...> &a, const fragment<...> &b, const
↪fragment<...> &c, bool satf=false);
```

`fragment`
: An overloaded class containing a section of a matrix distributed across all threads in the warp. The mapping of matrix elements into `fragment` internal storage is unspecified and subject to change in future architectures.

Only certain combinations of template arguments are allowed. The first template parameter specifies how the fragment will participate in the matrix operation. Acceptable values for `Use` are:

- `matrix_a` when the fragment is used as the first multiplicand, `A`,
- `matrix_b` when the fragment is used as the second multiplicand, `B`, or
- `accumulator` when the fragment is used as the source or destination accumulators (`C` or `D`, respectively).

  The `m`, `n` and `k` sizes describe the shape of the warp-wide matrix tiles participating in the multiply-accumulate operation. The dimension of each tile depends on its role. For `matrix_a` the tile takes dimension `m x k`; for `matrix_b` the dimension is `k x n`, and `accumulator` tiles are `m x n`.

  The data type, `T`, may be `double`, `float`, `__half`, `__nv_bfloat16`, `char`, or `unsigned char` for multiplicands and `double`, `float`, `int`, or `__half` for accumulators. As documented in *Element Types and Matrix Sizes*, limited combinations of accumulator and multiplicand types are supported. The `Layout` parameter must be specified for `matrix_a` and `matrix_b` fragments. `row_major` or `col_major` indicate that elements within a matrix row or column are contiguous in memory, respectively. The `Layout` parameter for an `accumulator` matrix should retain the default value of `void`. A row or column layout is specified only when the accumulator is loaded or stored as described below.

`load_matrix_sync`
: Waits until all warp lanes have arrived at load_matrix_sync and then loads the matrix fragment `a` from memory. `mptr` must be a 256-bit aligned pointer pointing to the first element of the matrix in memory. `ldm` describes the stride in elements between consecutive rows (for row major layout) or columns (for column major layout) and must be a multiple of 8 for `__half` element type or multiple of 4 for `float` element type. (i.e., multiple of 16 bytes in both cases). If the fragment is an `accumulator`, the `layout` argument must be specified as either `mem_row_major` or `mem_col_major`. For `matrix_a` and `matrix_b` fragments, the layout is inferred from the fragment's `layout` parameter. The values of `mptr`, `ldm`, `layout` and all template parameters for `a` must be the same for all threads in the warp. This function must be called by all threads in the warp, or the result is undefined.

store_matrix_sync
: Waits until all warp lanes have arrived at store_matrix_sync and then stores the matrix fragment a to memory. `mptr` must be a 256-bit aligned pointer pointing to the first element of the matrix in memory. `ldm` describes the stride in elements between consecutive rows (for row major layout) or columns (for column major layout) and must be a multiple of 8 for `__half` element type or multiple of 4 for `float` element type. (i.e., multiple of 16 bytes in both cases). The layout of the output matrix must be specified as either `mem_row_major` or `mem_col_major`. The values of `mptr`, `ldm`, `layout` and all template parameters for a must be the same for all threads in the warp.

fill_fragment
: Fill a matrix fragment with a constant value v. Because the mapping of matrix elements to each fragment is unspecified, this function is ordinarily called by all threads in the warp with a common value for v.

mma_sync
: Waits until all warp lanes have arrived at mma_sync, and then performs the warp-synchronous matrix multiply-accumulate operation `D=A*B+C`. The in-place operation, `C=A*B+C`, is also supported. The value of `satf` and template parameters for each matrix fragment must be the same for all threads in the warp. Also, the template parameters `m`, `n` and `k` must match between fragments `A`, `B`, `C` and `D`. This function must be called by all threads in the warp, or the result is undefined.

If `satf` (saturate to finite value) mode is `true`, the following additional numerical properties apply for the destination accumulator:

- If an element result is +Infinity, the corresponding accumulator will contain `+MAX_NORM`
- If an element result is -Infinity, the corresponding accumulator will contain `-MAX_NORM`
- If an element result is NaN, the corresponding accumulator will contain `+0`

Because the map of matrix elements into each thread's `fragment` is unspecified, individual matrix elements must be accessed from memory (shared or global) after calling `store_matrix_sync`. In the special case where all threads in the warp will apply an element-wise operation uniformly to all fragment elements, direct element access can be implemented using the following `fragment` class members.

```
enum fragment<Use, m, n, k, T, Layout>::num_elements;
T fragment<Use, m, n, k, T, Layout>::x[num_elements];
```

As an example, the following code scales an `accumulator` matrix tile by half.

```
wmma::fragment<wmma::accumulator, 16, 16, 16, float> frag;
float alpha = 0.5f; // Same value for all threads in warp
/*...*/
for(int t=0; t<frag.num_elements; t++)
frag.x[t] *= alpha;
```

10.24.2. Alternate Floating Point

Tensor Cores support alternate types of floating point operations on devices with compute capability 8.0 and higher.

`__nv_bfloat16`
: This data format is an alternate fp16 format that has the same range as f32 but reduced precision (7 bits). You can use this data format directly with the `__nv_bfloat16` type available in `cuda_bf16.h`. Matrix fragments with `__nv_bfloat16` data types are required to be composed with accumulators of `float` type. The shapes and operations supported are the same as with `__half`.

`tf32`
: This data format is a special floating point format supported by Tensor Cores, with the same range as f32 and reduced precision (>=10 bits). The internal layout of this format is implementation defined. In order to use this floating point format with WMMA operations, the input matrices must be manually converted to tf32 precision.

  To facilitate conversion, a new intrinsic `__float_to_tf32` is provided. While the input and output arguments to the intrinsic are of `float` type, the output will be `tf32` numerically. This new precision is intended to be used with Tensor Cores only, and if mixed with other `float`type operations, the precision and range of the result will be undefined.

  Once an input matrix (`matrix_a` or `matrix_b`) is converted to tf32 precision, the combination of a `fragment` with `precision::tf32` precision, and a data type of `float` to `load_matrix_sync` will take advantage of this new capability. Both the accumulator fragments must have `float` data types. The only supported matrix size is 16x16x8 (m-n-k).

  The elements of the fragment are represented as `float`, hence the mapping from `element_type<T>` to `storage_element_type<T>` is:

```
precision::tf32 -> float
```

10.24.3. Double Precision

Tensor Cores support double-precision floating point operations on devices with compute capability 8.0 and higher. To use this new functionality, a `fragment` with the `double` type must be used. The `mma_sync` operation will be performed with the .rn (rounds to nearest even) rounding modifier.

10.24.4. Sub-byte Operations

Sub-byte WMMA operations provide a way to access the low-precision capabilities of Tensor Cores. They are considered a preview feature i.e. the data structures and APIs for them are subject to change and may not be compatible with future releases. This functionality is available via the `nvcuda::wmma::experimental` namespace:

```
namespace experimental {
    namespace precision {
        struct u4; // 4-bit unsigned
        struct s4; // 4-bit signed
        struct b1; // 1-bit
```

(continues on next page)

(continued from previous page)

```
    }
     enum bmmaBitOp {
         bmmaBitOpXOR = 1, // compute_75 minimum
         bmmaBitOpAND = 2  // compute_80 minimum
     };
     enum bmmaAccumulateOp { bmmaAccumulateOpPOPC = 1 };
}
```

For 4 bit precision, the APIs available remain the same, but you must specify `experimental::precision::u4` or `experimental::precision::s4` as the fragment data type. Since the elements of the fragment are packed together, `num_storage_elements` will be smaller than `num_elements` for that fragment. The `num_elements` variable for a sub-byte fragment, hence returns the number of elements of sub-byte type `element_type<T>`. This is true for single bit precision as well, in which case, the mapping from `element_type<T>` to `storage_element_type<T>` is as follows:

```
experimental::precision::u4 -> unsigned (8 elements in 1 storage element)
experimental::precision::s4 -> int (8 elements in 1 storage element)
experimental::precision::b1 -> unsigned (32 elements in 1 storage element)
T -> T  //all other types
```

The allowed layouts for sub-byte fragments is always `row_major` for `matrix_a` and `col_major` for `matrix_b`.

For sub-byte operations the value of `ldm` in `load_matrix_sync` should be a multiple of 32 for element type `experimental::precision::u4` and `experimental::precision::s4` or a multiple of 128 for element type `experimental::precision::b1` (i.e., multiple of 16 bytes in both cases).

Note: Support for the following variants for MMA instructions is deprecated and will be removed in sm_90:

- `experimental::precision::u4`
- `experimental::precision::s4`
- `experimental::precision::b1` with `bmmaBitOp` set to `bmmaBitOpXOR`

`bmma_sync`

Waits until all warp lanes have executed bmma_sync, and then performs the warp-synchronous bit matrix multiply-accumulate operation `D = (A op B) + C`, where `op` consists of a logical operation `bmmaBitOp` followed by the accumulation defined by `bmmaAccumulateOp`. The available operations are:

`bmmaBitOpXOR`, a 128-bit XOR of a row in `matrix_a` with the 128-bit column of `matrix_b`

`bmmaBitOpAND`, a 128-bit AND of a row in `matrix_a` with the 128-bit column of `matrix_b`, available on devices with compute capability 8.0 and higher.

The accumulate op is always `bmmaAccumulateOpPOPC` which counts the number of set bits.

10.24.5. Restrictions

The special format required by tensor cores may be different for each major and minor device architecture. This is further complicated by threads holding only a fragment (opaque architecture-specific ABI data structure) of the overall matrix, with the developer not allowed to make assumptions on how the individual parameters are mapped to the registers participating in the matrix multiply-accumulate.

Since fragments are architecture-specific, it is unsafe to pass them from function A to function B if the functions have been compiled for different link-compatible architectures and linked together into the same device executable. In this case, the size and layout of the fragment will be specific to one architecture and using WMMA APIs in the other will lead to incorrect results or potentially, corruption.

An example of two link-compatible architectures, where the layout of the fragment differs, is sm_70 and sm_75.

```
fragA.cu: void foo() { wmma::fragment<...> mat_a; bar(&mat_a); }
fragB.cu: void bar(wmma::fragment<...> *mat_a) { // operate on mat_a }
```

```
// sm_70 fragment layout
$> nvcc -dc -arch=compute_70 -code=sm_70 fragA.cu -o fragA.o
// sm_75 fragment layout
$> nvcc -dc -arch=compute_75 -code=sm_75 fragB.cu -o fragB.o
// Linking the two together
$> nvcc -dlink -arch=sm_75 fragA.o fragB.o -o frag.o
```

This undefined behavior might also be undetectable at compilation time and by tools at runtime, so extra care is needed to make sure the layout of the fragments is consistent. This linking hazard is most likely to appear when linking with a legacy library that is both built for a different link-compatible architecture and expecting to be passed a WMMA fragment.

Note that in the case of weak linkages (for example, a CUDA C++ inline function), the linker may choose any available function definition which may result in implicit passes between compilation units.

To avoid these sorts of problems, the matrix should always be stored out to memory for transit through external interfaces (e.g. `wmma::store_matrix_sync(dst, …);`) and then it can be safely passed to `bar()` as a pointer type [e.g. `float *dst`].

Note that since sm_70 can run on sm_75, the above example sm_75 code can be changed to sm_70 and correctly work on sm_75. However, it is recommended to have sm_75 native code in your application when linking with other sm_75 separately compiled binaries.

10.24.6. Element Types and Matrix Sizes

Tensor Cores support a variety of element types and matrix sizes. The following table presents the various combinations of `matrix_a`, `matrix_b` and `accumulator` matrix supported:

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
__half	__half	float	16x16x16
__half	__half	float	32x8x16
__half	__half	float	8x32x16
__half	__half	__half	16x16x16
__half	__half	__half	32x8x16
__half	__half	__half	8x32x16
unsigned char	unsigned char	int	16x16x16
unsigned char	unsigned char	int	32x8x16
unsigned char	unsigned char	int	8x32x16
signed char	signed char	int	16x16x16
signed char	signed char	int	32x8x16
signed char	signed char	int	8x32x16

Alternate Floating Point support:

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
__nv_bfloat16	__nv_bfloat16	float	16x16x16
__nv_bfloat16	__nv_bfloat16	float	32x8x16
__nv_bfloat16	__nv_bfloat16	float	8x32x16
precision::tf32	precision::tf32	float	16x16x8

Double Precision Support:

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
double	double	double	8x8x4

Experimental support for sub-byte operations:

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
precision::u4	precision::u4	int	8x8x32
precision::s4	precision::s4	int	8x8x32
precision::b1	precision::b1	int	8x8x128

10.24.7. Example

The following code implements a 16x16x16 matrix multiplication in a single warp.

```
#include <mma.h>
using namespace nvcuda;

__global__ void wmma_ker(half *a, half *b, float *c) {
   // Declare the fragments
   wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::col_major> a_frag;
   wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b_frag;
   wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;

   // Initialize the output to zero
   wmma::fill_fragment(c_frag, 0.0f);

   // Load the inputs
   wmma::load_matrix_sync(a_frag, a, 16);
   wmma::load_matrix_sync(b_frag, b, 16);

   // Perform the matrix multiplication
   wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

   // Store the output
   wmma::store_matrix_sync(c, c_frag, 16, wmma::mem_row_major);
}
```

10.25. DPX

DPX is a set of functions that enable finding min and max values, as well as fused addition and min/max, for up to three 16 and 32-bit signed or unsigned integer parameters, with optional ReLU (clamping to zero):

- three parameters: `__vimax3_s32`, `__vimax3_s16x2`, `__vimax3_u32`, `__vimax3_u16x2`, `__vimin3_s32`, `__vimin3_s16x2`, `__vimin3_u32`, `__vimin3_u16x2`
- two parameters, with ReLU: `__vimax_s32_relu`, `__vimax_s16x2_relu`, `__vimin_s32_relu`, `__vimin_s16x2_relu`
- three parameters, with ReLU: `__vimax3_s32_relu`, `__vimax3_s16x2_relu`, `__vimin3_s32_relu`, `__vimin3_s16x2_relu`
- two parameters, also returning which parameter was smaller/larger: `__vibmax_s32`, `__vibmax_u32`, `__vibmin_s32`, `__vibmin_u32`, `__vibmax_s16x2`, `__vibmax_u16x2`, `__vibmin_s16x2`, `__vibmin_u16x2`
- three parameters, comparing (first + second) with the third: `__viaddmax_s32`, `__viaddmax_s16x2`, `__viaddmax_u32`, `__viaddmax_u16x2`, `__viaddmin_s32`, `__viaddmin_s16x2`, `__viaddmin_u32`, `__viaddmin_u16x2`
- three parameters, with ReLU, comparing (first + second) with the third and a zero: `__viaddmax_s32_relu`, `__viaddmax_s16x2_relu`, `__viaddmin_s32_relu`, `__viaddmin_s16x2_relu`

These instructions are hardware-accelerated on devices with compute capability 9 and higher, and software emulation on older devices.

Full API can be found in CUDA Math API documentation.

DPX is exceptionally useful when implementing dynamic programming algorithms, such as Smith-Waterman or Needleman–Wunsch in genomics and Floyd-Warshall in route optimization.

10.25.1. Examples

Max value of three signed 32-bit integers, with ReLU

```
const int a = -15;
const int b = 8;
const int c = 5;
int max_value_0 = __vimax3_s32_relu(a, b, c); // max(-15, 8, 5, 0) = 8
const int d = -2;
const int e = -4;
int max_value_1 = __vimax3_s32_relu(a, d, e); // max(-15, -2, -4, 0) = 0
```

Min value of the sum of two 32-bit signed integers, another 32-bit signed integer and a zero (ReLU)

```
const int a = -5;
const int b = 6;
const int c = -2;
int max_value_0 = __viaddmax_s32_relu(a, b, c); // max(-5 + 6, -2, 0) = max(1, -2, 0)
↪= 1
const int d = 4;
int max_value_1 = __viaddmax_s32_relu(a, d, c); // max(-5 + 4, -2, 0) = max(-1, -2, 0)
↪= 0
```

Min value of two unsigned 32-bit integers and determining which value is smaller

```
const unsigned int a = 9;
const unsigned int b = 6;
bool smaller_value;
unsigned int min_value = __vibmin_u32(a, b, &smaller_value); // min_value is 6,
↪smaller_value is true
```

Max values of three pairs of unsigned 16-bit integers

```
const unsigned a = 0x00050002;
const unsigned b = 0x00070004;
const unsigned c = 0x00020006;
unsigned int max_value = __vimax3_u16x2(a, b, c); // max(5, 7, 2) and max(2, 4, 6), so
↪max_value is 0x00070006
```

10.26. Asynchronous Barrier

The NVIDIA C++ standard library introduces a GPU implementation of std::barrier. Along with the implementation of `std::barrier` the library provides extensions that allow users to specify the scope of barrier objects. The barrier API scopes are documented under Thread Scopes. Devices of compute capability 8.0 or higher provide hardware acceleration for barrier operations and integration of these barriers with the *memcpy_async* feature. On devices with compute capability below 8.0 but starting 7.0, these barriers are available without hardware acceleration.

`nvcuda::experimental::awbarrier` is deprecated in favor of `cuda::barrier`.

10.26.1. Simple Synchronization Pattern

Without the arrive/wait barrier, synchronization is achieved using `__syncthreads()` (to synchronize all threads in a block) or `group.sync()` when using *Cooperative Groups*.

```
#include <cooperative_groups.h>

__global__ void simple_sync(int iteration_count) {
    auto block = cooperative_groups::this_thread_block();

    for (int i = 0; i < iteration_count; ++i) {
        /* code before arrive */
        block.sync(); /* wait for all threads to arrive here */
        /* code after wait */
    }
}
```

Threads are blocked at the synchronization point (`block.sync()`) until all threads have reached the synchronization point. In addition, memory updates that happened before the synchronization point are guaranteed to be visible to all threads in the block after the synchronization point, i.e., equivalent to `atomic_thread_fence(memory_order_seq_cst, thread_scope_block)` as well as the sync.

This pattern has three stages:

- Code **before** sync performs memory updates that will be read **after** the sync.
- Synchronization point
- Code **after** sync point with visibility of memory updates that happened **before** sync point.

10.26.2. Temporal Splitting and Five Stages of Synchronization

The temporally-split synchronization pattern with the `std::barrier` is as follows.

```
#include <cuda/barrier>
#include <cooperative_groups.h>

__device__ void compute(float* data, int curr_iteration);
```

(continues on next page)

(continued from previous page)

```
__global__ void split_arrive_wait(int iteration_count, float *data) {
    using barrier = cuda::barrier<cuda::thread_scope_block>;
    __shared__  barrier bar;
    auto block = cooperative_groups::this_thread_block();

    if (block.thread_rank() == 0) {
        init(&bar, block.size()); // Initialize the barrier with expected arrival count
    }
    block.sync();

    for (int curr_iter = 0; curr_iter < iteration_count; ++curr_iter) {
        /* code before arrive */
       barrier::arrival_token token = bar.arrive(); /* this thread arrives. Arrival
↪does not block a thread */
       compute(data, curr_iter);
       bar.wait(std::move(token)); /* wait for all threads participating in the barrier
↪to complete bar.arrive()*/
        /* code after wait */
    }
}
```

In this pattern, the synchronization point (`block.sync()`) is split into an arrive point (`bar.arrive()`) and a wait point (`bar.wait(std::move(token))`). A thread begins participating in a `cuda::barrier` with its first call to `bar.arrive()`. When a thread calls `bar.wait(std::move(token))` it will be blocked until participating threads have completed `bar.arrive()` the expected number of times as specified by the expected arrival count argument passed to `init()`. Memory updates that happen before participating threads' call to `bar.arrive()` are guaranteed to be visible to participating threads after their call to `bar.wait(std::move(token))`. Note that the call to `bar.arrive()` does not block a thread, it can proceed with other work that does not depend upon memory updates that happen before other participating threads' call to `bar.arrive()`.

The *arrive and then wait* pattern has five stages which may be iteratively repeated:

- Code **before** arrive performs memory updates that will be read **after** the wait.
- Arrive point with implicit memory fence (i.e., equivalent to `atomic_thread_fence(memory_order_seq_cst, thread_scope_block)`).
- Code **between** arrive and wait.
- Wait point.
- Code **after** the wait, with visibility of updates that were performed **before** the arrive.

10.26.3. Bootstrap Initialization, Expected Arrival Count, and Participation

Initialization must happen before any thread begins participating in a `cuda::barrier`.

```
#include <cuda/barrier>
#include <cooperative_groups.h>

__global__ void init_barrier() {
    __shared__ cuda::barrier<cuda::thread_scope_block> bar;
```

(continues on next page)

(continued from previous page)

```
    auto block = cooperative_groups::this_thread_block();

    if (block.thread_rank() == 0) {
        init(&bar, block.size()); // Single thread initializes the total expected
↪arrival count.
    }
    block.sync();
}
```

Before any thread can participate in `cuda::barrier`, the barrier must be initialized using `init()` with an **expected arrival count**, `block.size()` in this example. Initialization must happen before any thread calls `bar.arrive()`. This poses a bootstrapping challenge in that threads must synchronize before participating in the `cuda::barrier`, but threads are creating a `cuda::barrier` in order to synchronize. In this example, threads that will participate are part of a cooperative group and use `block.sync()` to bootstrap initialization. In this example a whole thread block is participating in initialization, hence `__syncthreads()` could also be used.

The second parameter of `init()` is the **expected arrival count**, i.e., the number of times `bar.arrive()` will be called by participating threads before a participating thread is unblocked from its call to `bar.wait(std::move(token))`. In the prior example the `cuda::barrier` is initialized with the number of threads in the thread block i.e., `cooperative_groups::this_thread_block().size()`, and all threads within the thread block participate in the barrier.

A `cuda::barrier` is flexible in specifying how threads participate (split arrive/wait) and which threads participate. In contrast `this_thread_block.sync()` from cooperative groups or `__syncthreads()` is applicable to whole-thread-block and `__syncwarp(mask)` is a specified subset of a warp. If the intention of the user is to synchronize a full thread block or a full warp we recommend using `__syncthreads()` and `__syncwarp(mask)` respectively for performance reasons.

10.26.4. A Barrier's Phase: Arrival, Countdown, Completion, and Reset

A `cuda::barrier` counts down from the expected arrival count to zero as participating threads call `bar.arrive()`. When the countdown reaches zero, a `cuda::barrier` is complete for the current phase. When the last call to `bar.arrive()` causes the countdown to reach zero, the countdown is automatically and atomically reset. The reset assigns the countdown to the expected arrival count, and moves the `cuda::barrier` to the next phase.

A `token` object of class `cuda::barrier::arrival_token`, as returned from `token=bar.arrive()`, is associated with the current phase of the barrier. A call to `bar.wait(std::move(token))` blocks the calling thread while the `cuda::barrier` is in the current phase, i.e., while the phase associated with the token matches the phase of the `cuda::barrier`. If the phase is advanced (because the countdown reaches zero) before the call to `bar.wait(std::move(token))` then the thread does not block; if the phase is advanced while the thread is blocked in `bar.wait(std::move(token))`, the thread is unblocked.

It is essential to know when a reset could or could not occur, especially in non-trivial arrive/wait synchronization patterns.

- A thread's calls to `token=bar.arrive()` and `bar.wait(std::move(token))` must be sequenced such that `token=bar.arrive()` occurs during the `cuda::barrier`'s current phase, and `bar.wait(std::move(token))` occurs during the same or next phase.

- A thread's call to `bar.arrive()` must occur when the barrier's counter is non-zero. After barrier initialization, if a thread's call to `bar.arrive()` causes the countdown to reach zero then a call to `bar.wait(std::move(token))` must happen before the barrier can be reused for a subsequent call to `bar.arrive()`.
- `bar.wait()` must only be called using a `token` object of the current phase or the immediately preceding phase. For any other values of the `token` object, the behavior is undefined.

For simple arrive/wait synchronization patterns, compliance with these usage rules is straightforward.

10.26.5. Spatial Partitioning (also known as Warp Specialization)

A thread block can be spatially partitioned such that warps are specialized to perform independent computations. Spatial partitioning is used in a producer or consumer pattern, where one subset of threads produces data that is concurrently consumed by the other (disjoint) subset of threads.

A producer/consumer spatial partitioning pattern requires two one sided synchronizations to manage a data buffer between the producer and consumer.

Producer	Consumer
wait for buffer to be ready to be filled	signal buffer is ready to be filled
produce data and fill the buffer	
signal buffer is filled	wait for buffer to be filled
	consume data in filled buffer

Producer threads wait for consumer threads to signal that the buffer is ready to be filled; however, consumer threads do not wait for this signal. Consumer threads wait for producer threads to signal that the buffer is filled; however, producer threads do not wait for this signal. For full producer/consumer concurrency this pattern has (at least) double buffering where each buffer requires two `cuda::barrier`s.

```
#include <cuda/barrier>
#include <cooperative_groups.h>

using barrier = cuda::barrier<cuda::thread_scope_block>;

__device__ void producer(barrier ready[], barrier filled[], float* buffer, float* in,
↪int N, int buffer_len)
{
    for (int i = 0; i < (N/buffer_len); ++i) {
        ready[i%2].arrive_and_wait(); /* wait for buffer_(i%2) to be ready to be filled
↪*/
        /* produce, i.e., fill in, buffer_(i%2)  */
        barrier::arrival_token token = filled[i%2].arrive(); /* buffer_(i%2) is filled
↪*/
    }
}

__device__ void consumer(barrier ready[], barrier filled[], float* buffer, float* out,
↪ int N, int buffer_len)
```

(continues on next page)

(continued from previous page)

```
{
    barrier::arrival_token token1 = ready[0].arrive(); /* buffer_0 is ready for
↪initial fill */
    barrier::arrival_token token2 = ready[1].arrive(); /* buffer_1 is ready for
↪initial fill */
    for (int i = 0; i < (N/buffer_len); ++i) {
        filled[i%2].arrive_and_wait(); /* wait for buffer_(i%2) to be filled */
        /* consume buffer_(i%2) */
        barrier::arrival_token token = ready[i%2].arrive(); /* buffer_(i%2) is ready
↪to be re-filled */
    }
}

//N is the total number of float elements in arrays in and out
__global__ void producer_consumer_pattern(int N, int buffer_len, float* in, float*
↪out) {

    // Shared memory buffer declared below is of size 2 * buffer_len
    // so that we can alternatively work between two buffers.
    // buffer_0 = buffer and buffer_1 = buffer + buffer_len
    __shared__ extern float buffer[];

    // bar[0] and bar[1] track if buffers buffer_0 and buffer_1 are ready to be filled,
    // while bar[2] and bar[3] track if buffers buffer_0 and buffer_1 are filled-in
↪respectively
    __shared__ barrier bar[4];


    auto block = cooperative_groups::this_thread_block();
    if (block.thread_rank() < 4)
        init(bar + block.thread_rank(), block.size());
    block.sync();

    if (block.thread_rank() < warpSize)
        producer(bar, bar+2, buffer, in, N, buffer_len);
    else
        consumer(bar, bar+2, buffer, out, N, buffer_len);
}
```

In this example the first warp is specialized as the producer and the remaining warps are specialized as the consumer. All producer and consumer threads participate (call `bar.arrive()` or `bar.arrive_and_wait()`) in each of the four `cuda::barriers` so the expected arrival counts are equal to `block.size()`.

A producer thread waits for the consumer threads to signal that the shared memory buffer can be filled. In order to wait for a `cuda::barrier` a producer thread must first arrive on that `ready[i%2].arrive()` to get a token and then `ready[i%2].wait(token)` with that token. For simplicity `ready[i%2].arrive_and_wait()` combines these operations.

```
bar.arrive_and_wait();
/* is equivalent to */
bar.wait(bar.arrive());
```

Producer threads compute and fill the ready buffer, they then signal that the buffer is filled by arriving on the filled barrier, `filled[i%2].arrive()`. A producer thread does not wait at this point, instead it waits until the next iteration's buffer (double buffering) is ready to be filled.

A consumer thread begins by signaling that both buffers are ready to be filled. A consumer thread does not wait at this point, instead it waits for this iteration's buffer to be filled, `filled[i%2].arrive_and_wait()`. After the consumer threads consume the buffer they signal that the buffer is ready to be filled again, `ready[i%2].arrive()`, and then wait for the next iteration's buffer to be filled.

10.26.6. Early Exit (Dropping out of Participation)

When a thread that is participating in a sequence of synchronizations must exit early from that sequence, that thread must explicitly drop out of participation before exiting. The remaining participating threads can proceed normally with subsequent `cuda::barrier` arrive and wait operations.

```
#include <cuda/barrier>
#include <cooperative_groups.h>

__device__ bool condition_check();

__global__ void early_exit_kernel(int N) {
    using barrier = cuda::barrier<cuda::thread_scope_block>;
    __shared__ barrier bar;
    auto block = cooperative_groups::this_thread_block();

    if (block.thread_rank() == 0)
        init(&bar , block.size());
    block.sync();

    for (int i = 0; i < N; ++i) {
        if (condition_check()) {
          bar.arrive_and_drop();
          return;
        }
        /* other threads can proceed normally */
        barrier::arrival_token token = bar.arrive();
        /* code between arrive and wait */
        bar.wait(std::move(token)); /* wait for all threads to arrive */
        /* code after wait */
    }
}
```

This operation arrives on the `cuda::barrier` to fulfill the participating thread's obligation to arrive in the **current** phase, and then decrements the expected arrival count for the **next** phase so that this thread is no longer expected to arrive on the barrier.

10.26.7. Completion Function

The `CompletionFunction` of `cuda::barrier<Scope, CompletionFunction>` is executed once per phase, after the last thread *arrives* and before any thread is unblocked from the `wait`. Memory operations performed by the threads that arrived at the `barrier` during the phase are visible to the thread executing the `CompletionFunction`, and all memory operations performed within the `CompletionFunction` are visible to all threads waiting at the `barrier` once they are unblocked from the `wait`.

```
#include <cuda/barrier>
#include <cooperative_groups.h>
#include <functional>
namespace cg = cooperative_groups;

__device__ int divergent_compute(int*, int);
__device__ int independent_computation(int*, int);

__global__ void psum(int* data, int n, int* acc) {
  auto block = cg::this_thread_block();

  constexpr int BlockSize = 128;
  __shared__ int smem[BlockSize];
  assert(BlockSize == block.size());
  assert(n % 128 == 0);

  auto completion_fn = [&] {
    int sum = 0;
    for (int i = 0; i < 128; ++i) sum += smem[i];
    *acc += sum;
  };

  // Barrier storage
  // Note: the barrier is not default-constructible because
  //       completion_fn is not default-constructible due
  //       to the capture.
  using completion_fn_t = decltype(completion_fn);
  using barrier_t = cuda::barrier<cuda::thread_scope_block,
                                  completion_fn_t>;
  __shared__ std::aligned_storage<sizeof(barrier_t),
                                  alignof(barrier_t)> bar_storage;

  // Initialize barrier:
  barrier_t* bar = (barrier_t*)&bar_storage;
  if (block.thread_rank() == 0) {
    assert(*acc == 0);
    assert(blockDim.x == blockDim.y == blockDim.y == 1);
    new (bar) barrier_t{block.size(), completion_fn};
    // equivalent to: init(bar, block.size(), completion_fn);
  }
  block.sync();

  // Main loop
  for (int i = 0; i < n; i += block.size()) {
    smem[block.thread_rank()] = data[i] + *acc;
    auto t = bar->arrive();
    // We can do independent computation here
    bar->wait(std::move(t));
    // shared-memory is safe to re-use in the next iteration
    // since all threads are done with it, including the one
    // that did the reduction
  }
}
```

10.26.8. Memory Barrier Primitives Interface

Memory barrier primitives are C-like interfaces to `cuda::barrier` functionality. These primitives are available through including the `<cuda_awbarrier_primitives.h>` header.

10.26.8.1 Data Types

```
typedef /* implementation defined */ __mbarrier_t;
typedef /* implementation defined */ __mbarrier_token_t;
```

10.26.8.2 Memory Barrier Primitives API

```
uint32_t __mbarrier_maximum_count();
void __mbarrier_init(__mbarrier_t* bar, uint32_t expected_count);
```

- `bar` must be a pointer to `__shared__` memory.
- `expected_count <= __mbarrier_maximum_count()`
- Initialize `*bar` expected arrival count for the current and next phase to `expected_count`.

```
void __mbarrier_inval(__mbarrier_t* bar);
```

- `bar` must be a pointer to the mbarrier object residing in shared memory.
- Invalidation of `*bar` is required before the corresponding shared memory can be repurposed.

```
__mbarrier_token_t __mbarrier_arrive(__mbarrier_t* bar);
```

- Initialization of `*bar` must happen before this call.
- Pending count must not be zero.
- Atomically decrement the pending count for the current phase of the barrier.
- Return an arrival token associated with the barrier state immediately prior to the decrement.

```
__mbarrier_token_t __mbarrier_arrive_and_drop(__mbarrier_t* bar);
```

- Initialization of `*bar` must happen before this call.
- Pending count must not be zero.
- Atomically decrement the pending count for the current phase and expected count for the next phase of the barrier.
- Return an arrival token associated with the barrier state immediately prior to the decrement.

```
bool __mbarrier_test_wait(__mbarrier_t* bar, __mbarrier_token_t token);
```

- `token` must be associated with the immediately preceding phase or current phase of `*this`.
- Returns `true` if `token` is associated with the immediately preceding phase of `*bar`, otherwise returns `false`.

```
//Note: This API has been deprecated in CUDA 11.1
uint32_t __mbarrier_pending_count(__mbarrier_token_t token);
```

10.27. Asynchronous Data Copies

CUDA 11 introduces Asynchronous Data operations with `memcpy_async` API to allow device code to explicitly manage the asynchronous copying of data. The `memcpy_async` feature enables CUDA kernels to overlap computation with data movement.

10.27.1. `memcpy_async` API

The `memcpy_async` APIs are provided in the `cuda/barrier`, `cuda/pipeline`, and `cooperative_groups/memcpy_async.h` header files.

The `cuda::memcpy_async` APIs work with `cuda::barrier` and `cuda::pipeline` synchronization primitives, while the `cooperative_groups::memcpy_async` synchronizes using `cooperative_groups::wait`.

These APIs have very similar semantics: copy objects from `src` to `dst` as-if performed by another thread which, on completion of the copy, can be synchronized through `cuda::pipeline`, `cuda::barrier`, or `cooperative_groups::wait`.

The complete API documentation of the `cuda::memcpy_async` overloads for `cuda::barrier` and `cuda::pipeline` is provided in the libcudacxx API documentation along with some examples.

The API documentation of *cooperative_groups::memcpy_async* is provided in the *Cooperative Groups* section.

The `memcpy_async` APIs that use *cuda::barrier* and `cuda::pipeline` require compute capability 7.0 or higher. On devices with compute capability 8.0 or higher, `memcpy_async` operations from global to shared memory can benefit from hardware acceleration.

10.27.2. Copy and Compute Pattern - Staging Data Through Shared Memory

CUDA applications often employ a *copy and compute* pattern that:

- fetches data from global memory,
- stores data to shared memory, and
- performs computations on shared memory data, and potentially writes results back to global memory.

The following sections illustrate how this pattern can be expressed without and with the `memcpy_async` feature:

- *Without memcpy_async* introduces an example that does not overlap computation with data movement and uses an intermediate register to copy data.
- *With memcpy_async* improves the previous example by introducing the *memcpy_async* and the `cuda::memcpy_async` APIs to directly copy data from global to shared memory without using intermediate registers.
- *Asynchronous Data Copies using cuda::barrier* shows memcpy with cooperative groups and barrier.

- *Single-Stage Asynchronous Data Copies using cuda::pipeline* shows memcpy with single stage pipeline.
- *Multi-Stage Asynchronous Data Copies using cuda::pipeline* shows memcpy with multi stage pipeline.

10.27.3. Without memcpy_async

Without `memcpy_async`, the *copy* phase of the *copy and compute* pattern is expressed as `shared[local_idx] = global[global_idx]`. This global to shared memory copy is expanded to a read from global memory into a register, followed by a write to shared memory from the register.

When this pattern occurs within an iterative algorithm, each thread block needs to synchronize after the `shared[local_idx] = global[global_idx]` assignment, to ensure all writes to shared memory have completed before the compute phase can begin. The thread block also needs to synchronize again after the compute phase, to prevent overwriting shared memory before all threads have completed their computations. This pattern is illustrated in the following code snippet.

```
#include <cooperative_groups.h>
__device__ void compute(int* global_out, int const* shared_in) {
    // Computes using all values of current batch from shared memory.
    // Stores this thread's result back to global memory.
}

__global__ void without_memcpy_async(int* global_out, int const* global_in, size_t
↪size, size_t batch_sz) {
  auto grid = cooperative_groups::this_grid();
  auto block = cooperative_groups::this_thread_block();
  assert(size == batch_sz * grid.size()); // Exposition: input size fits batch_sz *
↪grid_size

  extern __shared__ int shared[]; // block.size() * sizeof(int) bytes

  size_t local_idx = block.thread_rank();

  for (size_t batch = 0; batch < batch_sz; ++batch) {
    // Compute the index of the current batch for this block in global memory:
    size_t block_batch_idx = block.group_index().x * block.size() + grid.size() *
↪batch;
    size_t global_idx = block_batch_idx + threadIdx.x;
    shared[local_idx] = global_in[global_idx];

    block.sync(); // Wait for all copies to complete

    compute(global_out + block_batch_idx, shared); // Compute and write result to
↪global memory

    block.sync(); // Wait for compute using shared memory to finish
  }
}
```

10.27.4. With memcpy_async

With `memcpy_async`, the assignment of shared memory from global memory

```
shared[local_idx] = global_in[global_idx];
```

is replaced with an asynchronous copy operation from *cooperative groups*

```
cooperative_groups::memcpy_async(group, shared, global_in + batch_idx, sizeof(int) *
↪block.size());
```

The *cooperative_groups::memcpy_async* API copies `sizeof(int) * block.size()` bytes from global memory starting at `global_in + batch_idx` to the `shared` data. This operation happens as-if performed by another thread, which synchronizes with the current thread's call to *cooperative_groups::wait* after the copy has completed. Until the copy operation completes, modifying the global data or reading or writing the shared data introduces a data race.

On devices with compute capability 8.0 or higher, `memcpy_async` transfers from global to shared memory can benefit from hardware acceleration, which avoids transferring the data through an intermediate register.

```
#include <cooperative_groups.h>
#include <cooperative_groups/memcpy_async.h>

__device__ void compute(int* global_out, int const* shared_in);

__global__ void with_memcpy_async(int* global_out, int const* global_in, size_t size,
↪size_t batch_sz) {
  auto grid = cooperative_groups::this_grid();
  auto block = cooperative_groups::this_thread_block();
  assert(size == batch_sz * grid.size()); // Exposition: input size fits batch_sz *
↪grid_size

  extern __shared__ int shared[]; // block.size() * sizeof(int) bytes

  for (size_t batch = 0; batch < batch_sz; ++batch) {
    size_t block_batch_idx = block.group_index().x * block.size() + grid.size() *
↪batch;
    // Whole thread-group cooperatively copies whole batch to shared memory:
    cooperative_groups::memcpy_async(block, shared, global_in + block_batch_idx,
↪sizeof(int) * block.size());

    cooperative_groups::wait(block); // Joins all threads, waits for all copies to
↪complete

    compute(global_out + block_batch_idx, shared);

    block.sync();
  }
}}
```

10.27.5. Asynchronous Data Copies using `cuda::barrier`

The `cuda::memcpy_async` overload for *cuda::barrier* enables synchronizing asynchronous data transfers using a `barrier`. This overloads executes the copy operation as-if performed by another thread bound to the barrier by: incrementing the expected count of the current phase on creation, and decrementing it on completion of the copy operation, such that the phase of the `barrier` will only advance when all threads participating in the barrier have arrived, and all `memcpy_async` bound to the current phase of the barrier have completed. The following example uses a block-wide `barrier`, where all block threads participate, and swaps the wait operation with a barrier `arrive_and_wait`, while providing the same functionality as the previous example:

```
#include <cooperative_groups.h>
#include <cuda/barrier>
__device__ void compute(int* global_out, int const* shared_in);

__global__ void with_barrier(int* global_out, int const* global_in, size_t size, size_
↪t batch_sz) {
  auto grid = cooperative_groups::this_grid();
  auto block = cooperative_groups::this_thread_block();
  assert(size == batch_sz * grid.size()); // Assume input size fits batch_sz * grid_
↪size

  extern __shared__ int shared[]; // block.size() * sizeof(int) bytes

  // Create a synchronization object (C++20 barrier)
  __shared__ cuda::barrier<cuda::thread_scope::thread_scope_block> barrier;
  if (block.thread_rank() == 0) {
    init(&barrier, block.size()); // Friend function initializes barrier
  }
  block.sync();

  for (size_t batch = 0; batch < batch_sz; ++batch) {
    size_t block_batch_idx = block.group_index().x * block.size() + grid.size() *
↪batch;
    cuda::memcpy_async(block, shared, global_in + block_batch_idx, sizeof(int) *
↪block.size(), barrier);

    barrier.arrive_and_wait(); // Waits for all copies to complete

    compute(global_out + block_batch_idx, shared);

    block.sync();
  }
}
```

10.27.6. Performance Guidance for memcpy_async

For compute capability 8.x, the pipeline mechanism is shared among CUDA threads in the same CUDA warp. This sharing causes batches of `memcpy_async` to be entangled within a warp, which can impact performance under certain circumstances.

This section highlights the warp-entanglement effect on *commit*, *wait*, and *arrive* operations. Please refer to *Pipeline Interface* and the *Pipeline Primitives Interface* for an overview of the individual operations.

10.27.6.1 Alignment

On devices with compute capability 8.0, the cp.async family of instructions allows copying data from global to shared memory asynchronously. These instructions support copying 4, 8, and 16 bytes at a time. If the size provided to `memcpy_async` is a multiple of 4, 8, or 16, and both pointers passed to `memcpy_async` are aligned to a 4, 8, or 16 alignment boundary, then `memcpy_async` can be implemented using exclusively asynchronous memory operations.

Additionally for achieving best performance when using `memcpy_async` API, an alignment of 128 Bytes for both shared memory and global memory is required.

For pointers to values of types with an alignment requirement of 1 or 2, it is often not possible to prove that the pointers are always aligned to a higher alignment boundary. Determining whether the `cp.async` instructions can or cannot be used must be delayed until run-time. Performing such a runtime alignment check increases code-size and adds runtime overhead.

The cuda::aligned_size_t<size_t Align>(size_t size)Shape can be used to supply a proof that both pointers passed to `memcpy_async` are aligned to an `Align` alignment boundary and that `size` is a multiple of `Align`, by passing it as an argument where the `memcpy_async` APIs expect a Shape:

```
cuda::memcpy_async(group, dst, src, cuda::aligned_size_t<16>(N * block.size()),
↪pipeline);
```

If the proof is incorrect, the behavior is undefined.

10.27.6.2 Trivially copyable

On devices with compute capability 8.0, the cp.async family of instructions allows copying data from global to shared memory asynchronously. If the pointer types passed to `memcpy_async` do not point to TriviallyCopyable types, the copy constructor of each output element needs to be invoked, and these instructions cannot be used to accelerate `memcpy_async`.

10.27.6.3 Warp Entanglement - Commit

The sequence of `memcpy_async` batches is shared across the warp. The commit operation is coalesced such that the sequence is incremented once for all converged threads that invoke the commit operation. If the warp is fully converged, the sequence is incremented by one; if the warp is fully diverged, the sequence is incremented by 32.

- Let *PB* be the warp-shared pipeline's *actual* sequence of batches.

  `PB = {BP0, BP1, BP2, …, BPL}`

- Let *TB* be a thread's *perceived* sequence of batches, as if the sequence were only incremented by this thread's invocation of the commit operation.

  `TB = {BT0, BT1, BT2, …, BTL}`

  The `pipeline::producer_commit()` return value is from the thread's *perceived* batch sequence.

- An index in a thread's perceived sequence always aligns to an equal or larger index in the actual warp-shared sequence. The sequences are equal only when all commit operations are invoked from converged threads.

  `BTn  BPm` where `n <= m`

For example, when a warp is fully diverged:

- The warp-shared pipeline's actual sequence would be: `PB = {0, 1, 2, 3, ..., 31}` (PL=31).
- The perceived sequence for each thread of this warp would be:
  - Thread 0: `TB = {0}` (TL=0)
  - Thread 1: `TB = {0}` (TL=0)
  - …
  - Thread 31: `TB = {0}` (TL=0)

10.27.6.4 Warp Entanglement - Wait

A CUDA thread invokes either `pipeline_consumer_wait_prior<N>()` or `pipeline::consumer_wait()` to wait for batches in the *perceived* sequence TB to complete. Note that `pipeline::consumer_wait()` is equivalent to `pipeline_consumer_wait_prior<N>()`, where `N = PL`.

The `pipeline_consumer_wait_prior<N>()` function waits for batches in the *actual* sequence at least up to and including `PL-N`. Since `TL <= PL`, waiting for batch up to and including `PL-N` includes waiting for batch `TL-N`. Thus, when `TL < PL`, the thread will unintentionally wait for additional, more recent batches.

In the extreme fully-diverged warp example above, each thread could wait for all 32 batches.

10.27.6.5 Warp Entanglement - Arrive-On

Warp-divergence affects the number of times an `arrive_on(bar)` operation updates the barrier. If the invoking warp is fully converged, then the barrier is updated once. If the invoking warp is fully diverged, then 32 individual updates are applied to the barrier.

10.27.6.6 Keep Commit and Arrive-On Operations Converged

It is recommended that commit and arrive-on invocations are by converged threads:

- to not over-wait, by keeping threads' perceived sequence of batches aligned with the actual sequence, and
- to minimize updates to the barrier object.

When code preceding these operations diverges threads, then the warp should be re-converged, via `__syncwarp` before invoking commit or arrive-on operations.

10.28. Asynchronous Data Copies using `cuda::pipeline`

CUDA provides the `cuda::pipeline` synchronization object to manage and overlap asynchronous data movement with computation.

The API documentation for `cuda::pipeline` is provided in the libcudacxx API. A pipeline object is a double-ended N stage queue with a *head* and a *tail*, and is used to process work in a first-in first-out (FIFO) order. The pipeline object has following member functions to manage the stages of the pipeline.

Pipeline Class Member Function	Description
`producer_acquire`	Acquires an available stage in the pipeline internal queue.
`producer_commit`	Commits the asynchronous operations issued after the `producer_acquire` call on the currently acquired stage of the pipeline.
`consumer_wait`	Wait for completion of all asynchronous operations on the oldest stage of the pipeline.
`consumer_release`	Release the oldest stage of the pipeline to the pipeline object for reuse. The released stage can be then acquired by the producer.

10.28.1. Single-Stage Asynchronous Data Copies using `cuda::pipeline`

In previous examples we showed how to use *cooperative_groups* and *cuda::barrier* to do asynchronous data transfers. In this section, we will use the `cuda::pipeline` API with a single stage to schedule asynchronous copies. And later we will expand this example to show multi staged overlapped compute and copy.

```
#include <cooperative_groups/memcpy_async.h>
#include <cuda/pipeline>

__device__ void compute(int* global_out, int const* shared_in);
__global__ void with_single_stage(int* global_out, int const* global_in, size_t size,
↪size_t batch_sz) {
```

(continues on next page)

(continued from previous page)

```
    auto grid = cooperative_groups::this_grid();
    auto block = cooperative_groups::this_thread_block();
    assert(size == batch_sz * grid.size()); // Assume input size fits batch_sz * grid_
↪size

    constexpr size_t stages_count = 1; // Pipeline with one stage
    // One batch must fit in shared memory:
    extern __shared__ int shared[];  // block.size() * sizeof(int) bytes

    // Allocate shared storage for a single stage cuda::pipeline:
    __shared__ cuda::pipeline_shared_state<
        cuda::thread_scope::thread_scope_block,
        stages_count
    > shared_state;
    auto pipeline = cuda::make_pipeline(block, &shared_state);

    // Each thread processes `batch_sz` elements.
    // Compute offset of the batch `batch` of this thread block in global memory:
    auto block_batch = [&](size_t batch) -> int {
      return block.group_index().x * block.size() + grid.size() * batch;
    };

    for (size_t batch = 0; batch < batch_sz; ++batch) {
        size_t global_idx = block_batch(batch);

        // Collectively acquire the pipeline head stage from all producer threads:
        pipeline.producer_acquire();

        // Submit async copies to the pipeline's head stage to be
        // computed in the next loop iteration
        cuda::memcpy_async(block, shared, global_in + global_idx, sizeof(int) * block.
↪size(), pipeline);
        // Collectively commit (advance) the pipeline's head stage
        pipeline.producer_commit();

        // Collectively wait for the operations committed to the
        // previous `compute` stage to complete:
        pipeline.consumer_wait();

        // Computation overlapped with the memcpy_async of the "copy" stage:
        compute(global_out + global_idx, shared);

        // Collectively release the stage resources
        pipeline.consumer_release();
    }
}
```

10.28.2. Multi-Stage Asynchronous Data Copies using `cuda::pipeline`

In the previous examples with *cooperative_groups::wait* and *cuda::barrier*, the kernel threads immediately wait for the data transfer to shared memory to complete. This avoids data transfers from global memory into registers, but does not *hide* the latency of the `memcpy_async` operation by overlapping computation.

For that we use the CUDA *pipeline* feature in the following example. It provides a mechanism for managing a sequence of `memcpy_async` batches, enabling CUDA kernels to overlap memory transfers with computation. The following example implements a two-stage pipeline that overlaps data-transfer with computation. It:

- Initializes the pipeline shared state (more below)
- Kickstarts the pipeline by scheduling a `memcpy_async` for the first batch.
- Loops over all the batches: it schedules `memcpy_async` for the next batch, blocks all threads on the completion of the `memcpy_async` for the previous batch, and then overlaps the computation on the previous batch with the asynchronous copy of the memory for the next batch.
- Finally, it drains the pipeline by performing the computation on the last batch.

Note that, for interoperability with `cuda::pipeline`, `cuda::memcpy_async` from the `cuda/pipeline` header is used here.

```
#include <cooperative_groups/memcpy_async.h>
#include <cuda/pipeline>

__device__ void compute(int* global_out, int const* shared_in);
__global__ void with_staging(int* global_out, int const* global_in, size_t size, size_
↪t batch_sz) {
    auto grid = cooperative_groups::this_grid();
    auto block = cooperative_groups::this_thread_block();
    assert(size == batch_sz * grid.size()); // Assume input size fits batch_sz * grid_
↪size

    constexpr size_t stages_count = 2; // Pipeline with two stages
    // Two batches must fit in shared memory:
    extern __shared__ int shared[];  // stages_count * block.size() * sizeof(int) bytes
    size_t shared_offset[stages_count] = { 0, block.size() }; // Offsets to each batch

    // Allocate shared storage for a two-stage cuda::pipeline:
    __shared__ cuda::pipeline_shared_state<
        cuda::thread_scope::thread_scope_block,
        stages_count
    > shared_state;
    auto pipeline = cuda::make_pipeline(block, &shared_state);

    // Each thread processes `batch_sz` elements.
    // Compute offset of the batch `batch` of this thread block in global memory:
    auto block_batch = [&](size_t batch) -> int {
      return block.group_index().x * block.size() + grid.size() * batch;
    };

    // Initialize first pipeline stage by submitting a `memcpy_async` to fetch a whole
↪batch for the block:
```

(continues on next page)

(continued from previous page)

```
    if (batch_sz == 0) return;
    pipeline.producer_acquire();
    cuda::memcpy_async(block, shared + shared_offset[0], global_in + block_batch(0),
↪sizeof(int) * block.size(), pipeline);
    pipeline.producer_commit();

    // Pipelined copy/compute:
    for (size_t batch = 1; batch < batch_sz; ++batch) {
        // Stage indices for the compute and copy stages:
        size_t compute_stage_idx = (batch - 1) % 2;
        size_t copy_stage_idx = batch % 2;

        size_t global_idx = block_batch(batch);

        // Collectively acquire the pipeline head stage from all producer threads:
        pipeline.producer_acquire();

        // Submit async copies to the pipeline's head stage to be
        // computed in the next loop iteration
        cuda::memcpy_async(block, shared + shared_offset[copy_stage_idx], global_in +
↪global_idx, sizeof(int) * block.size(), pipeline);
        // Collectively commit (advance) the pipeline's head stage
        pipeline.producer_commit();

        // Collectively wait for the operations committed to the
        // previous `compute` stage to complete:
        pipeline.consumer_wait();

        // Computation overlapped with the memcpy_async of the "copy" stage:
        compute(global_out + global_idx, shared + shared_offset[compute_stage_idx]);

        // Collectively release the stage resources
        pipeline.consumer_release();
    }

    // Compute the data fetch by the last iteration
    pipeline.consumer_wait();
    compute(global_out + block_batch(batch_sz-1), shared + shared_offset[(batch_sz -
↪1) % 2]);
    pipeline.consumer_release();
}
```

A *pipeline object* is a double-ended queue with a *head* and a *tail*, and is used to process work in a first-in first-out (FIFO) order. Producer threads commit work to the pipeline's head, while consumer threads pull work from the pipeline's tail. In the example above, all threads are both producer and consumer threads. The threads first *commit*`memcpy_async` operations to fetch the *next* batch while they *wait* on the *previous* batch of `memcpy_async` operations to complete.

- Committing work to a pipeline stage involves:
  - Collectively *acquiring* the pipeline *head* from a set of producer threads using `pipeline.producer_acquire()`.
  - Submitting `memcpy_async` operations to the pipeline head.
  - Collectively *committing* (advancing) the pipeline head using `pipeline.producer_commit()`.
- Using a previously commited stage involves:

- Collectively waiting for the stage to complete, e.g., using `pipeline.consumer_wait()` to wait on the tail (oldest) stage.
- Collectively *releasing* the stage using `pipeline.consumer_release()`.

`cuda::pipeline_shared_state<scope, count>` encapsulates the finite resources that allow a pipeline to process up to `count` concurrent stages. If all resources are in use, `pipeline.producer_acquire()` blocks producer threads until the resources of the next pipeline stage are released by consumer threads.

This example can be written in a more concise manner by merging the prolog and epilog of the loop with the loop itself as follows:

```
template <size_t stages_count = 2 /* Pipeline with stages_count stages */>
__global__ void with_staging_unified(int* global_out, int const* global_in, size_t
↪size, size_t batch_sz) {
    auto grid = cooperative_groups::this_grid();
    auto block = cooperative_groups::this_thread_block();
    assert(size == batch_sz * grid.size()); // Assume input size fits batch_sz * grid_
↪size

    extern __shared__ int shared[]; // stages_count * block.size() * sizeof(int) bytes
    size_t shared_offset[stages_count];
    for (int s = 0; s < stages_count; ++s) shared_offset[s] = s * block.size();

    __shared__ cuda::pipeline_shared_state<
        cuda::thread_scope::thread_scope_block,
        stages_count
    > shared_state;
    auto pipeline = cuda::make_pipeline(block, &shared_state);

    auto block_batch = [&](size_t batch) -> int {
        return block.group_index().x * block.size() + grid.size() * batch;
    };

    // compute_batch: next batch to process
    // fetch_batch:  next batch to fetch from global memory
    for (size_t compute_batch = 0, fetch_batch = 0; compute_batch < batch_sz;
↪++compute_batch) {
        // The outer loop iterates over the computation of the batches
        for (; fetch_batch < batch_sz && fetch_batch < (compute_batch + stages_count);
↪ ++fetch_batch) {
            // This inner loop iterates over the memory transfers, making sure that the
↪pipeline is always full
            pipeline.producer_acquire();
            size_t shared_idx = fetch_batch % stages_count;
            size_t batch_idx = fetch_batch;
            size_t block_batch_idx = block_batch(batch_idx);
            cuda::memcpy_async(block, shared + shared_offset[shared_idx], global_in +
↪block_batch_idx, sizeof(int) * block.size(), pipeline);
            pipeline.producer_commit();
        }
        pipeline.consumer_wait();
        int shared_idx = compute_batch % stages_count;
        int batch_idx = compute_batch;
        compute(global_out + block_batch(batch_idx), shared + shared_offset[shared_
↪idx]);
        pipeline.consumer_release();
    }
```

(continues on next page)

(continued from previous page)

```
}
```

The `pipeline<thread_scope_block>` primitive used above is very flexible, and supports two features that our examples above are not using: any arbitrary subset of threads in the block can participate in the `pipeline`, and from the threads that participate, any subsets can be producers, consumers, or both. In the following example, threads with an "even" thread rank are producers, while other threads are consumers:

```
__device__ void compute(int* global_out, int shared_in);

template <size_t stages_count = 2>
__global__ void with_specialized_staging_unified(int* global_out, int const* global_
↪in, size_t size, size_t batch_sz) {
    auto grid = cooperative_groups::this_grid();
    auto block = cooperative_groups::this_thread_block();

    // In this example, threads with "even" thread rank are producers, while threads
↪with "odd" thread rank are consumers:
    const cuda::pipeline_role thread_role
      = block.thread_rank() % 2 == 0? cuda::pipeline_role::producer : cuda::pipeline_
↪role::consumer;

    // Each thread block only has half of its threads as producers:
    auto producer_threads = block.size() / 2;

    // Map adjacent even and odd threads to the same id:
    const int thread_idx = block.thread_rank() / 2;

    auto elements_per_batch = size / batch_sz;
    auto elements_per_batch_per_block = elements_per_batch / grid.group_dim().x;

    extern __shared__ int shared[]; // stages_count * elements_per_batch_per_block *
↪sizeof(int) bytes
    size_t shared_offset[stages_count];
    for (int s = 0; s < stages_count; ++s) shared_offset[s] = s * elements_per_batch_
↪per_block;

    __shared__ cuda::pipeline_shared_state<
        cuda::thread_scope::thread_scope_block,
        stages_count
    > shared_state;
    cuda::pipeline pipeline = cuda::make_pipeline(block, &shared_state, thread_role);

    // Each thread block processes `batch_sz` batches.
    // Compute offset of the batch `batch` of this thread block in global memory:
    auto block_batch = [&](size_t batch) -> int {
      return elements_per_batch * batch + elements_per_batch_per_block * blockIdx.x;
    };

    for (size_t compute_batch = 0, fetch_batch = 0; compute_batch < batch_sz;
↪++compute_batch) {
        // The outer loop iterates over the computation of the batches
        for (; fetch_batch < batch_sz && fetch_batch < (compute_batch + stages_count);
↪ ++fetch_batch) {
            // This inner loop iterates over the memory transfers, making sure that the
↪pipeline is always full
```

(continues on next page)

(continued from previous page)

```
            if (thread_role == cuda::pipeline_role::producer) {
                // Only the producer threads schedule asynchronous memcpys:
                pipeline.producer_acquire();
                size_t shared_idx = fetch_batch % stages_count;
                size_t batch_idx = fetch_batch;
                size_t global_batch_idx = block_batch(batch_idx) + thread_idx;
                size_t shared_batch_idx = shared_offset[shared_idx] + thread_idx;
                cuda::memcpy_async(shared + shared_batch_idx, global_in + global_
↪batch_idx, sizeof(int), pipeline);
                pipeline.producer_commit();
            }
        }
        if (thread_role == cuda::pipeline_role::consumer) {
            // Only the consumer threads compute:
            pipeline.consumer_wait();
            size_t shared_idx = compute_batch % stages_count;
            size_t global_batch_idx = block_batch(compute_batch) + thread_idx;
            size_t shared_batch_idx = shared_offset[shared_idx] + thread_idx;
            compute(global_out + global_batch_idx, *(shared + shared_batch_idx));
            pipeline.consumer_release();
        }
    }
}
```

There are some optimizations that `pipeline` performs, for example, when all threads are both producers and consumers, but in general, the cost of supporting all these features cannot be fully eliminated. For example, `pipeline` stores and uses a set of barriers in shared memory for synchronization, which is not really necessary if all threads in the block participate in the pipeline.

For the particular case in which all threads in the block participate in the `pipeline`, we can do better than `pipeline<thread_scope_block>` by using a `pipeline<thread_scope_thread>` combined with `__syncthreads()`:

```
template<size_t stages_count>
__global__ void with_staging_scope_thread(int* global_out, int const* global_in, size_
↪t size, size_t batch_sz) {
    auto grid = cooperative_groups::this_grid();
    auto block = cooperative_groups::this_thread_block();
    auto thread = cooperative_groups::this_thread();
    assert(size == batch_sz * grid.size()); // Assume input size fits batch_sz * grid_
↪size

    extern __shared__ int shared[]; // stages_count * block.size() * sizeof(int) bytes
    size_t shared_offset[stages_count];
    for (int s = 0; s < stages_count; ++s) shared_offset[s] = s * block.size();

    // No pipeline::shared_state needed
    cuda::pipeline<cuda::thread_scope_thread> pipeline = cuda::make_pipeline();

    auto block_batch = [&](size_t batch) -> int {
        return block.group_index().x * block.size() + grid.size() * batch;
    };

    for (size_t compute_batch = 0, fetch_batch = 0; compute_batch < batch_sz;
↪++compute_batch) {
        for (; fetch_batch < batch_sz && fetch_batch < (compute_batch + stages_count);
↪ ++fetch_batch) {
```

(continues on next page)

(continued from previous page)

```
            pipeline.producer_acquire();
            size_t shared_idx = fetch_batch % stages_count;
            size_t batch_idx = fetch_batch;
            // Each thread fetches its own data:
            size_t thread_batch_idx = block_batch(batch_idx) + threadIdx.x;
            // The copy is performed by a single `thread` and the size of the batch is
↪now that of a single element:
            cuda::memcpy_async(thread, shared + shared_offset[shared_idx] + threadIdx.
↪x, global_in + thread_batch_idx, sizeof(int), pipeline);
            pipeline.producer_commit();
        }
        pipeline.consumer_wait();
        block.sync(); // __syncthreads: All memcpy_async of all threads in the block
↪for this stage have completed here
        int shared_idx = compute_batch % stages_count;
        int batch_idx = compute_batch;
        compute(global_out + block_batch(batch_idx), shared + shared_offset[shared_
↪idx]);
        pipeline.consumer_release();
    }
}
```

If the `compute` operation only reads shared memory written to by other threads in the same warp as the current thread, `__syncwarp()` suffices.

10.28.3. Pipeline Interface

The complete API documentation for `cuda::memcpy_async` is provided in the libcudacxx API documentation along with some examples.

The `pipeline` interface requires

- at least CUDA 11.0,
- at least ISO C++ 2011 compatibility, e.g., to be compiled with `-std=c++11`, and
- `#include <cuda/pipeline>`.

For a C-like interface, when compiling without ISO C++ 2011 compatibility, see *Pipeline Primitives Interface*.

10.28.4. Pipeline Primitives Interface

Pipeline primitives are a C-like interface for `memcpy_async` functionality. The pipeline primitives interface is available by including the `<cuda_pipeline.h>` header. When compiling without ISO C++ 2011 compatibility, include the `<cuda_pipeline_primitives.h>` header.

10.28.4.1 memcpy_async Primitive

```
void __pipeline_memcpy_async(void* __restrict__ dst_shared,
                             const void* __restrict__ src_global,
                             size_t size_and_align,
                             size_t zfill=0);
```

- Request that the following operation be submitted for asynchronous evaluation:

```
size_t i = 0;
for (; i < size_and_align - zfill; ++i) ((char*)dst_shared)[i] = ((char*)src_
↪global)[i]; /* copy */
for (; i < size_and_align; ++i) ((char*)dst_shared)[i] = 0; /* zero-fill */
```

- Requirements:
  - `dst_shared` must be a pointer to the shared memory destination for the `memcpy_async`.
  - `src_global` must be a pointer to the global memory source for the `memcpy_async`.
  - `size_and_align` must be 4, 8, or 16.
  - `zfill <= size_and_align`.
  - `size_and_align` must be the alignment of `dst_shared` and `src_global`.
- It is a race condition for any thread to modify the source memory or observe the destination memory prior to waiting for the `memcpy_async` operation to complete. Between submitting a `memcpy_async` operation and waiting for its completion, any of the following actions introduces a race condition:
  - Loading from `dst_shared`.
  - Storing to `dst_shared` or `src_global`.
  - Applying an atomic update to `dst_shared` or `src_global`.

10.28.4.2 Commit Primitive

```
void __pipeline_commit();
```

- Commit submitted `memcpy_async` to the pipeline as the current batch.

10.28.4.3 Wait Primitive

```
void __pipeline_wait_prior(size_t N);
```

- Let `{0, 1, 2, ..., L}` be the sequence of indices associated with invocations of `__pipeline_commit()` by a given thread.
- Wait for completion of batches *at least* up to and including `L-N`.

10.28.4.4 Arrive On Barrier Primitive

```
void __pipeline_arrive_on(__mbarrier_t* bar);
```

- `bar` points to a barrier in shared memory.
- Increments the barrier arrival count by one, when all memcpy_async operations sequenced before this call have completed, the arrival count is decremented by one and hence the net effect on the arrival count is zero. It is user's responsibility to make sure that the increment on the arrival count does not exceed `__mbarrier_maximum_count()`.

10.29. Asynchronous Data Copies using the Tensor Memory Accelerator (TMA)

Many applications require movement of large amounts of data from and to global memory. Often, the data is laid out in global memory as a multi-dimensional array with non-sequential data acess patterns. To reduce global memory usage, sub-tiles of such arrays are copied to shared memory before use in computations. The loading and storing involves address-calculations that can be error-prone and repetitive. To offload these computations, Compute Capability 9.0 introduces the Tensor Memory Accelerator (TMA). The primary goal of TMA is to provide an efficient data transfer mechanism from global memory to shared memory for multi-dimensional arrays.

Naming. Tensor memory accelerator (TMA) is a broad term used to refer to the features described in this section. For the purpose of forward-compatibility and to reduce discrepancies with the PTX ISA, the text in this section refers to TMA operations as either bulk-asynchronous copies or bulk tensor asynchronous copies, depending on the specific type of copy used. The term "bulk" is used to contrast these operations with the asynchronous memory operations described in the previous sections.

Dimensions. TMA supports copying both one-dimensional and multi-dimensional arrays (up to 5-dimensional). The programming model for **bulk-asynchronous copies** of one-dimensional contiguous arrays is different from the programming model for **bulk tensor asynchronous copies** of multi-dimensional arrays. To perform a bulk tensor asynchronous copy of a multi-dimensional array, the hardware requires a tensor map. This object describes the layout of the multi-dimensional array in global and shared memory. A tensor map is typically created on the host using the cuTensorMapEncode API and then transferred from host to device as a `const` kernel parameter annotated with `__grid_constant__`. The tensor map is transferred from host to device as a `const` kernel parameter annotated with `__grid_constant__`, and can be used on the device to copy a tile of data between shared and global memory. In contrast, performing a bulk-asynchronous copy of a contiguous one-dimensional array does not require a tensor map: it can be performed on-device with a pointer and size parameter.

Source and destination. The source and destination addresses of bulk-asynchronous copy operations can be in shared or global memory. The operations can read data from global to shared memory, write data from shared to global memory, and also copy from shared memory to *Distributed Shared Memory* of another block in the same cluster. In addition, when in a cluster, a bulk-asynchronous operation can be specified as being multicast. In this case, data can be transferred from global memory to the shared memory of multiple blocks within the cluster. The multicast feature is optimized for target architecture `sm_90a` and may have significantly reduced performance on other targets. Hence, it is advised to be used with compute architecture `sm_90a`.

Asynchronous. Data transfers using TMA are *asynchronous*. This allows the initiating thread to continue computing while the hardware asynchronously copies the data. **Whether the data transfer oc-**

curs asynchronously in practice is up to the hardware implementation and may change in the future. There are several completion mechanisms that bulk-asynchronous operations can use to signal that they have completed. When the operation reads from global to shared memory, any thread in the block can wait for the data to be readable in shared memory by waiting on a *Shared Memory Barrier*. When the bulk-asynchronous operation writes data from shared memory to global or distributed shared memory, only the initiating thread can wait for the operation to have completed. This is accomplished using a *bulk async-group* based completion mechanism. A table describing the completion mechanisms can be found below and in the PTX ISA.

Table 8: Asynchronous copies with possible source and destinations memory spaces and completion mechanisms. An empty cell indicates that a source-destination pair is not supported.

Direction		**Completion mechanism**	
Destination	**Source**	**Asynchronous copy**	**Bulk-asynchronous copy (TMA)**
Global	Global		
Global	Shared::cta		Bulk async-group
Shared::cta	Global	Async-group, mbarrier	Mbarrier
Shared::cluster	Global		Mbarrier (multicast)
Shared::cta	Shared::cluster		Mbarrier
Shared::cta	Shared::cta		

10.29.1. Using TMA to transfer one-dimensional arrays

This section demonstrates how to write a simple kernel that read-modify-writes a one-dimensional array using TMA. This shows how to how to load and store data using bulk-asynchronous copies, as well as how to synchronize threads of execution with those copies.

The code of the kernel is included below. Some functionality requires inline PTX assembly that is currently made available through libcu++. The availability of these wrappers can be checked with the following code:

```
#if defined(__CUDA_MINIMUM_ARCH__) && __CUDA_MINIMUM_ARCH__ < 900
static_assert(false, "Device code is being compiled with older architectures that are
↪incompatible with TMA.");
#endif // __CUDA_MINIMUM_ARCH__
```

The kernel goes through the following stages:

1. Initialize shared memory barrier.
2. Initiate bulk-asynchronous copy of a block of memory from global to shared memory.
3. Arrive and wait on the shared memory barrier.
4. Increment the shared memory buffer values.
5. Wait for shared memory writes to be visible to the subsequent bulk-asynchronous copy, i.e., order the shared memory writes in the async proxy before the next step.
6. Initiate bulk-asynchronous copy of the buffer in shared memory to global memory.

7. Wait at end of kernel for bulk-asynchronous copy to have finished reading shared memory.

```
#include <cuda/barrier>
#include <cuda/ptx>
using barrier = cuda::barrier<cuda::thread_scope_block>;
namespace ptx = cuda::ptx;

static constexpr size_t buf_len = 1024;
__global__ void add_one_kernel(int* data, size_t offset)
{
  // Shared memory buffer. The destination shared memory buffer of
  // a bulk operations should be 16 byte aligned.
  __shared__ alignas(16) int smem_data[buf_len];

  // 1. a) Initialize shared memory barrier with the number of threads participating in
↪the barrier.
  //    b) Make initialized barrier visible in async proxy.
  #pragma nv_diag_suppress static_var_with_dynamic_init
  __shared__ barrier bar;
  if (threadIdx.x == 0) {
    init(&bar, blockDim.x);                      // a)
    ptx::fence_proxy_async(ptx::space_shared);   // b)
  }
  __syncthreads();

  // 2. Initiate TMA transfer to copy global to shared memory.
  if (threadIdx.x == 0) {
    // 3a. cuda::memcpy_async arrives on the barrier and communicates
    //     how many bytes are expected to come in (the transaction count)
    cuda::memcpy_async(
        smem_data,
        data + offset,
        cuda::aligned_size_t<16>(sizeof(smem_data)),
        bar
    );
  }
  // 3b. All threads arrive on the barrier
  barrier::arrival_token token = bar.arrive();

  // 3c. Wait for the data to have arrived.
  bar.wait(std::move(token));

  // 4. Compute saxpy and write back to shared memory
  for (int i = threadIdx.x; i < buf_len; i += blockDim.x) {
    smem_data[i] += 1;
  }

  // 5. Wait for shared memory writes to be visible to TMA engine.
  ptx::fence_proxy_async(ptx::space_shared);   // b)
  __syncthreads();
  // After syncthreads, writes by all threads are visible to TMA engine.

  // 6. Initiate TMA transfer to copy shared memory to global memory
  if (threadIdx.x == 0) {
    ptx::cp_async_bulk(
        ptx::space_global,
        ptx::space_shared,
        data + offset, smem_data, sizeof(smem_data));
```

(continues on next page)

(continued from previous page)

```
    // 7. Wait for TMA transfer to have finished reading shared memory.
    // Create a "bulk async-group" out of the previous bulk copy operation.
    ptx::cp_async_bulk_commit_group();
    // Wait for the group to have completed reading from shared memory.
    ptx::cp_async_bulk_wait_group_read(ptx::n32_t<0>());
  }
}
```

Barrier initialization. The barrier is initialized with the number of threads participating in the block. As a result, the barrier will flip only if all threads have arrived on this barrier. Shared memory barriers are described in more detail in *Asynchronous Data Copies using cuda::barrier*. To make the initialized barrier visible to subsequent bulk-asynchronous copies, the `fence.proxy.async.shared::cta` instruction is used. This instruction ensures that subsequent bulk-asynchronous copy operations operate on the initialized barrier.

TMA read. The bulk-asynchronous copy instruction directs the hardware to copy a large chunk of data into shared memory, and to update the transaction count of the shared memory barrier after completing the read. In general, issuing as few bulk copies with as big a size as possible results in the best performance. Because the copy can be performed asynchronously by the hardware, it is not necessary to split the copy into smaller chunks.

The thread that initiates the bulk-asynchronous copy operation arrives at the barrier using `mbarrier.expect_tx`. This is automatically performed by `cuda::memcpy_async`. This tells the barrier that the thread has arrived and also how many bytes (tx / transactions) are expected to arrive. Only a single thread has to update the expected transaction count. If multiple threads update the transaction count, the expected transaction will be the sum of the updates. The barrier will only flip once all threads have arrived **and** all bytes have arrived. Once the barrier has flipped, the bytes are safe to read from shared memory, both by the threads as well as by subsequent bulk-asynchronous copies. More information about barrier transaction accounting can be found in the PTX ISA.

Barrier wait. Waiting for the barrier to flip is done using `mbarrier.try_wait`. It can either return true, indicating that the wait is over, or return false, which may mean that the wait timed out. The while loop waits for completion, and retries on time-out.

SMEM write and sync. The increment of the buffer values reads and writes to shared memory. To make the writes visible to subsequent bulk-asynchronous copies, the `fence.proxy.async.shared::cta` instruction is used. This orders the writes to shared memory before subsequent reads from bulk-asynchronous copy operations, which read through the async proxy. So each thread first orders the writes to objects in shared memory in the async proxy via the `fence.proxy.async.shared::cta`, and these operations by all threads are ordered before the async operation performed in thread 0 using `__syncthreads()`.

TMA write and sync. The write from shared to global memory is again initiated by a single thread. The completion of the write is not tracked by a shared memory barrier. Instead, a thread-local mechanism is used. Multiple writes can be batched into a so-called *bulk async-group*. Afterwards, the thread can wait for all operations in this group to have completed reading from shared memory (as in the code above) or to have completed writing to global memory, making the writes visible to the initiating thread. For more information, refer to the PTX ISA documentation of cp.async.bulk.wait_group. Note that the bulk-asynchronous and non-bulk asynchronous copy instructions have different async-groups: there exist both `cp.async.wait_group` and `cp.async.bulk.wait_group` instructions.

The bulk-asynchronous instructions have specific alignment requirements on their source and destination addresses. More information can be found in the table below.

Table 9: Alignment requirements for one-dimensional bulk-asynchronous operations in Compute Capability 9.0.

Address / Size	Alignment
Global memory address	Must be 16 byte aligned.
Shared memory address	Must be 16 byte aligned.
Shared memory barrier address	Must be 8 byte aligned (this is guaranteed by `cuda::barrier`).
Size of transfer	Must be a multiple of 16 bytes.

10.29.2. Using TMA to transfer multi-dimensional arrays

The primary difference between the one-dimensional and multi-dimensional case is that a tensor map must be created on the host and passed to the CUDA kernel. This section describes how to create a tensor map using the CUDA driver API, how to pass it to device, and how to use it on device.

Driver API. A tensor map is created using the cuTensorMapEncodeTiled driver API. This API can be accessed by linking to the driver directly (`-lcuda`) or by using the cudaGetDriverEntryPointByVersion API. Below, we show how to get a pointer to the `cuTensorMapEncodeTiled` API. For more information, refer to *Driver Entry Point Access*.

```
#include <cudaTypedefs.h> // PFN_cuTensorMapEncodeTiled, CUtensorMap

PFN_cuTensorMapEncodeTiled_v12000 get_cuTensorMapEncodeTiled() {
  // Get pointer to cuTensorMapEncodeTiled
  cudaDriverEntryPointQueryResult driver_status;
  void* cuTensorMapEncodeTiled_ptr = nullptr;
  CUDA_CHECK(cudaGetDriverEntryPointByVersion("cuTensorMapEncodeTiled", &
↪cuTensorMapEncodeTiled_ptr, 12000, cudaEnableDefault, &driver_status));
  assert(driver_status == cudaDriverEntryPointSuccess);

  return reinterpret_cast<PFN_cuTensorMapEncodeTiled_v12000>(cuTensorMapEncodeTiled_
↪ptr);
}
```

Creation. Creating a tensor map requires many parameters. Among them are the base pointer to an array in global memory, the size of the array (in number of elements), the stride from one row to the next (in bytes), the size of the shared memory buffer (in number of elements). The code below creates a tensor map to describe a two-dimensional row-major array of size `GMEM_HEIGHT x GMEM_WIDTH`. Note the order of the parameters: the fastest moving dimension comes first.

```
  CUtensorMap tensor_map{};
  // rank is the number of dimensions of the array.
  constexpr uint32_t rank = 2;
  uint64_t size[rank] = {GMEM_WIDTH, GMEM_HEIGHT};
  // The stride is the number of bytes to traverse from the first element of one row to
↪the next.
  // It must be a multiple of 16.
  uint64_t stride[rank - 1] = {GMEM_WIDTH * sizeof(int)};
  // The box_size is the size of the shared memory buffer that is used as the
  // destination of a TMA transfer.
  uint32_t box_size[rank] = {SMEM_WIDTH, SMEM_HEIGHT};
```

(continues on next page)

(continued from previous page)

```
  // The distance between elements in units of sizeof(element). A stride of 2
  // can be used to load only the real component of a complex-valued tensor, for
↪instance.
  uint32_t elem_stride[rank] = {1, 1};

  // Get a function pointer to the cuTensorMapEncodeTiled driver API.
  auto cuTensorMapEncodeTiled = get_cuTensorMapEncodeTiled();

  // Create the tensor descriptor.
  CUresult res = cuTensorMapEncodeTiled(
    &tensor_map,                // CUtensorMap *tensorMap,
    CUtensorMapDataType::CU_TENSOR_MAP_DATA_TYPE_INT32,
    rank,                       // cuuint32_t tensorRank,
    tensor_ptr,                 // void *globalAddress,
    size,                       // const cuuint64_t *globalDim,
    stride,                     // const cuuint64_t *globalStrides,
    box_size,                   // const cuuint32_t *boxDim,
    elem_stride,                // const cuuint32_t *elementStrides,
    // Interleave patterns can be used to accelerate loading of values that
    // are less than 4 bytes long.
    CUtensorMapInterleave::CU_TENSOR_MAP_INTERLEAVE_NONE,
    // Swizzling can be used to avoid shared memory bank conflicts.
    CUtensorMapSwizzle::CU_TENSOR_MAP_SWIZZLE_NONE,
    // L2 Promotion can be used to widen the effect of a cache-policy to a wider
    // set of L2 cache lines.
    CUtensorMapL2promotion::CU_TENSOR_MAP_L2_PROMOTION_NONE,
    // Any element that is outside of bounds will be set to zero by the TMA transfer.
    CUtensorMapFloatOOBfill::CU_TENSOR_MAP_FLOAT_OOB_FILL_NONE
  );
```

Host-to-device transfer. There are three ways to make a tensor map accessible to device code. The recommended approach is to pass the tensor map as a const `__grid_constant__` parameter to a kernel. The other possibilities are copying the tensor map into device `__constant__` memory using `cudaMemcpyToSymbol` or accessing it via global memory. When passing the tensor map as a parameter, some versions of the GCC C++ compiler issue the warning "the ABI for passing parameters with 64-byte alignment has changed in GCC 4.6". This warning can be ignored.

```
#include <cuda.h>

__global__ void kernel(const __grid_constant__ CUtensorMap tensor_map)
{
   // Use tensor_map here.
}
int main() {
  CUtensorMap map;
  // [ ..Initialize map.. ]
  kernel<<<1, 1>>>(map);
}
```

As an alternative to the `__grid_constant__` kernel parameter, a global *constant* variable can be used. An example is included below.

```
#include <cuda.h>

__constant__ CUtensorMap global_tensor_map;
__global__ void kernel()
```

(continues on next page)

(continued from previous page)

```
{
  // Use global_tensor_map here.
}
int main() {
  CUtensorMap local_tensor_map;
  // [ ..Initialize map.. ]
  cudaMemcpyToSymbol(global_tensor_map, &local_tensor_map, sizeof(CUtensorMap));
  kernel<<<1, 1>>>();
}
```

Finally, it is possible to copy the tensor map to global memory. Using a pointer to a tensor map in global device memory requires a fence in each thread block before any thread in the block uses the updated tensor map. Further uses of the tensor map by that thread block do not need to be fenced unless the tensor map is modified again. Note that this mechanism may be slower than the two mechanisms described above.

```
#include <cuda.h>
#include <cuda/ptx>
namespace ptx = cuda::ptx;

__device__ CUtensorMap global_tensor_map;
__global__ void kernel(CUtensorMap *tensor_map)
{
  // Fence acquire tensor map:
  ptx::n32_t<128> size_bytes;
  // Since the tensor map was modified from the host using cudaMemcpy,
  // the scope should be .sys.
  ptx::fence_proxy_tensormap_generic(
     ptx::sem_acquire, ptx::scope_sys, tensor_map, size_bytes
 );
 // Safe to use tensor_map after fence inside this thread..
}
int main() {
  CUtensorMap local_tensor_map;
  // [ ..Initialize map.. ]
  cudaMemcpy(&global_tensor_map, &local_tensor_map, sizeof(CUtensorMap),
↪cudaMemcpyHostToDevice);
  kernel<<<1, 1>>>(global_tensor_map);
}
```

Use. The kernel below loads a 2D tile of size `SMEM_HEIGHT` x `SMEM_WIDTH` from a larger 2D array. The top-left corner of the tile is indicated by the indices `x` and `y`. The tile is loaded into shared memory, modified, and written back to global memory.

```
#include <cuda.h>          // CUtensormap
#include <cuda/barrier>
using barrier = cuda::barrier<cuda::thread_scope_block>;
namespace cde = cuda::device::experimental;

__global__ void kernel(const __grid_constant__ CUtensorMap tensor_map, int x, int y) {
  // The destination shared memory buffer of a bulk tensor operation should be
  // 128 byte aligned.
  __shared__ alignas(128) int smem_buffer[SMEM_HEIGHT][SMEM_WIDTH];

  // Initialize shared memory barrier with the number of threads participating in the
↪barrier.
```

(continues on next page)

(continued from previous page)

```
  #pragma nv_diag_suppress static_var_with_dynamic_init
  __shared__ barrier bar;

  if (threadIdx.x == 0) {
    // Initialize barrier. All `blockDim.x` threads in block participate.
    init(&bar, blockDim.x);
    // Make initialized barrier visible in async proxy.
    cde::fence_proxy_async_shared_cta();
  }
  // Syncthreads so initialized barrier is visible to all threads.
  __syncthreads();

  barrier::arrival_token token;
  if (threadIdx.x == 0) {
    // Initiate bulk tensor copy.
    cde::cp_async_bulk_tensor_2d_global_to_shared(&smem_buffer, &tensor_map, x, y,
↪bar);
    // Arrive on the barrier and tell how many bytes are expected to come in.
    token = cuda::device::barrier_arrive_tx(bar, 1, sizeof(smem_buffer));
  } else {
    // Other threads just arrive.
    token = bar.arrive();
  }
  // Wait for the data to have arrived.
  bar.wait(std::move(token));

  // Symbolically modify a value in shared memory.
  smem_buffer[0][threadIdx.x] += threadIdx.x;

  // Wait for shared memory writes to be visible to TMA engine.
  cde::fence_proxy_async_shared_cta();
  __syncthreads();
  // After syncthreads, writes by all threads are visible to TMA engine.

  // Initiate TMA transfer to copy shared memory to global memory
  if (threadIdx.x == 0) {
    cde::cp_async_bulk_tensor_2d_shared_to_global(&tensor_map, x, y, &smem_buffer);
    // Wait for TMA transfer to have finished reading shared memory.
    // Create a "bulk async-group" out of the previous bulk copy operation.
    cde::cp_async_bulk_commit_group();
    // Wait for the group to have completed reading from shared memory.
    cde::cp_async_bulk_wait_group_read<0>();
  }

  // Destroy barrier. This invalidates the memory region of the barrier. If
  // further computations were to take place in the kernel, this allows the
  // memory location of the shared memory barrier to be reused.
  if (threadIdx.x == 0) {
    (&bar)->~barrier();
  }
}
```

Negative indices and out of bounds. When part of the tile that is being *read* from global to shared memory is out of bounds, the shared memory that corresponds to the out of bounds area is zero-filled. The top-left corner indices of the tile may also be negative. When *writing* from shared to global memory, parts of the tile may be out of bounds, but the top left corner cannot have any negative indices.

Size and stride. The size of a tensor is the number of elements along one dimension. All sizes must be greater than one. The stride is the number of bytes between elements of the same dimension. For instance, a *4 x 4* matrix of integers has sizes 4 and 4. Since it has 4 bytes per element, the strides are 4 and 16 bytes. Due to alignment requirements, a *4 x 3* row-major matrix of integers must have strides of 4 and 16 bytes as well. Each row is padded with 4 extra bytes to ensure that the start of the next row is aligned to 16 bytes. For more information regarding alignment, refer to Table 10.

Table 10: Alignment requirements for multi-dimensional bulk tensor asynchronous copy operations in Compute Capability 9.0.

Address / Size	Alignment
Global memory address	Must be 16 byte aligned.
Global memory sizes	Must be greater than or equal to one. Does not have to be a multiple of 16 bytes.
Global memory strides	Must be multiples of 16 bytes.
Shared memory address	Must be 128 byte aligned.
Shared memory barrier address	Must be 8 byte aligned (this is guaranteed by `cuda::barrier`).
Size of transfer	Must be a multiple of 16 bytes.

10.29.2.1 Multi-dimensional TMA PTX wrappers

Below, the PTX instructions are ordered by their use in the example code above.

The cp.async.bulk.tensor instructions initiate a bulk tensor asynchronous copy between global and shared memory. The wrappers below read from global to shared memory and write from shared to global memory.

```
// https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#data-movement-and-
↪conversion-instructions-cp-async-bulk-tensor
inline __device__
void cuda::device::experimental::cp_async_bulk_tensor_1d_global_to_shared(
    void *dest, const CUtensorMap *tensor_map , int c0, cuda::barrier<cuda::thread_
↪scope_block> &bar
);

// https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#data-movement-and-
↪conversion-instructions-cp-async-bulk-tensor
inline __device__
void cuda::device::experimental::cp_async_bulk_tensor_2d_global_to_shared(
    void *dest, const CUtensorMap *tensor_map , int c0, int c1, cuda::barrier
↪<cuda::thread_scope_block> &bar
);

// https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#data-movement-and-
↪conversion-instructions-cp-async-bulk-tensor
inline __device__
void cuda::device::experimental::cp_async_bulk_tensor_3d_global_to_shared(
    void *dest, const CUtensorMap *tensor_map, int c0, int c1, int c2, cuda::barrier
↪<cuda::thread_scope_block> &bar
```

(continues on next page)

(continued from previous page)

```
);

// https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#data-movement-and-
↪conversion-instructions-cp-async-bulk-tensor
inline __device__
void cuda::device::experimental::cp_async_bulk_tensor_4d_global_to_shared(
    void *dest, const CUtensorMap *tensor_map , int c0, int c1, int c2, int c3,
↪cuda::barrier<cuda::thread_scope_block> &bar
);

// https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#data-movement-and-
↪conversion-instructions-cp-async-bulk-tensor
inline __device__
void cuda::device::experimental::cp_async_bulk_tensor_5d_global_to_shared(
    void *dest, const CUtensorMap *tensor_map , int c0, int c1, int c2, int c3, int
↪c4, cuda::barrier<cuda::thread_scope_block> &bar
);
```

```
// https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#data-movement-and-
↪conversion-instructions-cp-async-bulk-tensor
inline __device__
void cuda::device::experimental::cp_async_bulk_tensor_1d_shared_to_global(
    const CUtensorMap *tensor_map, int c0, const void *src
);

// https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#data-movement-and-
↪conversion-instructions-cp-async-bulk-tensor
inline __device__
void cuda::device::experimental::cp_async_bulk_tensor_2d_shared_to_global(
    const CUtensorMap *tensor_map, int c0, int c1, const void *src
);

// https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#data-movement-and-
↪conversion-instructions-cp-async-bulk-tensor
inline __device__
void cuda::device::experimental::cp_async_bulk_tensor_3d_shared_to_global(
    const CUtensorMap *tensor_map, int c0, int c1, int c2, const void *src
);

// https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#data-movement-and-
↪conversion-instructions-cp-async-bulk-tensor
inline __device__
void cuda::device::experimental::cp_async_bulk_tensor_4d_shared_to_global(
    const CUtensorMap *tensor_map, int c0, int c1, int c2, int c3, const void *src
);

// https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#data-movement-and-
↪conversion-instructions-cp-async-bulk-tensor
inline __device__
void cuda::device::experimental::cp_async_bulk_tensor_5d_shared_to_global(
    const CUtensorMap *tensor_map, int c0, int c1, int c2, int c3, int c4, const void
↪*src
);
```

10.29.3. TMA Swizzle

By default, the TMA engine loads data to shared memory in the same order as it is laid out in global memory. However, this layout may not be optimal for certain shared memory access patterns, as it could cause shared memory bank conflicts. To improve performance and reduce bank conflicts, we can change the shared memory layout by applying a 'swizzle pattern'.

Shared memory has 32 banks that are organized such that successive 32-bit words map to successive banks. Each bank has a bandwidth of 32 bits per clock cycle. When loading and storing shared memory, bank conflicts arise if the same bank is used multiple times within a transaction, resulting in reduced bandwidth. See *Shared Memory*, bank conflicts.

To ensure that data is laid out in shared memory in such a way that user code can avoid shared memory bank conflicts, the TMA engine can be instructed to 'swizzle' the data before storing it in shared memory and 'unswizzle' it when copying the data back from shared memory to global memory. The tensor map encodes the 'swizzle mode' indicating which swizzle pattern is used.

10.29.3.1 Example 'Matrix Transpose'

An example is the transpose of a matrix where data is mapped from row to column first access. The data is stored row major in global memory, but we want to also access it column wise in shared memory, which leads to bank conflicts. However, by using the 128 bytes 'swizzle' mode and new shared memory indices, they are eliminated.

In the example, we load an 8x8 matrix of type `int4`, stored as row major in global memory to shared memory. Then, each set of eight threads loads a row from the shared memory buffer and stores it to a column in a separate transpose shared memory buffer. This results in an eight-way bank conflict when storing. Finally, the transpose buffer is written back to global memory.

To avoid bank conflicts, the `CU_TENSOR_MAP_SWIZZLE_128B` layout can be used. This layout matches the 128 bytes row length and changes the shared memory layout in a way that both the column wise and row wise access don't require the same banks per transaction.

The two tables, Figure 27 and Figure 28, below show the normal and the swizzled shared memory layout of the 8x8 matrix of type `int4` and its transpose matrix. The colors indicate which of the eight groups of four banks the matrix element is mapped to, and the margin row and margin column list the global memory row and column indices. The entries show the shared memory indices of the 16-byte matrix elements.

```
__global__ void kernel_tma(const __grid_constant__ CUtensorMap tensor_map) {
   // The destination shared memory buffer of a bulk tensor operation
   // with the 128-byte swizzle mode, it should be 1024 bytes aligned.
   __shared__ alignas(1024) int4 smem_buffer[8][8];
   __shared__ alignas(1024) int4 smem_buffer_tr[8][8];

   // Initialize shared memory barrier
   #pragma nv_diag_suppress static_var_with_dynamic_init
   __shared__ barrier bar;

   if (threadIdx.x == 0) {
     init(&bar, blockDim.x);
     cde::fence_proxy_async_shared_cta();
   }

   __syncthreads();
```

(continues on next page)

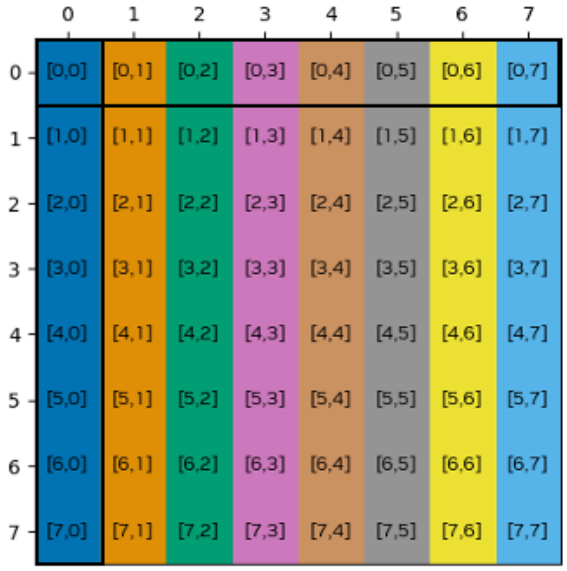


Figure 27: In the shared memory data layout without swizzle, the shared memory indices are equivalent to the global memory indices. Per load instruction, one row is read and stored in a column of the transpose buffer. Since all matrix elements of the column in the transpose fall in the same bank, the store must be serialized, resulting in eight store transactions, giving an eight-way bank conflict per stored column.

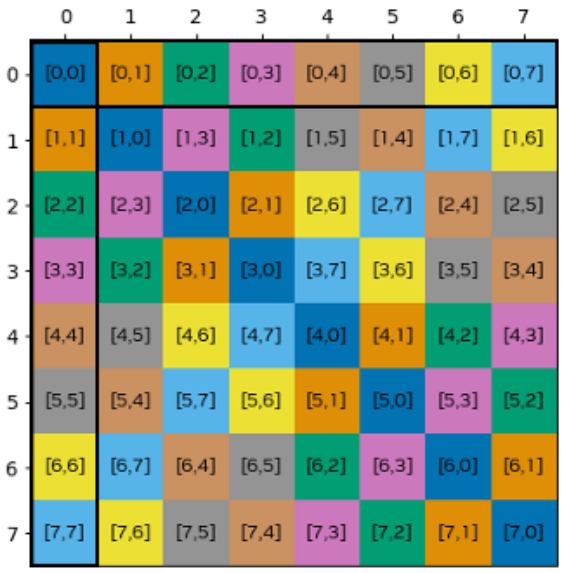


Figure 28: The shared memory data layout with CU_TENSOR_MAP_SWIZZLE_128B swizzle. One row is stored in a column, each matrix element is from a different bank for both the rows and columns, and so without any bank conflicts.

(continued from previous page)

```
  barrier::arrival_token token;
  if (threadIdx.x == 0) {
    // Initiate bulk tensor copy from global to shared memory,
    // in the same way as without swizzle.
    cde::cp_async_bulk_tensor_2d_global_to_shared(&smem_buffer, &tensor_map, 0, 0,
↪bar);
    token = cuda::device::barrier_arrive_tx(bar, 1, sizeof(smem_buffer));
  } else {
    token = bar.arrive();
  }

  bar.wait(std::move(token));

  /* Matrix transpose
   *  When using the normal shared memory layout, there are eight
   *  8-way shared memory bank conflict when storing to the transpose.
   *  When enabling the 128-byte swizzle pattern and using the according access
↪pattern,
   *  they are eliminated both for load and store. */
  for(int sidx_j =threadIdx.x; sidx_j < 8; sidx_j+= blockDim.x){
     for(int sidx_i = 0; sidx_i < 8; ++sidx_i){
        const int swiz_j_idx = (sidx_i % 8) ^ sidx_j;
        const int swiz_i_idx_tr = (sidx_j % 8) ^ sidx_i;
        smem_buffer_tr[sidx_j][swiz_i_idx_tr] = smem_buffer[sidx_i][swiz_j_idx];
     }
```

(continues on next page)

(continued from previous page)

```
   }

   // Wait for shared memory writes to be visible to TMA engine.
   cde::fence_proxy_async_shared_cta();
   __syncthreads();

   /* Initiate TMA transfer to copy the transposed shared memory buffer back to global
↪memory,
    * it will 'unswizzle' the data. */
   if (threadIdx.x == 0) {
      cde::cp_async_bulk_tensor_2d_shared_to_global(&tensor_map, 0, 0, &smem_buffer_
↪tr);
      cde::cp_async_bulk_commit_group();
      cde::cp_async_bulk_wait_group_read<0>();
   }

   // Destroy barrier
   if (threadIdx.x == 0) {
     (&bar)->~barrier();
   }
}

// --------------------------------- main ----------------------------------------

int main(){

...
   void* tensor_ptr = d_data;

   CUtensorMap tensor_map{};
   // rank is the number of dimensions of the array.
   constexpr uint32_t rank = 2;
   // global memory size
   uint64_t size[rank] = {4*8, 8};
   // global memory stride, must be a multiple of 16.
   uint64_t stride[rank - 1] = {8 * sizeof(int4)};
   // The inner shared memory box dimension in bytes, equal to the swizzle span.
   uint32_t box_size[rank] = {4*8, 8};

   uint32_t elem_stride[rank] = {1, 1};

   // Create the tensor descriptor.
   CUresult res = cuTensorMapEncodeTiled(
       &tensor_map,                // CUtensorMap *tensorMap,
       CUtensorMapDataType::CU_TENSOR_MAP_DATA_TYPE_INT32,
       rank,                       // cuuint32_t tensorRank,
       tensor_ptr,                 // void *globalAddress,
       size,                       // const cuuint64_t *globalDim,
       stride,                     // const cuuint64_t *globalStrides,
       box_size,                   // const cuuint32_t *boxDim,
       elem_stride,                // const cuuint32_t *elementStrides,
       CUtensorMapInterleave::CU_TENSOR_MAP_INTERLEAVE_NONE,
       // Using a swizzle pattern of 128 bytes.
       CUtensorMapSwizzle::CU_TENSOR_MAP_SWIZZLE_128B,
       CUtensorMapL2promotion::CU_TENSOR_MAP_L2_PROMOTION_NONE,
       CUtensorMapFloatOOBfill::CU_TENSOR_MAP_FLOAT_OOB_FILL_NONE
```

(continues on next page)

(continued from previous page)

```
    );

    kernel_tma<<<1, 8>>>(tensor_map);
  ...
}
```

Remark. This example is supposed to show the use of swizzle and 'as-is' is not performant nor does it scale beyond the given dimensions.

Explanation. During data transfer, the TMA engine shuffles the data according to the swizzle pattern, as described in the following tables. These swizzle patterns define the mapping of the 16-byte chunks along the swizzle width to subgroups of four banks. It is of type `CUtensorMapSwizzle` and has four options: none, 32 bytes, 64 bytes and 128 bytes. Note that the shared memory box's inner dimension must be less or equal to the span of the swizzle pattern.

10.29.3.2 The Swizzle Modes

As previously mentioned, there are four swizzle modes. The following tables show the different swizzle patterns, including the relation of the new shared memory indices. The tables define the mapping of the 16-byte chunks along the 128 bytes to eight subgroups of four banks.

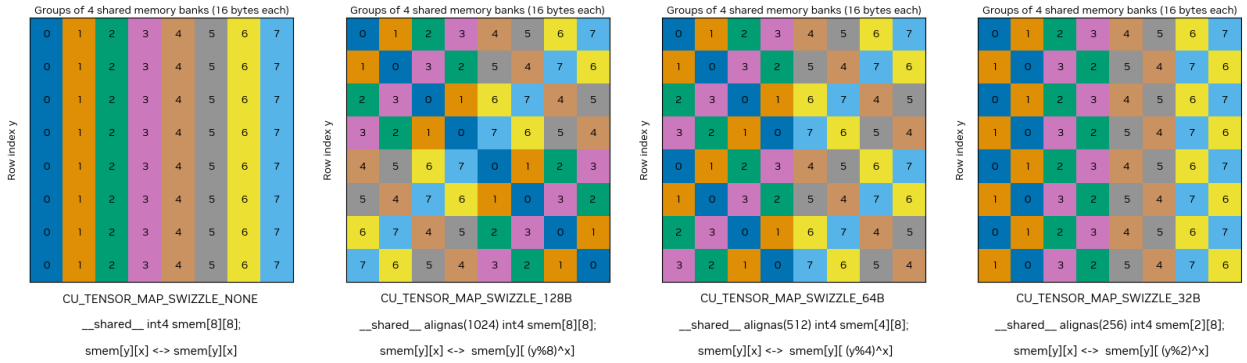


Figure 29: An Overview of TMA Swizzle Patterns

Considerations. When applying a TMA swizzle pattern, it is crucial to adhere to specific memory requirements:

- **Global memory alignment:** Global memory must be aligned to 128 bytes.
- **Shared memory alignment:** For simplicity shared memory should be aligned according to the number of bytes after which the swizzle pattern repeats. When the shared memory buffer is not aligned by the number of bytes by which the swizzle pattern repeats itself, there is an offset between the swizzle pattern and the shared memory. See *comment*, below.
- **Inner dimension:** The inner dimension of the shared memory block must meet the size requirements specified in Table 12. If these requirements are not met, the instruction is considered invalid. Additionally, if the swizzle width exceeds the inner dimension, ensure that the shared memory is allocated to accommodate the full swizzle width.
- **Granularity:** The granularity of swizzle mapping is fixed at 16 bytes. This means that data is organized and accessed in chunks of 16 bytes, which must be considered when planning memory layout and access patterns.

Swizzle Pattern Pointer Offset Computation. Here, we describe how to determine the offset between the swizzle pattern and the shared memory, when the shared memory buffer is not aligned by the number of bytes by which the swizzle pattern repeats itself. When using TMA, the shared memory is required to be aligned to 128 bytes. To find how many times the shared memory buffer relative to the swizzle pattern is shifted by that, apply the corresponding offset formula.

Table 11: Swizzle Pattern Pointer Offset Formula and Index Relation

Swizzle Mode	Offset Formula	Index Relation
CU_TENSOR_MAP_SWIZZLE_128B	(reinterpret_cast<uintptr_t>(smem_ptr)/128)%8	smem[y][x] <-> smem[y][((y+offset)%8)^x]
CU_TENSOR_MAP_SWIZZLE_64B	(reinterpret_cast<uintptr_t>(smem_ptr)/128)%4	smem[y][x] <-> smem[y][((y+offset)%4)^x]
CU_TENSOR_MAP_SWIZZLE_32B	(reinterpret_cast<uintptr_t>(smem_ptr)/128)%2	smem[y][x] <-> smem[y][((y+offset)%2)^x]

In Figure 29, this offset represents the initial row offset, thus, in the swizzle index calculation, it is added to the row index y. The following snippet shows how to access the swizzled shared memory in the CU_TENSOR_MAP_SWIZZLE_128B mode.

```
data_t* smem_ptr = &smem[0][0];
int offset = (reinterpret_cast<uintptr_t>(smem_ptr)/128)%8;
smem[y][((y+offset)%8)^x] = ...
```

Summary. The following Table 12 summarizes the requirements and properties of the different swizzle patterns for Compute Capability 9.

Table 12: Requirements and properties of the different swizzle patterns for Compute Capability 9

Pattern	Swizzle width	Shared box's inner dimension	Repeats after	Shared memory alignment	Global memory alignment
CU_TENSOR_MAP_SWIZZLE_128B	128 bytes	<=128 bytes	1024 bytes	128 bytes	128 bytes
CU_TENSOR_MAP_SWIZZLE_64B	64 bytes	<=64 bytes	512 bytes	128 bytes	128 bytes
CU_TENSOR_MAP_SWIZZLE_32B	32 bytes	<=32 bytes	256 bytes	128 bytes	128 bytes
CU_TENSOR_MAP_SWIZZLE_NONE (default)				128 bytes	16 bytes

10.30. Encoding a Tensor Map on Device

Previous sections have described how to create a tensor map on the host using the CUDA driver API.

This section explains how to encode a tiled-type tensor map on device. This is useful in situations where the typical way of transferring the tensor map (using `const __grid_constant__` kernel parameters) is undesirable, for instance, when processing a batch of tensors of various sizes in a single kernel launch.

The recommended pattern is as follows:

1. Create a tensor map "template", `template_tensor_map`, using the Driver API on the host.
2. In a device kernel, copy the `template_tensor_map`, modify the copy, store in global memory, and appropriately fence.
3. Use the tensor map in a kernel with appropriate fencing.

The high-level code structure is as follows:

```
// Initialize device context:
CUDA_CHECK(cudaDeviceSynchronize());

// Create a tensor map template using the cuTensorMapEncodeTiled driver function
CUtensorMap template_tensor_map = make_tensormap_template();

// Allocate tensor map and tensor in global memory
CUtensorMap* global_tensor_map;
CUDA_CHECK(cudaMalloc(&global_tensor_map, sizeof(CUtensorMap)));
char* global_buf;
CUDA_CHECK(cudaMalloc(&global_buf, 8 * 256));

// Fill global buffer with data.
fill_global_buf<<<1, 1>>>(global_buf);

// Define the parameters of the tensor map that will be created on device.
tensormap_params p{};
p.global_address    = global_buf;
p.rank              = 2;
p.box_dim[0]        = 128; // The box in shared memory has half the width of the full
↪buffer
p.box_dim[1]        = 4;   // The box in shared memory has half the height of the full
↪buffer
p.global_dim[0]     = 256; //
p.global_dim[1]     = 8;   //
p.global_stride[0]  = 256; //
p.element_stride[0] = 1;   //
p.element_stride[1] = 1;   //

// Encode global_tensor_map on device:
encode_tensor_map<<<1, 32>>>(template_tensor_map, p, global_tensor_map);

// Use it from another kernel:
consume_tensor_map<<<1, 1>>>(global_tensor_map);

// Check for errors:
CUDA_CHECK(cudaDeviceSynchronize());
```

The following sections describe the high-level steps. Throughout the examples, the following `tensormap_params` struct contains the new values of the fields to be updated. It is included here to reference when reading the examples.

```
struct tensormap_params {
  void* global_address;
  int rank;
  uint32_t box_dim[5];
  uint64_t global_dim[5];
  size_t global_stride[4];
  uint32_t element_stride[5];
};
```

10.30.1. Device-side Encoding and Modification of a Tensor Map

The recommended process of encoding a tensor map in global memory proceeds as follows.

1. Pass an existing tensor map, the `template_tensor_map`, to the kernel. In contrast to kernels that use the tensor map in a `cp.async.bulk.tensor` instruction, this may be done in any way: a pointer to global memory, kernel parameter, a `__const___` variable, and so on.
2. Copy-initialize a tensor map in shared memory with the template_tensor_map value.
3. Modify the tensor map in shared memory using the cuda::ptx::tensormap_replace functions. These functions wrap the tensormap.replace PTX instruction, which can be used to modify any field of a tiled-type tensor map, including the base address, size, stride, and so on.
4. Using the cuda::ptx::tensormap_copy_fenceproxy function, copy the modified tensor map from shared memory to global memory and perform any necessary fencing.

The following code contains a kernel that follows these steps. For completeness, it modifies all the fields of the tensor map. Typically, a kernel will modify just a few fields.

In this kernel, `template_tensor_map` is passed as a kernel parameter. This is the preferred way of moving `template_tensor_map` from the host to the device. If the kernel is intended to update an existing tensor map in device memory, it can take a pointer to the existing tensor map to modify.

Note: The format of the tensor map may change over time. Therefore, the cuda::ptx::tensormap_replace functions and corresponding tensormap.replace.tile PTX instructions are marked as specific to sm_90a. To use them, compile using `nvcc -arch sm_90a .....`

Tip: On sm_90a, a zero-initialized buffer in shared memory may also be used as the initial tensor map value. This enables encoding a tensor map purely on device, without using the driver API to encode the `template_tensor_map value`.

Note: On-device modification is only supported for tiled-type tensor maps; other tensor map types cannot be modified on device. For more information on the tensor map types, refer to the Driver API reference.

```
#include <cuda/ptx>

namespace ptx = cuda::ptx;

// launch with 1 warp.
__launch_bounds__(32)
__global__ void encode_tensor_map(const __grid_constant__ CUtensorMap template_tensor_
↪map, tensormap_params p, CUtensorMap* out) {
   __shared__ alignas(128) CUtensorMap smem_tmap;
   if (threadIdx.x == 0) {
      // Copy template to shared memory:
      smem_tmap = template_tensor_map;

      const auto space_shared = ptx::space_shared;
      ptx::tensormap_replace_global_address(space_shared, &smem_tmap, p.global_
↪address);
      // For field .rank, the operand new_val must be ones less than the desired
      // tensor rank as this field uses zero-based numbering.
      ptx::tensormap_replace_rank(space_shared, &smem_tmap, p.rank - 1);

      // Set box dimensions:
      if (0 < p.rank) { ptx::tensormap_replace_box_dim(space_shared, &smem_tmap,
↪ptx::n32_t<0>{}, p.box_dim[0]); }
      if (1 < p.rank) { ptx::tensormap_replace_box_dim(space_shared, &smem_tmap,
↪ptx::n32_t<1>{}, p.box_dim[1]); }
      if (2 < p.rank) { ptx::tensormap_replace_box_dim(space_shared, &smem_tmap,
↪ptx::n32_t<2>{}, p.box_dim[2]); }
      if (3 < p.rank) { ptx::tensormap_replace_box_dim(space_shared, &smem_tmap,
↪ptx::n32_t<3>{}, p.box_dim[3]); }
      if (4 < p.rank) { ptx::tensormap_replace_box_dim(space_shared, &smem_tmap,
↪ptx::n32_t<4>{}, p.box_dim[4]); }
      // Set global dimensions:
      if (0 < p.rank) { ptx::tensormap_replace_global_dim(space_shared, &smem_tmap,
↪ptx::n32_t<0>{}, (uint32_t) p.global_dim[0]); }
      if (1 < p.rank) { ptx::tensormap_replace_global_dim(space_shared, &smem_tmap,
↪ptx::n32_t<1>{}, (uint32_t) p.global_dim[1]); }
      if (2 < p.rank) { ptx::tensormap_replace_global_dim(space_shared, &smem_tmap,
↪ptx::n32_t<2>{}, (uint32_t) p.global_dim[2]); }
      if (3 < p.rank) { ptx::tensormap_replace_global_dim(space_shared, &smem_tmap,
↪ptx::n32_t<3>{}, (uint32_t) p.global_dim[3]); }
      if (4 < p.rank) { ptx::tensormap_replace_global_dim(space_shared, &smem_tmap,
↪ptx::n32_t<4>{}, (uint32_t) p.global_dim[4]); }
      // Set global stride:
      if (1 < p.rank) { ptx::tensormap_replace_global_stride(space_shared, &smem_tmap,
↪ ptx::n32_t<0>{}, p.global_stride[0]); }
      if (2 < p.rank) { ptx::tensormap_replace_global_stride(space_shared, &smem_tmap,
↪ ptx::n32_t<1>{}, p.global_stride[1]); }
      if (3 < p.rank) { ptx::tensormap_replace_global_stride(space_shared, &smem_tmap,
↪ ptx::n32_t<2>{}, p.global_stride[2]); }
      if (4 < p.rank) { ptx::tensormap_replace_global_stride(space_shared, &smem_tmap,
↪ ptx::n32_t<3>{}, p.global_stride[3]); }
      // Set element stride:
      if (0 < p.rank) { ptx::tensormap_replace_element_size(space_shared, &smem_tmap,
↪ptx::n32_t<0>{}, p.element_stride[0]); }
      if (1 < p.rank) { ptx::tensormap_replace_element_size(space_shared, &smem_tmap,
↪ptx::n32_t<1>{}, p.element_stride[1]); }
      if (2 < p.rank) { ptx::tensormap_replace_element_size(space_shared, &smem_tmap,
↪ptx::n32_t<2>{}, p.element_stride[2]); }
```

(continues on next page)

(continued from previous page)

```
      if (3 < p.rank) { ptx::tensormap_replace_element_size(space_shared, &smem_tmap,
↪ptx::n32_t<3>{}, p.element_stride[3]); }
      if (4 < p.rank) { ptx::tensormap_replace_element_size(space_shared, &smem_tmap,
↪ptx::n32_t<4>{}, p.element_stride[4]); }

      // These constants are documented in this table:
      // https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#tensormap-
↪new-val-validity
      auto u8_elem_type = ptx::n32_t<0>{};
      ptx::tensormap_replace_elemtype(space_shared, &smem_tmap, u8_elem_type);
      auto no_interleave = ptx::n32_t<0>{};
      ptx::tensormap_replace_interleave_layout(space_shared, &smem_tmap, no_
↪interleave);
      auto no_swizzle = ptx::n32_t<0>{};
      ptx::tensormap_replace_swizzle_mode(space_shared, &smem_tmap, no_swizzle);
      auto zero_fill = ptx::n32_t<0>{};
      ptx::tensormap_replace_fill_mode(space_shared, &smem_tmap, zero_fill);
   }
   // Synchronize the modifications with other threads in warp
   __syncwarp();
   // Copy the tensor map to global memory collectively with threads in the warp.
   // In addition: make the updated tensor map visible to other threads on device that
   // for use with cp.async.bulk.
   ptx::n32_t<128> bytes_128;
   ptx::tensormap_cp_fenceproxy(ptx::sem_release, ptx::scope_gpu, out, &smem_tmap,
↪bytes_128);
}
```

10.30.2. Usage of a Modified Tensor Map

In contrast to using a tensor map that is passed as a `const __grid_constant__` kernel parameter, using a tensor map in global memory requires explicitly establishing a release-acquire pattern in the tensor map proxy between the threads that modify the tensor map and the threads that use it.

The release part of the pattern was shown in the previous section. It is accomplished using the cuda::ptx::tensormap.cp_fenceproxy function.

The acquire part is accomplished using the cuda::ptx::fence_proxy_tensormap_generic function that wraps the fence.proxy.tensormap::generic.acquire instruction. If the two threads participating in the release-acquire pattern are on the same device, the `.gpu` scope suffices. If the threads are on different devices, the `.sys` scope must be used. Once a tensor map has been acquired by one thread, it can be used by other threads in the block after sufficient synchronization, for example, using `__syncthreads()`. The thread that uses the tensor map and the thread that performs the fence must be in the same block. That is, if the threads are in, for example, two different thread blocks of the same cluster, the same grid, or a different kernel, synchronization APIs such as `cooperative_groups::cluster` or `grid_group::sync()` or stream-order synchronization do not suffice to establish ordering for tensor map updates, that is, threads in these other thread blocks still need to acquire the tensor map proxy at the right scope before using the updated tensor map. If there are no intermediate modifications, the fence does not have to be repeated before each `cp.async.bulk.tensor` instruction.

The `fence` and subsequent use of the tensor map is shown in the following example.

```
// Consumer of tensor map in global memory:
__global__ void consume_tensor_map(CUtensorMap* tensor_map) {
  // Fence acquire tensor map:
  ptx::n32_t<128> size_bytes;
  ptx::fence_proxy_tensormap_generic(ptx::sem_acquire, ptx::scope_sys, tensor_map,
↪size_bytes);
  // Safe to use tensor_map after fence..

  __shared__ uint64_t bar;
  __shared__ alignas(128) char smem_buf[4][128];

  if (threadIdx.x == 0) {
    // Initialize barrier
    ptx::mbarrier_init(&bar, 1);
    // Make barrier init visible in async proxy, i.e., to TMA engine
    ptx::fence_proxy_async(ptx::space_shared);
    // Issue TMA request
    ptx::cp_async_bulk_tensor(ptx::space_cluster, ptx::space_global, smem_buf, tensor_
↪map, {0, 0}, &bar);

    // Arrive on barrier. Expect 4 * 128 bytes.
    ptx::mbarrier_arrive_expect_tx(ptx::sem_release, ptx::scope_cta, ptx::space_
↪shared, &bar, sizeof(smem_buf));
  }
  const int parity = 0;
  // Wait for load to have completed
  while (!ptx::mbarrier_try_wait_parity(&bar, parity)) {}

  // print items:
  printf("Got:\n\n");
  for (int j = 0; j < 4; ++j) {
    for (int i = 0; i < 128; ++i) {
      printf("%3d ", smem_buf[j][i]);
      if (i % 32 == 31) { printf("\n"); };
    }
    printf("\n");
  }
}
```

10.30.3. Creating a Template Tensor Map Value Using the Driver API

The following code creates a minimal tiled-type tensor map that can be subsequently modified on device.

```
CUtensorMap make_tensormap_template() {
  CUtensorMap template_tensor_map{};
  auto cuTensorMapEncodeTiled = get_cuTensorMapEncodeTiled();

  uint32_t dims_32         = 16;
  uint64_t dims_strides_64 = 16;
  uint32_t elem_strides    = 1;
```

(continues on next page)

(continued from previous page)

```
  // Create the tensor descriptor.
  CUresult res = cuTensorMapEncodeTiled(
    &template_tensor_map, // CUtensorMap *tensorMap,
    CUtensorMapDataType::CU_TENSOR_MAP_DATA_TYPE_UINT8,
    1,                // cuuint32_t tensorRank,
    nullptr,          // void *globalAddress,
    &dims_strides_64, // const cuuint64_t *globalDim,
    &dims_strides_64, // const cuuint64_t *globalStrides,
    &dims_32,         // const cuuint32_t *boxDim,
    &elem_strides,    // const cuuint32_t *elementStrides,
    CUtensorMapInterleave::CU_TENSOR_MAP_INTERLEAVE_NONE,
    CUtensorMapSwizzle::CU_TENSOR_MAP_SWIZZLE_NONE,
    CUtensorMapL2promotion::CU_TENSOR_MAP_L2_PROMOTION_NONE,
    CUtensorMapFloatOOBfill::CU_TENSOR_MAP_FLOAT_OOB_FILL_NONE);

  CU_CHECK(res);
  return template_tensor_map;
}
```

10.31. Profiler Counter Function

Each multiprocessor has a set of sixteen hardware counters that an application can increment with a single instruction by calling the `__prof_trigger()` function.

```
void __prof_trigger(int counter);
```

increments by one per warp the per-multiprocessor hardware counter of index `counter`. Counters 8 to 15 are reserved and should not be used by applications.

The value of counters 0, 1, ..., 7 can be obtained via `nvprof` by `nvprof --events prof_trigger_0x` where x is 0, 1, ..., 7. All counters are reset before each kernel launch (note that when collecting counters, kernel launches are synchronous as mentioned in *Concurrent Execution between Host and Device*).

10.32. Assertion

Assertion is only supported by devices of compute capability 2.x and higher.

```
void assert(int expression);
```

stops the kernel execution if `expression` is equal to zero. If the program is run within a debugger, this triggers a breakpoint and the debugger can be used to inspect the current state of the device. Otherwise, each thread for which `expression` is equal to zero prints a message to *stderr* after synchronization with the host via `cudaDeviceSynchronize()`, `cudaStreamSynchronize()`, or `cudaEventSynchronize()`. The format of this message is as follows:

```
<filename>:<line number>:<function>:
block: [blockId.x,blockId.x,blockIdx.z],
thread: [threadIdx.x,threadIdx.y,threadIdx.z]
Assertion `<expression>` failed.
```

Any subsequent host-side synchronization calls made for the same device will return `cudaErrorAssert`. No more commands can be sent to this device until `cudaDeviceReset()` is called to reinitialize the device.

If `expression` is different from zero, the kernel execution is unaffected.

For example, the following program from source file *test.cu*

```
#include <assert.h>

__global__ void testAssert(void)
{
    int is_one = 1;
    int should_be_one = 0;

    // This will have no effect
    assert(is_one);

    // This will halt kernel execution
    assert(should_be_one);
}

int main(int argc, char* argv[])
{
    testAssert<<<1,1>>>();
    cudaDeviceSynchronize();

    return 0;
}
```

will output:

```
test.cu:19: void testAssert(): block: [0,0,0], thread: [0,0,0] Assertion `should_be_
↪one` failed.
```

Assertions are for debugging purposes. They can affect performance and it is therefore recommended to disable them in production code. They can be disabled at compile time by defining the `NDEBUG` preprocessor macro before including `assert.h`. Note that `expression` should not be an expression with side effects (something like `(++i > 0)`, for example), otherwise disabling the assertion will affect the functionality of the code.

10.33. Trap function

A trap operation can be initiated by calling the `__trap()` function from any device thread.

```
void __trap();
```

The execution of the kernel is aborted and an interrupt is raised in the host program.

10.34. Breakpoint Function

Execution of a kernel function can be suspended by calling the `__brkpt()` function from any device thread.

```
void __brkpt();
```

10.35. Formatted Output

Formatted output is only supported by devices of compute capability 2.x and higher.

```
int printf(const char *format[, arg, ...]);
```

prints formatted output from a kernel to a host-side output stream.

The in-kernel `printf()` function behaves in a similar way to the standard C-library `printf()` function, and the user is referred to the host system's manual pages for a complete description of `printf()` behavior. In essence, the string passed in as `format` is output to a stream on the host, with substitutions made from the argument list wherever a format specifier is encountered. Supported format specifiers are listed below.

The `printf()` command is executed as any other device-side function: per-thread, and in the context of the calling thread. From a multi-threaded kernel, this means that a straightforward call to `printf()` will be executed by every thread, using that thread's data as specified. Multiple versions of the output string will then appear at the host stream, once for each thread which encountered the `printf()`.

It is up to the programmer to limit the output to a single thread if only a single output string is desired (see *Examples* for an illustrative example).

Unlike the C-standard `printf()`, which returns the number of characters printed, CUDA's `printf()` returns the number of arguments parsed. If no arguments follow the format string, 0 is returned. If the format string is NULL, -1 is returned. If an internal error occurs, -2 is returned.

10.35.1. Format Specifiers

As for standard `printf()`, format specifiers take the form: `%[flags][width][.precision][size]type`

The following fields are supported (see widely-available documentation for a complete description of all behaviors):

- Flags: `'#' ' ' '0' '+' '-'`
- Width: `'*' '0-9'`
- Precision: `'0-9'`
- Size: `'h' 'l' 'll'`
- Type: `"%cdiouxXpeEfgGaAs"`

Note that CUDA's `printf()` will accept any combination of flag, width, precision, size and type, whether or not overall they form a valid format specifier. In other words, "`%hd`" will be accepted and printf will expect a double-precision variable in the corresponding location in the argument list.

10.35.2. Limitations

Final formatting of the `printf()` output takes place on the host system. This means that the format string must be understood by the host-system's compiler and C library. Every effort has been made to ensure that the format specifiers supported by CUDA's printf function form a universal subset from the most common host compilers, but exact behavior will be host-OS-dependent.

As described in *Format Specifiers*, `printf()` will accept *all* combinations of valid flags and types. This is because it cannot determine what will and will not be valid on the host system where the final output is formatted. The effect of this is that output may be undefined if the program emits a format string which contains invalid combinations.

The `printf()` command can accept at most 32 arguments in addition to the format string. Additional arguments beyond this will be ignored, and the format specifier output as-is.

Owing to the differing size of the `long` type on 64-bit Windows platforms (four bytes on 64-bit Windows platforms, eight bytes on other 64-bit platforms), a kernel which is compiled on a non-Windows 64-bit machine but then run on a win64 machine will see corrupted output for all format strings which include "`%ld`". It is recommended that the compilation platform matches the execution platform to ensure safety.

The output buffer for `printf()` is set to a fixed size before kernel launch (see *Associated Host-Side API*). It is circular and if more output is produced during kernel execution than can fit in the buffer, older output is overwritten. It is flushed only when one of these actions is performed:

- Kernel launch via `<<<>>>` or `cuLaunchKernel()` (at the start of the launch, and if the CUDA_LAUNCH_BLOCKING environment variable is set to 1, at the end of the launch as well),
- Synchronization via `cudaDeviceSynchronize()`, `cuCtxSynchronize()`, `cudaStreamSynchronize()`, `cuStreamSynchronize()`, `cudaEventSynchronize()`, or `cuEventSynchronize()`,
- Memory copies via any blocking version of `cudaMemcpy*()` or `cuMemcpy*()`,
- Module loading/unloading via `cuModuleLoad()` or `cuModuleUnload()`,
- Context destruction via `cudaDeviceReset()` or `cuCtxDestroy()`.
- Prior to executing a stream callback added by `cudaLaunchHostFunc` or `cuLaunchHostFunc`.

Note that the buffer is not flushed automatically when the program exits. The user must call `cudaDeviceReset()` or `cuCtxDestroy()` explicitly, as shown in the examples below.

Internally `printf()` uses a shared data structure and so it is possible that calling `printf()` might change the order of execution of threads. In particular, a thread which calls `printf()` might take a longer execution path than one which does not call `printf()`, and that path length is dependent upon the parameters of the `printf()`. Note, however, that CUDA makes no guarantees of thread execution order except at explicit `__syncthreads()` barriers, so it is impossible to tell whether execution order has been modified by `printf()` or by other scheduling behavior in the hardware.

10.35.3. Associated Host-Side API

The following API functions get and set the size of the buffer used to transfer the `printf()` arguments and internal metadata to the host (default is 1 megabyte):

- `cudaDeviceGetLimit(size_t* size,cudaLimitPrintfFifoSize)`
- `cudaDeviceSetLimit(cudaLimitPrintfFifoSize, size_t size)`

10.35.4. Examples

The following code sample:

```
#include <stdio.h>

__global__ void helloCUDA(float f)
{
    printf("Hello thread %d, f=%f\n", threadIdx.x, f);
}

int main()
{
    helloCUDA<<<1, 5>>>(1.2345f);
    cudaDeviceSynchronize();
    return 0;
}
```

will output:

```
Hello thread 2, f=1.2345
Hello thread 1, f=1.2345
Hello thread 4, f=1.2345
Hello thread 0, f=1.2345
Hello thread 3, f=1.2345
```

Notice how each thread encounters the `printf()` command, so there are as many lines of output as there were threads launched in the grid. As expected, global values (i.e., `float f`) are common between all threads, and local values (i.e., `threadIdx.x`) are distinct per-thread.

The following code sample:

```
#include <stdio.h>

__global__ void helloCUDA(float f)
{
    if (threadIdx.x == 0)
        printf("Hello thread %d, f=%f\n", threadIdx.x, f) ;
}

int main()
{
    helloCUDA<<<1, 5>>>(1.2345f);
    cudaDeviceSynchronize();
    return 0;
}
```

will output:

```
Hello thread 0, f=1.2345
```

Self-evidently, the `if()` statement limits which threads will call `printf`, so that only a single line of output is seen.

10.36. Dynamic Global Memory Allocation and Operations

Dynamic global memory allocation and operations are only supported by devices of compute capability 2.x and higher.

```
__host__ __device__ void* malloc(size_t size);
__device__ void *__nv_aligned_device_malloc(size_t size, size_t align);
__host__ __device__  void free(void* ptr);
```

allocate and free memory dynamically from a fixed-size heap in global memory.

```
__host__ __device__ void* memcpy(void* dest, const void* src, size_t size);
```

copy `size` bytes from the memory location pointed by `src` to the memory location pointed by `dest`.

```
__host__ __device__ void* memset(void* ptr, int value, size_t size);
```

set `size` bytes of memory block pointed by `ptr` to `value` (interpreted as an unsigned char).

The CUDA in-kernel `malloc()` function allocates at least `size` bytes from the device heap and returns a pointer to the allocated memory or NULL if insufficient memory exists to fulfill the request. The returned pointer is guaranteed to be aligned to a 16-byte boundary.

The CUDA in-kernel `__nv_aligned_device_malloc()` function allocates at least `size` bytes from the device heap and returns a pointer to the allocated memory or NULL if insufficient memory exists to fulfill the requested size or alignment. The address of the allocated memory will be a multiple of `align`. `align` must be a non-zero power of 2.

The CUDA in-kernel `free()` function deallocates the memory pointed to by `ptr`, which must have been returned by a previous call to `malloc()` or `__nv_aligned_device_malloc()`. If `ptr` is NULL, the call to `free()` is ignored. Repeated calls to `free()` with the same `ptr` has undefined behavior.

The memory allocated by a given CUDA thread via `malloc()` or `__nv_aligned_device_malloc()` remains allocated for the lifetime of the CUDA context, or until it is explicitly released by a call to `free()`. It can be used by any other CUDA threads even from subsequent kernel launches. Any CUDA thread may free memory allocated by another thread, but care should be taken to ensure that the same pointer is not freed more than once.

10.36.1. Heap Memory Allocation

The device memory heap has a fixed size that must be specified before any program using `malloc()`, `__nv_aligned_device_malloc()` or `free()` is loaded into the context. A default heap of eight megabytes is allocated if any program uses `malloc()` or `__nv_aligned_device_malloc()` without explicitly specifying the heap size.

The following API functions get and set the heap size:

- `cudaDeviceGetLimit(size_t* size, cudaLimitMallocHeapSize)`
- `cudaDeviceSetLimit(cudaLimitMallocHeapSize, size_t size)`

The heap size granted will be at least `size` bytes. `cuCtxGetLimit()`and `cudaDeviceGetLimit()` return the currently requested heap size.

The actual memory allocation for the heap occurs when a module is loaded into the context, either explicitly via the CUDA driver API (see *Module*), or implicitly via the CUDA runtime API (see *CUDA Runtime*). If the memory allocation fails, the module load will generate a `CUDA_ERROR_SHARED_OBJECT_INIT_FAILED` error.

Heap size cannot be changed once a module load has occurred and it does not resize dynamically according to need.

Memory reserved for the device heap is in addition to memory allocated through host-side CUDA API calls such as `cudaMalloc()`.

10.36.2. Interoperability with Host Memory API

Memory allocated via device `malloc()` or `__nv_aligned_device_malloc()` cannot be freed using the runtime (i.e., by calling any of the free memory functions from *Device Memory*).

Similarly, memory allocated via the runtime (i.e., by calling any of the memory allocation functions from *Device Memory*) cannot be freed via `free()`.

In addition, memory allocated by a call to `malloc()` or `__nv_aligned_device_malloc()` in device code cannot be used in any runtime or driver API calls (i.e. cudaMemcpy, cudaMemset, etc).

10.36.3. Examples

10.36.3.1 Per Thread Allocation

The following code sample:

```
#include <stdlib.h>
#include <stdio.h>

__global__ void mallocTest()
{
    size_t size = 123;
    char* ptr = (char*)malloc(size);
    memset(ptr, 0, size);
    printf("Thread %d got pointer: %p\n", threadIdx.x, ptr);
```

(continues on next page)

(continued from previous page)

```
    free(ptr);
}

int main()
{
    // Set a heap size of 128 megabytes. Note that this must
    // be done before any kernel is launched.
    cudaDeviceSetLimit(cudaLimitMallocHeapSize, 128*1024*1024);
    mallocTest<<<1, 5>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

will output:

```
Thread 0 got pointer: 00057020
Thread 1 got pointer: 0005708c
Thread 2 got pointer: 000570f8
Thread 3 got pointer: 00057164
Thread 4 got pointer: 000571d0
```

Notice how each thread encounters the `malloc()` and `memset()` commands and so receives and initializes its own allocation. (Exact pointer values will vary: these are illustrative.)

10.36.3.2 Per Thread Block Allocation

```
#include <stdlib.h>

__global__ void mallocTest()
{
    __shared__ int* data;

    // The first thread in the block does the allocation and then
    // shares the pointer with all other threads through shared memory,
    // so that access can easily be coalesced.
    // 64 bytes per thread are allocated.
    if (threadIdx.x == 0) {
        size_t size = blockDim.x * 64;
        data = (int*)malloc(size);
    }
    __syncthreads();

    // Check for failure
    if (data == NULL)
        return;

    // Threads index into the memory, ensuring coalescence
    int* ptr = data;
    for (int i = 0; i < 64; ++i)
        ptr[i * blockDim.x + threadIdx.x] = threadIdx.x;

    // Ensure all threads complete before freeing
    __syncthreads();

    // Only one thread may free the memory!
```

(continues on next page)

(continued from previous page)

```
    if (threadIdx.x == 0)
        free(data);
}

int main()
{
    cudaDeviceSetLimit(cudaLimitMallocHeapSize, 128*1024*1024);
    mallocTest<<<10, 128>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

10.36.3.3 Allocation Persisting Between Kernel Launches

```
#include <stdlib.h>
#include <stdio.h>

#define NUM_BLOCKS 20

__device__ int* dataptr[NUM_BLOCKS]; // Per-block pointer

__global__ void allocmem()
{
    // Only the first thread in the block does the allocation
    // since we want only one allocation per block.
    if (threadIdx.x == 0)
        dataptr[blockIdx.x] = (int*)malloc(blockDim.x * 4);
    __syncthreads();

    // Check for failure
    if (dataptr[blockIdx.x] == NULL)
        return;

    // Zero the data with all threads in parallel
    dataptr[blockIdx.x][threadIdx.x] = 0;
}

// Simple example: store thread ID into each element
__global__ void usemem()
{
    int* ptr = dataptr[blockIdx.x];
    if (ptr != NULL)
        ptr[threadIdx.x] += threadIdx.x;
}

// Print the content of the buffer before freeing it
__global__ void freemem()
{
    int* ptr = dataptr[blockIdx.x];
    if (ptr != NULL)
        printf("Block %d, Thread %d: final value = %d\n",
                        blockIdx.x, threadIdx.x, ptr[threadIdx.x]);

    // Only free from one thread!
```

(continues on next page)

(continued from previous page)

```
    if (threadIdx.x == 0)
        free(ptr);
}

int main()
{
    cudaDeviceSetLimit(cudaLimitMallocHeapSize, 128*1024*1024);

    // Allocate memory
    allocmem<<< NUM_BLOCKS, 10 >>>();

    // Use memory
    usemem<<< NUM_BLOCKS, 10 >>>();
    usemem<<< NUM_BLOCKS, 10 >>>();
    usemem<<< NUM_BLOCKS, 10 >>>();

    // Free memory
    freemem<<< NUM_BLOCKS, 10 >>>();

    cudaDeviceSynchronize();

    return 0;
}
```

10.37. Execution Configuration

Any call to a `__global__` function must specify the *execution configuration* for that call. The execution configuration defines the dimension of the grid and blocks that will be used to execute the function on the device, as well as the associated stream (see *CUDA Runtime* for a description of streams).

The execution configuration is specified by inserting an expression of the form `<<< Dg, Db, Ns, S >>>` between the function name and the parenthesized argument list, where:

- `Dg` is of type `dim3` (see *dim3*) and specifies the dimension and size of the grid, such that `Dg.x * Dg.y * Dg.z` equals the number of blocks being launched;
- `Db` is of type `dim3` (see *dim3*) and specifies the dimension and size of each block, such that `Db.x * Db.y * Db.z` equals the number of threads per block;
- `Ns` is of type `size_t` and specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory; this dynamically allocated memory is used by any of the variables declared as an external array as mentioned in *__shared__*; `Ns` is an optional argument which defaults to 0;
- `S` is of type `cudaStream_t` and specifies the associated stream; `S` is an optional argument which defaults to 0.

As an example, a function declared as

```
__global__ void Func(float* parameter);
```

must be called like this:

```
Func<<< Dg, Db, Ns >>>(parameter);
```

The arguments to the execution configuration are evaluated before the actual function arguments.

The function call will fail if `Dg` or `Db` are greater than the maximum sizes allowed for the device as specified in *Compute Capabilities*, or if `Ns` is greater than the maximum amount of shared memory available on the device, minus the amount of shared memory required for static allocation.

Compute capability 9.0 and above allows users to specify compile time thread block cluster dimensions, so that the kernel can use the cluster hierarchy in CUDA. Compile time cluster dimension can be specified using `__cluster_dims__([x, [y, [z]]])`. The example below shows compile time cluster size of 2 in X dimension and 1 in Y and Z dimension.

```
__global__ void __cluster_dims__(2, 1, 1) Func(float* parameter);
```

The default form of `__cluster_dims__()` specifies that a kernel is to be launched as a cluster grid. By not specifying a cluster dimension, the user is free to specify the dimension at launch time. Not specifying a dimension at launch time will result in a launch time error.

Thread block cluster dimensions can also be specified at runtime and kernel with the cluster can be launched using `cudaLaunchKernelEx` API. The API takes a configuration argument of type `cudaLaunchConfig_t`, kernel function pointer and kernel arguments. Runtime kernel configuration is shown in the example below.

```
__global__ void Func(float* parameter);


// Kernel invocation with runtime cluster size
{
    cudaLaunchConfig_t config = {0};
    // The grid dimension is not affected by cluster launch, and is still enumerated
    // using number of blocks.
    // The grid dimension should be a multiple of cluster size.
    config.gridDim = Dg;
    config.blockDim = Db;
    config.dynamicSmemBytes = Ns;

    cudaLaunchAttribute attribute[1];
    attribute[0].id = cudaLaunchAttributeClusterDimension;
    attribute[0].val.clusterDim.x = 2; // Cluster size in X-dimension
    attribute[0].val.clusterDim.y = 1;
    attribute[0].val.clusterDim.z = 1;
    config.attrs = attribute;
    config.numAttrs = 1;

    float* parameter;
    cudaLaunchKernelEx(&config, Func, parameter);
}
```

10.38. Launch Bounds

As discussed in detail in *Multiprocessor Level*, the fewer registers a kernel uses, the more threads and thread blocks are likely to reside on a multiprocessor, which can improve performance.

Therefore, the compiler uses heuristics to minimize register usage while keeping register spilling (see *Device Memory Accesses*) and instruction count to a minimum. An application can optionally aid these heuristics by providing additional information to the compiler in the form of launch bounds that are specified using the `__launch_bounds__()` qualifier in the definition of a `__global__` function:

```
__global__ void
__launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor, maxBlocksPerCluster)
MyKernel(...)
{
    ...
}
```

- `maxThreadsPerBlock` specifies the maximum number of threads per block with which the application will ever launch `MyKernel()`; it compiles to the `.maxntid`*PTX* directive.
- `minBlocksPerMultiprocessor` is optional and specifies the desired minimum number of resident blocks per multiprocessor; it compiles to the `.minnctapersm`*PTX* directive.
- `maxBlocksPerCluster` is optional and specifies the desired maximum number thread blocks per cluster with which the application will ever launch `MyKernel()`; it compiles to the `.maxclusterrank`*PTX* directive.

If launch bounds are specified, the compiler first derives from them the upper limit *L* on the number of registers the kernel should use to ensure that `minBlocksPerMultiprocessor` blocks (or a single block if `minBlocksPerMultiprocessor` is not specified) of `maxThreadsPerBlock` threads can reside on the multiprocessor (see *Hardware Multithreading* for the relationship between the number of registers used by a kernel and the number of registers allocated per block). The compiler then optimizes register usage in the following way:

- If the initial register usage is higher than *L*, the compiler reduces it further until it becomes less or equal to *L*, usually at the expense of more local memory usage and/or higher number of instructions;
- If the initial register usage is lower than *L*
  - If `maxThreadsPerBlock` is specified and `minBlocksPerMultiprocessor` is not, the compiler uses `maxThreadsPerBlock` to determine the register usage thresholds for the transitions between `n` and `n+1` resident blocks (i.e., when using one less register makes room for an additional resident block as in the example of *Multiprocessor Level*) and then applies similar heuristics as when no launch bounds are specified;
  - If both `minBlocksPerMultiprocessor` and `maxThreadsPerBlock` are specified, the compiler may increase register usage as high as *L* to reduce the number of instructions and better hide single thread instruction latency.

A kernel will fail to launch if it is executed with more threads per block than its launch bound `maxThreadsPerBlock`.

A kernel will fail to launch if it is executed with more thread blocks per cluster than its launch bound `maxBlocksPerCluster`.

Per thread resources required by a CUDA kernel might limit the maximum block size in an unwanted way. In order to maintain forward compatibility to future hardware and toolkits and to

ensure that at least one thread block can run on an SM, developers should include the single argument `__launch_bounds__(maxThreadsPerBlock)` which specifies the largest block size that the kernel will be launched with. Failure to do so could lead to "too many resources requested for launch" errors. Providing the two argument version of `__launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor)` can improve performance in some cases. The right value for `minBlocksPerMultiprocessor` should be determined using a detailed per kernel analysis.

Optimal launch bounds for a given kernel will usually differ across major architecture revisions. The sample code below shows how this is typically handled in device code using the `__CUDA_ARCH__` macro introduced in *Application Compatibility*.

```
#define THREADS_PER_BLOCK          256
#if __CUDA_ARCH__ >= 200
    #define MY_KERNEL_MAX_THREADS  (2 * THREADS_PER_BLOCK)
    #define MY_KERNEL_MIN_BLOCKS   3
#else
    #define MY_KERNEL_MAX_THREADS  THREADS_PER_BLOCK
    #define MY_KERNEL_MIN_BLOCKS   2
#endif

// Device code
__global__ void
__launch_bounds__(MY_KERNEL_MAX_THREADS, MY_KERNEL_MIN_BLOCKS)
MyKernel(...)
{
    ...
}
```

In the common case where `MyKernel` is invoked with the maximum number of threads per block (specified as the first parameter of `__launch_bounds__()`), it is tempting to use `MY_KERNEL_MAX_THREADS` as the number of threads per block in the execution configuration:

```
// Host code
MyKernel<<<blocksPerGrid, MY_KERNEL_MAX_THREADS>>>(...);
```

This will not work however since `__CUDA_ARCH__` is undefined in host code as mentioned in *Application Compatibility*, so `MyKernel` will launch with 256 threads per block even when `__CUDA_ARCH__` is greater or equal to 200. Instead the number of threads per block should be determined:

- Either at compile time using a macro that does not depend on `__CUDA_ARCH__`, for example

  ```
  // Host code
  MyKernel<<<blocksPerGrid, THREADS_PER_BLOCK>>>(...);
  ```

- Or at runtime based on the compute capability

  ```
  // Host code
  cudaGetDeviceProperties(&deviceProp, device);
  int threadsPerBlock =
            (deviceProp.major >= 2 ?
                      2 * THREADS_PER_BLOCK : THREADS_PER_BLOCK);
  MyKernel<<<blocksPerGrid, threadsPerBlock>>>(...);
  ```

Register usage is reported by the `--ptxas-options=-v` compiler option. The number of resident blocks can be derived from the occupancy reported by the CUDA profiler (see *Device Memory Accesses* for a definition of occupancy).

10.39. Maximum Number of Registers per Thread

To provide a mechanism for low-level performance tuning, CUDA C++ provides the `__maxnreg__()` function qualifier to pass performance tuning information to the backend optimizing compiler. The `__maxnreg__()` qualifier specifies the maximum number of registers to be allocated to a single thread in a thread block. In the definition of a `__global__` function:

```
__global__ void
__maxnreg__(maxNumberRegistersPerThread)
MyKernel(...)
{
    ...
}
```

- `maxNumberRegistersPerThread` specifies the maximum number of registers to be allocated to a single thread in a thread block of the kernel `MyKernel()`; it compiles to the `.maxnreg`*PTX* directive.

The `__launch_bounds__()` and `__maxnreg__()` qualifiers cannot be applied to the same kernel.

Register usage can also be controlled for all `__global__` functions in a file using the `maxrregcount` compiler option. The value of `maxrregcount` is ignored for functions with the `__maxnreg__` qualifier.

10.40. #pragma unroll

By default, the compiler unrolls small loops with a known trip count. The `#pragma unroll` directive however can be used to control unrolling of any given loop. It must be placed immediately before the loop and only applies to that loop. It is optionally followed by an integral constant expression (ICE)[6]. If the ICE is absent, the loop will be completely unrolled if its trip count is constant. If the ICE evaluates to 1, the compiler will not unroll the loop. The pragma will be ignored if the ICE evaluates to a non-positive integer or to an integer greater than the maximum value representable by the `int` data type.

Examples:

```
struct S1_t { static const int value = 4; };
template <int X, typename T2>
__device__ void foo(int *p1, int *p2) {

// no argument specified, loop will be completely unrolled
#pragma unroll
for (int i = 0; i < 12; ++i)
  p1[i] += p2[i]*2;

// unroll value = 8
#pragma unroll (X+1)
for (int i = 0; i < 12; ++i)
  p1[i] += p2[i]*4;

// unroll value = 1, loop unrolling disabled
```

(continues on next page)

[6] See the C++ Standard for definition of integral constant expression.

(continued from previous page)

```
#pragma unroll 1
for (int i = 0; i < 12; ++i)
  p1[i] += p2[i]*8;

// unroll value = 4
#pragma unroll (T2::value)
for (int i = 0; i < 12; ++i)
  p1[i] += p2[i]*16;
}

__global__ void bar(int *p1, int *p2) {
foo<7, S1_t>(p1, p2);
}
```

10.41. SIMD Video Instructions

PTX ISA version 3.0 includes SIMD (Single Instruction, Multiple Data) video instructions which operate on pairs of 16-bit values and quads of 8-bit values. These are available on devices of compute capability 3.0.

The SIMD video instructions are:

- vadd2, vadd4
- vsub2, vsub4
- vavrg2, vavrg4
- vabsdiff2, vabsdiff4
- vmin2, vmin4
- vmax2, vmax4
- vset2, vset4

PTX instructions, such as the SIMD video instructions, can be included in CUDA programs by way of the assembler, `asm()`, statement.

The basic syntax of an `asm()` statement is:

```
asm("template-string" : "constraint"(output) : "constraint"(input)"));
```

An example of using the `vabsdiff4` PTX instruction is:

```
asm("vabsdiff4.u32.u32.u32.add" " %0, %1, %2, %3;": "=r" (result):"r" (A), "r" (B), "r
↪" (C));
```

This uses the `vabsdiff4` instruction to compute an integer quad byte SIMD sum of absolute differences. The absolute difference value is computed for each byte of the unsigned integers A and B in SIMD fashion. The optional accumulate operation (`.add`) is specified to sum these differences.

Refer to the document “Using Inline PTX Assembly in CUDA” for details on using the assembly statement in your code. Refer to the PTX ISA documentation (“Parallel Thread Execution ISA Version 3.0” for example) for details on the PTX instructions for the version of PTX that you are using.

10.42. Diagnostic Pragmas

The following pragmas may be used to control the error severity used when a given diagnostic message is issued.

```
#pragma nv_diag_suppress
#pragma nv_diag_warning
#pragma nv_diag_error
#pragma nv_diag_default
#pragma nv_diag_once
```

Uses of these pragmas have the following form:

```
#pragma nv_diag_xxx error_number, error_number ...
```

The diagnostic affected is specified using an error number showed in a warning message. Any diagnostic may be overridden to be an error, but only warnings may have their severity suppressed or be restored to a warning after being promoted to an error. The `nv_diag_default` pragma is used to return the severity of a diagnostic to the one that was in effect before any pragmas were issued (i.e., the normal severity of the message as modified by any command-line options). The following example suppresses the "`declared but never referenced`" warning on the declaration of `foo`:

```
#pragma nv_diag_suppress 177
void foo()
{
  int i=0;
}
#pragma nv_diag_default 177
void bar()
{
  int i=0;
}
```

The following pragmas may be used to save and restore the current diagnostic pragma state:

```
#pragma nv_diagnostic push
#pragma nv_diagnostic pop
```

Examples:

```
#pragma nv_diagnostic push
#pragma nv_diag_suppress 177
void foo()
{
  int i=0;
}
#pragma nv_diagnostic pop
void bar()
{
  int i=0;
}
```

Note that the pragmas only affect the nvcc CUDA frontend compiler; they have no effect on the host compiler.

Removal Notice: The support of diagnostic pragmas without `nv_` prefix are removed from CUDA 12.0, if the pragmas are inside the device code, warning `unrecognized #pragma in device code` will

be emitted, otherwise they will be passed to the host compiler. If they are intended for CUDA code, use the pragmas with `nv_` prefix instead.

10.43. Custom ABI Pragmas

The `#pragma nv_abi` directive enables applications compiled in separate compilation mode to achieve performance similar to that of whole program compilation.

The syntax for using this pragma is as follows, where ICE refers to any integral constant expression (ICE):[Page 276, 6].

```
#pragma nv_abi preserve_n_data(ICE) preserve_n_control(ICE)
```

Note, the arguments that follow `#pragma nv_abi` are optional and can be provided in any order; however, at least one argument is required.

The `preserve_n` arguments set a limit on the number of registers preserved during a function call:

- `preserve_n_data(ICE)` limits the number of data registers, and
- `preserve_n_control(ICE)` limits the number of control registers.

`#pragma nv_abi` can be placed immediately before a device function declaration or definition. Alternatively, it can be placed directly before an indirect function call within a C++ expression statement inside a device function. Note, indirect function calls to free functions are supported, but indirect calls through function argument references or class member functions are not.

When the pragma is applied to a device function declaration or definition, it modifies the custom ABI properties for any calls to that function. When placed at an indirect function call site, the pragma affects the ABI properties for that indirect function call. The key point is that unlike direct function calls, where you can place the pragma before a function declaration or definition, `#pragma nv_abi` only affects indirect function calls when the pragma is placed before a call site.

As shown in the following example, we have two device functions, `foo()` and `bar()`. In this example the pragma is placed before the call site of the function pointer fptr to modify the ABI properties of the indirect function call. Notice that placing the pragma before the direct call does not affect the ABI properties of the call. To alter the ABI properties of a direct function call, the pragma must be placed before the function declaration or definition.

```
__device__ int foo()
{
  int value{0};
  ...
  return value;
}

__device__ int bar()
{
  int value{0};
  ...
  return value;
}

__device__ void baz()
{
  int result{0};
```

(continues on next page)

(continued from previous page)

```
  int (*fptr)() = foo;  // function pointer

  #pragma nv_abi preserve_n_data(16) preserve_n_control(8)
  result = fptr();      // The pragma affects the indirect call to foo() via fptr

  #pragma nv_abi preserve_n_data(16) preserve_n_control(8)
  result = (*fptr)();   // Alternate syntax for the indirect call to foo()

  #pragma nv_abi preserve_n_data(16) preserve_n_control(8)
  result += bar();      // The pragma does NOT affect the direct call to bar()
}
```

As shown in the following example, to modify direct function calls, you must apply the pragma to the function declaration or definition.

```
#pragma nv_abi preserve_n_data(16)
__device__ void foo();
```

Note that a program is ill-formed if the pragma arguments for a function declaration and its corresponding definition do not match.

10.44. CUDA C++ Memory Model

The CUDA C++ Memory Model extends the ISO C++ Memory Model as documented in the CUDA C++ Memory Model documentation.

10.45. CUDA C++ Execution Model

The CUDA C++ Exeuction Model extends the ISO C++ execution model as documented in the CUDA C++ Execution Model documentation.

Chapter 11. Cooperative Groups

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

11.1. Introduction

Cooperative Groups is an extension to the CUDA programming model, introduced in CUDA 9, for organizing groups of communicating threads. Cooperative Groups allows developers to express the granularity at which threads are communicating, helping them to express richer, more efficient parallel decompositions.

Historically, the CUDA programming model has provided a single, simple construct for synchronizing cooperating threads: a barrier across all threads of a thread block, as implemented with the `__syncthreads()` intrinsic function. However, programmers would like to define and synchronize groups of threads at other granularities to enable greater performance, design flexibility, and software reuse in the form of "collective" group-wide function interfaces. In an effort to express broader patterns of parallel interaction, many performance-oriented programmers have resorted to writing their own ad hoc and unsafe primitives for synchronizing threads within a single warp, or across sets of thread blocks running on a single GPU. Whilst the performance improvements achieved have often been valuable, this has resulted in an ever-growing collection of brittle code that is expensive to write, tune, and maintain over time and across GPU generations. Cooperative Groups addresses this by providing a safe and future-proof mechanism to enable performant code.

11.2. What's New in Cooperative Groups

11.2.1. CUDA 13.0

- `multi_grid_group` was removed.

11.2.2. CUDA 12.2

- `barrier_arrive` and `barrier_wait` member functions were added for *grid_group* and *thread_block*. Description of the API is available *here*.

11.2.3. CUDA 12.1

- *invoke_one and invoke_one_broadcast* APIs were added.

11.2.4. CUDA 12.0

- The following experimental APIs are now moved to the main namespace:
  - asynchronous reduce and scan update added in CUDA 11.7
  - `thread_block_tile` larger than 32 added in CUDA 11.1
- It is no longer required to provide memory using the `block_tile_memory` object in order to create these large tiles on Compute Capability 8.0 or higher.

11.3. Programming Model Concept

The Cooperative Groups programming model describes synchronization patterns both within and across CUDA thread blocks. It provides both the means for applications to define their own groups of threads, and the interfaces to synchronize them. It also provides new launch APIs that enforce certain restrictions and therefore can guarantee the synchronization will work. These primitives enable new patterns of cooperative parallelism within CUDA, including producer-consumer parallelism, opportunistic parallelism, and global synchronization across the entire Grid.

The Cooperative Groups programming model consists of the following elements:

- Data types for representing groups of cooperating threads;
- Operations to obtain implicit groups defined by the CUDA launch API (e.g., thread blocks);
- Collectives for partitioning existing groups into new groups;
- Collective Algorithms for data movement and manipulation (e.g. memcpy_async, reduce, scan);
- An operation to synchronize all threads within the group;
- Operations to inspect the group properties;
- Collectives that expose low-level, group-specific and often HW accelerated, operations.

The main concept in Cooperative Groups is that of objects naming the set of threads that are part of it. This expression of groups as first-class program objects improves software composition, since collective functions can receive an explicit object representing the group of participating threads. This object also makes programmer intent explicit, which eliminates unsound architectural assumptions that result in brittle code, undesirable restrictions upon compiler optimizations, and better compatibility with new GPU generations.

To write efficient code, its best to use specialized groups (going generic loses a lot of compile time optimizations), and pass these group objects by reference to functions that intend to use these threads in some cooperative fashion.

Cooperative Groups requires CUDA 9.0 or later. To use Cooperative Groups, include the header file:

```
// Primary header is compatible with pre-C++11, collective algorithm headers require
↪C++11
#include <cooperative_groups.h>
// Optionally include for memcpy_async() collective
#include <cooperative_groups/memcpy_async.h>
// Optionally include for reduce() collective
#include <cooperative_groups/reduce.h>
// Optionally include for inclusive_scan() and exclusive_scan() collectives
#include <cooperative_groups/scan.h>
```

and use the Cooperative Groups namespace:

```
using namespace cooperative_groups;
// Alternatively use an alias to avoid polluting the namespace with collective
↪algorithms
namespace cg = cooperative_groups;
```

The code can be compiled in a normal way using nvcc, however if you wish to use memcpy_async, reduce or scan functionality and your host compiler's default dialect is not C++11 or higher, then you must add `--std=c++11` to the command line.

11.3.1. Composition Example

To illustrate the concept of groups, this example attempts to perform a block-wide sum reduction. Previously, there were hidden constraints on the implementation when writing this code:

```
__device__ int sum(int *x, int n) {
    // ...
    __syncthreads();
    return total;
}

__global__ void parallel_kernel(float *x) {
    // ...
    // Entire thread block must call sum
    sum(x, n);
}
```

All threads in the thread block must arrive at the `__syncthreads()` barrier, however, this constraint is hidden from the developer who might want to use `sum(…)`. With Cooperative Groups, a better way of writing this would be:

```
__device__ int sum(const thread_block& g, int *x, int n) {
    // ...
    g.sync()
    return total;
}

__global__ void parallel_kernel(...) {
```

(continues on next page)

(continued from previous page)

```
    // ...
    // Entire thread block must call sum
    thread_block tb = this_thread_block();
    sum(tb, x, n);
    // ...
}
```

11.4. Group Types

11.4.1. Implicit Groups

Implicit groups represent the launch configuration of the kernel. Regardless of how your kernel is written, it always has a set number of threads, blocks and block dimensions, a single grid and grid dimensions. In addition, if the multi-device cooperative launch API is used, it can have multiple grids (single grid per device). These groups provide a starting point for decomposition into finer grained groups which are typically HW accelerated and are more specialized for the problem the developer is solving.

Although you can create an implicit group anywhere in the code, it is dangerous to do so. Creating a handle for an implicit group is a collective operation—all threads in the group must participate. If the group was created in a conditional branch that not all threads reach, this can lead to deadlocks or data corruption. For this reason, it is recommended that you create a handle for the implicit group upfront (as early as possible, before any branching has occurred) and use that handle throughout the kernel. Group handles must be initialized at declaration time (there is no default constructor) for the same reason and copy-constructing them is discouraged.

11.4.1.1 Thread Block Group

Any CUDA programmer is already familiar with a certain group of threads: the thread block. The Cooperative Groups extension introduces a new datatype, `thread_block`, to explicitly represent this concept within the kernel.

`class thread_block;`

Constructed via:

```
thread_block g = this_thread_block();
```

Public Member Functions:

`static void sync()`: Synchronize the threads named in the group, equivalent to `g.barrier_wait(g.barrier_arrive())`

`thread_block::arrival_token barrier_arrive()`: Arrive on the thread_block barrier, returns a token that needs to be passed into `barrier_wait()`. More details *here*

`void barrier_wait(thread_block::arrival_token&& t)`: Wait on the `thread_block` barrier, takes arrival token returned from `barrier_arrive()` as an rvalue reference. More details *here*

`static unsigned int thread_rank()`: Rank of the calling thread within [0, num_threads)

`static dim3 group_index()`: 3-Dimensional index of the block within the launched grid

`static dim3 thread_index()`: 3-Dimensional index of the thread within the launched block

`static dim3 dim_threads()`: Dimensions of the launched block in units of threads

`static unsigned int num_threads()`: Total number of threads in the group

Legacy member functions (aliases):

`static unsigned int size()`: Total number of threads in the group (alias of `num_threads()`)

`static dim3 group_dim()`: Dimensions of the launched block (alias of `dim_threads()`)

Example:

```
/// Loading an integer from global into shared memory
__global__ void kernel(int *globalInput) {
    __shared__ int x;
    thread_block g = this_thread_block();
    // Choose a leader in the thread block
    if (g.thread_rank() == 0) {
        // load from global into shared for all threads to work with
        x = (*globalInput);
    }
    // After loading data into shared memory, you want to synchronize
    // if all threads in your thread block need to see it
    g.sync(); // equivalent to __syncthreads();
}
```

Note: that all threads in the group must participate in collective operations, or the behavior is undefined.

Related: The `thread_block` datatype is derived from the more generic `thread_group` datatype, which can be used to represent a wider class of groups.

11.4.1.2 Cluster Group

This group object represents all the threads launched in a single cluster. Refer to *Thread Block Clusters*. The APIs are available on all hardware with Compute Capability 9.0+. In such cases, when a non-cluster grid is launched, the APIs assume a 1x1x1 cluster.

`class cluster_group;`

Constructed via:

```
cluster_group g = this_cluster();
```

Public Member Functions:

`static void sync()`: Synchronize the threads named in the group, equivalent to `g.barrier_wait(g.barrier_arrive())`

`static cluster_group::arrival_token barrier_arrive()`: Arrive on the cluster barrier, returns a token that needs to be passed into `barrier_wait()`. More details *here*

`static void barrier_wait(cluster_group::arrival_token&& t)`: Wait on the cluster barrier, takes arrival token returned from `barrier_arrive()` as a rvalue reference. More details *here*

`static unsigned int thread_rank()`: Rank of the calling thread within [0, num_threads)

`static unsigned int block_rank()`: Rank of the calling block within [0, num_blocks)

`static unsigned int num_threads()`: Total number of threads in the group

`static unsigned int num_blocks()`: Total number of blocks in the group

`static dim3 dim_threads()`: Dimensions of the launched cluster in units of threads

`static dim3 dim_blocks()`: Dimensions of the launched cluster in units of blocks

`static dim3 block_index()`: 3-Dimensional index of the calling block within the launched cluster

`static unsigned int query_shared_rank(const void *addr)`: Obtain the block rank to which a shared memory address belongs

`static T* map_shared_rank(T *addr, int rank)`: Obtain the address of a shared memory variable of another block in the cluster

Legacy member functions (aliases):

`static unsigned int size()`: Total number of threads in the group (alias of `num_threads()`)

11.4.1.3 Grid Group

This group object represents all the threads launched in a single grid. APIs other than `sync()` are available at all times, but to be able to synchronize across the grid, you need to use the cooperative launch API.

`class grid_group;`

Constructed via:

```
grid_group g = this_grid();
```

Public Member Functions:

`bool is_valid() const`: Returns whether the grid_group can synchronize

`void sync() const`: Synchronize the threads named in the group, equivalent to `g.barrier_wait(g.barrier_arrive())`

`grid_group::arrival_token barrier_arrive()`: Arrive on the grid barrier, returns a token that needs to be passed into `barrier_wait()`. More details *here*

`void barrier_wait(grid_group::arrival_token&& t)`: Wait on the grid barrier, takes arrival token returned from `barrier_arrive()` as a rvalue reference. More details *here*

`static unsigned long long thread_rank()`: Rank of the calling thread within [0, num_threads)

`static unsigned long long block_rank()`: Rank of the calling block within [0, num_blocks)

`static unsigned long long cluster_rank()`: Rank of the calling cluster within [0, num_clusters)

`static unsigned long long num_threads()`: Total number of threads in the group

`static unsigned long long num_blocks()`: Total number of blocks in the group

`static unsigned long long num_clusters()`: Total number of clusters in the group

`static dim3 dim_blocks()`: Dimensions of the launched grid in units of blocks

`static dim3 dim_clusters()`: Dimensions of the launched grid in units of clusters

`static dim3 block_index()`: 3-Dimensional index of the block within the launched grid

`static dim3 cluster_index()`: 3-Dimensional index of the cluster within the launched grid

Legacy member functions (aliases):

`static unsigned long long size()`: Total number of threads in the group (alias of `num_threads()`)

`static dim3 group_dim()`: Dimensions of the launched grid (alias of `dim_blocks()`)

11.4.2. Explicit Groups

11.4.2.1 Thread Block Tile

A templated version of a tiled group, where a template parameter is used to specify the size of the tile - with this known at compile time there is the potential for more optimal execution.

```
template <unsigned int Size, typename ParentT = void>
class thread_block_tile;
```

Constructed via:

```
template <unsigned int Size, typename ParentT>
_CG_QUALIFIER thread_block_tile<Size, ParentT> tiled_partition(const ParentT& g)
```

`Size` must be a power of 2 and less than or equal to 1024. Notes section describes extra steps needed to create tiles of size larger than 32 on hardware with Compute Capability 7.5 or lower.

`ParentT` is the parent-type from which this group was partitioned. It is automatically inferred, but a value of void will store this information in the group handle rather than in the type.

Public Member Functions:

`void sync() const`: Synchronize the threads named in the group

`unsigned long long num_threads() const`: Total number of threads in the group

`unsigned long long thread_rank() const`: Rank of the calling thread within [0, num_threads)

`unsigned long long meta_group_size() const`: Returns the number of groups created when the parent group was partitioned.

`unsigned long long meta_group_rank() const`: Linear rank of the group within the set of tiles partitioned from a parent group (bounded by meta_group_size)

`T shfl(T var, unsigned int src_rank) const`: Refer to *Warp Shuffle Functions*, **Note: For sizes larger than 32 all threads in the group have to specify the same src_rank, otherwise the behavior is undefined.**

`T shfl_up(T var, int delta) const`: Refer to *Warp Shuffle Functions*, available only for sizes lower or equal to 32.

`T shfl_down(T var, int delta) const`: Refer to *Warp Shuffle Functions*, available only for sizes lower or equal to 32.

`T shfl_xor(T var, int delta) const`: Refer to *Warp Shuffle Functions*, available only for sizes lower or equal to 32.

`int any(int predicate) const`: Refer to Warp Vote Functions

`int all(int predicate) const`: Refer to Warp Vote Functions

`unsigned int ballot(int predicate) const`: Refer to Warp Vote Functions, available only for sizes lower or equal to 32.

`unsigned int match_any(T val) const`: Refer to *Warp Match Functions*, available only for sizes lower or equal to 32.

`unsigned int match_all(T val, int &pred) const`: Refer to *Warp Match Functions*, available only for sizes lower or equal to 32.

Legacy member functions (aliases):

`unsigned long long size() const`: Total number of threads in the group (alias of `num_threads()`)

Notes:

- `thread_block_tile` templated data structure is being used here, the size of the group is passed to the `tiled_partition` call as a template parameter rather than an argument.
- `shfl`, `shfl_up`, `shfl_down`, and `shfl_xor` functions accept objects of any type when compiled with C++11 or later. This means it's possible to shuffle non-integral types as long as they satisfy the below constraints:
  - Qualifies as trivially copyable i.e., `is_trivially_copyable<T>::value == true`
  - `sizeof(T) <= 32` for tile sizes lower or equal 32, `sizeof(T) <= 8` for larger tiles
- On hardware with Compute Capability 7.5 or lower tiles of size larger than 32 need small amount of memory reserved for them. This can be done using `cooperative_groups::block_tile_memory` struct template that has to reside in either shared or global memory.

  ```
  template <unsigned int MaxBlockSize = 1024>
  struct block_tile_memory;
  ```

  `MaxBlockSize` Specifies the maximal number of threads in the current thread block. This parameter can be used to minimize the shared memory usage of `block_tile_memory` in kernels launched only with smaller thread counts.

  This `block_tile_memory` needs be then passed into `cooperative_groups::this_thread_block`, allowing the resulting `thread_block` to be partitioned into tiles of sizes larger than 32. Overload of `this_thread_block` accepting `block_tile_memory` argument is a collective operation and has to be called with all threads in the `thread_block`.

  `block_tile_memory` can be used on hardware with Compute Capability 8.0 or higher in order to be able to write one source targeting multiple different Compute Capabilities. It should consume no memory when instantiated in shared memory in cases where its not required.

Examples:

```
/// The following code will create two sets of tiled groups, of size 32 and 4
↪respectively:
/// The latter has the provenance encoded in the type, while the first stores it in the
↪handle
thread_block block = this_thread_block();
thread_block_tile<32> tile32 = tiled_partition<32>(block);
thread_block_tile<4, thread_block> tile4 = tiled_partition<4>(block);
```

```
/// The following code will create tiles of size 128 on all Compute Capabilities.
/// block_tile_memory can be omitted on Compute Capability 8.0 or higher.
__global__ void kernel(...) {
    // reserve shared memory for thread_block_tile usage,
```

(continues on next page)

(continued from previous page)

```
    //   specify that block size will be at most 256 threads.
    __shared__ block_tile_memory<256> shared;
    thread_block thb = this_thread_block(shared);

    // Create tiles with 128 threads.
    auto tile = tiled_partition<128>(thb);

    // ...
}
```

11.4.2.1.1 Warp-Synchronous Code Pattern

Developers might have had warp-synchronous codes that they previously made implicit assumptions about the warp size and would code around that number. Now this needs to be specified explicitly.

```
__global__ void cooperative_kernel(...) {
    // obtain default "current thread block" group
    thread_block my_block = this_thread_block();

    // subdivide into 32-thread, tiled subgroups
    // Tiled subgroups evenly partition a parent group into
    // adjacent sets of threads - in this case each one warp in size
    auto my_tile = tiled_partition<32>(my_block);

    // This operation will be performed by only the
    // first 32-thread tile of each block
    if (my_tile.meta_group_rank() == 0) {
        // ...
        my_tile.sync();
    }
}
```

11.4.2.1.2 Single Thread Group

Group representing the current thread can be obtained from `this_thread` function:

```
thread_block_tile<1> this_thread();
```

The following `memcpy_async` API uses a `thread_group`, to copy an int element from source to destination:

```
#include <cooperative_groups.h>
#include <cooperative_groups/memcpy_async.h>

cooperative_groups::memcpy_async(cooperative_groups::this_thread(), dest, src,
↪sizeof(int));
```

More detailed examples of using `this_thread` to perform asynchronous copies can be found in the *Single-Stage Asynchronous Data Copies using cuda::pipeline* and *Multi-Stage Asynchronous Data Copies using cuda::pipeline* sections.

11.4.2.2 Coalesced Groups

In CUDA's SIMT architecture, at the hardware level the multiprocessor executes threads in groups of 32 called warps. If there exists a data-dependent conditional branch in the application code such that threads within a warp diverge, then the warp serially executes each branch disabling threads not on that path. The threads that remain active on the path are referred to as coalesced. Cooperative Groups has functionality to discover, and create, a group containing all coalesced threads.

Constructing the group handle via `coalesced_threads()` is opportunistic. It returns the set of active threads at that point in time, and makes no guarantee about which threads are returned (as long as they are active) or that they will stay coalesced throughout execution (they will be brought back together for the execution of a collective but can diverge again afterwards).

`class coalesced_group;`

Constructed via:

```
coalesced_group active = coalesced_threads();
```

Public Member Functions:

`void sync() const`: Synchronize the threads named in the group

`unsigned long long num_threads() const`: Total number of threads in the group

`unsigned long long thread_rank() const`: Rank of the calling thread within [0, num_threads)

`unsigned long long meta_group_size() const`: Returns the number of groups created when the parent group was partitioned. If this group was created by querying the set of active threads, for example `coalesced_threads()` the value of `meta_group_size()` will be 1.

`unsigned long long meta_group_rank() const`: Linear rank of the group within the set of tiles partitioned from a parent group (bounded by meta_group_size). If this group was created by querying the set of active threads, e.g. `coalesced_threads()` the value of `meta_group_rank()` will always be 0.

`T shfl(T var, unsigned int src_rank) const`: Refer to *Warp Shuffle Functions*

`T shfl_up(T var, int delta) const`: Refer to *Warp Shuffle Functions*

`T shfl_down(T var, int delta) const`: Refer to *Warp Shuffle Functions*

`int any(int predicate) const`: Refer to Warp Vote Functions

`int all(int predicate) const`: Refer to Warp Vote Functions

`unsigned int ballot(int predicate) const`: Refer to Warp Vote Functions

`unsigned int match_any(T val) const`: Refer to *Warp Match Functions*

`unsigned int match_all(T val, int &pred) const`: Refer to *Warp Match Functions*

Legacy member functions (aliases):

`unsigned long long size() const`: Total number of threads in the group (alias of `num_threads()`)

Notes:

`shfl`, `shfl_up`, and `shfl_down` functions accept objects of any type when compiled with C++11 or later. This means it's possible to shuffle non-integral types as long as they satisfy the below constraints:

- Qualifies as trivially copyable i.e. `is_trivially_copyable<T>::value == true`

- sizeof(T) <= 32

Example:

```
/// Consider a situation whereby there is a branch in the
/// code in which only the 2nd, 4th and 8th threads in each warp are
/// active. The coalesced_threads() call, placed in that branch, will create (for each
/// warp) a group, active, that has three threads (with
/// ranks 0-2 inclusive).
__global__ void kernel(int *globalInput) {
    // Lets say globalInput says that threads 2, 4, 8 should handle the data
    if (threadIdx.x == *globalInput) {
        coalesced_group active = coalesced_threads();
        // active contains 0-2 inclusive
        active.sync();
    }
}
```

11.4.2.2.1 Discovery Pattern

Commonly developers need to work with the current active set of threads. No assumption is made about the threads that are present, and instead developers work with the threads that happen to be there. This is seen in the following “aggregating atomic increment across threads in a warp” example (written using the correct CUDA 9.0 set of intrinsics):

```
{
    unsigned int writemask = __activemask();
    unsigned int total = __popc(writemask);
    unsigned int prefix = __popc(writemask & __lanemask_lt());
    // Find the lowest-numbered active lane
    int elected_lane = __ffs(writemask) - 1;
    int base_offset = 0;
    if (prefix == 0) {
        base_offset = atomicAdd(p, total);
    }
    base_offset = __shfl_sync(writemask, base_offset, elected_lane);
    int thread_offset = prefix + base_offset;
    return thread_offset;
}
```

This can be re-written with Cooperative Groups as follows:

```
{
    cg::coalesced_group g = cg::coalesced_threads();
    int prev;
    if (g.thread_rank() == 0) {
        prev = atomicAdd(p, g.num_threads());
    }
    prev = g.thread_rank() + g.shfl(prev, 0);
    return prev;
}
```

11.5. Group Partitioning

11.5.1. tiled_partition

```
template <unsigned int Size, typename ParentT>
thread_block_tile<Size, ParentT> tiled_partition(const ParentT& g);
```

```
thread_group tiled_partition(const thread_group& parent, unsigned int tilesz);
```

The `tiled_partition` method is a collective operation that partitions the parent group into a one-dimensional, row-major, tiling of subgroups. A total of ((size(parent)/tilesz) subgroups will be created, therefore the parent group size must be evenly divisible by the `Size`. The allowed parent groups are `thread_block` or `thread_block_tile`.

The implementation may cause the calling thread to wait until all the members of the parent group have invoked the operation before resuming execution. Functionality is limited to native hardware sizes, 1/2/4/8/16/32 and the `cg::size(parent)` must be greater than the `Size` parameter. The templated version of `tiled_partition` supports 64/128/256/512 sizes as well, but some additional steps are required on Compute Capability 7.5 or lower, refer to *Thread Block Tile* for details.

Codegen Requirements: Compute Capability 5.0 minimum, C++11 for sizes larger than 32

Example:

```
/// The following code will create a 32-thread tile
thread_block block = this_thread_block();
thread_block_tile<32> tile32 = tiled_partition<32>(block);
```

We can partition each of these groups into even smaller groups, each of size 4 threads:

```
auto tile4 = tiled_partition<4>(tile32);
// or using a general group
// thread_group tile4 = tiled_partition(tile32, 4);
```

If, for instance, if we were to then include the following line of code:

```
if (tile4.thread_rank()==0) printf("Hello from tile4 rank 0\n");
```

then the statement would be printed by every fourth thread in the block: the threads of rank 0 in each `tile4` group, which correspond to those threads with ranks 0,4,8,12,etc. in the `block` group.

11.5.2. labeled_partition

```
template <typename Label>
coalesced_group labeled_partition(const coalesced_group& g, Label label);
```

```
template <unsigned int Size, typename Label>
coalesced_group labeled_partition(const thread_block_tile<Size>& g, Label label);
```

The `labeled_partition` method is a collective operation that partitions the parent group into one-dimensional subgroups within which the threads are coalesced. The implementation will evaluate a condition label and assign threads that have the same value for label into the same group.

`Label` can be any integral type.

The implementation may cause the calling thread to wait until all the members of the parent group have invoked the operation before resuming execution.

Note: This functionality is still being evaluated and may slightly change in the future.

Codegen Requirements: Compute Capability 7.0 minimum, C++11

11.5.3. binary_partition

```
coalesced_group binary_partition(const coalesced_group& g, bool pred);
```

```
template <unsigned int Size>
coalesced_group binary_partition(const thread_block_tile<Size>& g, bool pred);
```

The `binary_partition()` method is a collective operation that partitions the parent group into one-dimensional subgroups within which the threads are coalesced. The implementation will evaluate a predicate and assign threads that have the same value into the same group. This is a specialized form of `labeled_partition()`, where the label can only be 0 or 1.

The implementation may cause the calling thread to wait until all the members of the parent group have invoked the operation before resuming execution.

Note: This functionality is still being evaluated and may slightly change in the future.

Codegen Requirements: Compute Capability 7.0 minimum, C++11

Example:

```
/// This example divides a 32-sized tile into a group with odd
/// numbers and a group with even numbers
_global__ void oddEven(int *inputArr) {
    auto block = cg::this_thread_block();
    auto tile32 = cg::tiled_partition<32>(block);

    // inputArr contains random integers
    int elem = inputArr[block.thread_rank()];
    // after this, tile32 is split into 2 groups,
    // a subtile where elem&1 is true and one where its false
    auto subtile = cg::binary_partition(tile32, (elem & 1));
}
```

11.6. Group Collectives

Cooperative Groups library provides a set of collective operations that can be performed by a group of threads. These operations require participation of all threads in the specified group in order to complete the operation. All threads in the group need to pass the same values for corresponding arguments to each collective call, unless different values are explicitly allowed in the argument description. Otherwise the behavior of the call is undefined.

11.6.1. Synchronization

11.6.1.1 barrier_arrive and barrier_wait

```
T::arrival_token T::barrier_arrive();
void T::barrier_wait(T::arrival_token&&);
```

`barrier_arrive` and `barrier_wait` member functions provide a synchronization API similar to `cuda::barrier` *(read more)*. Cooperative Groups automatically initializes the group barrier, but arrive and wait operations have an additional restriction resulting from collective nature of those operations: All threads in the group must arrive and wait at the barrier once per phase. When `barrier_arrive` is called with a group, result of calling any collective operation or another barrier arrival with that group is undefined until completion of the barrier phase is observed with `barrier_wait` call. Threads blocked on `barrier_wait` might be released from the synchronization before other threads call `barrier_wait`, but only after all threads in the group called `barrier_arrive`. Group type T can be any of the *implicit groups*.This allows threads to do independent work after they arrive and before they wait for the synchronization to resolve, allowing to hide some of the synchronization latency. `barrier_arrive` returns an `arrival_token` object that must be passed into the corresponding `barrier_wait`. Token is consumed this way and can not be used for another `barrier_wait` call.

Example of barrier_arrive and barrier_wait used to synchronize initalization of shared memory across the cluster:

```
#include <cooperative_groups.h>

using namespace cooperative_groups;

void __device__ init_shared_data(const thread_block& block, int *data);
void __device__ local_processing(const thread_block& block);
void __device__ process_shared_data(const thread_block& block, int *data);

__global__ void cluster_kernel() {
    extern __shared__ int array[];
    auto cluster = this_cluster();
    auto block   = this_thread_block();

    // Use this thread block to initialize some shared state
    init_shared_data(block, &array[0]);

    auto token = cluster.barrier_arrive(); // Let other blocks know this block is
↪running and data was initialized

    // Do some local processing to hide the synchronization latency
    local_processing(block);

    // Map data in shared memory from the next block in the cluster
    int *dsmem = cluster.map_shared_rank(&array[0], (cluster.block_rank() + 1) %
↪cluster.num_blocks());

    // Make sure all other blocks in the cluster are running and initialized shared
↪data before accessing dsmem
    cluster.barrier_wait(std::move(token));

    // Consume data in distributed shared memory
    process_shared_data(block, dsmem);
```

(continues on next page)

(continued from previous page)

```
    cluster.sync();
}
```

11.6.1.2 sync

```
static void T::sync();

template <typename T>
void sync(T& group);
```

sync synchronizes the threads named in the group. Group type T can be any of the existing group types, as all of them support synchronization. Its available as a member function in every group type or as a free function taking a group as parameter. If the group is a `grid_group` the kernel must have been launched using the appropriate cooperative launch APIs. Equivalent to `T.barrier_wait(T.barrier_arrive())`.

11.6.2. Data Transfer

11.6.2.1 memcpy_async

`memcpy_async` is a group-wide collective memcpy that utilizes hardware accelerated support for non-blocking memory transactions from global to shared memory. Given a set of threads named in the group, `memcpy_async` will move specified amount of bytes or elements of the input type through a single pipeline stage. Additionally for achieving best performance when using the `memcpy_async` API, an alignment of 16 bytes for both shared memory and global memory is required. It is important to note that while this is a memcpy in the general case, it is only asynchronous if the source is global memory and the destination is shared memory and both can be addressed with 16, 8, or 4 byte alignments. Asynchronously copied data should only be read following a call to wait or wait_prior which signals that the corresponding stage has completed moving data to shared memory.

Having to wait on all outstanding requests can lose some flexibility (but gain simplicity). In order to efficiently overlap data transfer and execution, its important to be able to kick off an **N+1**`memcpy_async` request while waiting on and operating on request **N**. To do so, use `memcpy_async` and wait on it using the collective stage-based `wait_prior` API. See *wait and wait_prior* for more details.

Usage 1

```
template <typename TyGroup, typename TyElem, typename TyShape>
void memcpy_async(
  const TyGroup &group,
  TyElem *__restrict__ _dst,
  const TyElem *__restrict__ _src,
  const TyShape &shape
);
```

Performs a copy of ``**shape**`` **bytes**.

Usage 2

```
template <typename TyGroup, typename TyElem, typename TyDstLayout, typename
↪TySrcLayout>
void memcpy_async(
  const TyGroup &group,
  TyElem *__restrict__ dst,
  const TyDstLayout &dstLayout,
  const TyElem *__restrict__ src,
  const TySrcLayout &srcLayout
);
```

Performs a copy of **``min(dstLayout, srcLayout)`` elements**. If layouts are of type `cuda::aligned_size_t<N>`, both must specify the same alignment.

Errata The `memcpy_async` API introduced in CUDA 11.1 with both src and dst input layouts, expects the layout to be provided in elements rather than bytes. The element type is inferred from `TyElem` and has the size `sizeof(TyElem)`. If `cuda::aligned_size_t<N>` type is used as the layout, the number of elements specified times `sizeof(TyElem)` must be a multiple of N and it is recommended to use `std::byte` or `char` as the element type.

If specified shape or layout of the copy is of type `cuda::aligned_size_t<N>`, alignment will be guaranteed to be at least `min(16, N)`. In that case both `dst` and `src` pointers need to be aligned to N bytes and the number of bytes copied needs to be a multiple of N.

Codegen Requirements: Compute Capability 5.0 minimum, Compute Capability 8.0 for asynchronicity, C++11

`cooperative_groups/memcpy_async.h` header needs to be included.

Example:

```
/// This example streams elementsPerThreadBlock worth of data from global memory
/// into a limited sized shared memory (elementsInShared) block to operate on.
#include <cooperative_groups.h>
#include <cooperative_groups/memcpy_async.h>

namespace cg = cooperative_groups;

__global__ void kernel(int* global_data) {
    cg::thread_block tb = cg::this_thread_block();
    const size_t elementsPerThreadBlock = 16 * 1024;
    const size_t elementsInShared = 128;
    __shared__ int local_smem[elementsInShared];

    size_t copy_count;
    size_t index = 0;
    while (index < elementsPerThreadBlock) {
        cg::memcpy_async(tb, local_smem, elementsInShared, global_data + index,
↪elementsPerThreadBlock - index);
        copy_count = min(elementsInShared, elementsPerThreadBlock - index);
        cg::wait(tb);
        // Work with local_smem
        index += copy_count;
    }
}
```

11.6.2.2 wait and wait_prior

```
template <typename TyGroup>
void wait(TyGroup & group);

template <unsigned int NumStages, typename TyGroup>
void wait_prior(TyGroup & group);
```

`wait` and `wait_prior` collectives allow to wait for memcpy_async copies to complete. `wait` blocks calling threads until all previous copies are done. `wait_prior` allows that the latest NumStages are still not done and waits for all the previous requests. So with `N` total copies requested, it waits until the first `N-NumStages` are done and the last `NumStages` might still be in progress. Both `wait` and `wait_prior` will synchronize the named group.

Codegen Requirements: Compute Capability 5.0 minimum, Compute Capability 8.0 for asynchronicity, C++11

`cooperative_groups/memcpy_async.h` header needs to be included.

Example:

```
/// This example streams elementsPerThreadBlock worth of data from global memory
/// into a limited sized shared memory (elementsInShared) block to operate on in
/// multiple (two) stages. As stage N is kicked off, we can wait on and operate on
↪stage N-1.
#include <cooperative_groups.h>
#include <cooperative_groups/memcpy_async.h>

namespace cg = cooperative_groups;

__global__ void kernel(int* global_data) {
    cg::thread_block tb = cg::this_thread_block();
    const size_t elementsPerThreadBlock = 16 * 1024 + 64;
    const size_t elementsInShared = 128;
    __align__(16) __shared__ int local_smem[2][elementsInShared];
    int stage = 0;
    // First kick off an extra request
    size_t copy_count = elementsInShared;
    size_t index = copy_count;
    cg::memcpy_async(tb, local_smem[stage], elementsInShared, global_data,
↪elementsPerThreadBlock - index);
    while (index < elementsPerThreadBlock) {
        // Now we kick off the next request...
        cg::memcpy_async(tb, local_smem[stage ^ 1], elementsInShared, global_data +
↪index, elementsPerThreadBlock - index);
        // ... but we wait on the one before it
        cg::wait_prior<1>(tb);

        // Its now available and we can work with local_smem[stage] here
        // (...)
        //

        // Calculate the amount fo data that was actually copied, for the next
↪iteration.
        copy_count = min(elementsInShared, elementsPerThreadBlock - index);
        index += copy_count;

        // A cg::sync(tb) might be needed here depending on whether
```

(continues on next page)

(continued from previous page)

```
        // the work done with local_smem[stage] can release threads to race ahead or not
        // Wrap to the next stage
        stage ^= 1;
    }
    cg::wait(tb);
    // The last local_smem[stage] can be handled here
}
```

11.6.3. Data Manipulation

11.6.3.1 reduce

```
template <typename TyGroup, typename TyArg, typename TyOp>
auto reduce(const TyGroup& group, TyArg&& val, TyOp&& op) -> decltype(op(val, val));
```

`reduce` performs a reduction operation on the data provided by each thread named in the group passed in. This takes advantage of hardware acceleration (on compute 80 and higher devices) for the arithmetic add, min, or max operations and the logical AND, OR, or XOR, as well as providing a software fallback on older generation hardware. Only 4B types are accelerated by hardware.

`group`: Valid group types are `coalesced_group` and `thread_block_tile`.

`val`: Any type that satisfies the below requirements:

- Qualifies as trivially copyable i.e. `is_trivially_copyable<TyArg>::value == true`
- `sizeof(T) <= 32` for `coalesced_group` and tiles of size lower or equal 32, `sizeof(T) <= 8` for larger tiles
- Has suitable arithmetic or comparative operators for the given function object.

Note: Different threads in the group can pass different values for this argument.

op: Valid function objects that will provide hardware acceleration with integral types are `plus()`, `less()`, `greater()`, `bit_and()`, `bit_xor()`, `bit_or()`. These must be constructed, hence the TyVal template argument is required, i.e. `plus<int>()`. Reduce also supports lambdas and other function objects that can be invoked using `operator()`

Asynchronous reduce

```
template <typename TyGroup, typename TyArg, typename TyAtomic, typename TyOp>
void reduce_update_async(const TyGroup& group, TyAtomic& atomic, TyArg&& val, TyOp&&
↪op);

template <typename TyGroup, typename TyArg, typename TyAtomic, typename TyOp>
void reduce_store_async(const TyGroup& group, TyAtomic& atomic, TyArg&& val, TyOp&&
↪op);

template <typename TyGroup, typename TyArg, typename TyOp>
void reduce_store_async(const TyGroup& group, TyArg* ptr, TyArg&& val, TyOp&& op);
```

`*_async` variants of the API are asynchronously calculating the result to either store to or update a specified destination by one of the participating threads, instead of returning it by each thread. To observe the effect of these asynchronous calls, calling group of threads or a larger group containing them need to be synchronized.

- In case of the atomic store or update variant, `atomic` argument can be either of `cuda::atomic` or `cuda::atomic_ref` available in CUDA C++ Standard Library. This variant of the API is available only on platforms and devices, where these types are supported by the CUDA C++ Standard Library. Result of the reduction is used to atomically update the atomic according to the specified `op`, eg. the result is atomically added to the atomic in case of `cg::plus()`. Type held by the `atomic` must match the type of `TyArg`. Scope of the atomic must include all the threads in the group and if multiple groups are using the same atomic concurrently, scope must include all threads in all groups using it. Atomic update is performed with relaxed memory ordering.
- In case of the pointer store variant, result of the reduction will be weakly stored into the `dst` pointer.

Codegen Requirements: Compute Capability 5.0 minimum, Compute Capability 8.0 for HW acceleration, C++11.

`cooperative_groups/reduce.h` header needs to be included.

Example of approximate standard deviation for integer vector:

```
#include <cooperative_groups.h>
#include <cooperative_groups/reduce.h>
namespace cg = cooperative_groups;

/// Calculate approximate standard deviation of integers in vec
__device__ int std_dev(const cg::thread_block_tile<32>& tile, int *vec, int length) {
    int thread_sum = 0;

    // calculate average first
    for (int i = tile.thread_rank(); i < length; i += tile.num_threads()) {
        thread_sum += vec[i];
    }
    // cg::plus<int> allows cg::reduce() to know it can use hardware acceleration for
↪addition
    int avg = cg::reduce(tile, thread_sum, cg::plus<int>()) / length;

    int thread_diffs_sum = 0;
    for (int i = tile.thread_rank(); i < length; i += tile.num_threads()) {
        int diff = vec[i] - avg;
        thread_diffs_sum += diff * diff;
    }

    // temporarily use floats to calculate the square root
    float diff_sum = static_cast<float>(cg::reduce(tile, thread_diffs_sum, cg::plus
↪<int>())) / length;

    return static_cast<int>(sqrtf(diff_sum));
}
```

Example of block wide reduction:

```
#include <cooperative_groups.h>
#include <cooperative_groups/reduce.h>
namespace cg=cooperative_groups;

/// The following example accepts input in *A and outputs a result into *sum
/// It spreads the data equally within the block
__device__ void block_reduce(const int* A, int count, cuda::atomic<int, cuda::thread_
↪scope_block>& total_sum) {
    auto block = cg::this_thread_block();
```

(continues on next page)

(continued from previous page)

```
    auto tile = cg::tiled_partition<32>(block);
    int thread_sum = 0;

    // Stride loop over all values, each thread accumulates its part of the array.
    for (int i = block.thread_rank(); i < count; i += block.size()) {
        thread_sum += A[i];
    }

    // reduce thread sums across the tile, add the result to the atomic
    // cg::plus<int> allows cg::reduce() to know it can use hardware acceleration for
↪addition
 cg::reduce_update_async(tile, total_sum, thread_sum, cg::plus<int>());

 // synchronize the block, to ensure all async reductions are ready
    block.sync();
}
```

11.6.3.2 Reduce Operators

Below are the prototypes of function objects for some of the basic operations that can be done with `reduce`

```
namespace cooperative_groups {
  template <typename Ty>
  struct cg::plus;

  template <typename Ty>
  struct cg::less;

  template <typename Ty>
  struct cg::greater;

  template <typename Ty>
  struct cg::bit_and;

  template <typename Ty>
  struct cg::bit_xor;

  template <typename Ty>
  struct cg::bit_or;
}
```

Reduce is limited to the information available to the implementation at compile time. Thus in order to make use of intrinsics introduced in CC 8.0, the `cg::` namespace exposes several functional objects that mirror the hardware. These objects appear similar to those presented in the C++ STL, with the exception of `less/greater`. The reason for any difference from the STL is that these function objects are designed to actually mirror the operation of the hardware intrinsics.

Functional description:

- `cg::plus:` Accepts two values and returns the sum of both using operator+.
- `cg::less:` Accepts two values and returns the lesser using operator<. This differs in that the **lower value is returned** rather than a Boolean.

- `cg::greater`: Accepts two values and returns the greater using operator<. This differs in that the **greater value is returned** rather than a Boolean.
- `cg::bit_and`: Accepts two values and returns the result of operator&.
- `cg::bit_xor`: Accepts two values and returns the result of operator^.
- `cg::bit_or`: Accepts two values and returns the result of operator|.

Example:

```
{
    // cg::plus<int> is specialized within cg::reduce and calls __reduce_add_sync(...)
↪on CC 8.0+
    cg::reduce(tile, (int)val, cg::plus<int>());

    // cg::plus<float> fails to match with an accelerator and instead performs a
↪standard shuffle based reduction
    cg::reduce(tile, (float)val, cg::plus<float>());

    // While individual components of a vector are supported, reduce will not use
↪hardware intrinsics for the following
    // It will also be necessary to define a corresponding operator for vector and any
↪custom types that may be used
    int4 vec = {...};
    cg::reduce(tile, vec, cg::plus<int4>())

    // Finally lambdas and other function objects cannot be inspected for dispatch
    // and will instead perform shuffle based reductions using the provided function
↪object.
    cg::reduce(tile, (int)val, [](int l, int r) -> int {return l + r;});
}
```

11.6.3.3 inclusive_scan and exclusive_scan

```
template <typename TyGroup, typename TyVal, typename TyFn>
auto inclusive_scan(const TyGroup& group, TyVal&& val, TyFn&& op) -> decltype(op(val,
↪val));

template <typename TyGroup, typename TyVal>
TyVal inclusive_scan(const TyGroup& group, TyVal&& val);

template <typename TyGroup, typename TyVal, typename TyFn>
auto exclusive_scan(const TyGroup& group, TyVal&& val, TyFn&& op) -> decltype(op(val,
↪val));

template <typename TyGroup, typename TyVal>
TyVal exclusive_scan(const TyGroup& group, TyVal&& val);
```

`inclusive_scan` and `exclusive_scan` performs a scan operation on the data provided by each thread named in the group passed in. Result for each thread is a reduction of data from threads with lower `thread_rank` than that thread in case of `exclusive_scan`. `inclusive_scan` result also includes the calling thread data in the reduction.

`group`: Valid group types are `coalesced_group` and `thread_block_tile`.

`val`: Any type that satisfies the below requirements:

- Qualifies as trivially copyable i.e. `is_trivially_copyable<TyArg>::value == true`

- `sizeof(T) <= 32` for `coalesced_group` and tiles of size lower or equal 32, `sizeof(T) <= 8` for larger tiles
- Has suitable arithmetic or comparative operators for the given function object.

Note: Different threads in the group can pass different values for this argument.

op: Function objects defined for convenience are `plus()`, `less()`, `greater()`, `bit_and()`, `bit_xor()`, `bit_or()` described in *Reduce Operators*. These must be constructed, hence the TyVal template argument is required, i.e. `plus<int>()`. `inclusive_scan` and `exclusive_scan` also supports lambdas and other function objects that can be invoked using `operator()`. Overloads without this argument use `cg::plus<TyVal>()`.

Scan update

```
template <typename TyGroup, typename TyAtomic, typename TyVal, typename TyFn>
auto inclusive_scan_update(const TyGroup& group, TyAtomic& atomic, TyVal&& val, TyFn&&
↪ op) -> decltype(op(val, val));

template <typename TyGroup, typename TyAtomic, typename TyVal>
TyVal inclusive_scan_update(const TyGroup& group, TyAtomic& atomic, TyVal&& val);

template <typename TyGroup, typename TyAtomic, typename TyVal, typename TyFn>
auto exclusive_scan_update(const TyGroup& group, TyAtomic& atomic, TyVal&& val, TyFn&&
↪ op) -> decltype(op(val, val));

template <typename TyGroup, typename TyAtomic, typename TyVal>
TyVal exclusive_scan_update(const TyGroup& group, TyAtomic& atomic, TyVal&& val);
```

`*_scan_update` collectives take an additional argument `atomic` that can be either of `cuda::atomic` or `cuda::atomic_ref` available in CUDA C++ Standard Library. These variants of the API are available only on platforms and devices, where these types are supported by the CUDA C++ Standard Library. These variants will perform an update to the `atomic` according to `op` with value of the sum of input values of all threads in the group. Previous value of the `atomic` will be combined with the result of scan by each thread and returned. Type held by the `atomic` must match the type of `TyVal`. Scope of the atomic must include all the threads in the group and if multiple groups are using the same atomic concurrently, scope must include all threads in all groups using it. Atomic update is performed with relaxed memory ordering.

Following pseudocode illustrates how the update variant of scan works:

```
/*
 inclusive_scan_update behaves as the following block,
 except both reduce and inclusive_scan is calculated simultaneously.
auto total = reduce(group, val, op);
TyVal old;
if (group.thread_rank() == selected_thread) {
    atomically {
        old = atomic.load();
        atomic.store(op(old, total));
    }
}
old = group.shfl(old, selected_thread);
return op(inclusive_scan(group, val, op), old);
*/
```

Codegen Requirements: Compute Capability 5.0 minimum, C++11.

`cooperative_groups/scan.h` header needs to be included.

Example:

```
#include <stdio.h>
#include <cooperative_groups.h>
#include <cooperative_groups/scan.h>
namespace cg = cooperative_groups;

__global__ void kernel() {
    auto thread_block = cg::this_thread_block();
    auto tile = cg::tiled_partition<8>(thread_block);
    unsigned int val = cg::inclusive_scan(tile, tile.thread_rank());
    printf("%u: %u\n", tile.thread_rank(), val);
}

/*  prints for each group:
    0: 0
    1: 1
    2: 3
    3: 6
    4: 10
    5: 15
    6: 21
    7: 28
*/
```

Example of stream compaction using exclusive_scan:

```
#include <cooperative_groups.h>
#include <cooperative_groups/scan.h>
namespace cg = cooperative_groups;

// put data from input into output only if it passes test_fn predicate
template<typename Group, typename Data, typename TyFn>
__device__ int stream_compaction(Group &g, Data *input, int count, TyFn&& test_fn,
↪Data *output) {
    int per_thread = count / g.num_threads();
    int thread_start = min(g.thread_rank() * per_thread, count);
    int my_count = min(per_thread, count - thread_start);

    // get all passing items from my part of the input
    //  into a contagious part of the array and count them.
    int i = thread_start;
    while (i < my_count + thread_start) {
        if (test_fn(input[i])) {
            i++;
        }
        else {
            my_count--;
            input[i] = input[my_count + thread_start];
        }
    }

    // scan over counts from each thread to calculate my starting
    //  index in the output
    int my_idx = cg::exclusive_scan(g, my_count);

    for (i = 0; i < my_count; ++i) {
        output[my_idx + i] = input[thread_start + i];
```

(continues on next page)

(continued from previous page)

```
    }
    // return the total number of items in the output
    return g.shfl(my_idx + my_count, g.num_threads() - 1);
}
```

Example of dynamic buffer space allocation using exclusive_scan_update:

```
#include <cooperative_groups.h>
#include <cooperative_groups/scan.h>
namespace cg = cooperative_groups;

// Buffer partitioning is static to make the example easier to follow,
// but any arbitrary dynamic allocation scheme can be implemented by replacing this
↪function.
__device__ int calculate_buffer_space_needed(cg::thread_block_tile<32>& tile) {
    return tile.thread_rank() % 2 + 1;
}

__device__ int my_thread_data(int i) {
    return i;
}

__global__ void kernel() {
    __shared__ extern int buffer[];
    __shared__ cuda::atomic<int, cuda::thread_scope_block> buffer_used;

    auto block = cg::this_thread_block();
    auto tile = cg::tiled_partition<32>(block);
    buffer_used = 0;
    block.sync();

    // each thread calculates buffer size it needs
    int buf_needed = calculate_buffer_space_needed(tile);

    // scan over the needs of each thread, result for each thread is an offset
    // of that thread's part of the buffer. buffer_used is atomically updated with
    // the sum of all thread's inputs, to correctly offset other tile's allocations
    int buf_offset =
        cg::exclusive_scan_update(tile, buffer_used, buf_needed);

    // each thread fills its own part of the buffer with thread specific data
    for (int i = 0 ; i < buf_needed ; ++i) {
        buffer[buf_offset + i] = my_thread_data(i);
    }

    block.sync();
    // buffer_used now holds total amount of memory allocated
    // buffer is {0, 0, 1, 0, 0, 1 ...};
}
```

11.6.4. Execution control

11.6.4.1 invoke_one and invoke_one_broadcast

```
template<typename Group, typename Fn, typename... Args>
void invoke_one(const Group& group, Fn&& fn, Args&&... args);

template<typename Group, typename Fn, typename... Args>
auto invoke_one_broadcast(const Group& group, Fn&& fn, Args&&... args) ->
↪decltype(fn(args...));
```

`invoke_one` selects a single arbitrary thread from the calling `group` and uses that thread to call the supplied invocable `fn` with the supplied arguments `args`. In case of `invoke_one_broadcast` the result of the call is also distributed to all threads in the group and returned from this collective.

Calling group can be synchronized with the selected thread before and/or after it calls the supplied invocable. It means that communication within the calling group is not allowed inside the supplied invocable body, otherwise forward progress is not guaranteed. Communication with threads outside of the calling group is allowed in the body of the supplied invocable. Thread selection mechanism is **not** guaranteed to be deterministic.

On devices with Compute Capability 9.0 or higher hardware acceleration might be used to select the thread when called with *explicit group types*.

`group`: All group types are valid for `invoke_one`, `coalesced_group` and `thread_block_tile` are valid for `invoke_one_broadcast`.

`fn`: Function or object that can be invoked using `operator()`.

`args`: Parameter pack of types matching types of parameters of the supplied invocable `fn`.

In case of `invoke_one_broadcast` the return type of the supplied invocable `fn` must satisfy the below requirements:

- Qualifies as trivially copyable i.e. `is_trivially_copyable<T>::value == true`
- `sizeof(T) <= 32` for `coalesced_group` and tiles of size lower or equal 32, `sizeof(T) <= 8` for larger tiles

Codegen Requirements: Compute Capability 5.0 minimum, Compute Capability 9.0 for hardware acceleration, C++11.

Aggregated atomic example from *Discovery pattern section* re-written to use invoke_one_broadcast:

```
#include <cooperative_groups.h>
#include <cuda/atomic>
namespace cg = cooperative_groups;

template<cuda::thread_scope Scope>
__device__ unsigned int atomicAddOneRelaxed(cuda::atomic<unsigned int, Scope>&
↪atomic) {
    auto g = cg::coalesced_threads();
    auto prev = cg::invoke_one_broadcast(g, [&] () {
        return atomic.fetch_add(g.num_threads(), cuda::memory_order_relaxed);
    });
    return prev + g.thread_rank();
}
```

11.7. Grid Synchronization

Prior to the introduction of Cooperative Groups, the CUDA programming model only allowed synchronization between thread blocks at a kernel completion boundary. The kernel boundary carries with it an implicit invalidation of state, and with it, potential performance implications.

For example, in certain use cases, applications have a large number of small kernels, with each kernel representing a stage in a processing pipeline. The presence of these kernels is required by the current CUDA programming model to ensure that the thread blocks operating on one pipeline stage have produced data before the thread block operating on the next pipeline stage is ready to consume it. In such cases, the ability to provide global inter thread block synchronization would allow the application to be restructured to have persistent thread blocks, which are able to synchronize on the device when a given stage is complete.

To synchronize across the grid, from within a kernel, you would simply use the `grid.sync()` function:

```
grid_group grid = this_grid();
grid.sync();
```

And when launching the kernel it is necessary to use, instead of the `<<<...>>>` execution configuration syntax, the `cudaLaunchCooperativeKernel` CUDA runtime launch API or the `CUDA driver equivalent`.

Example:

To guarantee co-residency of the thread blocks on the GPU, the number of blocks launched needs to be carefully considered. For example, as many blocks as there are SMs can be launched as follows:

```
int dev = 0;
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, dev);
// initialize, then launch
cudaLaunchCooperativeKernel((void*)my_kernel, deviceProp.multiProcessorCount,
↪numThreads, args);
```

Alternatively, you can maximize the exposed parallelism by calculating how many blocks can fit simultaneously per-SM using the occupancy calculator as follows:

```
/// This will launch a grid that can maximally fill the GPU, on the default stream with
↪kernel arguments
int numBlocksPerSm = 0;
 // Number of threads my_kernel will be launched with
int numThreads = 128;
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, dev);
cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocksPerSm, my_kernel, numThreads,
↪0);
// launch
void *kernelArgs[] = { /* add kernel args */ };
dim3 dimBlock(numThreads, 1, 1);
dim3 dimGrid(deviceProp.multiProcessorCount*numBlocksPerSm, 1, 1);
cudaLaunchCooperativeKernel((void*)my_kernel, dimGrid, dimBlock, kernelArgs);
```

It is good practice to first ensure the device supports cooperative launches by querying the device attribute `cudaDevAttrCooperativeLaunch`:

```
int dev = 0;
int supportsCoopLaunch = 0;
cudaDeviceGetAttribute(&supportsCoopLaunch, cudaDevAttrCooperativeLaunch, dev);
```

which will set `supportsCoopLaunch` to 1 if the property is supported on device 0. Only devices with compute capability of 6.0 and higher are supported. In addition, you need to be running on either of these:

- The Linux platform without MPS
- The Linux platform with MPS and on a device with compute capability 7.0 or higher
- The latest Windows platform

Chapter 12. Cluster Launch Control

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

12.1. Introduction

Compute Capability 10.0 introduces Cluster Launch Control, a new feature that provides developer with more control over thread block scheduling by cancelling thread blocks or thread block clusters.

When dealing with problems of variable size, there are two main approaches to determining the number of kernel thread blocks.

Approach 1: Fixed Work per Thread Block:

In this approach, the number of thread blocks is determined by the problem size, while the amount of work done by each thread block remains constant or is limited.

Key advantages of this approach:

- Load balancing between SMs.

  In particular, when thread block run-times exhibit variability and/or when the number of thread blocks is much larger than what the GPU can execute simultaneously (resulting in a low-tail effect), this approach allows the GPU scheduler to run more thread blocks on some SMs than others.

- Preemption.

  The GPU scheduler can start executing a *higher-priority kernel*, even if it is launched after the execution of a lower-priority kernel has already begun, by scheduling the higher-priority kernel's thread blocks as the currently running lower-priority kernel's thread blocks finish. It can then return to the lower-priority kernel once the higher-priority kernel has finished.

Approach 2: Fixed Number of Thread Blocks:

In this approach, often implemented as a block-stride or grid-stride loop, the number of thread blocks does not directly depend on the problem size. Instead, the amount of work done by each thread block is a function of the problem size. Typically, the number of thread blocks is based on the number of SMs on the GPU where the kernel is executed and the desired occupancy.

Key advantage of this approach:

- Reduced thread block overheads.

This approach not only reduces amortized thread block launch latency but also minimizes the computational overhead associated with shared operations across all thread blocks. These overheads can be significantly higher than launch latency overheads.

For example, in convolution kernels, a prologue for calculating convolution coefficients – independent of the thread block index – can be computed fewer times due to the fixed number of thread blocks, thus reducing redundant computations.

Cluster Launch Control Approach:

Cluster Launch Control allows a kernel to request (**cancel**) the thread block index of a block that has not yet started execution.

This mechanism enables work-stealing among thread blocks: a thread block attempts to cancel the launch of another thread block that has not started running yet. If cancellation succeeds, it "steals" the other thread block's work by using cancelled block index to perform the task.

The cancellation will fail if there are no more thread block indices available and may fail for other reasons, such as a higher-priority kernel being scheduled. In the latter case, if a thread block exits after a cancellation failure, the scheduler can start executing the higher-priority kernel, after which it will continue scheduling the remaining thread blocks of the current kernel for execution.

The table below summarizes advantages and disadvantages of the three approaches:

	Fixed Work per Thread Block	**Fixed Number of Thread Blocks**	**Cluster Launch Control**
Reduced overheads	x	v	v
Preemption	v	x	v
Load balancing	v	x	v

12.2. Cluster Launch Control API Details

Cancelling a thread block via the Cluster Launch Control API is done asynchronously and synchronized using a memory barrier, following a programming pattern similar to *asynchronous data copies*.

The API, currently available through libcu++, provides a request instruction that writes the encoded cancellation result into a `__shared__` variable, along with instructions to decode the result into a *Success*/*Fail* flag and the index of the cancelled thread block in case of *Success*.

12.2.1. Thread block cancellation steps

The preferred way to use Cluster Launch Control is from a single thread, i.e., one request at a time.

The following are the five steps of the thread block cancellation process. The first two steps are declarations and initialization of cancellation result and synchronization variables, which are done before the work-stealing. The last three steps are typically executed inside a work-stealing loop over thread block indices.

1. Declare variables for thread block cancellation:

```
__shared__ uint4 result; // Request result.
__shared__ uint64_t bar; // Synchronization barrier.
int phase = 0;           // Synchronization barrier phase.
```

2. Initialize shared memory barrier with a single arrival count:

```
if (cg::thread_block::thread_rank() == 0)
    ptx::mbarrier_init(&bar, 1);
__syncthreads();
```

3. Submit asynchronous cancellation request by a single thread and set transaction count:

```
if (cg::thread_block::thread_rank() == 0) {
    cg::invoke_one(cg::coalesced_threads(), ptx::clusterlaunchcontrol_try_cancel,
↪&result, &bar);
    ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_cta, ptx::space_
↪shared, &bar, sizeof(uint4));
}
```

Note: Since thread block cancellation is a uniform instruction, it is recommended to submit it inside *invoke_one* thread selector. This allows the compliler to optimize out the peeling loop.

---

4. Synchronize (complete) asynchronous cancellation request:

```
while (!ptx::mbarrier_try_wait_parity(&bar, phase))
{}
phase ^= 1;
```

5. Retrieve cancellation status and cancelled thread block index:

```
bool success = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result);
if (success) {
    // Don't need all three for 1D/2D thread blocks:
    int bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x(result);
    int by = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_y(result);
    int bz = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_z(result);
}
```

6. Ensure visibility of shared memory operations between async and generic proxies, and protect against data races between iterations of the work-stealing loop.

12.2.2. Thread block cancellation constraints

The constraints are related to failed cancellation requests:

- Submitting another cancellation request after **observing** a previously failed request is Undefined Behavior.

  In the two code examples below, assuming the first cancellation request fails, only the first example exhibits undefined behavior. The second example is correct because there is no observation between the cancellation requests:

  Invalid code:

```
// First request:
ptx::clusterlaunchcontrol_try_cancel(&result0, &bar0);

// First request query:
[Synchronize bar0 code here.]
bool success0 = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result0);
assert(!success0); // Observed failure; second cacellation will be invalid.

// Second request - next line is Undefined Behavior:
ptx::clusterlaunchcontrol_try_cancel(&result1, &bar1);
```

  Valid code:

```
// First request:
ptx::clusterlaunchcontrol_try_cancel(&result0, &bar0);

// Second request:
ptx::clusterlaunchcontrol_try_cancel(&result1, &bar1);

// First request query:
[Synchronize bar0 code here.]
bool success0 = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result0);
assert(!success0); // Observed failure; second cacellation was valid.
```

- Retrieving the thread block index of a failed cancellation request is Undefined Behavior.
- Submitting a cancellation request from multiple threads is not recommended. It results in the cancellation of multiple thread blocks and requires careful handling, such as:
  - Each submitting thread must proivde a unique `__shared__` result pointer to avoid data races.
  - If the same barrier is used for synchronization, the arrival and transaction counts must be adjusted accordingly.

12.2.3. Kernel Example: Vector-Scalar Multiplication

The three kernels below demonstrate the Fixed Work per Thread Block, Fixed Number of Thread Blocks, and Cluster Launch Control approaches for vector-scalar multiplication $\overline{v} := \alpha\overline{v}$.

- Fixed Work per Thread Block:

```
__global__
void kernel_fixed_work (float* data, int n)
{
    // Prologue:
    float alpha = compute_scalar();

    // Computation:
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        data[i] *= alpha;
}

// Launch: kernel_fixed_work<<<1024, (n + 1023) / 1024>>>(data, n);
```

- Fixed Number of Thread Blocks:

```
__global__
void kernel_fixed_blocks (float* data, int n)
{
    // Prologue:
    float alpha = compute_scalar();

    // Computation:
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    while (i < n) {
        data[i] *= alpha;
        i += gridDim.x * blockDim.x;
    }
}

// Launch: kernel_fixed_blocks<<<1024, SM_COUNT>>>(data, n);
```

- Cluster Launch Control:

```
#include <cooperative_groups.h>
#include <cuda/ptx>

namespace cg = cooperative_groups;
namespace ptx = cuda::ptx;

__global__
void kernel_cluster_launch_control (float* data, int n)
{
    // Cluster launch control initialization:
    __shared__ uint4 result;
    __shared__ uint64_t bar;
    int phase = 0;

    if (cg::thread_block::thread_rank() == 0)
        ptx::mbarrier_init(&bar, 1);
```

(continues on next page)

(continued from previous page)

```
    // Prologue:
    float alpha = compute_scalar(); // Device function not shown in this code
↪snippet.

    // Work-stealing loop:
    int bx = blockIdx.x; // Assuming 1D x-axis thread blocks.

    while (true) {
        // Protect result from overwrite in the next iteration,
        // (also ensure barrier initialization at 1st iteration):
        __syncthreads();

        // Cancellation request:
        if (cg::thread_block::thread_rank() == 0) {
            // Acquire write of result in the async proxy:
            ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_acquire,
↪ptx::space_cluster, ptx::scope_cluster);

            cg::invoke_one(cg::coalesced_threads(), [&]()
↪{ptx::clusterlaunchcontrol_try_cancel(&result, &bar);});
            ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_cta,
↪ptx::space_shared, &bar, sizeof(uint4));
        }

        // Computation:
        int i = bx * blockDim.x + threadIdx.x;
        if (i < n)
            data[i] *= alpha;

        // Cancellation request synchronization:
        while (!ptx::mbarrier_try_wait_parity(ptx::sem_acquire, ptx::scope_cta, &
↪bar, phase))
        {}
        phase ^= 1;

        // Cancellation request decoding:
        bool success = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result);
        if (!success)
            break;

        bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x<int>
↪(result);

        // Release read of result to the async proxy:
        ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_release, ptx::space_
↪shared, ptx::scope_cluster);
    }
}

// Launch: kernel_cluster_launch_control<<<1024, (n + 1023) / 1024>>>(data, n);
```

12.2.4. Cluster Launch Control for Thread Block Clusters

In the case of a *thread block clusters*, the thread block cancellation steps are the same as in a non-cluster setting, with minor adjustments. As in the non-cluster case, submitting a cancellation request from multiple threads **within a cluster** is not recommended, as this will attempt to cancel multiple clusters.

- The cancellation is submitted by a single cluster thread.
- The shared memory result of each cluster's thread block will receive the same (encoded) value of the cancelled thread block index (i.e., the result value is multicasted). The result received by all thread blocks corresponds to the local block index `{0, 0, 0}` within a cluster. Therefore, thread blocks within the cluster need to add the local block index.
- Synchronization is performed by each cluster's thread block using a local `__shared__` memory barrier. Barrier operations must be performed with the `ptx::scope_cluster` scope.
- Cancelling in the cluster case requires all the thread blocks to exist. A user can guarantee that all thread blocks are running by using `cg::cluster_group::sync()` from *Cluster Group* API.

The kernel below demonstrates Cluster Launch Control approach in a thread block cluster case:

```
#include <cooperative_groups.h>
#include <cuda/ptx>

namespace cg = cooperative_groups;
namespace ptx = cuda::ptx;

__global__ __cluster_dims__(2, 1, 1)
void kernel_cluster_launch_control (float* data, int n)
{
    // Cluster launch control initialization:
    __shared__ uint4 result;
    __shared__ uint64_t bar;
    int phase = 0;

    if (cg::thread_block::thread_rank() == 0) {
        ptx::mbarrier_init(&bar, 1);
        ptx::fence_mbarrier_init(ptx::sem_release, ptx::scope_cluster); // CGA-level
↪fence.
    }

    // Prologue:
    float alpha = compute_scalar(); // Device function not shown in this code snippet.

    // Work-stealing loop:
    int bx = blockIdx.x; // Assuming 1D x-axis thread blocks.

    while (true) {
        // Protect result from overwrite in the next iteration,
        // (also ensure all thread blocks have started at 1st iteration):
        cg::cluster_group::sync();

        // Cancellation request by a single cluster thread:
        if (cg::cluster_group::thread_rank() == 0) {
            // Acquire write of result in the async proxy:
            ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_acquire, ptx::space_
↪cluster, ptx::scope_cluster);
```

(continues on next page)

(continued from previous page)

```
            cg::invoke_one(cg::coalesced_threads(), [&](){ptx::clusterlaunchcontrol_
↪try_cancel_multicast(&result, &bar);});
        }

        // Cancellation completion tracked by each thread block:
        if (cg::thread_block::thread_rank() == 0)
            ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_cluster,
↪ptx::space_shared, &bar, sizeof(uint4));

        // Computation:
        int i = bx * blockDim.x + threadIdx.x;
        if (i < n)
            data[i] *= alpha;

        // Cancellation request synchronization:
        while (!ptx::mbarrier_try_wait_parity(ptx::sem_acquire, ptx::scope_cluster, &
↪bar, phase))
        {}
        phase ^= 1;

        // Cancellation request decoding:
        bool success = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result);
        if (!success)
            break;

        bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x<int>(result);
        bx += cg::cluster_group::block_index().x; // Add local offset.

        // Release read of result to the async proxy:
        ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_release, ptx::space_
↪shared, ptx::scope_cluster);
    }
}

// Launch: kernel_cluster_launch_control<<<1024, (n + 1023) / 1024>>>(data, n);
```

Chapter 13. CUDA Dynamic Parallelism

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

13.1. Introduction

13.1.1. Overview

Dynamic Parallelism is an extension to the CUDA programming model enabling a CUDA kernel to create and synchronize with new work directly on the GPU. The creation of parallelism dynamically at whichever point in a program that it is needed offers exciting capabilities.

The ability to create work directly from the GPU can reduce the need to transfer execution control and data between host and device, as launch configuration decisions can now be made at runtime by threads executing on the device. Additionally, data-dependent parallel work can be generated inline within a kernel at run-time, taking advantage of the GPU's hardware schedulers and load balancers dynamically and adapting in response to data-driven decisions or workloads. Algorithms and programming patterns that had previously required modifications to eliminate recursion, irregular loop structure, or other constructs that do not fit a flat, single-level of parallelism may more transparently be expressed.

This document describes the extended capabilities of CUDA which enable Dynamic Parallelism, including the modifications and additions to the CUDA programming model necessary to take advantage of these, as well as guidelines and best practices for exploiting this added capacity.

Dynamic Parallelism is only supported by devices of compute capability 3.5 and higher.

13.1.2. Glossary

Definitions for terms used in this guide.

Grid
: A Grid is a collection of *Threads*. Threads in a Grid execute a *Kernel Function* and are divided into *Thread Blocks*.

Thread Block
A Thread Block is a group of threads which execute on the same multiprocessor (*SM*). Threads within a Thread Block have access to shared memory and can be explicitly synchronized.

Kernel Function
A Kernel Function is an implicitly parallel subroutine that executes under the CUDA execution and memory model for every Thread in a Grid.

Host
The Host refers to the execution environment that initially invoked CUDA. Typically the thread running on a system's CPU processor.

Parent
A *Parent Thread*, Thread Block, or Grid is one that has launched new grid(s), the *Child* Grid(s). The Parent is not considered completed until all of its launched Child Grids have also completed.

Child
A Child thread, block, or grid is one that has been launched by a Parent grid. A Child grid must complete before the Parent Thread, Thread Block, or Grid are considered complete.

Thread Block Scope
Objects with Thread Block Scope have the lifetime of a single Thread Block. They only have defined behavior when operated on by Threads in the Thread Block that created the object and are destroyed when the Thread Block that created them is complete.

Device Runtime
The Device Runtime refers to the runtime system and APIs available to enable Kernel Functions to use Dynamic Parallelism.

13.2. Execution Environment and Memory Model

13.2.1. Execution Environment

The CUDA execution model is based on primitives of threads, thread blocks, and grids, with kernel functions defining the program executed by individual threads within a thread block and grid. When a kernel function is invoked the grid's properties are described by an execution configuration, which has a special syntax in CUDA. Support for dynamic parallelism in CUDA extends the ability to configure, launch, and implicitly synchronize upon new grids to threads that are running on the device.

13.2.1.1 Parent and Child Grids

A device thread that configures and launches a new grid belongs to the parent grid, and the grid created by the invocation is a child grid.

The invocation and completion of child grids is properly nested, meaning that the parent grid is not considered complete until all child grids created by its threads have completed, and the runtime guarantees an implicit synchronization between the parent and child.

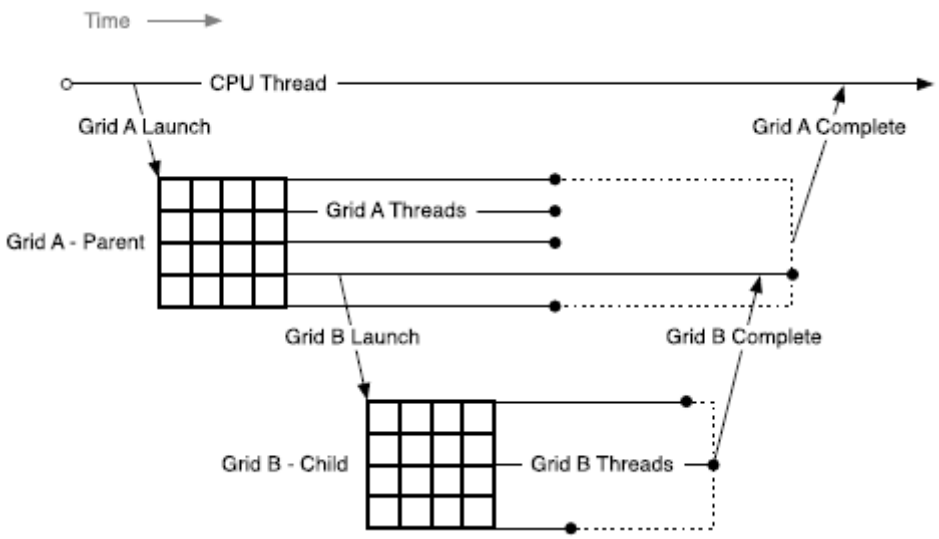


Figure 30: Parent-Child Launch Nesting

13.2.1.2 Scope of CUDA Primitives

On both host and device, the CUDA runtime offers an API for launching kernels and for tracking dependencies between launches via streams and events. On the host system, the state of launches and the CUDA primitives referencing streams and events are shared by all threads within a process; however processes execute independently and may not share CUDA objects.

On the device, launched kernels and CUDA objects are visible to all threads in a grid. This means, for example, that a stream may be created by one thread and used by any other thread in the grid.

13.2.1.3 Synchronization

Warning: Explicit synchronization with child kernels from a parent block (i.e. using `cudaDeviceSynchronize()` in device code) is deprecated in CUDA 11.6 and removed for compute_90+ compilation. For compute capability < 9.0, compile-time opt-in by specifying `-DCUDA_FORCE_CDP1_IF_SUPPORTED` is required to continue using `cudaDeviceSynchronize()` in device code. Note that this is slated for full removal in a future CUDA release.

CUDA runtime operations from any thread, including kernel launches, are visible across all the threads in a grid. This means that an invoking thread in the parent grid may perform synchronization to control the launch order of grids launched by any thread in the grid on streams created by any thread in the grid. Execution of a grid is not considered complete until all launches by all threads in the grid have completed. If all threads in a grid exit before all child launches have completed, an implicit synchronization operation will automatically be triggered.

13.2.1.4 Streams and Events

CUDA *Streams* and *Events* allow control over dependencies between grid launches: grids launched into the same stream execute in-order, and events may be used to create dependencies between streams. Streams and events created on the device serve this exact same purpose.

Streams and events created within a grid exist within grid scope, but have undefined behavior when used outside of the grid where they were created. As described above, all work launched by a grid is implicitly synchronized when the grid exits; work launched into streams is included in this, with all dependencies resolved appropriately. The behavior of operations on a stream that has been modified outside of grid scope is undefined.

Streams and events created on the host have undefined behavior when used within any kernel, just as streams and events created by a parent grid have undefined behavior if used within a child grid.

13.2.1.5 Ordering and Concurrency

The ordering of kernel launches from the device runtime follows CUDA Stream ordering semantics. Within a grid, all kernel launches into the same stream (with the exception of the fire-and-forget stream discussed later) are executed in-order. With multiple threads in the same grid launching into the same stream, the ordering within the stream is dependent on the thread scheduling within the grid, which may be controlled with synchronization primitives such as `__syncthreads()`.

Note that while named streams are shared by all threads within a grid, the implicit *NULL* stream is only shared by all threads within a thread block. If multiple threads in a thread block launch into the implicit stream, then these launches will be executed in-order. If multiple threads in different thread blocks launch into the implicit stream, then these launches may be executed concurrently. If concurrency is desired for launches by multiple threads within a thread block, explicit named streams should be used.

Dynamic Parallelism enables concurrency to be expressed more easily within a program; however, the device runtime introduces no new concurrency guarantees within the CUDA execution model. There is no guarantee of concurrent execution between any number of different thread blocks on a device.

The lack of concurrency guarantee extends to a parent grid and their child grids. When a parent grid launches a child grid, the child may start to execute once stream dependencies are satisfied and hardware resources are available to host the child, but is not guaranteed to begin execution until the parent grid reaches an implicit synchronization point.

While concurrency will often easily be achieved, it may vary as a function of device configuration, application workload, and runtime scheduling. It is therefore unsafe to depend upon any concurrency between different thread blocks.

13.2.1.6 Device Management

There is no multi-GPU support from the device runtime; the device runtime is only capable of operating on the device upon which it is currently executing. It is permitted, however, to query properties for any CUDA capable device in the system.

13.2.2. Memory Model

Parent and child grids share the same global and constant memory storage, but have distinct local and shared memory.

13.2.2.1 Coherence and Consistency

13.2.2.1.1 Global Memory

Parent and child grids have coherent access to global memory, with weak consistency guarantees between child and parent. There is only one point of time in the execution of a child grid when its view of memory is fully consistent with the parent thread: at the point when the child grid is invoked by the parent.

All global memory operations in the parent thread prior to the child grid's invocation are visible to the child grid. With the removal of `cudaDeviceSynchronize()`, it is no longer possible to access the modifications made by the threads in the child grid from the parent grid. The only way to access the modifications made by the threads in the child grid before the parent grid exits is via a kernel launched into the `cudaStreamTailLaunch` stream.

In the following example, the child grid executing `child_launch` is only guaranteed to see the modifications to `data` made before the child grid was launched. Since thread 0 of the parent is performing the launch, the child will be consistent with the memory seen by thread 0 of the parent. Due to the first `__syncthreads()` call, the child will see `data[0]=0`, `data[1]=1`, ..., `data[255]=255` (without the `__syncthreads()` call, only `data[0]=0` would be guaranteed to be seen by the child). The child grid is only guaranteed to return at an implicit synchronization. This means that the modifications made by the threads in the child grid are never guaranteed to become available to the parent grid. To access modifications made by `child_launch`, a `tail_launch` kernel is launched into the `cudaStreamTailLaunch` stream.

```
__global__ void tail_launch(int *data) {
   data[threadIdx.x] = data[threadIdx.x]+1;
}

__global__ void child_launch(int *data) {
   data[threadIdx.x] = data[threadIdx.x]+1;
}

__global__ void parent_launch(int *data) {
   data[threadIdx.x] = threadIdx.x;

   __syncthreads();

   if (threadIdx.x == 0) {
       child_launch<<< 1, 256 >>>(data);
       tail_launch<<< 1, 256, 0, cudaStreamTailLaunch >>>(data);
   }
}

void host_launch(int *data) {
    parent_launch<<< 1, 256 >>>(data);
}
```

13.2.2.1.2 Zero Copy Memory

Zero-copy system memory has identical coherence and consistency guarantees to global memory, and follows the semantics detailed above. A kernel may not allocate or free zero-copy memory, but may use pointers to zero-copy passed in from the host program.

13.2.2.1.3 Constant Memory

Constants may not be modified from the device. They may only be modified from the host, but the behavior of modifying a constant from the host while there is a concurrent grid that access that constant at any point during its lifetime is undefined.

13.2.2.1.4 Shared and Local Memory

Shared and Local memory is private to a thread block or thread, respectively, and is not visible or coherent between parent and child. Behavior is undefined when an object in one of these locations is referenced outside of the scope within which it belongs, and may cause an error.

The NVIDIA compiler will attempt to warn if it can detect that a pointer to local or shared memory is being passed as an argument to a kernel launch. At runtime, the programmer may use the `__isGlobal()` intrinsic to determine whether a pointer references global memory and so may safely be passed to a child launch.

Note that calls to `cudaMemcpy*Async()` or `cudaMemset*Async()` may invoke new child kernels on the device in order to preserve stream semantics. As such, passing shared or local memory pointers to these APIs is illegal and will return an error.

13.2.2.1.5 Local Memory

Local memory is private storage for an executing thread, and is not visible outside of that thread. It is illegal to pass a pointer to local memory as a launch argument when launching a child kernel. The result of dereferencing such a local memory pointer from a child will be undefined.

For example the following is illegal, with undefined behavior if `x_array` is accessed by `child_launch`:

```
int x_array[10];       // Creates x_array in parent's local memory
child_launch<<< 1, 1 >>>(x_array);
```

It is sometimes difficult for a programmer to be aware of when a variable is placed into local memory by the compiler. As a general rule, all storage passed to a child kernel should be allocated explicitly from the global-memory heap, either with `cudaMalloc()`, `new()` or by declaring `__device__` storage at global scope. For example:

```
// Correct - "value" is global storage
__device__ int value;
__device__ void x() {
    value = 5;
    child<<< 1, 1 >>>(&value);
}
```

```
// Invalid - "value" is local storage
__device__ void y() {
    int value = 5;
    child<<< 1, 1 >>>(&value);
}
```

13.2.2.1.6 Texture Memory

Writes to the global memory region over which a texture is mapped are incoherent with respect to texture accesses. Coherence for texture memory is enforced at the invocation of a child grid and when a child grid completes. This means that writes to memory prior to a child kernel launch are reflected in texture memory accesses of the child. Similarly to Global Memory above, writes to memory by a child are never guaranteed to be reflected in the texture memory accesses by a parent. The only way to access the modifications made by the threads in the child grid before the parent grid exits is via a kernel launched into the `cudaStreamTailLaunch` stream. Concurrent accesses by parent and child may result in inconsistent data.

13.3. Programming Interface

13.3.1. CUDA C++ Reference

This section describes changes and additions to the CUDA C++ language extensions for supporting *Dynamic Parallelism*.

The language interface and API available to CUDA kernels using CUDA C++ for Dynamic Parallelism, referred to as the *Device Runtime*, is substantially like that of the CUDA Runtime API available on the host. Where possible the syntax and semantics of the CUDA Runtime API have been retained in order to facilitate ease of code reuse for routines that may run in either the host or device environments.

As with all code in CUDA C++, the APIs and code outlined here is per-thread code. This enables each thread to make unique, dynamic decisions regarding what kernel or operation to execute next. There are no synchronization requirements between threads within a block to execute any of the provided device runtime APIs, which enables the device runtime API functions to be called in arbitrarily divergent kernel code without deadlock.

13.3.1.1 Device-Side Kernel Launch

Kernels may be launched from the device using the standard CUDA <<< >>> syntax:

```
kernel_name<<< Dg, Db, Ns, S >>>([kernel arguments]);
```

- `Dg` is of type `dim3` and specifies the dimensions and size of the grid
- `Db` is of type `dim3` and specifies the dimensions and size of each thread block
- `Ns` is of type `size_t` and specifies the number of bytes of shared memory that is dynamically allocated per thread block for this call in addition to statically allocated memory. `Ns` is an optional argument that defaults to 0.

- S is of type `cudaStream_t` and specifies the stream associated with this call. The stream must have been allocated in the same grid where the call is being made. S is an optional argument that defaults to the NULL stream.

13.3.1.1.1 Launches are Asynchronous

Identical to host-side launches, all device-side kernel launches are asynchronous with respect to the launching thread. That is to say, the `<<<>>>` launch command will return immediately and the launching thread will continue to execute until it hits an implicit launch-synchronization point (such as at a kernel launched into the `cudaStreamTailLaunch` stream).

The child grid launch is posted to the device and will execute independently of the parent thread. The child grid may begin execution at any time after launch, but is not guaranteed to begin execution until the launching thread reaches an implicit launch-synchronization point.

13.3.1.1.2 Launch Environment Configuration

All global device configuration settings (for example, shared memory and `L1` cache size as returned from `cudaDeviceGetCacheConfig()`, and device limits returned from `cudaDeviceGetLimit()`) will be inherited from the parent. Likewise, device limits such as stack size will remain as-configured.

For host-launched kernels, per-kernel configurations set from the host will take precedence over the global setting. These configurations will be used when the kernel is launched from the device as well. It is not possible to reconfigure a kernel's environment from the device.

13.3.1.2 Streams

Both named and unnamed (NULL) streams are available from the device runtime. Named streams may be used by any thread within a grid, but stream handles may not be passed to other child/parent kernels. In other words, a stream should be treated as private to the grid in which it is created.

Similar to host-side launch, work launched into separate streams may run concurrently, but actual concurrency is not guaranteed. Programs that depend upon concurrency between child kernels are not supported by the CUDA programming model and will have undefined behavior.

The host-side NULL stream's cross-stream barrier semantic is not supported on the device (see below for details). In order to retain semantic compatibility with the host runtime, all device streams must be created using the `cudaStreamCreateWithFlags()` API, passing the `cudaStreamNonBlocking` flag. The `cudaStreamCreate()` call is a host-runtime- only API and will fail to compile for the device.

As `cudaStreamSynchronize()` and `cudaStreamQuery()` are unsupported by the device runtime, a kernel launched into the `cudaStreamTailLaunch` stream should be used instead when the application needs to know that stream-launched child kernels have completed.

13.3.1.2.1 The Implicit (NULL) Stream

Within a host program, the unnamed (NULL) stream has additional barrier synchronization semantics with other streams (see *Default Stream* for details). The device runtime offers a single implicit, unnamed stream shared between all threads in a thread block, but as all named streams must be created with the `cudaStreamNonBlocking` flag, work launched into the NULL stream will not insert an implicit dependency on pending work in any other streams (including NULL streams of other thread blocks).

13.3.1.2.2 The Fire-and-Forget Stream

The fire-and-forget named stream (`cudaStreamFireAndForget`) allows the user to launch fire-and-forget work with less boilerplate and without stream tracking overhead. It is functionally identical to, but faster than, creating a new stream per launch, and launching into that stream.

Fire-and-forget launches are immediately scheduled for launch without any dependency on the completion of previously launched grids. No other grid launches can depend on the completion of a fire-and-forget launch, except through the implicit synchronization at the end of the parent grid. So a tail launch or the next grid in parent grid's stream won't launch before a parent grid's fire-and-forget work has completed.

```
// In this example, C2's launch will not wait for C1's completion
__global__ void P( ... ) {
   C1<<< ... , cudaStreamFireAndForget >>>( ... );
   C2<<< ... , cudaStreamFireAndForget >>>( ... );
}
```

The fire-and-forget stream cannot be used to record or wait on events. Attempting to do so results in `cudaErrorInvalidValue`. The fire-and-forget stream is not supported when compiled with `CUDA_FORCE_CDP1_IF_SUPPORTED` defined. Fire-and-forget stream usage requires compilation to be in 64-bit mode.

13.3.1.2.3 The Tail Launch Stream

The tail launch named stream (`cudaStreamTailLaunch`) allows a grid to schedule a new grid for launch after its completion. It should be possible to to use a tail launch to achieve the same functionality as a `cudaDeviceSynchronize()` in most cases.

Each grid has its own tail launch stream. All non-tail launch work launched by a grid is implicitly synchronized before the tail stream is kicked off. I.e. A parent grid's tail launch does not launch until the parent grid and all work launched by the parent grid to ordinary streams or per-thread or fire-and-forget streams have completed. If two grids are launched to the same grid's tail launch stream, the later grid does not launch until the earlier grid and all its descendent work has completed.

```
// In this example, C2 will only launch after C1 completes.
__global__ void P( ... ) {
   C1<<< ... , cudaStreamTailLaunch >>>( ... );
   C2<<< ... , cudaStreamTailLaunch >>>( ... );
}
```

Grids launched into the tail launch stream will not launch until the completion of all work by the parent grid, including all other grids (and their descendants) launched by the parent in all non-tail launched streams, including work executed or launched after the tail launch.

```
// In this example, C will only launch after all X, F and P complete.
__global__ void P( ... ) {
   C<<< ... , cudaStreamTailLaunch >>>( ... );
   X<<< ... , cudaStreamPerThread >>>( ... );
   F<<< ... , cudaStreamFireAndForget >>>( ... )
}
```

The next grid in the parent grid's stream will not be launched before a parent grid's tail launch work has completed. In other words, the tail launch stream behaves as if it were inserted between its parent grid and the next grid in its parent grid's stream.

```
// In this example, P2 will only launch after C completes.
__global__ void P1( ... ) {
   C<<< ... , cudaStreamTailLaunch >>>( ... );
}

__global__ void P2( ... ) {
}

int main ( ... ) {
   ...
   P1<<< ... >>>( ... );
   P2<<< ... >>>( ... );
   ...
}
```

Each grid only gets one tail launch stream. To tail launch concurrent grids, it can be done like the example below.

```
// In this example,  C1 and C2 will launch concurrently after P's completion
__global__ void T( ... ) {
   C1<<< ... , cudaStreamFireAndForget >>>( ... );
   C2<<< ... , cudaStreamFireAndForget >>>( ... );
}

__global__ void P( ... ) {
   ...
   T<<< ... , cudaStreamTailLaunch >>>( ... );
}
```

The tail launch stream cannot be used to record or wait on events. Attempting to do so results in `cudaErrorInvalidValue`. The tail launch stream is not supported when compiled with `CUDA_FORCE_CDP1_IF_SUPPORTED` defined. Tail launch stream usage requires compilation to be in 64-bit mode.

13.3.1.3 Events

Only the inter-stream synchronization capabilities of CUDA events are supported. This means that `cudaStreamWaitEvent()` is supported, but `cudaEventSynchronize()`, `cudaEventElapsedTime()`, and `cudaEventQuery()` are not. As `cudaEventElapsedTime()` is not supported, cudaEvents must be created via `cudaEventCreateWithFlags()`, passing the `cudaEventDisableTiming` flag.

As with named streams, event objects may be shared between all threads within the grid which created them but are local to that grid and may not be passed to other kernels. Event handles are not guaranteed to be unique between grids, so using an event handle within a grid that did not create it will result in undefined behavior.

13.3.1.4 Synchronization

It is up to the program to perform sufficient inter-thread synchronization, for example via a CUDA Event, if the calling thread is intended to synchronize with child grids invoked from other threads.

As it is not possible to explicitly synchronize child work from a parent thread, there is no way to guarantee that changes occurring in child grids are visible to threads within the parent grid.

13.3.1.5 Device Management

Only the device on which a kernel is running will be controllable from that kernel. This means that device APIs such as `cudaSetDevice()` are not supported by the device runtime. The active device as seen from the GPU (returned from `cudaGetDevice()`) will have the same device number as seen from the host system. The `cudaDeviceGetAttribute()` call may request information about another device as this API allows specification of a device ID as a parameter of the call. Note that the catch-all `cudaGetDeviceProperties()` API is not offered by the device runtime - properties must be queried individually.

13.3.1.6 Memory Declarations

13.3.1.6.1 Device and Constant Memory

Memory declared at file scope with `__device__` or `__constant__` memory space specifiers behaves identically when using the device runtime. All kernels may read or write device variables, whether the kernel was initially launched by the host or device runtime. Equivalently, all kernels will have the same view of `__constant__`s as declared at the module scope.

13.3.1.6.2 Textures and Surfaces

CUDA supports dynamically created texture and surface objects[7], where a texture object may be created on the host, passed to a kernel, used by that kernel, and then destroyed from the host. The device runtime does not allow creation or destruction of texture or surface objects from within device code, but texture and surface objects created from the host may be used and passed around freely on the device. Regardless of where they are created, dynamically created texture objects are always valid and may be passed to child kernels from a parent.

Note: The device runtime does not support legacy module-scope (i.e., Fermi-style) textures and surfaces within a kernel launched from the device. Module-scope (legacy) textures may be created from the host and used in device code as for any kernel, but may only be used by a top-level kernel (i.e., the one which is launched from the host).

---

[7] Dynamically created texture and surface objects are an addition to the CUDA memory model introduced with CUDA 5.0. Please see the *CUDA Programming Guide* for details.

13.3.1.6.3 Shared Memory Variable Declarations

In CUDA C++ shared memory can be declared either as a statically sized file-scope or function-scoped variable, or as an `extern` variable with the size determined at runtime by the kernel's caller via a launch configuration argument. Both types of declarations are valid under the device runtime.

```
__global__ void permute(int n, int *data) {
   extern __shared__ int smem[];
   if (n <= 1)
       return;

   smem[threadIdx.x] = data[threadIdx.x];
   __syncthreads();

   permute_data(smem, n);
   __syncthreads();

   // Write back to GMEM since we can't pass SMEM to children.
   data[threadIdx.x] = smem[threadIdx.x];
   __syncthreads();

   if (threadIdx.x == 0) {
       permute<<< 1, 256, n/2*sizeof(int) >>>(n/2, data);
       permute<<< 1, 256, n/2*sizeof(int) >>>(n/2, data+n/2);
   }
}

void host_launch(int *data) {
    permute<<< 1, 256, 256*sizeof(int) >>>(256, data);
}
```

13.3.1.6.4 Symbol Addresses

Device-side symbols (i.e., those marked `__device__`) may be referenced from within a kernel simply via the & operator, as all global-scope device variables are in the kernel's visible address space. This also applies to `__constant__` symbols, although in this case the pointer will reference read-only data.

Given that device-side symbols can be referenced directly, those CUDA runtime APIs which reference symbols (e.g., `cudaMemcpyToSymbol()` or `cudaGetSymbolAddress()`) are redundant and hence not supported by the device runtime. Note this implies that constant data cannot be altered from within a running kernel, even ahead of a child kernel launch, as references to `__constant__` space are read-only.

13.3.1.7 API Errors and Launch Failures

As usual for the CUDA runtime, any function may return an error code. The last error code returned is recorded and may be retrieved via the `cudaGetLastError()` call. Errors are recorded per-thread, so that each thread can identify the most recent error that it has generated. The error code is of type `cudaError_t`.

Similar to a host-side launch, device-side launches may fail for many reasons (invalid arguments, etc). The user must call `cudaGetLastError()` to determine if a launch generated an error, however lack of an error after launch does not imply the child kernel completed successfully.

For device-side exceptions, e.g., access to an invalid address, an error in a child grid will be returned to the host.

13.3.1.7.1 Launch Setup APIs

Kernel launch is a system-level mechanism exposed through the device runtime library, and as such is available directly from PTX via the underlying `cudaGetParameterBuffer()` and `cudaLaunchDevice()` APIs. It is permitted for a CUDA application to call these APIs itself, with the same requirements as for PTX. In both cases, the user is then responsible for correctly populating all necessary data structures in the correct format according to specification. Backwards compatibility is guaranteed in these data structures.

As with host-side launch, the device-side operator `<<<>>>` maps to underlying kernel launch APIs. This is so that users targeting PTX will be able to enact a launch, and so that the compiler front-end can translate `<<<>>>` into these calls.

Table 13: New Device-only Launch Implementation Functions

Runtime API Launch Functions	**Description of Difference From Host Runtime Behaviour (behavior is identical if no description)**
`cudaGetParameterBuffer`	Generated automatically from `<<<>>>`. Note different API to host equivalent.
`cudaLaunchDevice`	Generated automatically from `<<<>>>`. Note different API to host equivalent.

The APIs for these launch functions are different to those of the CUDA Runtime API, and are defined as follows:

```
extern   device   cudaError_t cudaGetParameterBuffer(void **params);
extern __device__ cudaError_t cudaLaunchDevice(void *kernel,
                                               void *params, dim3 gridDim,
                                               dim3 blockDim,
                                               unsigned int sharedMemSize = 0,
                                               cudaStream_t stream = 0);
```

13.3.1.8 API Reference

The portions of the CUDA Runtime API supported in the device runtime are detailed here. Host and device runtime APIs have identical syntax; semantics are the same except where indicated. The following table provides an overview of the API relative to the version available from the host.

Table 14: Supported API Functions

Runtime API Functions	Details
cudaDeviceGetCacheConfig	
cudaDeviceGetLimit	
cudaGetLastError	Last error is per-thread state, not per-block state
cudaPeekAtLastError	
cudaGetErrorString	
cudaGetDeviceCount	
cudaDeviceGetAttribute	Will return attributes for any device
cudaGetDevice	Always returns current device ID as would be seen from host
cudaStreamCreateWithFlags	Must pass cudaStreamNonBlocking flag
cudaStreamDestroy	
cudaStreamWaitEvent	
cudaEventCreateWithFlags	Must pass cudaEventDisableTiming flag
cudaEventRecord	
cudaEventDestroy	
cudaFuncGetAttributes	
cudaMemcpyAsync	Notes about all memcpy/memset functions: ► Only async memcpy/set functions are supported ► Only device-to-device memcpy is permitted ► May not pass in local or shared memory pointers
cudaMemcpy2DAsync	
cudaMemcpy3DAsync	
cudaMemsetAsync	
cudaMemset2DAsync	
cudaMemset3DAsync	
cudaRuntimeGetVersion	
cudaMalloc	May not call cudaFree on the device on a pointer created on the host, and vice-versa
cudaFree	
cudaOccupancyMaxActiveBlocksPerMultiprocessor	
cudaOccupancyMaxPotentialBlockSize	
cudaOccupancyMaxPotentialBlockSizeVariableSMem	

13.3.2. Device-side Launch from PTX

This section is for the programming language and compiler implementers who target *Parallel Thread Execution* (PTX) and plan to support *Dynamic Parallelism* in their language. It provides the low-level details related to supporting kernel launches at the PTX level.

13.3.2.1 Kernel Launch APIs

Device-side kernel launches can be implemented using the following two APIs accessible from PTX: `cudaLaunchDevice()` and `cudaGetParameterBuffer()`. `cudaLaunchDevice()` launches the specified kernel with the parameter buffer that is obtained by calling `cudaGetParameterBuffer()` and filled with the parameters to the launched kernel. The parameter buffer can be NULL, i.e., no need to invoke `cudaGetParameterBuffer()`, if the launched kernel does not take any parameters.

13.3.2.1.1 cudaLaunchDevice

At the PTX level, `cudaLaunchDevice()`needs to be declared in one of the two forms shown below before it is used.

```
// PTX-level Declaration of cudaLaunchDevice() when .address_size is 64
.extern .func(.param .b32 func_retval0) cudaLaunchDevice
(
  .param .b64 func,
  .param .b64 parameterBuffer,
  .param .align 4 .b8 gridDimension[12],
  .param .align 4 .b8 blockDimension[12],
  .param .b32 sharedMemSize,
  .param .b64 stream
)
;
```

The CUDA-level declaration below is mapped to one of the aforementioned PTX-level declarations and is found in the system header file `cuda_device_runtime_api.h`. The function is defined in the `cudadevrt` system library, which must be linked with a program in order to use device-side kernel launch functionality.

```
// CUDA-level declaration of cudaLaunchDevice()
extern "C" __device__
cudaError_t cudaLaunchDevice(void *func, void *parameterBuffer,
                             dim3 gridDimension, dim3 blockDimension,
                             unsigned int sharedMemSize,
                             cudaStream_t stream);
```

The first parameter is a pointer to the kernel to be is launched, and the second parameter is the parameter buffer that holds the actual parameters to the launched kernel. The layout of the parameter buffer is explained in *Parameter Buffer Layout*, below. Other parameters specify the launch configuration, i.e., as grid dimension, block dimension, shared memory size, and the stream associated with the launch (please refer to *Execution Configuration* for the detailed description of launch configuration.

13.3.2.1.2 cudaGetParameterBuffer

`cudaGetParameterBuffer()` needs to be declared at the PTX level before it's used. The PTX-level declaration must be in one of the two forms given below, depending on address size:

```
// PTX-level Declaration of cudaGetParameterBuffer() when .address_size is 64
.extern .func(.param .b64 func_retval0) cudaGetParameterBuffer
(
  .param .b64 alignment,
  .param .b64 size
)
;
```

The following CUDA-level declaration of `cudaGetParameterBuffer()` is mapped to the aforementioned PTX-level declaration:

```
// CUDA-level Declaration of cudaGetParameterBuffer()
extern "C" __device__
void *cudaGetParameterBuffer(size_t alignment, size_t size);
```

The first parameter specifies the alignment requirement of the parameter buffer and the second parameter the size requirement in bytes. In the current implementation, the parameter buffer returned by `cudaGetParameterBuffer()` is always guaranteed to be 64- byte aligned, and the alignment requirement parameter is ignored. However, it is recommended to pass the correct alignment requirement value - which is the largest alignment of any parameter to be placed in the parameter buffer - to `cudaGetParameterBuffer()` to ensure portability in the future.

13.3.2.2 Parameter Buffer Layout

Parameter reordering in the parameter buffer is prohibited, and each individual parameter placed in the parameter buffer is required to be aligned. That is, each parameter must be placed at the n^{th} byte in the parameter buffer, where n is the smallest multiple of the parameter size that is greater than the offset of the last byte taken by the preceding parameter. The maximum size of the parameter buffer is 4KB.

For a more detailed description of PTX code generated by the CUDA compiler, please refer to the PTX-3.5 specification.

13.3.3. Toolkit Support for Dynamic Parallelism

13.3.3.1 Including Device Runtime API in CUDA Code

Similar to the host-side runtime API, prototypes for the CUDA device runtime API are included automatically during program compilation. There is no need to include`cuda_device_runtime_api.h` explicitly.

13.3.3.2 Compiling and Linking

When compiling and linking CUDA programs using dynamic parallelism with `nvcc`, the program will automatically link against the static device runtime library `libcudadevrt`.

The device runtime is offered as a static library (`cudadevrt.lib` on Windows, `libcudadevrt.a` under Linux), against which a GPU application that uses the device runtime must be linked. Linking of device libraries can be accomplished through `nvcc` and/or `nvlink`. Two simple examples are shown below.

A device runtime program may be compiled and linked in a single step, if all required source files can be specified from the command line:

```
$ nvcc -arch=sm_75 -rdc=true hello_world.cu -o hello -lcudadevrt
```

It is also possible to compile CUDA .cu source files first to object files, and then link these together in a two-stage process:

```
$ nvcc -arch=sm_75 -dc hello_world.cu -o hello_world.o
$ nvcc -arch=sm_75 -rdc=true hello_world.o -o hello -lcudadevrt
```

Please see the Using Separate Compilation section of The CUDA Driver Compiler NVCC guide for more details.

13.4. Programming Guidelines

13.4.1. Basics

The device runtime is a functional subset of the host runtime. API level device management, kernel launching, device memcpy, stream management, and event management are exposed from the device runtime.

Programming for the device runtime should be familiar to someone who already has experience with CUDA. Device runtime syntax and semantics are largely the same as that of the host API, with any exceptions detailed earlier in this document.

The following example shows a simple *Hello World* program incorporating dynamic parallelism:

```
#include <stdio.h>

__global__ void childKernel()
{
    printf("Hello ");
}

__global__ void tailKernel()
{
    printf("World!\n");
}

__global__ void parentKernel()
{
    // launch child
    childKernel<<<1,1>>>();
```

(continues on next page)

(continued from previous page)

```
    if (cudaSuccess != cudaGetLastError()) {
        return;
    }

    // launch tail into cudaStreamTailLaunch stream
    // implicitly synchronizes: waits for child to complete
    tailKernel<<<1,1,0,cudaStreamTailLaunch>>>();

}

int main(int argc, char *argv[])
{
    // launch parent
    parentKernel<<<1,1>>>();
    if (cudaSuccess != cudaGetLastError()) {
        return 1;
    }

    // wait for parent to complete
    if (cudaSuccess != cudaDeviceSynchronize()) {
        return 2;
    }

    return 0;
}
```

This program may be built in a single step from the command line as follows:

```
$ nvcc -arch=sm_75 -rdc=true hello_world.cu -o hello -lcudadevrt
```

13.4.2. Performance

13.4.2.1 Dynamic-parallelism-enabled Kernel Overhead

System software which is active when controlling dynamic launches may impose an overhead on any kernel which is running at the time, whether or not it invokes kernel launches of its own. This overhead arises from the device runtime's execution tracking and management software and may result in decreased performance. This overhead is, in general, incurred for applications that link against the device runtime library.

13.4.3. Implementation Restrictions and Limitations

Dynamic Parallelism guarantees all semantics described in this document, however, certain hardware and software resources are implementation-dependent and limit the scale, performance and other properties of a program which uses the device runtime.

13.4.3.1 Runtime

13.4.3.1.1 Memory Footprint

The device runtime system software reserves memory for various management purposes, in particular a reservation for tracking pending grid launches. Configuration controls are available to reduce the size of this reservation in exchange for certain launch limitations. See *Configuration Options*, below, for details.

13.4.3.1.2 Pending Kernel Launches

When a kernel is launched, all associated configuration and parameter data is tracked until the kernel completes. This data is stored within a system-managed launch pool.

The size of the fixed-size launch pool is configurable by calling `cudaDeviceSetLimit()` from the host and specifying `cudaLimitDevRuntimePendingLaunchCount`.

13.4.3.1.3 Configuration Options

Resource allocation for the device runtime system software is controlled via the `cudaDeviceSetLimit()` API from the host program. Limits must be set before any kernel is launched, and may not be changed while the GPU is actively running programs.

The following named limits may be set:

Limit	Behavior
`cudaLimitDevRuntimePendingLaunchCount`	Controls the amount of memory set aside for buffering kernel launches and events which have not yet begun to execute, due either to unresolved dependencies or lack of execution resources. When the buffer is full, an attempt to allocate a launch slot during a device side kernel launch will fail and return `cudaErrorLaunchOutOfResources`, while an attempt to allocate an event slot will fail and return `cudaErrorMemoryAllocation`. The default number of launch slots is 2048. Applications may increase the number of launch and/or event slots by setting `cudaLimitDevRuntimePendingLaunchCount`. The number of event slots allocated is twice the value of that limit.
`cudaLimitStackSize`	Controls the stack size in bytes of each GPU thread. The CUDA driver automatically increases the per-thread stack size for each kernel launch as needed. This size isn't reset back to the original value after each launch. To set the per-thread stack size to a different value, `cudaDeviceSetLimit()` can be called to set this limit. The stack will be immediately resized, and if necessary, the device will block until all preceding requested tasks are complete. `cudaDeviceGetLimit()` can be called to get the current per-thread stack size.

13.4.3.1.4 Memory Allocation and Lifetime

`cudaMalloc()` and `cudaFree()` have distinct semantics between the host and device environments. When invoked from the host, `cudaMalloc()` allocates a new region from unused device memory. When invoked from the device runtime these functions map to device-side `malloc()` and `free()`. This implies that within the device environment the total allocatable memory is limited to the device `malloc()` heap size, which may be smaller than the available unused device memory. Also, it is an error to invoke `cudaFree()` from the host program on a pointer which was allocated by `cudaMalloc()` on the device or vice-versa.

	`cudaMalloc()` on Host	**`cudaMalloc()` on Device**
`cudaFree()` on Host	Supported	Not Supported
`cudaFree()` on Device	Not Supported	Supported
Allocation limit	Free device memory	`cudaLimitMallocHeapSize`

13.4.3.1.5 SM Id and Warp Id

Note that in PTX `%smid` and `%warpid` are defined as volatile values. The device runtime may reschedule thread blocks onto different SMs in order to more efficiently manage resources. As such, it is unsafe to rely upon `%smid` or `%warpid` remaining unchanged across the lifetime of a thread or thread block.

13.4.3.1.6 ECC Errors

No notification of ECC errors is available to code within a CUDA kernel. ECC errors are reported at the host side once the entire launch tree has completed. Any ECC errors which arise during execution of a nested program will either generate an exception or continue execution (depending upon error and configuration).

13.5. CDP2 vs CDP1

This section summarises the differences between, and the compatibility and interoperability of, the new (CDP2) and legacy (CDP1) CUDA Dynamic Parallelism interfaces. It also shows how to opt-out of the CDP2 interface on devices of compute capability less than 9.0.

13.5.1. Differences Between CDP1 and CDP2

Explicit device-side synchronization is no longer possible with CDP2 or on devices of compute capability 9.0 or higher. Implicit synchronization (such as tail launches) must be used instead.

Attempting to query or set `cudaLimitDevRuntimeSyncDepth` (or `CU_LIMIT_DEV_RUNTIME_SYNC_DEPTH`) with CDP2 or on devices of compute capability 9.0 or higher results in `cudaErrorUnsupportedLimit`.

CDP2 no longer has a virtualized pool for pending launches that don't fit in the fixed-sized pool. `cudaLimitDevRuntimePendingLaunchCount` must be set to be large enough to avoid running out of launch slots.

For CDP2, there is a limit to the total number of events existing at once (note that events are destroyed only after a launch completes), equal to twice the pending launch count. `cudaLimitDevRuntimePendingLaunchCount` must be set to be large enough to avoid running out of event slots.

Streams are tracked per grid with CDP2 or on devices of compute capability 9.0 or higher, not per thread block. This allows work to be launched into a stream created by another thread block. Attempting to do so with the CDP1 results in `cudaErrorInvalidValue`.

CDP2 introduces the tail launch (`cudaStreamTailLaunch`) and fire-and-forget (`cudaStreamFireAndForget`) named streams.

CDP2 is supported only under 64-bit compilation mode.

13.5.2. Compatibility and Interoperability

CDP2 is the default. Functions can be compiled with `-DCUDA_FORCE_CDP1_IF_SUPPORTED` to opt-out of using CDP2 on devices of compute capability less than 9.0.

	Function compiler with CUDA 12.0 and newer (default)	**Function compiled with pre-CUDA 12.0 or with CUDA 12.0 and newer with `-DCUDA_FORCE_CDP1_IF_SUPPORTED` specified**
Compilation	Compile error if device code references `cudaDeviceSynchronize`.	Compile error if code references `cudaStreamTailLaunch` or `cudaStreamFireAndForget`. Compile error if device code references `cudaDeviceSynchronize` and code is compiled for sm_90 or newer.
Compute capability < 9.0	New interface is used.	Legacy interface is used.
Compute capability 9.0 and higher	New interface is used.	New interface is used. If function references `cudaDeviceSynchronize` in device code, function load returns `cudaErrorSymbolNotFound` (this could happen if the code is compiled for devices of compute capability less than 9.0, but run on devices of compute capability 9.0 or higher using JIT).

Functions using CDP1 and CDP2 may be loaded and run simultaneously in the same context. The CDP1 functions are able to use CDP1-specific features (e.g. `cudaDeviceSynchronize`) and CDP2 functions are able to use CDP2-specific features (e.g. tail launch and fire-and-forget launch).

A function using CDP1 cannot launch a function using CDP2, and vice versa. If a function that would use CDP1 contains in its call graph a function that would use CDP2, or vice versa, `cudaErrorCdpVersionMismatch` would result during function load.

13.6. Legacy CUDA Dynamic Parallelism (CDP1)

See *CUDA Dynamic Parallelism*, above, for CDP2 version of document.

13.6.1. Execution Environment and Memory Model (CDP1)

See *Execution Environment and Memory Model*, above, for CDP2 version of document.

13.6.1.1 Execution Environment (CDP1)

See *Execution Environment*, above, for CDP2 version of document.

The CUDA execution model is based on primitives of threads, thread blocks, and grids, with kernel functions defining the program executed by individual threads within a thread block and grid. When a kernel function is invoked the grid's properties are described by an execution configuration, which has a special syntax in CUDA. Support for dynamic parallelism in CUDA extends the ability to configure, launch, and synchronize upon new grids to threads that are running on the device.

Warning: Explicit synchronization with child kernels from a parent block (i.e. using `cudaDeviceSynchronize()` in device code) block is deprecated in CUDA 11.6, removed for compute_90+ compilation, and is slated for full removal in a future CUDA release.

13.6.1.1.1 Parent and Child Grids (CDP1)

See *Parent and Child Grids*, above, for CDP2 version of document.

A device thread that configures and launches a new grid belongs to the parent grid, and the grid created by the invocation is a child grid.

The invocation and completion of child grids is properly nested, meaning that the parent grid is not considered complete until all child grids created by its threads have completed. Even if the invoking threads do not explicitly synchronize on the child grids launched, the runtime guarantees an implicit synchronization between the parent and child.

Warning: Explicit synchronization with child kernels from a parent block (i.e. using `cudaDeviceSynchronize()` in device code) is deprecated in CUDA 11.6, removed for compute_90+ compilation, and is slated for full removal in a future CUDA release.

13.6.1.1.2 Scope of CUDA Primitives (CDP1)

See *Scope of CUDA Primitives*, above, for CDP2 version of document.

On both host and device, the CUDA runtime offers an API for launching kernels, for waiting for launched work to complete, and for tracking dependencies between launches via streams and events. On the host system, the state of launches and the CUDA primitives referencing streams and events are shared by all threads within a process; however processes execute independently and may not share CUDA objects.

A similar hierarchy exists on the device: launched kernels and CUDA objects are visible to all threads in a thread block, but are independent between thread blocks. This means for example that a stream may be created by one thread and used by any other thread in the same thread block, but may not be shared with threads in any other thread block.

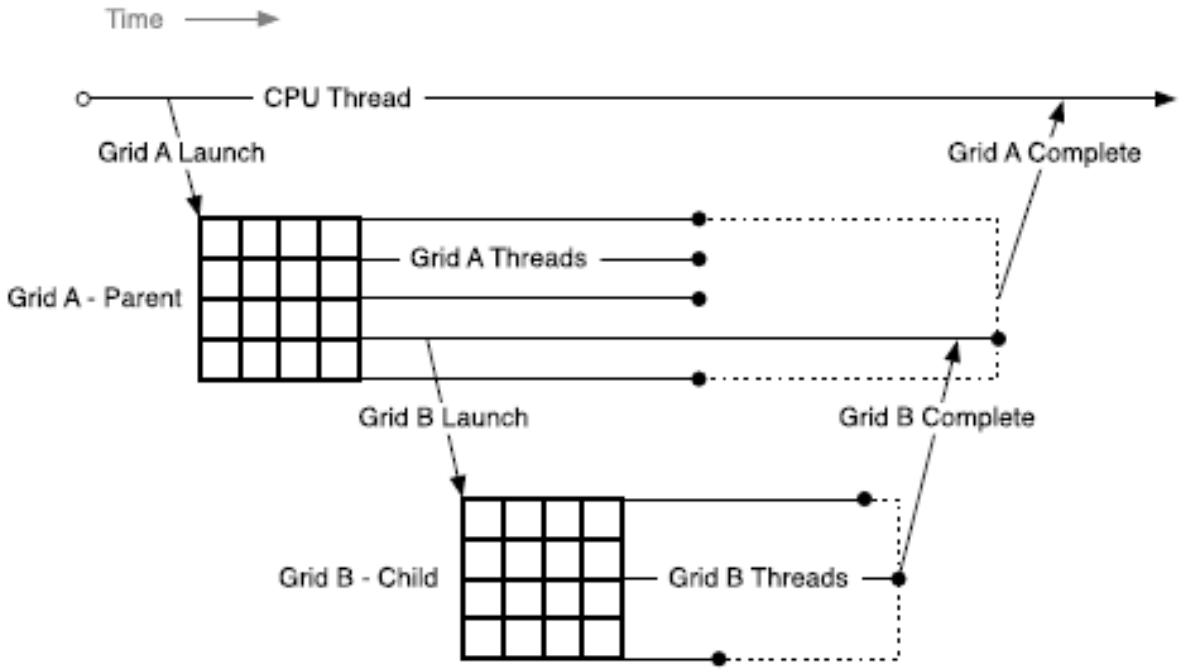


Figure 31: Parent-Child Launch Nesting

13.6.1.1.3 Synchronization (CDP1)

See *Synchronization*, above, for CDP2 version of document.

Warning: Explicit synchronization with child kernels from a parent block (i.e. using `cudaDeviceSynchronize()` in device code) is deprecated in CUDA 11.6, removed for compute_90+ compilation, and is slated for full removal in a future CUDA release.

CUDA runtime operations from any thread, including kernel launches, are visible across a thread block. This means that an invoking thread in the parent grid may perform synchronization on the grids launched by that thread, by other threads in the thread block, or on streams created within the same thread block. Execution of a thread block is not considered complete until all launches by all threads in the block have completed. If all threads in a block exit before all child launches have completed, a synchronization operation will automatically be triggered.

13.6.1.1.4 Streams and Events (CDP1)

See *Streams and Events*, above, for CDP2 version of document.

CUDA *Streams* and *Events* allow control over dependencies between grid launches: grids launched into the same stream execute in-order, and events may be used to create dependencies between streams. Streams and events created on the device serve this exact same purpose.

Streams and events created within a grid exist within thread block scope but have undefined behavior when used outside of the thread block where they were created. As described above, all work launched by a thread block is implicitly synchronized when the block exits; work launched into streams is included in this, with all dependencies resolved appropriately. The behavior of operations on a stream

that has been modified outside of thread block scope is undefined.

Streams and events created on the host have undefined behavior when used within any kernel, just as streams and events created by a parent grid have undefined behavior if used within a child grid.

13.6.1.1.5 Ordering and Concurrency (CDP1)

See *Ordering and Concurrency*, above, for CDP2 version of document.

The ordering of kernel launches from the device runtime follows CUDA Stream ordering semantics. Within a thread block, all kernel launches into the same stream are executed in-order. With multiple threads in the same thread block launching into the same stream, the ordering within the stream is dependent on the thread scheduling within the block, which may be controlled with synchronization primitives such as `__syncthreads()`.

Note that because streams are shared by all threads within a thread block, the implicit *NULL* stream is also shared. If multiple threads in a thread block launch into the implicit stream, then these launches will be executed in-order. If concurrency is desired, explicit named streams should be used.

Dynamic Parallelism enables concurrency to be expressed more easily within a program; however, the device runtime introduces no new concurrency guarantees within the CUDA execution model. There is no guarantee of concurrent execution between any number of different thread blocks on a device.

The lack of concurrency guarantee extends to parent thread blocks and their child grids. When a parent thread block launches a child grid, the child is not guaranteed to begin execution until the parent thread block reaches an explicit synchronization point (such as `cudaDeviceSynchronize()`).

Warning: Explicit synchronization with child kernels from a parent block (i.e. using `cudaDeviceSynchronize()` in device code) is deprecated in CUDA 11.6, removed for compute_90+ compilation, and is slated for full removal in a future CUDA release.

While concurrency will often easily be achieved, it may vary as a function of deviceconfiguration, application workload, and runtime scheduling. It is therefore unsafe to depend upon any concurrency between different thread blocks.

13.6.1.1.6 Device Management (CDP1)

See *Device Management*, above, for CDP2 version of document.

There is no multi-GPU support from the device runtime; the device runtime is only capable of operating on the device upon which it is currently executing. It is permitted, however, to query properties for any CUDA capable device in the system.

13.6.1.2 Memory Model (CDP1)

See *Memory Model*, above, for CDP2 version of document.

Parent and child grids share the same global and constant memory storage, but have distinct local and shared memory.

13.6.1.2.1 Coherence and Consistency (CDP1)

See *Coherence and Consistency*, above, for CDP2 version of document.

13.6.1.2.1.1 Global Memory (CDP1)

See *Global Memory*, above, for CDP2 version of document.

Parent and child grids have coherent access to global memory, with weak consistency guarantees between child and parent. There are two points in the execution of a child grid when its view of memory is fully consistent with the parent thread: when the child grid is invoked by the parent, and when the child grid completes as signaled by a synchronization API invocation in the parent thread.

Warning: Explicit synchronization with child kernels from a parent block (i.e. using `cudaDeviceSynchronize()` in device code) is deprecated in CUDA 11.6, removed for compute_90+ compilation, and is slated for full removal in a future CUDA release.

All global memory operations in the parent thread prior to the child grid's invocation are visible to the child grid. All memory operations of the child grid are visible to the parent after the parent has synchronized on the child grid's completion.

In the following example, the child grid executing `child_launch` is only guaranteed to see the modifications to `data` made before the child grid was launched. Since thread 0 of the parent is performing the launch, the child will be consistent with the memory seen by thread 0 of the parent. Due to the first `__syncthreads()` call, the child will see `data[0]=0`, `data[1]=1`, ..., `data[255]=255` (without the `__syncthreads()` call, only `data[0]` would be guaranteed to be seen by the child). When the child grid returns, thread 0 is guaranteed to see modifications made by the threads in its child grid. Those modifications become available to the other threads of the parent grid only after the second `__syncthreads()` call:

```
__global__ void child_launch(int *data) {
   data[threadIdx.x] = data[threadIdx.x]+1;
}

__global__ void parent_launch(int *data) {
   data[threadIdx.x] = threadIdx.x;

   __syncthreads();

   if (threadIdx.x == 0) {
       child_launch<<< 1, 256 >>>(data);
       cudaDeviceSynchronize();
   }

   __syncthreads();
```

(continues on next page)

(continued from previous page)

```
}

void host_launch(int *data) {
    parent_launch<<< 1, 256 >>>(data);
}
```

13.6.1.2.1.2 Zero Copy Memory (CDP1)

See *Zero Copy Memory*, above, for CDP2 version of document.

Zero-copy system memory has identical coherence and consistency guarantees to global memory, and follows the semantics detailed above. A kernel may not allocate or free zero-copy memory, but may use pointers to zero-copy passed in from the host program.

13.6.1.2.1.3 Constant Memory (CDP1)

See *Constant Memory*, above, for CDP2 version of document.

Constants are immutable and may not be modified from the device, even between parent and child launches. That is to say, the value of all `__constant__` variables must be set from the host prior to launch. Constant memory is inherited automatically by all child kernels from their respective parents.

Taking the address of a constant memory object from within a kernel thread has the same semantics as for all CUDA programs, and passing that pointer from parent to child or from a child to parent is naturally supported.

13.6.1.2.1.4 Shared and Local Memory (CDP1)

See *Shared and Local Memory*, above, for CDP2 version of document.

Shared and Local memory is private to a thread block or thread, respectively, and is not visible or coherent between parent and child. Behavior is undefined when an object in one of these locations is referenced outside of the scope within which it belongs, and may cause an error.

The NVIDIA compiler will attempt to warn if it can detect that a pointer to local or shared memory is being passed as an argument to a kernel launch. At runtime, the programmer may use the `__isGlobal()` intrinsic to determine whether a pointer references global memory and so may safely be passed to a child launch.

Note that calls to `cudaMemcpy*Async()` or `cudaMemset*Async()` may invoke new child kernels on the device in order to preserve stream semantics. As such, passing shared or local memory pointers to these APIs is illegal and will return an error.

13.6.1.2.1.5 Local Memory (CDP1)

See *Local Memory*, above, for CDP2 version of document.

Local memory is private storage for an executing thread, and is not visible outside of that thread. It is illegal to pass a pointer to local memory as a launch argument when launching a child kernel. The result of dereferencing such a local memory pointer from a child will be undefined.

For example the following is illegal, with undefined behavior if `x_array` is accessed by `child_launch`:

```
int x_array[10];       // Creates x_array in parent's local memory
child_launch<<< 1, 1 >>>(x_array);
```

It is sometimes difficult for a programmer to be aware of when a variable is placed into local memory by the compiler. As a general rule, all storage passed to a child kernel should be allocated explicitly from the global-memory heap, either with `cudaMalloc()`, `new()` or by declaring `__device__` storage at global scope. For example:

```
// Correct - "value" is global storage
__device__ int value;
__device__ void x() {
    value = 5;
    child<<< 1, 1 >>>(&value);
}
```

```
// Invalid - "value" is local storage
__device__ void y() {
    int value = 5;
    child<<< 1, 1 >>>(&value);
}
```

13.6.1.2.1.6 Texture Memory (CDP1)

See *Texture Memory*, above, for CDP2 version of document.

Writes to the global memory region over which a texture is mapped are incoherent with respect to texture accesses. Coherence for texture memory is enforced at the invocation of a child grid and when a child grid completes. This means that writes to memory prior to a child kernel launch are reflected in texture memory accesses of the child. Similarly, writes to memory by a child will be reflected in the texture memory accesses by a parent, but only after the parent synchronizes on the child's completion. Concurrent accesses by parent and child may result in inconsistent data.

Warning: Explicit synchronization with child kernels from a parent block (i.e. using `cudaDeviceSynchronize()` in device code) is deprecated in CUDA 11.6, removed for compute_90+ compilation, and is slated for full removal in a future CUDA release.

13.6.2. Programming Interface (CDP1)

See *Programming Interface*, above, for CDP2 version of document.

13.6.2.1 CUDA C++ Reference (CDP1)

See *CUDA C++ Reference*, above, for CDP2 version of document.

This section describes changes and additions to the CUDA C++ language extensions for supporting *Dynamic Parallelism*.

The language interface and API available to CUDA kernels using CUDA C++ for Dynamic Parallelism, referred to as the *Device Runtime*, is substantially like that of the CUDA Runtime API available on the host. Where possible the syntax and semantics of the CUDA Runtime API have been retained in order to facilitate ease of code reuse for routines that may run in either the host or device environments.

As with all code in CUDA C++, the APIs and code outlined here is per-thread code. This enables each thread to make unique, dynamic decisions regarding what kernel or operation to execute next. There are no synchronization requirements between threads within a block to execute any of the provided device runtime APIs, which enables the device runtime API functions to be called in arbitrarily divergent kernel code without deadlock.

13.6.2.1.1 Device-Side Kernel Launch (CDP1)

See *Kernel Launch APIs*, above, for CDP2 version of document.

Kernels may be launched from the device using the standard CUDA <<< >>> syntax:

```
kernel_name<<< Dg, Db, Ns, S >>>([kernel arguments]);
```

- `Dg` is of type `dim3` and specifies the dimensions and size of the grid
- `Db` is of type `dim3` and specifies the dimensions and size of each thread block
- `Ns` is of type `size_t` and specifies the number of bytes of shared memory that is dynamically allocated per thread block for this call and addition to statically allocated memory. `Ns` is an optional argument that defaults to 0.
- `S` is of type `cudaStream_t` and specifies the stream associated with this call. The stream must have been allocated in the same thread block where the call is being made. `S` is an optional argument that defaults to 0.

13.6.2.1.1.1 Launches are Asynchronous (CDP1)

See *Launches are Asynchronous*, above, for CDP2 version of document.

Identical to host-side launches, all device-side kernel launches are asynchronous with respect to the launching thread. That is to say, the <<<>>> launch command will return immediately and the launching thread will continue to execute until it hits an explicit launch-synchronization point such as `cudaDeviceSynchronize()`.

Warning: Explicit synchronization with child kernels from a parent block (i.e. using `cudaDeviceSynchronize()` in device code) is deprecated in CUDA 11.6, removed for compute_90+ compilation, and is slated for full removal in a future CUDA release.

The grid launch is posted to the device and will execute independently of the parent thread. The child grid may begin execution at any time after launch, but is not guaranteed to begin execution until the launching thread reaches an explicit launch-synchronization point.

13.6.2.1.1.2 Launch Environment Configuration (CDP1)

See *Launch Environment Configuration*, above, for CDP2 version of document.

All global device configuration settings (for example, shared memory and L1 cache size as returned from `cudaDeviceGetCacheConfig()`, and device limits returned from `cudaDeviceGetLimit()`) will be inherited from the parent. Likewise, device limits such as stack size will remain as-configured.

For host-launched kernels, per-kernel configurations set from the host will take precedence over the global setting. These configurations will be used when the kernel is launched from the device as well. It is not possible to reconfigure a kernel's environment from the device.

13.6.2.1.2 Streams (CDP1)

See *Streams*, above, for CDP2 version of document.

Both named and unnamed (NULL) streams are available from the device runtime. Named streams may be used by any thread within a thread-block, but stream handles may not be passed to other blocks or child/parent kernels. In other words, a stream should be treated as private to the block in which it is created. Stream handles are not guaranteed to be unique between blocks, so using a stream handle within a block that did not allocate it will result in undefined behavior.

Similar to host-side launch, work launched into separate streams may run concurrently, but actual concurrency is not guaranteed. Programs that depend upon concurrency between child kernels are not supported by the CUDA programming model and will have undefined behavior.

The host-side NULL stream's cross-stream barrier semantic is not supported on the device (see below for details). In order to retain semantic compatibility with the host runtime, all device streams must be created using the `cudaStreamCreateWithFlags()` API, passing the `cudaStreamNonBlocking` flag. The `cudaStreamCreate()` call is a host-runtime- only API and will fail to compile for the device.

As `cudaStreamSynchronize()` and `cudaStreamQuery()` are unsupported by the device runtime, `cudaDeviceSynchronize()` should be used instead when the application needs to know that stream-launched child kernels have completed.

Warning: Explicit synchronization with child kernels from a parent block (i.e. using `cudaDeviceSynchronize()` in device code) is deprecated in CUDA 11.6, removed for compute_90+ compilation, and is slated for full removal in a future CUDA release.

13.6.2.1.2.1 The Implicit (NULL) Stream (CDP1)

See *The Implicit (NULL) Stream*, above, for CDP2 version of document.

Within a host program, the unnamed (NULL) stream has additional barrier synchronization semantics with other streams (see *Default Stream* for details). The device runtime offers a single implicit, unnamed stream shared between all threads in a block, but as all named streams must be created with the `cudaStreamNonBlocking` flag, work launched into the NULL stream will not insert an implicit dependency on pending work in any other streams (including NULL streams of other thread blocks).

13.6.2.1.3 Events (CDP1)

See *Events*, above, for CDP2 version of document.

Only the inter-stream synchronization capabilities of CUDA events are supported. This means that `cudaStreamWaitEvent()` is supported, but `cudaEventSynchronize()`, `cudaEventElapsedTime()`, and `cudaEventQuery()` are not. As `cudaEventElapsedTime()` is not supported, cudaEvents must be created via `cudaEventCreateWithFlags()`, passing the `cudaEventDisableTiming` flag.

As for all device runtime objects, event objects may be shared between all threads within the thread-block which created them but are local to that block and may not be passed to other kernels, or between blocks within the same kernel. Event handles are not guaranteed to be unique between blocks, so using an event handle within a block that did not create it will result in undefined behavior.

13.6.2.1.4 Synchronization (CDP1)

See *Synchronization*, above, for CDP2 version of document.

> **Warning:** Explicit synchronization with child kernels from a parent block (i.e. using `cudaDeviceSynchronize()` in device code) is deprecated in CUDA 11.6, removed for compute_90+ compilation, and is slated for full removal in a future CUDA release.

The `cudaDeviceSynchronize()` function will synchronize on all work launched by any thread in the thread-block up to the point where `cudaDeviceSynchronize()` was called. Note that `cudaDeviceSynchronize()` may be called from within divergent code (see *Block Wide Synchronization (CDP1)*).

It is up to the program to perform sufficient additional inter-thread synchronization, for example via a call to `__syncthreads()`, if the calling thread is intended to synchronize with child grids invoked from other threads.

13.6.2.1.4.1 Block Wide Synchronization (CDP1)

See *CUDA Dynamic Parallelism*, above, for CDP2 version of document.

The `cudaDeviceSynchronize()` function does not imply intra-block synchronization. In particular, without explicit synchronization via a `__syncthreads()` directive the calling thread can make no assumptions about what work has been launched by any thread other than itself. For example if multiple threads within a block are each launching work and synchronization is desired for all this work at once (perhaps because of event-based dependencies), it is up to the program to guarantee that this work is submitted by all threads before calling `cudaDeviceSynchronize()`.

Because the implementation is permitted to synchronize on launches from any thread in the block, it is quite possible that simultaneous calls to `cudaDeviceSynchronize()` by multiple threads will drain all work in the first call and then have no effect for the later calls.

13.6.2.1.5 Device Management (CDP1)

See *Device Management*, above, for CDP2 version of document.

Only the device on which a kernel is running will be controllable from that kernel. This means that device APIs such as `cudaSetDevice()` are not supported by the device runtime. The active device as seen from the GPU (returned from `cudaGetDevice()`) will have the same device number as seen from the host system. The `cudaDeviceGetAttribute()` call may request information about another device as this API allows specification of a device ID as a parameter of the call. Note that the catch-all `cudaGetDeviceProperties()` API is not offered by the device runtime - properties must be queried individually.

13.6.2.1.6 Memory Declarations (CDP1)

See *Memory Declarations*, above, for CDP2 version of document.

13.6.2.1.6.1 Device and Constant Memory (CDP1)

See *Device and Constant Memory*, above, for CDP2 version of document.

Memory declared at file scope with `__device__` or `__constant__` memory space specifiers behaves identically when using the device runtime. All kernels may read or write device variables, whether the kernel was initially launched by the host or device runtime. Equivalently, all kernels will have the same view of `__constant__`s as declared at the module scope.

13.6.2.1.6.2 Textures and Surfaces (CDP1)

See *Textures and Surfaces*, above, for CDP2 version of document.

CUDA supports dynamically created texture and surface objects[Page 327, 7], where a texture object may be created on the host, passed to a kernel, used by that kernel, and then destroyed from the host. The device runtime does not allow creation or destruction of texture or surface objects from within device code, but texture and surface objects created from the host may be used and passed around freely on the device. Regardless of where they are created, dynamically created texture objects are always valid and may be passed to child kernels from a parent.

Note: The device runtime does not support legacy module-scope (i.e., Fermi-style) textures and surfaces within a kernel launched from the device. Module-scope (legacy) textures may be created from the host and used in device code as for any kernel, but may only be used by a top-level kernel (i.e., the one which is launched from the host).

13.6.2.1.6.3 Shared Memory Variable Declarations (CDP1)

See *Shared Memory Variable Declarations*, above, for CDP2 version of document.

In CUDA C++ shared memory can be declared either as a statically sized file-scope or function-scoped variable, or as an `extern` variable with the size determined at runtime by the kernel's caller via a launch configuration argument. Both types of declarations are valid under the device runtime.

```
__global__ void permute(int n, int *data) {
   extern __shared__ int smem[];
   if (n <= 1)
       return;

   smem[threadIdx.x] = data[threadIdx.x];
   __syncthreads();

   permute_data(smem, n);
   __syncthreads();

   // Write back to GMEM since we can't pass SMEM to children.
   data[threadIdx.x] = smem[threadIdx.x];
   __syncthreads();

   if (threadIdx.x == 0) {
       permute<<< 1, 256, n/2*sizeof(int) >>>(n/2, data);
       permute<<< 1, 256, n/2*sizeof(int) >>>(n/2, data+n/2);
   }
}

void host_launch(int *data) {
    permute<<< 1, 256, 256*sizeof(int) >>>(256, data);
}
```

13.6.2.1.6.4 Symbol Addresses (CDP1)

See *Symbol Addresses*, above, for CDP2 version of document.

Device-side symbols (i.e., those marked `__device__`) may be referenced from within a kernel simply via the & operator, as all global-scope device variables are in the kernel's visible address space. This also applies to `__constant__` symbols, although in this case the pointer will reference read-only data.

Given that device-side symbols can be referenced directly, those CUDA runtime APIs which reference symbols (for example, `cudaMemcpyToSymbol()` or `cudaGetSymbolAddress()`) are redundant and hence not supported by the device runtime. Note this implies that constant data cannot be altered from within a running kernel, even ahead of a child kernel launch, as references to `__constant__` space are read-only.

13.6.2.1.7 API Errors and Launch Failures (CDP1)

See *API Errors and Launch Failures*, above, for CDP2 version of document.

As usual for the CUDA runtime, any function may return an error code. The last error code returned is recorded and may be retrieved via the `cudaGetLastError()` call. Errors are recorded per-thread, so that each thread can identify the most recent error that it has generated. The error code is of type `cudaError_t`.

Similar to a host-side launch, device-side launches may fail for many reasons (invalid arguments, and so on). The user must call `cudaGetLastError()` to determine if a launch generated an error, however lack of an error after launch does not imply the child kernel completed successfully.

For device-side exceptions, for example, access to an invalid address, an error in a child grid will be returned to the host instead of being returned by the parent's call to `cudaDeviceSynchronize()`.

13.6.2.1.7.1 Launch Setup APIs (CDP1)

See *Launch Setup APIs*, above, for CDP2 version of document.

Kernel launch is a system-level mechanism exposed through the device runtime library, and as such is available directly from PTX via the underlying `cudaGetParameterBuffer()` and `cudaLaunchDevice()` APIs. It is permitted for a CUDA application to call these APIs itself, with the same requirements as for PTX. In both cases, the user is then responsible for correctly populating all necessary data structures in the correct format according to specification. Backwards compatibility is guaranteed in these data structures.

As with host-side launch, the device-side operator `<<<>>>` maps to underlying kernel launch APIs. This is so that users targeting PTX will be able to enact a launch, and so that the compiler front-end can translate `<<<>>>` into these calls.

Table 15: New Device-only Launch Implementation Functions

Runtime API Launch Functions	Description of Difference From Host Runtime Behaviour (behavior is identical if no description)
`cudaGetParameterBuffer`	Generated automatically from `<<<>>>`. Note different API to host equivalent.
`cudaLaunchDevice`	Generated automatically from `<<<>>>`. Note different API to host equivalent.

The APIs for these launch functions are different to those of the CUDA Runtime API, and are defined as follows:

```
extern   device   cudaError_t cudaGetParameterBuffer(void **params);
extern __device__ cudaError_t cudaLaunchDevice(void *kernel,
                                               void *params, dim3 gridDim,
                                               dim3 blockDim,
                                               unsigned int sharedMemSize = 0,
                                               cudaStream_t stream = 0);
```

13.6.2.1.8 API Reference (CDP1)

See *API Reference*, above, for CDP2 version of document.

The portions of the CUDA Runtime API supported in the device runtime are detailed here. Host and device runtime APIs have identical syntax; semantics are the same except where indicated. The table below provides an overview of the API relative to the version available from the host.

Table 16: Supported API Functions

Runtime API Functions	Details
`cudaDeviceSynchronize`	Synchronizes on work launched from thread's own block only. Warning: Note that calling this API from device code is deprecated in CUDA 11.6, removed for compute_90+ compilation, and is slated for full removal in a future CUDA release.
`cudaDeviceGetCacheConfig`	
`cudaDeviceGetLimit`	
`cudaGetLastError`	Last error is per-thread state, not per-block state
`cudaPeekAtLastError`	
`cudaGetErrorString`	
`cudaGetDeviceCount`	
`cudaDeviceGetAttribute`	Will return attributes for any device
`cudaGetDevice`	Always returns current device ID as would be seen from host
`cudaStreamCreateWithFlags`	Must pass `cudaStreamNonBlocking` flag
`cudaStreamDestroy`	
`cudaStreamWaitEvent`	
`cudaEventCreateWithFlags`	Must pass `cudaEventDisableTiming` flag
`cudaEventRecord`	
`cudaEventDestroy`	
`cudaFuncGetAttributes`	
`cudaMemcpyAsync`	Notes about all `memcpy/memset` functions: ► Only async `memcpy/set` functions are supported ► Only device-to-device `memcpy` is permitted ► May not pass in local or shared memory pointers
`cudaMemcpy2DAsync`	
`cudaMemcpy3DAsync`	
`cudaMemsetAsync`	
`cudaMemset2DAsync`	
`cudaMemset3DAsync`	
`cudaRuntimeGetVersion`	
`cudaMalloc`	May not call `cudaFree` on the device on a pointer created on the host, and vice-versa
`cudaFree`	
`cudaOccupancyMaxActiveBlocksPerMultiprocessor`	
`cudaOccupancyMaxPotentialBlockSize`	
`cudaOccupancyMaxPotentialBlockSizeVariableSMem`	

13.6.2.2 Device-side Launch from PTX (CDP1)

See *Device-side Launch from PTX*, above, for CDP2 version of document.

This section is for the programming language and compiler implementers who target *Parallel Thread Execution* (PTX) and plan to support *Dynamic Parallelism* in their language. It provides the low-level details related to supporting kernel launches at the PTX level.

13.6.2.2.1 Kernel Launch APIs (CDP1)

See *Kernel Launch APIs*, above, for CDP2 version of document.

Device-side kernel launches can be implemented using the following two APIs accessible from PTX: `cudaLaunchDevice()` and `cudaGetParameterBuffer()`. `cudaLaunchDevice()` launches the specified kernel with the parameter buffer that is obtained by calling `cudaGetParameterBuffer()` and filled with the parameters to the launched kernel. The parameter buffer can be NULL, i.e., no need to invoke `cudaGetParameterBuffer()`, if the launched kernel does not take any parameters.

13.6.2.2.1.1 cudaLaunchDevice (CDP1)

See *cudaLaunchDevice*, above, for CDP2 version of document.

At the PTX level, `cudaLaunchDevice()`needs to be declared in one of the two forms shown below before it is used.

```
// PTX-level Declaration of cudaLaunchDevice() when .address_size is 64
.extern .func(.param .b32 func_retval0) cudaLaunchDevice
(
  .param .b64 func,
  .param .b64 parameterBuffer,
  .param .align 4 .b8 gridDimension[12],
  .param .align 4 .b8 blockDimension[12],
  .param .b32 sharedMemSize,
  .param .b64 stream
)
;
```

```
// PTX-level Declaration of cudaLaunchDevice() when .address_size is 32
.extern .func(.param .b32 func_retval0) cudaLaunchDevice
(
  .param .b32 func,
  .param .b32 parameterBuffer,
  .param .align 4 .b8 gridDimension[12],
  .param .align 4 .b8 blockDimension[12],
  .param .b32 sharedMemSize,
  .param .b32 stream
)
;
```

The CUDA-level declaration below is mapped to one of the aforementioned PTX-level declarations and is found in the system header file `cuda_device_runtime_api.h`. The function is defined in the `cudadevrt` system library, which must be linked with a program in order to use device-side kernel launch functionality.

```
// CUDA-level declaration of cudaLaunchDevice()
extern "C" __device__
cudaError_t cudaLaunchDevice(void *func, void *parameterBuffer,
                             dim3 gridDimension, dim3 blockDimension,
                             unsigned int sharedMemSize,
                             cudaStream_t stream);
```

The first parameter is a pointer to the kernel to be is launched, and the second parameter is the parameter buffer that holds the actual parameters to the launched kernel. The layout of the parameter buffer is explained in *Parameter Buffer Layout (CDP1)*, below. Other parameters specify the launch configuration, i.e., as grid dimension, block dimension, shared memory size, and the stream associated with the launch (please refer to *Execution Configuration* for the detailed description of launch configuration.

13.6.2.2.1.2 cudaGetParameterBuffer (CDP1)

See *cudaGetParameterBuffer*, above, for CDP2 version of document.

`cudaGetParameterBuffer()` needs to be declared at the PTX level before it's used. The PTX-level declaration must be in one of the two forms given below, depending on address size:

```
// PTX-level Declaration of cudaGetParameterBuffer() when .address_size is 64
// When .address_size is 64
.extern .func(.param .b64 func_retval0) cudaGetParameterBuffer
(
  .param .b64 alignment,
  .param .b64 size
)
;
```

```
// PTX-level Declaration of cudaGetParameterBuffer() when .address_size is 32
.extern .func(.param .b32 func_retval0) cudaGetParameterBuffer
(
  .param .b32 alignment,
  .param .b32 size
)
;
```

The following CUDA-level declaration of `cudaGetParameterBuffer()` is mapped to the aforementioned PTX-level declaration:

```
// CUDA-level Declaration of cudaGetParameterBuffer()
extern "C" __device__
void *cudaGetParameterBuffer(size_t alignment, size_t size);
```

The first parameter specifies the alignment requirement of the parameter buffer and the second parameter the size requirement in bytes. In the current implementation, the parameter buffer returned by `cudaGetParameterBuffer()` is always guaranteed to be 64- byte aligned, and the alignment requirement parameter is ignored. However, it is recommended to pass the correct alignment requirement value - which is the largest alignment of any parameter to be placed in the parameter buffer - to `cudaGetParameterBuffer()` to ensure portability in the future.

13.6.2.2.2 Parameter Buffer Layout (CDP1)

See *Parameter Buffer Layout*, above, for CDP2 version of document.

Parameter reordering in the parameter buffer is prohibited, and each individual parameter placed in the parameter buffer is required to be aligned. That is, each parameter must be placed at the n^{th} byte in the parameter buffer, where n is the smallest multiple of the parameter size that is greater than the offset of the last byte taken by the preceding parameter. The maximum size of the parameter buffer is 4KB.

For a more detailed description of PTX code generated by the CUDA compiler, please refer to the PTX-3.5 specification.

13.6.2.3 Toolkit Support for Dynamic Parallelism (CDP1)

See *Toolkit Support for Dynamic Parallelism*, above, for CDP2 version of document.

13.6.2.3.1 Including Device Runtime API in CUDA Code (CDP1)

See *Including Device Runtime API in CUDA Code*, above, for CDP2 version of document.

Similar to the host-side runtime API, prototypes for the CUDA device runtime API are included automatically during program compilation. There is no need to include `cuda_device_runtime_api.h` explicitly.

13.6.2.3.2 Compiling and Linking (CDP1)

See *Compiling and Linking*, above, for CDP2 version of document.

When compiling and linking CUDA programs using dynamic parallelism with `nvcc`, the program will automatically link against the static device runtime library `libcudadevrt`.

The device runtime is offered as a static library (`cudadevrt.lib` on Windows, `libcudadevrt.a` under Linux), against which a GPU application that uses the device runtime must be linked. Linking of device libraries can be accomplished through `nvcc` and/or `nvlink`. Two simple examples are shown below.

A device runtime program may be compiled and linked in a single step, if all required source files can be specified from the command line:

```
$ nvcc -arch=sm_75 -rdc=true hello_world.cu -o hello -lcudadevrt
```

It is also possible to compile CUDA .cu source files first to object files, and then link these together in a two-stage process:

```
$ nvcc -arch=sm_75 -dc hello_world.cu -o hello_world.o
$ nvcc -arch=sm_75 -rdc=true hello_world.o -o hello -lcudadevrt
```

Please see the Using Separate Compilation section of The CUDA Driver Compiler NVCC guide for more details.

13.6.3. Programming Guidelines (CDP1)

See *Programming Guidelines*, above, for CDP2 version of document.

13.6.3.1 Basics (CDP1)

See *Basics*, above, for CDP2 version of document.

The device runtime is a functional subset of the host runtime. API level device management, kernel launching, device memcpy, stream management, and event management are exposed from the device runtime.

Programming for the device runtime should be familiar to someone who already has experience with CUDA. Device runtime syntax and semantics are largely the same as that of the host API, with any exceptions detailed earlier in this document.

Warning: Explicit synchronization with child kernels from a parent block (i.e. using `cudaDeviceSynchronize()` in device code) is deprecated in CUDA 11.6, removed for compute_90+ compilation, and is slated for full removal in a future CUDA release.

The following example shows a simple *Hello World* program incorporating dynamic parallelism:

```
#include <stdio.h>

__global__ void childKernel()
{
    printf("Hello ");
}

__global__ void parentKernel()
{
    // launch child
    childKernel<<<1,1>>>();
    if (cudaSuccess != cudaGetLastError()) {
        return;
    }

    // wait for child to complete
    if (cudaSuccess != cudaDeviceSynchronize()) {
        return;
    }

    printf("World!\n");
}

int main(int argc, char *argv[])
{
    // launch parent
    parentKernel<<<1,1>>>();
    if (cudaSuccess != cudaGetLastError()) {
        return 1;
    }

    // wait for parent to complete
```

(continues on next page)

(continued from previous page)

```
    if (cudaSuccess != cudaDeviceSynchronize()) {
        return 2;
    }

    return 0;
}
```

This program may be built in a single step from the command line as follows:

```
$ nvcc -arch=sm_75 -rdc=true hello_world.cu -o hello -lcudadevrt
```

13.6.3.2 Performance (CDP1)

See *Performance*, above, for CDP2 version of document.

13.6.3.2.1 Synchronization (CDP1)

See *CUDA Dynamic Parallelism*, above, for CDP2 version of document.

> **Warning:** Explicit synchronization with child kernels from a parent block (such as using `cudaDeviceSynchronize()` in device code) is deprecated in CUDA 11.6, removed for compute_90+ compilation, and is slated for full removal in a future CUDA release.

Synchronization by one thread may impact the performance of other threads in the same *Thread Block*, even when those other threads do not call `cudaDeviceSynchronize()` themselves. This impact will depend upon the underlying implementation. In general the implicit synchronization of child kernels done when a thread block ends is more efficient compared to calling `cudaDeviceSynchronize()` explicitly. It is therefore recommended to only call `cudaDeviceSynchronize()` if it is needed to synchronize with a child kernel before a thread block ends.

13.6.3.2.2 Dynamic-parallelism-enabled Kernel Overhead (CDP1)

See *Dynamic-parallelism-enabled Kernel Overhead*, above, for CDP2 version of document.

System software which is active when controlling dynamic launches may impose an overhead on any kernel which is running at the time, whether or not it invokes kernel launches of its own. This overhead arises from the device runtime's execution tracking and management software and may result in decreased performance for example, library calls when made from the device compared to from the host side. This overhead is, in general, incurred for applications that link against the device runtime library.

13.6.3.3 Implementation Restrictions and Limitations (CDP1)

See *Implementation Restrictions and Limitations*, above, for CDP2 version of document.

Dynamic Parallelism guarantees all semantics described in this document, however, certain hardware and software resources are implementation-dependent and limit the scale, performance and other properties of a program which uses the device runtime.

13.6.3.3.1 Runtime (CDP1)

See *Runtime*, above, for CDP2 version of document.

13.6.3.3.1.1 Memory Footprint (CDP1)

See *Memory Footprint*, above, for CDP2 version of document.

The device runtime system software reserves memory for various management purposes, in particular one reservation which is used for saving parent-grid state during synchronization, and a second reservation for tracking pending grid launches. Configuration controls are available to reduce the size of these reservations in exchange for certain launch limitations. See *Configuration Options (CDP1)*, below, for details.

The majority of reserved memory is allocated as backing-store for parent kernel state, for use when synchronizing on a child launch. Conservatively, this memory must support storing of state for the maximum number of live threads possible on the device. This means that each parent generation at which `cudaDeviceSynchronize()` is callable may require up to 860MB of device memory, depending on the device configuration, which will be unavailable for program use even if it is not all consumed.

13.6.3.3.1.2 Nesting and Synchronization Depth (CDP1)

See *CUDA Dynamic Parallelism*, above, for CDP2 version of document.

Using the device runtime, one kernel may launch another kernel, and that kernel may launch another, and so on. Each subordinate launch is considered a new *nesting level*, and the total number of levels is the *nesting depth* of the program. The *synchronization depth* is defined as the deepest level at which the program will explicitly synchronize on a child launch. Typically this is one less than the nesting depth of the program, but if the program does not need to call `cudaDeviceSynchronize()` at all levels then the synchronization depth might be substantially different to the nesting depth.

> **Warning:** Explicit synchronization with child kernels from a parent block (i.e. using `cudaDeviceSynchronize()` in device code) is deprecated in CUDA 11.6, removed for compute_90+ compilation, and is slated for full removal in a future CUDA release.

The overall maximum nesting depth is limited to 24, but practically speaking the real limit will be the amount of memory required by the system for each new level (see *Memory Footprint (CDP1)* above). Any launch which would result in a kernel at a deeper level than the maximum will fail. Note that this may also apply to `cudaMemcpyAsync()`, which might itself generate a kernel launch. See *Configuration Options (CDP1)* for details.

By default, sufficient storage is reserved for two levels of synchronization. This maximum synchronization depth (and hence reserved storage) may be controlled by calling `cudaDeviceSetLimit()`

and specifying `cudaLimitDevRuntimeSyncDepth`. The number of levels to be supported must be configured before the top-level kernel is launched from the host, in order to guarantee successful execution of a nested program. Calling `cudaDeviceSynchronize()` at a depth greater than the specified maximum synchronization depth will return an error.

An optimization is permitted where the system detects that it need not reserve space for the parent's state in cases where the parent kernel never calls `cudaDeviceSynchronize()`. In this case, because explicit parent/child synchronization never occurs, the memory footprint required for a program will be much less than the conservative maximum. Such a program could specify a shallower maximum synchronization depth to avoid over-allocation of backing store.

13.6.3.3.1.3 Pending Kernel Launches (CDP1)

See *Pending Kernel Launches*, above, for CDP2 version of document.

When a kernel is launched, all associated configuration and parameter data is tracked until the kernel completes. This data is stored within a system-managed launch pool.

The launch pool is divided into a fixed-size pool and a virtualized pool with lower performance. The device runtime system software will try to track launch data in the fixed-size pool first. The virtualized pool will be used to track new launches when the fixed-size pool is full.

The size of the fixed-size launch pool is configurable by calling `cudaDeviceSetLimit()` from the host and specifying `cudaLimitDevRuntimePendingLaunchCount`.

13.6.3.3.1.4 Configuration Options (CDP1)

See *Configuration Options*, above, for CDP2 version of document.

Resource allocation for the device runtime system software is controlled via the `cudaDeviceSetLimit()` API from the host program. Limits must be set before any kernel is launched, and may not be changed while the GPU is actively running programs.

> **Warning:** Explicit synchronization with child kernels from a parent block (i.e. using `cudaDeviceSynchronize()` in device code) is deprecated in CUDA 11.6, removed for compute_90+ compilation, and is slated for full removal in a future CUDA release.

The following named limits may be set:

Limit	Behavior
`cudaLimitDevRuntimeSyncDepth`	Sets the maximum depth at which `cudaDeviceSynchronize()` may be called. Launches may be performed deeper than this, but explicit synchronization deeper than this limit will return the `cudaErrorLaunchMaxDepthExceeded`. The default maximum sync depth is 2.
`cudaLimitDevRuntimePendingLaunchCount`	Controls the amount of memory set aside for buffering kernel launches which have not yet begun to execute, due either to unresolved dependencies or lack of execution resources. When the buffer is full, the device runtime system software will attempt to track new pending launches in a lower performance virtualized buffer. If the virtualized buffer is also full, i.e. when all available heap space is consumed, launches will not occur, and the thread's last error will be set to `cudaErrorLaunchPendingCountExceeded`. The default pending launch count is 2048 launches.
`cudaLimitStackSize`	Controls the stack size in bytes of each GPU thread. The CUDA driver automatically increases the per-thread stack size for each kernel launch as needed. This size isn't reset back to the original value after each launch. To set the per-thread stack size to a different value, `cudaDeviceSetLimit()` can be called to set this limit. The stack will be immediately resized, and if necessary, the device will block until all preceding requested tasks are complete. `cudaDeviceGetLimit()` can be called to get the current per-thread stack size.

13.6.3.3.1.5 Memory Allocation and Lifetime (CDP1)

See *Memory Allocation and Lifetime*, above, for CDP2 version of document.

`cudaMalloc()` and `cudaFree()` have distinct semantics between the host and device environments. When invoked from the host, `cudaMalloc()` allocates a new region from unused device memory. When invoked from the device runtime these functions map to device-side `malloc()` and `free()`. This implies that within the device environment the total allocatable memory is limited to the device `malloc()` heap size, which may be smaller than the available unused device memory. Also, it is an error to invoke `cudaFree()` from the host program on a pointer which was allocated by `cudaMalloc()` on the device or vice-versa.

	`cudaMalloc()` on Host	**`cudaMalloc()` on Device**
`cudaFree()` on Host	Supported	Not Supported
`cudaFree()` on Device	Not Supported	Supported
Allocation limit	Free device memory	`cudaLimitMallocHeapSize`

13.6.3.3.1.6 SM Id and Warp Id (CDP1)

See *SM Id and Warp Id*, above, for CDP2 version of document.

Note that in PTX `%smid` and `%warpid` are defined as volatile values. The device runtime may reschedule thread blocks onto different SMs in order to more efficiently manage resources. As such, it is unsafe to rely upon `%smid` or `%warpid` remaining unchanged across the lifetime of a thread or thread block.

13.6.3.3.1.7 ECC Errors (CDP1)

See *ECC Errors*, above, for CDP2 version of document.

No notification of ECC errors is available to code within a CUDA kernel. ECC errors are reported at the host side once the entire launch tree has completed. Any ECC errors which arise during execution of a nested program will either generate an exception or continue execution (depending upon error and configuration).

Chapter 14. Virtual Memory Management

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

14.1. Introduction

The Virtual Memory Management APIs provide a way for the application to directly manage the unified virtual address space that CUDA provides to map physical memory to virtual addresses accessible by the GPU. Introduced in CUDA 10.2, these APIs additionally provide a new way to interop with other processes and graphics APIs like OpenGL and Vulkan, as well as provide newer memory attributes that a user can tune to fit their applications.

Historically, memory allocation calls (such as `cudaMalloc()`) in the CUDA programming model have returned a memory address that points to the GPU memory. The address thus obtained could be used with any CUDA API or inside a device kernel. However, the memory allocated could not be resized depending on the user's memory needs. In order to increase an allocation's size, the user had to explicitly allocate a larger buffer, copy data from the initial allocation, free it and then continue to keep track of the newer allocation's address. This often leads to lower performance and higher peak memory utilization for applications. Essentially, users had a malloc-like interface for allocating GPU memory, but did not have a corresponding realloc to complement it. The Virtual Memory Management APIs decouple the idea of an address and memory and allow the application to handle them separately. The APIs allow applications to map and unmap memory from a virtual address range as they see fit.

In the case of enabling peer device access to memory allocations by using `cudaEnablePeerAccess`, all past and future user allocations are mapped to the target peer device. This lead to users unwittingly paying runtime cost of mapping all cudaMalloc allocations to peer devices. However, in most situations applications communicate by sharing only a few allocations with another device and not all allocations are required to be mapped to all the devices. With Virtual Memory Management, applications can specifically choose certain allocations to be accessible from target devices.

The CUDA Virtual Memory Management APIs expose fine grained control to the user for managing the GPU memory in applications. It provides APIs that let users:

- Place memory allocated on different devices into a contiguous VA range.
- Perform interprocess communication for memory sharing using platform-specific mechanisms.
- Opt into newer memory types on the devices that support them.

In order to allocate memory, the Virtual Memory Management programming model exposes the following functionality:

- Allocating physical memory.
- Reserving a VA range.
- Mapping allocated memory to the VA range.
- Controlling access rights on the mapped range.

Note that the suite of APIs described in this section require a system that supports UVA.

14.2. Query for Support

Before attempting to use Virtual Memory Management APIs, applications must ensure that the devices they want to use support CUDA Virtual Memory Management. The following code sample shows querying for Virtual Memory Management support:

```
int deviceSupportsVmm;
CUresult result = cuDeviceGetAttribute(&deviceSupportsVmm, CU_DEVICE_ATTRIBUTE_
↪VIRTUAL_MEMORY_MANAGEMENT_SUPPORTED, device);
if (deviceSupportsVmm != 0) {
    // `device` supports Virtual Memory Management
}
```

14.3. Allocating Physical Memory

The first step in memory allocation using Virtual Memory Management APIs is to create a physical memory chunk that will provide a backing for the allocation. In order to allocate physical memory, applications must use the `cuMemCreate` API. The allocation created by this function does not have any device or host mappings. The function argument `CUmemGenericAllocationHandle` describes the properties of the memory to allocate such as the location of the allocation, if the allocation is going to be shared to another process (or other Graphics APIs), or the physical attributes of the memory to be allocated. Users must ensure the requested allocation's size must be aligned to appropriate granularity. Information regarding an allocation's granularity requirements can be queried using `cuMemGetAllocationGranularity`. The following code snippet shows allocating physical memory with `cuMemCreate`:

```
CUmemGenericAllocationHandle allocatePhysicalMemory(int device, size_t size) {
    CUmemAllocationProp prop = {};
    prop.type = CU_MEM_ALLOCATION_TYPE_PINNED;
    prop.location.type = CU_MEM_LOCATION_TYPE_DEVICE;
    prop.location.id = device;

    size_t granularity = 0;
    cuMemGetAllocationGranularity(&granularity, &prop, CU_MEM_ALLOC_GRANULARITY_
↪MINIMUM);

    // Ensure size matches granularity requirements for the allocation
    size_t padded_size = ROUND_UP(size, granularity);
```

(continues on next page)

(continued from previous page)

```
    // Allocate physical memory
    CUmemGenericAllocationHandle allocHandle;
    cuMemCreate(&allocHandle, padded_size, &prop, 0);

    return allocHandle;
}
```

The memory allocated by `cuMemCreate` is referenced by the `CUmemGenericAllocationHandle` it returns. This is a departure from the cudaMalloc-style of allocation, which returns a pointer to the GPU memory, which was directly accessible by CUDA kernel executing on the device. The memory allocated cannot be used for any operations other than querying properties using `cuMemGetAllocationPropertiesFromHandle`. In order to make this memory accessible, applications must map this memory into a VA range reserved by `cuMemAddressReserve` and provide suitable access rights to it. Applications must free the allocated memory using the `cuMemRelease` API.

14.3.1. Shareable Memory Allocations

With `cuMemCreate` users now have the facility to indicate to CUDA, at allocation time, that they have earmarked a particular allocation for Inter process communication and graphics interop purposes. Applications can do this by setting `CUmemAllocationProp::requestedHandleTypes` to a platform-specific field. On Windows, when `CUmemAllocationProp::requestedHandleTypes` is set to `CU_MEM_HANDLE_TYPE_WIN32` applications must also specify an LPSECURITYATTRIBUTES attribute in `CUmemAllocationProp::win32HandleMetaData`. This security attribute defines the scope of which exported allocations may be transferred to other processes.

The CUDA Virtual Memory Management API functions do not support the legacy interprocess communication functions with their memory. Instead, they expose a new mechanism for interprocess communication that uses OS-specific handles. Applications can obtain these OS-specific handles corresponding to the allocations by using `cuMemExportToShareableHandle`. The handles thus obtained can be transferred by using the usual OS native mechanisms for inter process communication. The recipient process should import the allocation by using `cuMemImportFromShareableHandle`.

Users must ensure they query for support of the requested handle type before attempting to export memory allocated with `cuMemCreate`. The following code snippet illustrates query for handle type support in a platform-specific way.

```
int deviceSupportsIpcHandle;
#if defined(__linux__)
    cuDeviceGetAttribute(&deviceSupportsIpcHandle, CU_DEVICE_ATTRIBUTE_HANDLE_TYPE_
↪POSIX_FILE_DESCRIPTOR_SUPPORTED, device));
#else
    cuDeviceGetAttribute(&deviceSupportsIpcHandle, CU_DEVICE_ATTRIBUTE_HANDLE_TYPE_
↪WIN32_HANDLE_SUPPORTED, device));
#endif
```

Users should set the `CUmemAllocationProp::requestedHandleTypes` appropriately as shown below:

```
#if defined(__linux__)
    prop.requestedHandleTypes = CU_MEM_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR;
#else
    prop.requestedHandleTypes = CU_MEM_HANDLE_TYPE_WIN32;
```

(continues on next page)

(continued from previous page)

```
    prop.win32HandleMetaData = // Windows specific LPSECURITYATTRIBUTES attribute.
#endif
```

The memMapIpcDrv sample can be used as an example for using IPC with Virtual Memory Management allocations.

14.3.2. Memory Type

Before CUDA 10.2, applications had no user-controlled way of allocating any special type of memory that certain devices may support. With `cuMemCreate`, applications can additionally specify memory type requirements using the `CUmemAllocationProp::allocFlags` to opt into any specific memory features. Applications must also ensure that the requested memory type is supported on the device of allocation.

14.3.2.1 Compressible Memory

Compressible memory can be used to accelerate accesses to data with unstructured sparsity and other compressible data patterns. Compression can save DRAM bandwidth, L2 read bandwidth and L2 capacity depending on the data being operated on. Applications that want to allocate compressible memory on devices that support Compute Data Compression can do so by setting `CUmemAllocationProp::allocFlags::compressionType` to `CU_MEM_ALLOCATION_COMP_GENERIC`. Users must query if device supports Compute Data Compression by using `CU_DEVICE_ATTRIBUTE_GENERIC_COMPRESSION_SUPPORTED`. The following code snippet illustrates querying compressible memory support `cuDeviceGetAttribute`.

```
int compressionSupported = 0;
cuDeviceGetAttribute(&compressionSupported, CU_DEVICE_ATTRIBUTE_GENERIC_COMPRESSION_
↪SUPPORTED, device);
```

On devices that support Compute Data Compression, users must opt in at allocation time as shown below:

```
prop.allocFlags.compressionType = CU_MEM_ALLOCATION_COMP_GENERIC;
```

Due to various reasons such as limited HW resources, the allocation may not have compression attributes, the user is expected to query back the properties of the allocated memory using `cuMemGetAllocationPropertiesFromHandle` and check for compression attribute.

```
CUmemAllocationProp allocationProp = {};
cuMemGetAllocationPropertiesFromHandle(&allocationProp, allocationHandle);

if (allocationProp.allocFlags.compressionType == CU_MEM_ALLOCATION_COMP_GENERIC)
{
    // Obtained compressible memory allocation
}
```

14.4. Reserving a Virtual Address Range

Since with Virtual Memory Management the notions of address and memory are distinct, applications must carve out an address range that can hold the memory allocations made by `cuMemCreate`. The address range reserved must be at least as large as the sum of the sizes of all the physical memory allocations the user plans to place in them.

Applications can reserve a virtual address range by passing appropriate parameters to `cuMemAddressReserve`. The address range obtained will not have any device or host physical memory associated with it. The reserved virtual address range can be mapped to memory chunks belonging to any device in the system, thus providing the application a continuous VA range backed and mapped by memory belonging to different devices. Applications are expected to return the virtual address range back to CUDA using `cuMemAddressFree`. Users must ensure that the entire VA range is unmapped before calling `cuMemAddressFree`. These functions are conceptually similar to mmap/munmap (on Linux) or VirtualAlloc/VirtualFree (on Windows) functions. The following code snippet illustrates the usage for the function:

```
CUdeviceptr ptr;
// `ptr` holds the returned start of virtual address range reserved.
CUresult result = cuMemAddressReserve(&ptr, size, 0, 0, 0); // alignment = 0 for
↪default alignment
```

14.5. Virtual Aliasing Support

The Virtual Memory Management APIs provide a way to create multiple virtual memory mappings or "proxies" to the same allocation using multiple calls to `cuMemMap` with different virtual addresses, so-called virtual aliasing. Unless otherwise noted in the PTX ISA, writes to one proxy of the allocation are considered inconsistent and incoherent with any other proxy of the same memory until the writing device operation (grid launch, memcpy, memset, and so on) completes. Grids present on the GPU prior to a writing device operation but reading after the writing device operation completes are also considered to have inconsistent and incoherent proxies.

For example, the following snippet is considered undefined, assuming device pointers A and B are virtual aliases of the same memory allocation:

```
__global__ void foo(char *A, char *B) {
  *A = 0x1;
  printf("%d\n", *B);    // Undefined behavior!  *B can take on either
// the previous value or some value in-between.
}
```

The following is defined behavior, assuming these two kernels are ordered monotonically (by streams or events).

```
__global__ void foo1(char *A) {
  *A = 0x1;
}

__global__ void foo2(char *B) {
  printf("%d\n", *B);    // *B == *A == 0x1 assuming foo2 waits for foo1
// to complete before launching
```

(continues on next page)

(continued from previous page)

```
}

cudaMemcpyAsync(B, input, size, stream1);    // Aliases are allowed at
// operation boundaries
foo1<<<1,1,0,stream1>>>(A);                  // allowing foo1 to access A.
cudaEventRecord(event, stream1);
cudaStreamWaitEvent(stream2, event);
foo2<<<1,1,0,stream2>>>(B);
cudaStreamWaitEvent(stream3, event);
cudaMemcpyAsync(output, B, size, stream3);  // Both launches of foo2 and
                                            // cudaMemcpy (which both
                                            // read) wait for foo1 (which writes)
                                            // to complete before proceeding
```

If accessing same allocation through different “proxies” is required in the same kernel a `fence.proxy.alias` can be used between the two accesses. The above example can thus be made legal with inline PTX assembly:

```
__global__ void foo(char *A, char *B) {
  *A = 0x1;
  asm volatile ("fence.proxy.alias;" ::: "memory");
  printf("%d\n", *B);    // *B == *A == 0x1
}
```

14.6. Mapping Memory

The allocated physical memory and the carved out virtual address space from the previous two sections represent the memory and address distinction introduced by the Virtual Memory Management APIs. For the allocated memory to be useable, the user must first place the memory in the address space. The address range obtained from `cuMemAddressReserve` and the physical allocation obtained from `cuMemCreate` or `cuMemImportFromShareableHandle` must be associated with each other by using `cuMemMap`.

Users can associate allocations from multiple devices to reside in contiguous virtual address ranges as long as they have carved out enough address space. In order to decouple the physical allocation and the address range, users must unmap the address of the mapping by using `cuMemUnmap`. Users can map and unmap memory to the same address range as many times as they want, as long as they ensure that they don’t attempt to create mappings on VA range reservations that are already mapped. The following code snippet illustrates the usage for the function:

```
CUdeviceptr ptr;
// `ptr`: address in the address range previously reserved by cuMemAddressReserve.
// `allocHandle`: CUmemGenericAllocationHandle obtained by a previous call to
↪cuMemCreate.
CUresult result = cuMemMap(ptr, size, 0, allocHandle, 0);
```

14.7. Controlling Access Rights

The Virtual Memory Management APIs enable applications to explicitly protect their VA ranges with access control mechanisms. Mapping the allocation to a region of the address range using `cuMemMap` does not make the address accessible, and would result in a program crash if accessed by a CUDA kernel. Users must specifically select access control using the `cuMemSetAccess` function, which allows or restricts access for specific devices to a mapped address range. The following code snippet illustrates the usage for the function:

```
void setAccessOnDevice(int device, CUdeviceptr ptr, size_t size) {
    CUmemAccessDesc accessDesc = {};
    accessDesc.location.type = CU_MEM_LOCATION_TYPE_DEVICE;
    accessDesc.location.id = device;
    accessDesc.flags = CU_MEM_ACCESS_FLAGS_PROT_READWRITE;

    // Make the address accessible
    cuMemSetAccess(ptr, size, &accessDesc, 1);
}
```

The access control mechanism exposed with Virtual Memory Management allows users to be explicit about which allocations they want to share with other peer devices on the system. As specified earlier, `cudaEnablePeerAccess` forces all prior and future cudaMalloc'd allocations to be mapped to the target peer device. This can be convenient in many cases as user doesn't have to worry about tracking the mapping state of every allocation to every device in the system. But for users concerned with performance of their applications this approach has performance implications. With access control at allocation granularity Virtual Memory Management exposes a mechanism to have peer mappings with minimal overhead.

The `vectorAddMMAP` sample can be used as an example for using the Virtual Memory Management APIs.

14.8. Fabric Memory

CUDA 12.4 introduced a new VMM allocation handle type `CU_MEM_HANDLE_TYPE_FABRIC`. On supported platforms and provided the NVIDIA IMEX daemon is running this allocation handle type enables sharing allocations not only intra node with any communication mechanism, e.g. MPI, but also inter node. This allows GPUs in a Multi Node NVLINK System to map the memory of all other GPUs part of the same NVLINK fabric even if they are in different nodes greatly increasing the scale of multi-GPU Programming with NVLINK.

14.8.1. Query for Support

Before attempting to use Fabric Memory, applications must ensure that the devices they want to use support Fabric Memory. The following code sample shows querying for Fabric Memory support:

```
int deviceSupportsFabricMem;
CUresult result = cuDeviceGetAttribute(&deviceSupportsFabricMem, CU_DEVICE_ATTRIBUTE_
↪HANDLE_TYPE_FABRIC_SUPPORTED, device);
if (deviceSupportsFabricMem != 0) {
```

(continues on next page)

(continued from previous page)

```
    // `device` supports Fabric Memory
}
```

Aside from using `CU_MEM_HANDLE_TYPE_FABRIC` as handle type and not requiring OS native mechanisms for inter process communication to exchange sharable handles there is no difference in using Fabric Memory compared to other allocation handle types.

14.9. Multicast Support

The Multicast Object Management APIs provide a way for the application to create Multicast Objects and in combination with the Virtual Memory Management APIs described above allow applications to leverage NVLINK SHARP on supported NVLINK connected GPUs if they are connected with NVSWITCH. NVLINK SHARP allows CUDA applications to leverage in fabric computing to accelerate operations like broadcast and reductions between GPUs connected with NVSWITCH. For this to work multiple NVLINK connected GPUs form a Multicast Team and each GPU from the team backs up a Multicast Object with physical memory. So a Multicast Team of N GPUs has N physical replicas, each local to one participating GPU, of a Multicast Object. The multimem PTX instructions using mappings of Multicast Objects work with all replicas of the Multicast Object.

To work with Multicast Objects an application needs to

- Query Multicast Support
- Create a Multicast Handle with `cuMulticastCreate`.
- Share the Multicast Handle with all processes that control a GPU which should participate in a Multicast Team. This works with `cuMemExportToShareableHandle` as described above.
- Add all GPUs that should participate in the Multicast Team with `cuMulticastAddDevice`.
- For each participating GPU bind physical memory allocated with `cuMemCreate` as described above to the Multicast Handle. All devices need to be added to the Multicast Team before binding memory on any device.
- Reserve an address range, map the Multicast Handle and set Access Rights as described above for regular Unicast mappings. Unicast and Multicast mappings to the same physical memory are possible. See the *Virtual Aliasing Support* section above how to ensure consistency between multiple mappings to the same physical memory.
- Use the multimem PTX instructions with the multicast mappings.

The `multi_node_p2p` example in the Multi GPU Programming Models GitHub repository contains a complete example using Fabric Memory including Multicast Objects to leverage NVLINK SHARP. Please note that this example is for developers of libraries like NCCL or NVSHMEM. It shows how higher-level programming models like NVSHMEM work internally within a (multinode) NVLINK domain. Application developers generally should use the higher-level MPI, NCCL, or NVSHMEM interfaces instead of this API.

14.9.1. Query for Support

Before attempting to use Multicast Objects, applications must ensure that the devices they want to use support them. The following code sample shows querying for Fabric Memory support:

```
int deviceSupportsMultiCast;
CUresult result = cuDeviceGetAttribute(&deviceSupportsMultiCast, CU_DEVICE_ATTRIBUTE_
↪MULTICAST_SUPPORTED, device);
if (deviceSupportsMultiCast != 0) {
    // `device` supports Multicast Objects
}
```

14.9.2. Allocating Multicast Objects

Multicast Objects can be created with `cuMulticastCreate`:

```
CUmemGenericAllocationHandle createMCHandle(int numDevices, size_t size) {
    CUmemAllocationProp mcProp = {};
    mcProp.numDevices = numDevices;
    mcProp.handleTypes = CU_MEM_HANDLE_TYPE_FABRIC; // or on single node CU_MEM_HANDLE_
↪TYPE_POSIX_FILE_DESCRIPTOR

    size_t granularity = 0;
    cuMulticastGetGranularity(&granularity, &mcProp, CU_MEM_ALLOC_GRANULARITY_
↪MINIMUM);

    // Ensure size matches granularity requirements for the allocation
    size_t padded_size = ROUND_UP(size, granularity);

    mcProp.size = padded_size;

    // Create Multicast Object this has no devices and no physical memory associated yet
    CUmemGenericAllocationHandle mcHandle;
    cuMulticastCreate(&mcHandle, &mcProp);

    return mcHandle;
}
```

14.9.3. Add Devices to Multicast Objects

Devices can be added to a Multicast Team with `cuMulticastAddDevice`:

```
cuMulticastAddDevice(&mcHandle, device);
```

This step needs to be completed on all processes controlling devices that should participate in a Multicast Team before memory on any device is bound to the Multicast Object.

14.9.4. Bind Memory to Multicast Objects

After a Multicast Object has been created and all participating devices have been added to the Multicast Object it needs to be backed with physical memory allocated with `cuMemCreate` for each device:

```
cuMulticastBindMem(mcHandle, mcOffset, memHandle, memOffset, size, 0 /*flags*/);
```

14.9.5. Use Multicast Mappings

To use Multicast Mappings in CUDA C++ it is required to use the multimem PTX instructions with Inline PTX Assembly:

```
__global__ void all_reduce_norm_barrier_kernel(float* l2_norm,
                                               float* partial_l2_norm_mc,
                                               unsigned int* arrival_counter_uc,
↪unsigned int* arrival_counter_mc,
                                               const unsigned int expected_count) {
    assert( 1 == blockDim.x * blockDim.y * blockDim.z * gridDim.x * gridDim.y *
↪gridDim.z );
    float l2_norm_sum = 0.0;
#if __CUDA_ARCH__ >= 900

    // atomic reduction to all replicas
    // this can be conceptually thought of as __threadfence_system(); atomicAdd_
↪system(arrival_counter_mc, 1);
    asm volatile ("multimem.red.release.sys.global.add.u32 [%0], %1;" :: "l"(arrival_
↪counter_mc), "n"(1) : "memory");

    // Need a fence between Multicast (mc) and Unicast (uc) access to the same memory
↪`arrival_counter_uc` and `arrival_counter_mc`:
    // - fence.proxy instructions establish an ordering between memory accesses that
↪may happen through different proxies
    // - Value .alias of the .proxykind qualifier refers to memory accesses performed
↪using virtually aliased addresses to the same memory location.
    // from https://docs.nvidia.com/cuda/parallel-thread-execution/#parallel-
↪synchronization-and-communication-instructions-membar
    asm volatile ("fence.proxy.alias;" ::: "memory");

    // spin wait with acquire ordering on UC mapping till all peers have arrived in
↪this iteration
    // Note: all ranks need to reach another barrier after this kernel, such that it is
↪not possible for the barrier to be unblocked by an
    // arrival of a rank for the next iteration if some other rank is slow.
    cuda::atomic_ref<unsigned int,cuda::thread_scope_system> ac(arrival_counter_uc);
    while (expected_count > ac.load(cuda::memory_order_acquire));

    // Atomic load reduction from all replicas. It does not provide ordering so it can
↪be relaxed.
    asm volatile ("multimem.ld_reduce.relaxed.sys.global.add.f32 %0, [%1];" : "=f"(l2_
↪norm_sum) : "l"(partial_l2_norm_mc) : "memory");

#else
    #error "ERROR: multimem instructions require compute capability 9.0 or larger."
#endif
```

(continues on next page)

(continued from previous page)

```
    *l2_norm = std::sqrt(l2_norm_sum);
}
```

Chapter 15. Stream Ordered Memory Allocator

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

15.1. Introduction

Managing memory allocations using `cudaMalloc` and `cudaFree` causes GPU to synchronize across all executing CUDA streams. The Stream Order Memory Allocator enables applications to order memory allocation and deallocation with other work launched into a CUDA stream such as kernel launches and asynchronous copies. This improves application memory use by taking advantage of stream-ordering semantics to reuse memory allocations. The allocator also allows applications to control the allocator's memory caching behavior. When set up with an appropriate release threshold, the caching behavior allows the allocator to avoid expensive calls into the OS when the application indicates it is willing to accept a bigger memory footprint. The allocator also supports the easy and secure sharing of allocations between processes.

For many applications, the Stream Ordered Memory Allocator reduces the need for custom memory management abstractions, and makes it easier to create high-performance custom memory management for applications that need it. For applications and libraries that already have custom memory allocators, adopting the Stream Ordered Memory Allocator enables multiple libraries to share a common pool of memory managed by the driver, thus reducing excess memory consumption. Additionally, the driver can perform optimizations based on its awareness of the allocator and other stream management APIs. Finally, Nsight Compute and the Next-Gen CUDA debugger is aware of the allocator as part of their CUDA 11.3 toolkit support.

15.2. Query for Support

The user can determine whether or not a device supports the stream ordered memory allocator by calling `cudaDeviceGetAttribute()` with the device attribute `cudaDevAttrMemoryPoolsSupported`.

Starting with CUDA 11.3, IPC memory pool support can be queried with the `cudaDevAttrMemoryPoolSupportedHandleTypes` device attribute. Previous drivers will return `cudaErrorInvalidValue` as those drivers are unaware of the attribute enum.

```
int driverVersion = 0;
int deviceSupportsMemoryPools = 0;
int poolSupportedHandleTypes = 0;
cudaDriverGetVersion(&driverVersion);
if (driverVersion >= 11020) {
    cudaDeviceGetAttribute(&deviceSupportsMemoryPools,
                           cudaDevAttrMemoryPoolsSupported, device);
}
if (deviceSupportsMemoryPools != 0) {
    // `device` supports the Stream Ordered Memory Allocator
}

if (driverVersion >= 11030) {
    cudaDeviceGetAttribute(&poolSupportedHandleTypes,
               cudaDevAttrMemoryPoolSupportedHandleTypes, device);
}
if (poolSupportedHandleTypes & cudaMemHandleTypePosixFileDescriptor) {
   // Pools on the specified device can be created with posix file descriptor-based IPC
}
```

Performing the driver version check before the query avoids hitting a `cudaErrorInvalidValue` error on drivers where the attribute was not yet defined. One can use `cudaGetLastError` to clear the error instead of avoiding it.

15.3. API Fundamentals (cudaMallocAsync and cudaFreeAsync)

The APIs `cudaMallocAsync` and `cudaFreeAsync` form the core of the allocator. `cudaMallocAsync` returns an allocation and `cudaFreeAsync` frees an allocation. Both APIs accept stream arguments to define when the allocation will become and stop being available for use. The pointer value returned by `cudaMallocAsync` is determined synchronously and is available for constructing future work. It is important to note that `cudaMallocAsync` ignores the current device/context when determining where the allocation will reside. Instead, `cudaMallocAsync` determines the resident device based on the specified memory pool or the supplied stream. The simplest use pattern is when the memory is allocated, used, and freed back into the same stream.

```
void *ptr;
size_t size = 512;
cudaMallocAsync(&ptr, size, cudaStreamPerThread);
// do work using the allocation
kernel<<<..., cudaStreamPerThread>>>(ptr, ...);
```

(continues on next page)

(continued from previous page)

```
// An asynchronous free can be specified without synchronizing the cpu and GPU
cudaFreeAsync(ptr, cudaStreamPerThread);
```

When using an allocation in a stream other than the allocating stream, the user must guarantee that the access will happen after the allocation operation, otherwise the behavior is undefined. The user may make this guarantee either by synchronizing the allocating stream, or by using CUDA events to synchronize the producing and consuming streams.

`cudaFreeAsync()` inserts a free operation into the stream. The user must guarantee that the free operation happens after the allocation operation and any use of the allocation. Also, any use of the allocation after the free operation starts results in undefined behavior. Events and/or stream synchronizing operations should be used to guarantee any access to the allocation on other streams is complete before the freeing stream begins the free operation.

```
cudaMallocAsync(&ptr, size, stream1);
cudaEventRecord(event1, stream1);
//stream2 must wait for the allocation to be ready before accessing
cudaStreamWaitEvent(stream2, event1);
kernel<<<..., stream2>>>(ptr, ...);
cudaEventRecord(event2, stream2);
// stream3 must wait for stream2 to finish accessing the allocation before
// freeing the allocation
cudaStreamWaitEvent(stream3, event2);
cudaFreeAsync(ptr, stream3);
```

The user can free allocations allocated with `cudaMalloc()` with `cudaFreeAsync()`. The user must make the same guarantees about accesses being complete before the free operation begins.

```
cudaMalloc(&ptr, size);
kernel<<<..., stream>>>(ptr, ...);
cudaFreeAsync(ptr, stream);
```

The user can free memory allocated with `cudaMallocAsync` with `cudaFree()`. When freeing such allocations through the `cudaFree()` API, the driver assumes that all accesses to the allocation are complete and performs no further synchronization. The user can use `cudaStreamQuery` / `cudaStreamSynchronize` / `cudaEventQuery` / `cudaEventSynchronize` / `cudaDeviceSynchronize` to guarantee that the appropriate asynchronous work is complete and that the GPU will not try to access the allocation.

```
cudaMallocAsync(&ptr, size,stream);
kernel<<<..., stream>>>(ptr, ...);
// synchronize is needed to avoid prematurely freeing the memory
cudaStreamSynchronize(stream);
cudaFree(ptr);
```

15.4. Memory Pools and the cudaMemPool_t

Memory pools encapsulate virtual address and physical memory resources that are allocated and managed according to the pools attributes and properties. The primary aspect of a memory pool is the kind and location of memory it manages.

All calls to `cudaMallocAsync` use the resources of a memory pool. In the absence of a specified memory pool, `cudaMallocAsync` uses the current memory pool of the supplied stream's device. The current memory pool for a device may be set with `cudaDeviceSetMempool` and queried with `cudaDeviceGetMempool`. By default (in the absence of a `cudaDeviceSetMempool` call), the current memory pool is the default memory pool of a device. The API `cudaMallocFromPoolAsync` and c++ overloads of cudaMallocAsync allow a user to specify the pool to be used for an allocation without setting it as the current pool. The APIs `cudaDeviceGetDefaultMempool` and `cudaMemPoolCreate` give users handles to memory pools.

Note: The mempool current to a device will be local to that device. So allocating without specifying a memory pool will always yield an allocation local to the stream's device.

Note: `cudaMemPoolSetAttribute` and `cudaMemPoolGetAttribute` control the attributes of the memory pools.

15.5. Default/Implicit Pools

The default memory pool of a device may be retrieved with the `cudaDeviceGetDefaultMempool` API. Allocations from the default memory pool of a device are non-migratable device allocation located on that device. These allocations will always be accessible from that device. The accessibility of the default memory pool may be modified with `cudaMemPoolSetAccess` and queried by `cudaMemPoolGetAccess`. Since the default pools do not need to be explicitly created, they are sometimes referred to as implicit pools. The default memory pool of a device does not support IPC.

15.6. Explicit Pools

The API `cudaMemPoolCreate` creates an explicit pool. This allows applications to request properties for their allocation beyond what is provided by the default/implict pools. These include properties such as IPC capability, maximum pool size, allocations resident on a specific CPU NUMA node on supported platforms etc.

```
// create a pool similar to the implicit pool on device 0
int device = 0;
cudaMemPoolProps poolProps = { };
poolProps.allocType = cudaMemAllocationTypePinned;
poolProps.location.id = device;
poolProps.location.type = cudaMemLocationTypeDevice;

cudaMemPoolCreate(&memPool, &poolProps));
```

The following code snippet illustrates an example of creating an IPC capable memory pool on a valid CPU NUMA node.

```
// create a pool resident on a CPU NUMA node that is capable of IPC sharing (via a file
↪descriptor).
int cpu_numa_id = 0;
cudaMemPoolProps poolProps = { };
poolProps.allocType = cudaMemAllocationTypePinned;
poolProps.location.id = cpu_numa_id;
poolProps.location.type = cudaMemLocationTypeHostNuma;
poolProps.handleType = cudaMemHandleTypePosixFileDescriptor;

cudaMemPoolCreate(&ipcMemPool, &poolProps));
```

15.7. Physical Page Caching Behavior

By default, the allocator tries to minimize the physical memory owned by a pool. To minimize the OS calls to allocate and free physical memory, applications must configure a memory footprint for each pool. Applications can do this with the release threshold attribute (`cudaMemPoolAttrReleaseThreshold`).

The release threshold is the amount of memory in bytes a pool should hold onto before trying to release memory back to the OS. When more than the release threshold bytes of memory are held by the memory pool, the allocator will try to release memory back to the OS on the next call to stream, event or device synchronize. Setting the release threshold to UINT64_MAX will prevent the driver from attempting to shrink the pool after every synchronization.

```
Cuuint64_t setVal = UINT64_MAX;
cudaMemPoolSetAttribute(memPool, cudaMemPoolAttrReleaseThreshold, &setVal);
```

Applications that set `cudaMemPoolAttrReleaseThreshold` high enough to effectively disable memory pool shrinking may wish to explicitly shrink a memory pool's memory footprint. `cudaMemPoolTrimTo` allows such applications to do so. When trimming a memory pool's footprint, the `minBytesToKeep` parameter allows an application to hold onto an amount of memory it expects to need in a subsequent phase of execution.

```
Cuuint64_t setVal = UINT64_MAX;
cudaMemPoolSetAttribute(memPool, cudaMemPoolAttrReleaseThreshold, &setVal);

// application phase needing a lot of memory from the stream ordered allocator
for (i=0; i<10; i++) {
    for (j=0; j<10; j++) {
        cudaMallocAsync(&ptrs[j],size[j], stream);
    }
    kernel<<<...,stream>>>(ptrs,...);
    for (j=0; j<10; j++) {
        cudaFreeAsync(ptrs[j], stream);
    }
}

// Process does not need as much memory for the next phase.
// Synchronize so that the trim operation will know that the allocations are no
// longer in use.
cudaStreamSynchronize(stream);
```

(continues on next page)

(continued from previous page)

```
cudaMemPoolTrimTo(mempool, 0);

// Some other process/allocation mechanism can now use the physical memory
// released by the trimming operation.
```

15.8. Resource Usage Statistics

In CUDA 11.3, the pool attributes `cudaMemPoolAttrReservedMemCurrent`, `cudaMemPoolAttrReservedMemHigh`, `cudaMemPoolAttrUsedMemCurrent`, and `cudaMemPoolAttrUsedMemHigh` were added to query the memory usage of a pool.

Querying the `cudaMemPoolAttrReservedMemCurrent` attribute of a pool reports the current total physical GPU memory consumed by the pool. Querying the `cudaMemPoolAttrUsedMemCurrent` of a pool returns the total size of all of the memory allocated from the pool and not available for reuse.

The`cudaMemPoolAttr*MemHigh` attributes are watermarks recording the max value achieved by the respective `cudaMemPoolAttr*MemCurrent` attribute since last reset. They can be reset to the current value by using the `cudaMemPoolSetAttribute` API.

```
// sample helper functions for getting the usage statistics in bulk
struct usageStatistics {
    cuuint64_t reserved;
    cuuint64_t reservedHigh;
    cuuint64_t used;
    cuuint64_t usedHigh;
};

void getUsageStatistics(cudaMemoryPool_t memPool, struct usageStatistics *statistics)
{
    cudaMemPoolGetAttribute(memPool, cudaMemPoolAttrReservedMemCurrent, statistics->
↪reserved);
    cudaMemPoolGetAttribute(memPool, cudaMemPoolAttrReservedMemHigh, statistics->
↪reservedHigh);
    cudaMemPoolGetAttribute(memPool, cudaMemPoolAttrUsedMemCurrent, statistics->used);
    cudaMemPoolGetAttribute(memPool, cudaMemPoolAttrUsedMemHigh, statistics->
↪usedHigh);
}


// resetting the watermarks will make them take on the current value.
void resetStatistics(cudaMemoryPool_t memPool)
{
    cuuint64_t value = 0;
    cudaMemPoolSetAttribute(memPool, cudaMemPoolAttrReservedMemHigh, &value);
    cudaMemPoolSetAttribute(memPool, cudaMemPoolAttrUsedMemHigh, &value);
}
```

15.9. Memory Reuse Policies

In order to service an allocation request, the driver attempts to reuse memory that was previously freed via `cudaFreeAsync()` before attempting to allocate more memory from the OS. For example, memory freed in a stream can immediately be reused for a subsequent allocation request in the same stream. Similarly, when a stream is synchronized with the CPU, the memory that was previously freed in that stream becomes available for reuse for an allocation in any stream.

The stream ordered allocator has a few controllable allocation policies. The pool attributes `cudaMemPoolReuseFollowEventDependencies`, `cudaMemPoolReuseAllowOpportunistic`, and `cudaMemPoolReuseAllowInternalDependencies` control these policies. Upgrading to a newer CUDA driver may change, enhance, augment and/or reorder the reuse policies.

15.9.1. cudaMemPoolReuseFollowEventDependencies

Before allocating more physical GPU memory, the allocator examines dependency information established by CUDA events and tries to allocate from memory freed in another stream.

```
cudaMallocAsync(&ptr, size, originalStream);
kernel<<<..., originalStream>>>(ptr, ...);
cudaFreeAsync(ptr, originalStream);
cudaEventRecord(event,originalStream);

// waiting on the event that captures the free in another stream
// allows the allocator to reuse the memory to satisfy
// a new allocation request in the other stream when
// cudaMemPoolReuseFollowEventDependencies is enabled.
cudaStreamWaitEvent(otherStream, event);
cudaMallocAsync(&ptr2, size, otherStream);
```

15.9.2. cudaMemPoolReuseAllowOpportunistic

According to the `cudaMemPoolReuseAllowOpportunistic` policy, the allocator examines freed allocations to see if the free's stream order semantic has been met (such as the stream has passed the point of execution indicated by the free). When this is disabled, the allocator will still reuse memory made available when a stream is synchronized with the CPU. Disabling this policy does not stop the `cudaMemPoolReuseFollowEventDependencies` from applying.

```
cudaMallocAsync(&ptr, size, originalStream);
kernel<<<..., originalStream>>>(ptr, ...);
cudaFreeAsync(ptr, originalStream);


// after some time, the kernel finishes running
wait(10);

// When cudaMemPoolReuseAllowOpportunistic is enabled this allocation request
// can be fulfilled with the prior allocation based on the progress of originalStream.
cudaMallocAsync(&ptr2, size, otherStream);
```

15.9.3. cudaMemPoolReuseAllowInternalDependencies

Failing to allocate and map more physical memory from the OS, the driver will look for memory whose availability depends on another stream's pending progress. If such memory is found, the driver will insert the required dependency into the allocating stream and reuse the memory.

```
cudaMallocAsync(&ptr, size, originalStream);
kernel<<<..., originalStream>>>(ptr, ...);
cudaFreeAsync(ptr, originalStream);

// When cudaMemPoolReuseAllowInternalDependencies is enabled
// and the driver fails to allocate more physical memory, the driver may
// effectively perform a cudaStreamWaitEvent in the allocating stream
// to make sure that future work in 'otherStream' happens after the work
// in the original stream that would be allowed to access the original allocation.
cudaMallocAsync(&ptr2, size, otherStream);
```

15.9.4. Disabling Reuse Policies

While the controllable reuse policies improve memory reuse, users may want to disable them. Allowing opportunistic reuse (such as `cudaMemPoolReuseAllowOpportunistic`) introduces run to run variance in allocation patterns based on the interleaving of CPU and GPU execution. Internal dependency insertion (such as `cudaMemPoolReuseAllowInternalDependencies`) can serialize work in unexpected and potentially non-deterministic ways when the user would rather explicitly synchronize an event or stream on allocation failure.

15.10. Device Accessibility for Multi-GPU Support

Just like allocation accessibility controlled through the virtual memory management APIs, memory pool allocation accessibility does not follow `cudaDeviceEnablePeerAccess` or `cuCtxEnablePeerAccess`. Instead, the API `cudaMemPoolSetAccess` modifies what devices can access allocations from a pool. By default, allocations are accessible from the device where the allocations are located. This access cannot be revoked. To enable access from other devices, the accessing device must be peer capable with the memory pool's device; check with `cudaDeviceCanAccessPeer`. If the peer capability is not checked, the set access may fail with `cudaErrorInvalidDevice`. If no allocations had been made from the pool, the `cudaMemPoolSetAccess` call may succeed even when the devices are not peer capable; in this case, the next allocation from the pool will fail.

It is worth noting that `cudaMemPoolSetAccess` affects all allocations from the memory pool, not just future ones. Also the accessibility reported by `cudaMemPoolGetAccess` applies to all allocations from the pool, not just future ones. It is recommended that the accessibility settings of a pool for a given GPU not be changed frequently; once a pool is made accessible from a given GPU, it should remain accessible from that GPU for the lifetime of the pool.

```
// snippet showing usage of cudaMemPoolSetAccess:
cudaError_t setAccessOnDevice(cudaMemPool_t memPool, int residentDevice,
              int accessingDevice) {
```

(continues on next page)

(continued from previous page)

```
    cudaMemAccessDesc accessDesc = {};
    accessDesc.location.type = cudaMemLocationTypeDevice;
    accessDesc.location.id = accessingDevice;
    accessDesc.flags = cudaMemAccessFlagsProtReadWrite;

    int canAccess = 0;
    cudaError_t error = cudaDeviceCanAccessPeer(&canAccess, accessingDevice,
              residentDevice);
    if (error != cudaSuccess) {
        return error;
    } else if (canAccess == 0) {
        return cudaErrorPeerAccessUnsupported;
    }

    // Make the address accessible
    return cudaMemPoolSetAccess(memPool, &accessDesc, 1);
}
```

15.11. IPC Memory Pools

IPC capable memory pools allow easy, efficient and secure sharing of GPU memory between processes. CUDA's IPC memory pools provide the same security benefits as CUDA's virtual memory management APIs.

There are two phases to sharing memory between processes with memory pools. The processes first need to share access to the pool, then share specific allocations from that pool. The first phase establishes and enforces security. The second phase coordinates what virtual addresses are used in each process and when mappings need to be valid in the importing process.

15.11.1. Creating and Sharing IPC Memory Pools

Sharing access to a pool involves retrieving an OS native handle to the pool (with the `cudaMemPoolExportToShareableHandle()` API), transferring the handle to the importing process using the usual OS native IPC mechanisms, and creating an imported memory pool (with the `cudaMemPoolImportFromShareableHandle()` API). For `cudaMemPoolExportToShareableHandle` to succeed, the memory pool had to be created with the requested handle type specified in the pool properties structure. Please reference samples for the appropriate IPC mechanisms to transfer the OS native handle between processes. The rest of the procedure can be found in the following code snippets.

```
// in exporting process
// create an exportable IPC capable pool on device 0
cudaMemPoolProps poolProps = { };
poolProps.allocType = cudaMemAllocationTypePinned;
poolProps.location.id = 0;
poolProps.location.type = cudaMemLocationTypeDevice;

// Setting handleTypes to a non zero value will make the pool exportable (IPC capable)
poolProps.handleTypes = CU_MEM_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR;
```

(continues on next page)

(continued from previous page)

```
cudaMemPoolCreate(&memPool, &poolProps));

// FD based handles are integer types
int fdHandle = 0;


// Retrieve an OS native handle to the pool.
// Note that a pointer to the handle memory is passed in here.
cudaMemPoolExportToShareableHandle(&fdHandle,
            memPool,
            CU_MEM_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR,
            0);

// The handle must be sent to the importing process with the appropriate
// OS specific APIs.
```

```
// in importing process
 int fdHandle;
// The handle needs to be retrieved from the exporting process with the
// appropriate OS specific APIs.
// Create an imported pool from the shareable handle.
// Note that the handle is passed by value here.
cudaMemPoolImportFromShareableHandle(&importedMemPool,
          (void*)fdHandle,
          CU_MEM_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR,
          0);
```

15.11.2. Set Access in the Importing Process

Imported memory pools are initially only accessible from their resident device. The imported memory pool does not inherit any accessibility set by the exporting process. The importing process needs to enable access (with `cudaMemPoolSetAccess`) from any GPU it plans to access the memory from.

If the imported memory pool belongs to a non-visible device in the importing process, the user must use the `cudaMemPoolSetAccess` API to enable access from the GPUs the allocations will be used on.

15.11.3. Creating and Sharing Allocations from an Exported Pool

Once the pool has been shared, allocations made with `cudaMallocAsync()` from the pool in the exporting process can be shared with other processes that have imported the pool. Since the pool's security policy is established and verified at the pool level, the OS does not need extra bookkeeping to provide security for specific pool allocations; In other words, the opaque `cudaMemPoolPtrExportData` required to import a pool allocation may be sent to the importing process using any mechanism.

While allocations may be exported and even imported without synchronizing with the allocating stream in any way, the importing process must follow the same rules as the exporting process when accessing the allocation. Namely, access to the allocation must happen after the stream ordering of the allocation operation in the allocating stream. The two following code snippets show `cudaMemPoolEx-`

portPointer() and cudaMemPoolImportPointer() sharing the allocation with an IPC event used to guarantee that the allocation isn't accessed in the importing process before the allocation is ready.

```
// preparing an allocation in the exporting process
cudaMemPoolPtrExportData exportData;
cudaEvent_t readyIpcEvent;
cudaIpcEventHandle_t readyIpcEventHandle;

// ipc event for coordinating between processes
// cudaEventInterprocess flag makes the event an ipc event
// cudaEventDisableTiming  is set for performance reasons

cudaEventCreate(
        &readyIpcEvent, cudaEventDisableTiming | cudaEventInterprocess)

// allocate from the exporting mem pool
cudaMallocAsync(&ptr, size,exportMemPool, stream);

// event for sharing when the allocation is ready.
cudaEventRecord(readyIpcEvent, stream);
cudaMemPoolExportPointer(&exportData, ptr);
cudaIpcGetEventHandle(&readyIpcEventHandle, readyIpcEvent);

// Share IPC event and pointer export data with the importing process using
//  any mechanism. Here we copy the data into shared memory
shmem->ptrData = exportData;
shmem->readyIpcEventHandle = readyIpcEventHandle;
// signal consumers data is ready
```

```
// Importing an allocation
cudaMemPoolPtrExportData *importData = &shmem->prtData;
cudaEvent_t readyIpcEvent;
cudaIpcEventHandle_t *readyIpcEventHandle = &shmem->readyIpcEventHandle;

// Need to retrieve the ipc event handle and the export data from the
// exporting process using any mechanism.  Here we are using shmem and just
// need synchronization to make sure the shared memory is filled in.

cudaIpcOpenEventHandle(&readyIpcEvent, readyIpcEventHandle);

// import the allocation. The operation does not block on the allocation being ready.
cudaMemPoolImportPointer(&ptr, importedMemPool, importData);

// Wait for the prior stream operations in the allocating stream to complete before
// using the allocation in the importing process.
cudaStreamWaitEvent(stream, readyIpcEvent);
kernel<<<..., stream>>>(ptr, ...);
```

When freeing the allocation, the allocation needs to be freed in the importing process before it is freed in the exporting process. The following code snippet demonstrates the use of CUDA IPC events to provide the required synchronization between the cudaFreeAsync operations in both processes. Access to the allocation from the importing process is obviously restricted by the free operation in the importing process side. It is worth noting that cudaFree can be used to free the allocation in both processes and that other stream synchronization APIs may be used instead of CUDA IPC events.

```
// The free must happen in importing process before the exporting process
kernel<<<..., stream>>>(ptr, ...);
```

(continues on next page)

(continued from previous page)

```
// Last access in importing process
cudaFreeAsync(ptr, stream);

// Access not allowed in the importing process after the free
cudaIpcEventRecord(finishedIpcEvent, stream);
```

```
// Exporting process
// The exporting process needs to coordinate its free with the stream order
// of the importing process's free.
cudaStreamWaitEvent(stream, finishedIpcEvent);
kernel<<<..., stream>>>(ptrInExportingProcess, ...);

// The free in the importing process doesn't stop the exporting process
// from using the allocation.
cudFreeAsync(ptrInExportingProcess,stream);
```

15.11.4. IPC Export Pool Limitations

IPC pools currently do not support releasing physical blocks back to the OS. As a result the `cudaMemPoolTrimTo` API acts as a no-op and the `cudaMemPoolAttrReleaseThreshold` effectively gets ignored. This behavior is controlled by the driver, not the runtime and may change in a future driver update.

15.11.5. IPC Import Pool Limitations

Allocating from an import pool is not allowed; specifically, import pools cannot be set current and cannot be used in the `cudaMallocFromPoolAsync` API. As such, the allocation reuse policy attributes are meaningless for these pools.

IPC pools currently do not support releasing physical blocks back to the OS. As a result the `cudaMemPoolTrimTo` API acts as a no-op and the `cudaMemPoolAttrReleaseThreshold` effectively gets ignored.

The resource usage stat attribute queries only reflect the allocations imported into the process and the associated physical memory.

15.12. Synchronization API Actions

One of the optimizations that comes with the allocator being part of the CUDA driver is integration with the synchronize APIs. When the user requests that the CUDA driver synchronize, the driver waits for asynchronous work to complete. Before returning, the driver will determine what frees the synchronization guaranteed to be completed. These allocations are made available for allocation regardless of specified stream or disabled allocation policies. The driver also checks `cudaMemPoolAttrReleaseThreshold` here and releases any excess physical memory that it can.

15.13. Addendums

15.13.1. cudaMemcpyAsync Current Context/Device Sensitivity

In the current CUDA driver, any async `memcpy` involving memory from `cudaMallocAsync` should be done using the specified stream's context as the calling thread's current context. This is not necessary for `cudaMemcpyPeerAsync`, as the device primary contexts specified in the API are referenced instead of the current context.

15.13.2. cuPointerGetAttribute Query

Invoking `cuPointerGetAttribute` on an allocation after invoking `cudaFreeAsync` on it results in undefined behavior. Specifically, it does not matter if an allocation is still accessible from a given stream: the behavior is still undefined.

15.13.3. cuGraphAddMemsetNode

`cuGraphAddMemsetNode` does not work with memory allocated via the stream ordered allocator. However, memsets of the allocations can be stream captured.

15.13.4. Pointer Attributes

The `cuPointerGetAttributes` query works on stream ordered allocations. Since stream ordered allocations are not context associated, querying `CU_POINTER_ATTRIBUTE_CONTEXT` will succeed but return NULL in `*data`. The attribute `CU_POINTER_ATTRIBUTE_DEVICE_ORDINAL` can be used to determine the location of the allocation: this can be useful when selecting a context for making p2h2p copies using `cudaMemcpyPeerAsync`. The attribute `CU_POINTER_ATTRIBUTE_MEMPOOL_HANDLE` was added in CUDA 11.3 and can be useful for debugging and for confirming which pool an allocation comes from before doing IPC.

15.13.5. CPU Virtual Memory

When using CUDA stream-ordered memory allocator APIs, avoid setting VRAM limitations with "ulimit -v" as this is not supported.

Chapter 16. Graph Memory Nodes

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

16.1. Introduction

Graph memory nodes allow graphs to create and own memory allocations. Graph memory nodes have GPU ordered lifetime semantics, which dictate when memory is allowed to be accessed on the device. These GPU ordered lifetime semantics enable driver-managed memory reuse, and match those of the stream ordered allocation APIs `cudaMallocAsync` and `cudaFreeAsync`, which may be captured when creating a graph.

Graph allocations have fixed addresses over the life of a graph including repeated instantiations and launches. This allows the memory to be directly referenced by other operations within the graph without the need of a graph update, even when CUDA changes the backing physical memory. Within a graph, allocations whose graph ordered lifetimes do not overlap may use the same underlying physical memory.

CUDA may reuse the same physical memory for allocations across multiple graphs, aliasing virtual address mappings according to the GPU ordered lifetime semantics. For example when different graphs are launched into the same stream, CUDA may virtually alias the same physical memory to satisfy the needs of allocations which have single-graph lifetimes.

16.2. Support and Compatibility

Graph memory nodes require an 11.4 capable CUDA driver and support for the stream ordered allocator on the GPU. The following snippet shows how to check for support on a given device.

```
int driverVersion = 0;
int deviceSupportsMemoryPools = 0;
int deviceSupportsMemoryNodes = 0;
cudaDriverGetVersion(&driverVersion);
if (driverVersion >= 11020) { // avoid invalid value error in cudaDeviceGetAttribute
    cudaDeviceGetAttribute(&deviceSupportsMemoryPools,
↪cudaDevAttrMemoryPoolsSupported, device);
```

(continues on next page)

(continued from previous page)

```
}
deviceSupportsMemoryNodes = (driverVersion >= 11040) && (deviceSupportsMemoryPools !=
↪0);
```

Doing the attribute query inside the driver version check avoids an invalid value return code on 11.0 and 11.1 drivers. Be aware that the compute sanitizer emits warnings when it detects CUDA returning error codes, and a version check before reading the attribute will avoid this. Graph memory nodes are only supported on driver versions 11.4 and newer.

16.3. API Fundamentals

Graph memory nodes are graph nodes representing either memory allocation or free actions. As a shorthand, nodes that allocate memory are called allocation nodes. Likewise, nodes that free memory are called free nodes. Allocations created by allocation nodes are called graph allocations. CUDA assigns virtual addresses for the graph allocation at node creation time. While these virtual addresses are fixed for the lifetime of the allocation node, the allocation contents are not persistent past the freeing operation and may be overwritten by accesses referring to a different allocation.

Graph allocations are considered recreated every time a graph runs. A graph allocation's lifetime, which differs from the node's lifetime, begins when GPU execution reaches the allocating graph node and ends when one of the following occurs:

- GPU execution reaches the freeing graph node
- GPU execution reaches the freeing `cudaFreeAsync()` stream call
- immediately upon the freeing call to `cudaFree()`

Note: Graph destruction does not automatically free any live graph-allocated memory, even though it ends the lifetime of the allocation node. The allocation must subsequently be freed in another graph, or using `cudaFreeAsync()/cudaFree()`.

Just like other *Graph Structure*, graph memory nodes are ordered within a graph by dependency edges. A program must guarantee that operations accessing graph memory:

- are ordered after the allocation node
- are ordered before the operation freeing the memory

Graph allocation lifetimes begin and usually end according to GPU execution (as opposed to API invocation). GPU ordering is the order that work runs on the GPU as opposed to the order that the work is enqueued or described. Thus, graph allocations are considered 'GPU ordered.'

16.3.1. Graph Node APIs

Graph memory nodes may be explicitly created with the memory node creation APIs, `cudaGraphAddMemAllocNode` and `cudaGraphAddMemFreeNode`. The address allocated by `cudaGraphAddMemAllocNode` is returned to the user in the `dptr` field of the passed `CUDA_MEM_ALLOC_NODE_PARAMS` structure. All operations using graph allocations inside the allocating graph must be ordered after the allocating node. Similarly, any free nodes must be ordered after all uses of the allocation within the graph. `cudaGraphAddMemFreeNode` creates free nodes.

In the following figure, there is an example graph with an alloc and a free node. Kernel nodes **a**, **b**, and **c** are ordered after the allocation node and before the free node such that the kernels can access the allocation. Kernel node **e** is not ordered after the alloc node and therefore cannot safely access the memory. Kernel node **d** is not ordered before the free node, therefore it cannot safely access the memory.

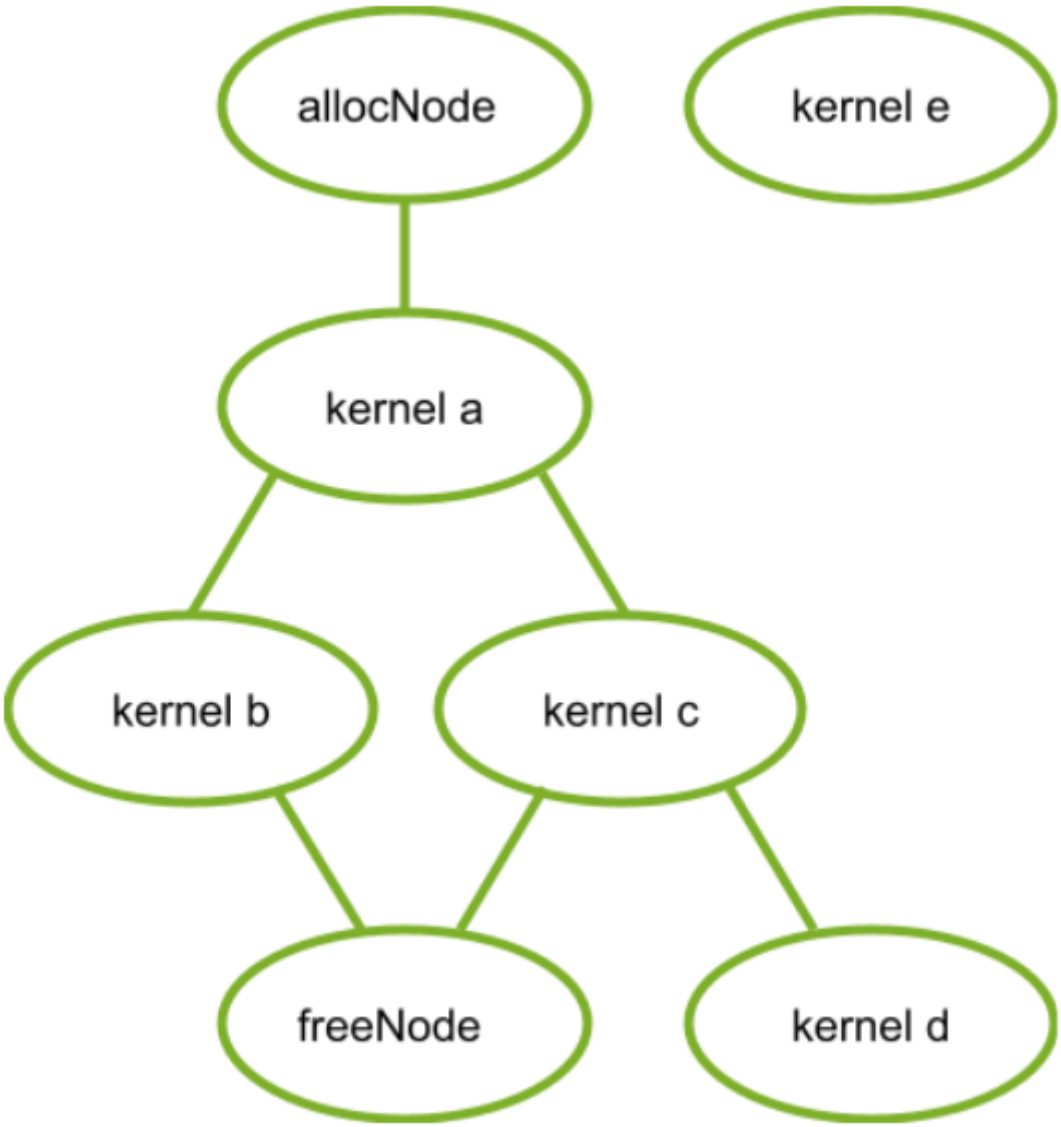


Figure 32: Kernel Nodes

The following code snippet establishes the graph in this figure:

```
// Create the graph - it starts out empty
cudaGraphCreate(&graph, 0);

// parameters for a basic allocation
cudaMemAllocNodeParams params = {};
params.poolProps.allocType = cudaMemAllocationTypePinned;
params.poolProps.location.type = cudaMemLocationTypeDevice;
// specify device 0 as the resident device
params.poolProps.location.id = 0;
params.bytesize = size;

cudaGraphAddMemAllocNode(&allocNode, graph, NULL, 0, &params);
nodeParams->kernelParams[0] = params.dptr;
cudaGraphAddKernelNode(&a, graph, &allocNode, 1, &nodeParams);
cudaGraphAddKernelNode(&b, graph, &a, 1, &nodeParams);
cudaGraphAddKernelNode(&c, graph, &a, 1, &nodeParams);
cudaGraphNode_t dependencies[2];
// kernel nodes b and c are using the graph allocation, so the freeing node must
↪depend on them.  Since the dependency of node b on node a establishes an indirect
↪dependency, the free node does not need to explicitly depend on node a.
dependencies[0] = b;
dependencies[1] = c;
cudaGraphAddMemFreeNode(&freeNode, graph, dependencies, 2, params.dptr);
// free node does not depend on kernel node d, so it must not access the freed graph
↪allocation.
cudaGraphAddKernelNode(&d, graph, &c, 1, &nodeParams);

// node e does not depend on the allocation node, so it must not access the
↪allocation.  This would be true even if the freeNode depended on kernel node e.
cudaGraphAddKernelNode(&e, graph, NULL, 0, &nodeParams);
```

16.3.2. Stream Capture

Graph memory nodes can be created by capturing the corresponding stream ordered allocation and free calls `cudaMallocAsync` and `cudaFreeAsync`. In this case, the virtual addresses returned by the captured allocation API can be used by other operations inside the graph. Since the stream ordered dependencies will be captured into the graph, the ordering requirements of the stream ordered allocation APIs guarantee that the graph memory nodes will be properly ordered with respect to the captured stream operations (for correctly written stream code).

Ignoring kernel nodes **d** and **e**, for clarity, the following code snippet shows how to use stream capture to create the graph from the previous figure:

```
cudaMallocAsync(&dptr, size, stream1);
kernel_A<<< ..., stream1 >>>(dptr, ...);

// Fork into stream2
cudaEventRecord(event1, stream1);
cudaStreamWaitEvent(stream2, event1);

kernel_B<<< ..., stream1 >>>(dptr, ...);
// event dependencies translated into graph dependencies, so the kernel node created
↪by the capture of kernel C will depend on the allocation node created by capturing
↪the cudaMallocAsync call.
```

(continues on next page)

(continued from previous page)

```
kernel_C<<< ..., stream2 >>>(dptr, ...);

// Join stream2 back to origin stream (stream1)
cudaEventRecord(event2, stream2);
cudaStreamWaitEvent(stream1, event2);

// Free depends on all work accessing the memory.
cudaFreeAsync(dptr, stream1);

// End capture in the origin stream
cudaStreamEndCapture(stream1, &graph);
```

16.3.3. Accessing and Freeing Graph Memory Outside of the Allocating Graph

Graph allocations do not have to be freed by the allocating graph. When a graph does not free an allocation, that allocation persists beyond the execution of the graph and can be accessed by subsequent CUDA operations. These allocations may be accessed in another graph or directly using a stream operation as long as the accessing operation is ordered after the allocation through CUDA events and other stream ordering mechanisms. An allocation may subsequently be freed by regular calls to `cudaFree`, `cudaFreeAsync`, or by the launch of another graph with a corresponding free node, or a subsequent launch of the allocating graph (if it was instantiated with the *cudaGraphInstantiateFlagAutoFreeOnLaunch* flag). It is illegal to access memory after it has been freed - the free operation must be ordered after all operations accessing the memory using graph dependencies, CUDA events, and other stream ordering mechanisms.

Note: Because graph allocations may share underlying physical memory with each other, the *Virtual Aliasing Support* rules relating to consistency and coherency must be considered. Simply put, the free operation must be ordered after the full device operation (for example, compute kernel / memcpy) completes. Specifically, out of band synchronization - for example a handshake through memory as part of a compute kernel that accesses the graph-allocated memory - is not sufficient for providing ordering guarantees between the memory writes to graph memory and the free operation of that graph memory.

The following code snippets demonstrate accessing graph allocations outside of the allocating graph with ordering properly established by: using a single stream, using events between streams, and using events baked into the allocating and freeing graph.

Ordering established by using a single stream:

```
void *dptr;
cudaGraphAddMemAllocNode(&allocNode, allocGraph, NULL, 0, &params);
dptr = params.dptr;

cudaGraphInstantiate(&allocGraphExec, allocGraph, NULL, NULL, 0);

cudaGraphLaunch(allocGraphExec, stream);
kernel<<< …, stream >>>(dptr, …);
cudaFreeAsync(dptr, stream);
```

Ordering established by recording and waiting on CUDA events:

```
void *dptr;

// Contents of allocating graph
cudaGraphAddMemAllocNode(&allocNode, allocGraph, NULL, 0, &params);
dptr = params.dptr;

// contents of consuming/freeing graph
nodeParams->kernelParams[0] = params.dptr;
cudaGraphAddKernelNode(&a, graph, NULL, 0, &nodeParams);
cudaGraphAddMemFreeNode(&freeNode, freeGraph, &a, 1, dptr);

cudaGraphInstantiate(&allocGraphExec, allocGraph, NULL, NULL, 0);
cudaGraphInstantiate(&freeGraphExec, freeGraph, NULL, NULL, 0);

cudaGraphLaunch(allocGraphExec, allocStream);

// establish the dependency of stream2 on the allocation node
// note: the dependency could also have been established with a stream synchronize
↪operation
cudaEventRecord(allocEvent, allocStream)
cudaStreamWaitEvent(stream2, allocEvent);

kernel<<< …, stream2 >>> (dptr, …);

// establish the dependency between the stream 3 and the allocation use
cudaStreamRecordEvent(streamUseDoneEvent, stream2);
cudaStreamWaitEvent(stream3, streamUseDoneEvent);

// it is now safe to launch the freeing graph, which may also access the memory
cudaGraphLaunch(freeGraphExec, stream3);
```

Ordering established by using graph external event nodes:

```
void *dptr;
cudaEvent_t allocEvent; // event indicating when the allocation will be ready for use.
cudaEvent_t streamUseDoneEvent; // event indicating when the stream operations are
↪done with the allocation.

// Contents of allocating graph with event record node
cudaGraphAddMemAllocNode(&allocNode, allocGraph, NULL, 0, &params);
dptr = params.dptr;
// note: this event record node depends on the alloc node
cudaGraphAddEventRecordNode(&recordNode, allocGraph, &allocNode, 1, allocEvent);
cudaGraphInstantiate(&allocGraphExec, allocGraph, NULL, NULL, 0);

// contents of consuming/freeing graph with event wait nodes
cudaGraphAddEventWaitNode(&streamUseDoneEventNode, waitAndFreeGraph, NULL, 0,
↪streamUseDoneEvent);
cudaGraphAddEventWaitNode(&allocReadyEventNode, waitAndFreeGraph, NULL, 0,
↪allocEvent);
nodeParams->kernelParams[0] = params.dptr;

// The allocReadyEventNode provides ordering with the alloc node for use in a
↪consuming graph.
cudaGraphAddKernelNode(&kernelNode, waitAndFreeGraph, &allocReadyEventNode, 1, &
↪nodeParams);
```

(continues on next page)

(continued from previous page)

```
// The free node has to be ordered after both external and internal users.
// Thus the node must depend on both the kernelNode and the
// streamUseDoneEventNode.
dependencies[0] = kernelNode;
dependencies[1] = streamUseDoneEventNode;
cudaGraphAddMemFreeNode(&freeNode, waitAndFreeGraph, &dependencies, 2, dptr);
cudaGraphInstantiate(&waitAndFreeGraphExec, waitAndFreeGraph, NULL, NULL, 0);

cudaGraphLaunch(allocGraphExec, allocStream);

// establish the dependency of stream2 on the event node satisfies the ordering
↪requirement
cudaStreamWaitEvent(stream2, allocEvent);
kernel<<< …, stream2 >>> (dptr, …);
cudaStreamRecordEvent(streamUseDoneEvent, stream2);

// the event wait node in the waitAndFreeGraphExec establishes the dependency on the
↪"readyForFreeEvent" that is needed to prevent the kernel running in stream two from
↪accessing the allocation after the free node in execution order.
cudaGraphLaunch(waitAndFreeGraphExec, stream3);
```

16.3.4. cudaGraphInstantiateFlagAutoFreeOnLaunch

Under normal circumstances, CUDA will prevent a graph from being relaunched if it has unfreed memory allocations because multiple allocations at the same address will leak memory. Instantiating a graph with the `cudaGraphInstantiateFlagAutoFreeOnLaunch` flag allows the graph to be relaunched while it still has unfreed allocations. In this case, the launch automatically inserts an asynchronous free of the unfreed allocations.

Auto free on launch is useful for single-producer multiple-consumer algorithms. At each iteration, a producer graph creates several allocations, and, depending on runtime conditions, a varying set of consumers accesses those allocations. This type of variable execution sequence means that consumers cannot free the allocations because a subsequent consumer may require access. Auto free on launch means that the launch loop does not need to track the producer's allocations - instead, that information remains isolated to the producer's creation and destruction logic. In general, auto free on launch simplifies an algorithm which would otherwise need to free all the allocations owned by a graph before each relaunch.

Note: The `cudaGraphInstantiateFlagAutoFreeOnLaunch` flag does not change the behavior of graph destruction. The application must explicitly free the unfreed memory in order to avoid memory leaks, even for graphs instantiated with the flag. The following code shows the use of `cudaGraphInstantiateFlagAutoFreeOnLaunch` to simplify a single-producer / multiple-consumer algorithm:

```
// Create producer graph which allocates memory and populates it with data
cudaStreamBeginCapture(cudaStreamPerThread, cudaStreamCaptureModeGlobal);
cudaMallocAsync(&data1, blocks * threads, cudaStreamPerThread);
cudaMallocAsync(&data2, blocks * threads, cudaStreamPerThread);
produce<<<blocks, threads, 0, cudaStreamPerThread>>>(data1, data2);
...
cudaStreamEndCapture(cudaStreamPerThread, &graph);
cudaGraphInstantiateWithFlags(&producer,
```

(continues on next page)

(continued from previous page)

```
                                graph,
                                cudaGraphInstantiateFlagAutoFreeOnLaunch);
cudaGraphDestroy(graph);

// Create first consumer graph by capturing an asynchronous library call
cudaStreamBeginCapture(cudaStreamPerThread, cudaStreamCaptureModeGlobal);
consumerFromLibrary(data1, cudaStreamPerThread);
cudaStreamEndCapture(cudaStreamPerThread, &graph);
cudaGraphInstantiateWithFlags(&consumer1, graph, 0); //regular instantiation
cudaGraphDestroy(graph);

// Create second consumer graph
cudaStreamBeginCapture(cudaStreamPerThread, cudaStreamCaptureModeGlobal);
consume2<<<blocks, threads, 0, cudaStreamPerThread>>>(data2);
...
cudaStreamEndCapture(cudaStreamPerThread, &graph);
cudaGraphInstantiateWithFlags(&consumer2, graph, 0);
cudaGraphDestroy(graph);

// Launch in a loop
bool launchConsumer2 = false;
do {
    cudaGraphLaunch(producer, myStream);
    cudaGraphLaunch(consumer1, myStream);
    if (launchConsumer2) {
        cudaGraphLaunch(consumer2, myStream);
    }
} while (determineAction(&launchConsumer2));

cudaFreeAsync(data1, myStream);
cudaFreeAsync(data2, myStream);

cudaGraphExecDestroy(producer);
cudaGraphExecDestroy(consumer1);
cudaGraphExecDestroy(consumer2);
```

16.4. Optimized Memory Reuse

CUDA reuses memory in two ways:

- Virtual and physical memory reuse within a graph is based on virtual address assignment, like in the stream ordered allocator.
- Physical memory reuse between graphs is done with virtual aliasing: different graphs can map the same physical memory to their unique virtual addresses.

16.4.1. Address Reuse within a Graph

CUDA may reuse memory within a graph by assigning the same virtual address ranges to different allocations whose lifetimes do not overlap. Since virtual addresses may be reused, pointers to different allocations with disjoint lifetimes are not guaranteed to be unique.

The following figure shows adding a new allocation node (2) that can reuse the address freed by a dependent node (1).



Figure 33: Adding New Alloc Node 2

The following figure shows adding a new alloc node (4). The new alloc node is not dependent on the free node (2) so cannot reuse the address from the associated alloc node (2). If the alloc node (2) used the address freed by free node (1), the new alloc node 3 would need a new address.

16.4.2. Physical Memory Management and Sharing

CUDA is responsible for mapping physical memory to the virtual address before the allocating node is reached in GPU order. As an optimization for memory footprint and mapping overhead, multiple graphs may use the same physical memory for distinct allocations if they will not run simultaneously; however, physical pages cannot be reused if they are bound to more than one executing graph at the same time, or to a graph allocation which remains unfreed.

CUDA may update physical memory mappings at any time during graph instantiation, launch, or execution. CUDA may also introduce synchronization between future graph launches in order to prevent live graph allocations from referring to the same physical memory. As for any allocate-free-allocate



Figure 34: Adding New Alloc Node 3

pattern, if a program accesses a pointer outside of an allocation's lifetime, the erroneous access may silently read or write live data owned by another allocation (even if the virtual address of the allocation is unique). Use of compute sanitizer tools can catch this error.

The following figure shows graphs sequentially launched in the same stream. In this example, each graph frees all the memory it allocates. Since the graphs in the same stream never run concurrently, CUDA can and should use the same physical memory to satisfy all the allocations.

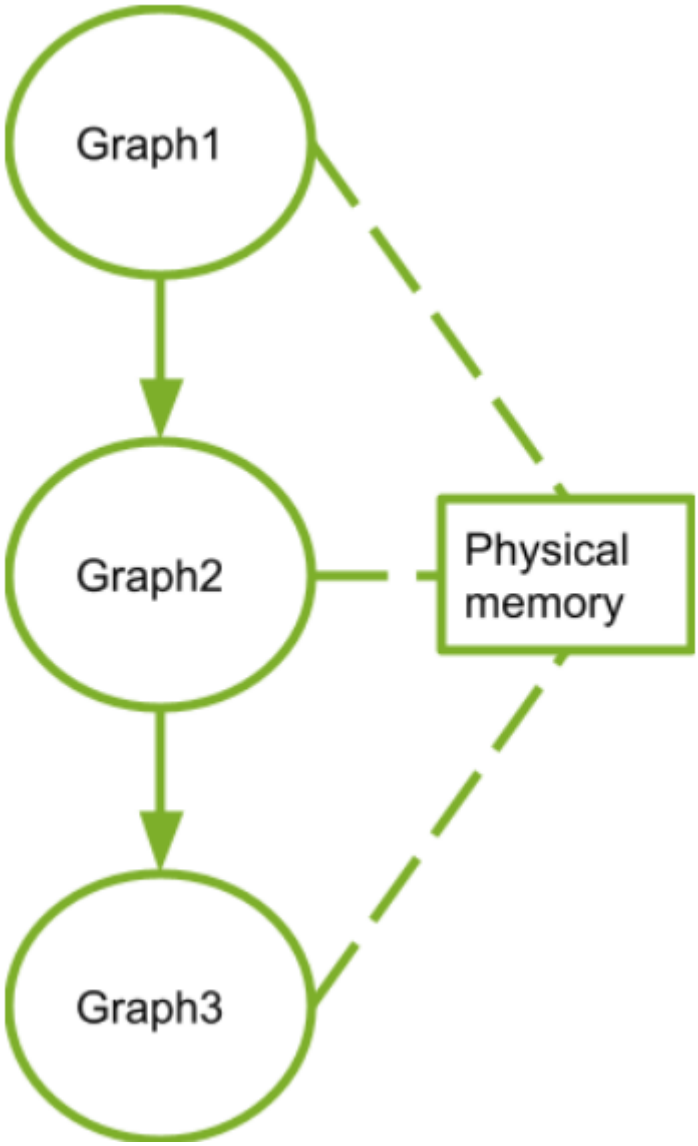


Figure 35: Sequentially Launched Graphs

16.5. Performance Considerations

When multiple graphs are launched into the same stream, CUDA attempts to allocate the same physical memory to them because the execution of these graphs cannot overlap. Physical mappings for a graph are retained between launches as an optimization to avoid the cost of remapping. If, at a later time, one of the graphs is launched such that its execution may overlap with the others (for example if it is launched into a different stream) then CUDA must perform some remapping because concurrent graphs require distinct memory to avoid data corruption.

In general, remapping of graph memory in CUDA is likely caused by these operations:

- Changing the stream into which a graph is launched
- A trim operation on the graph memory pool, which explicitly frees unused memory (discussed in *Physical Memory Footprint*)
- Relaunching a graph while an unfreed allocation from another graph is mapped to the same memory will cause a remap of memory before relaunch

Remapping must happen in execution order, but after any previous execution of that graph is complete (otherwise memory that is still in use could be unmapped). Due to this ordering dependency, as well as because mapping operations are OS calls, mapping operations can be relatively expensive. Applications can avoid this cost by launching graphs containing allocation memory nodes consistently into the same stream.

16.5.1. First Launch / cudaGraphUpload

Physical memory cannot be allocated or mapped during graph instantiation because the stream in which the graph will execute is unknown. Mapping is done instead during graph launch. Calling `cudaGraphUpload` can separate out the cost of allocation from the launch by performing all mappings for that graph immediately and associating the graph with the upload stream. If the graph is then launched into the same stream, it will launch without any additional remapping.

Using different streams for graph upload and graph launch behaves similarly to switching streams, likely resulting in remap operations. In addition, unrelated memory pool management is permitted to pull memory from an idle stream, which could negate the impact of the uploads.

16.6. Physical Memory Footprint

The pool-management behavior of asynchronous allocation means that destroying a graph which contains memory nodes (even if their allocations are free) will not immediately return physical memory to the OS for use by other processes. To explicitly release memory back to the OS, an application should use the `cudaDeviceGraphMemTrim` API.

`cudaDeviceGraphMemTrim` will unmap and release any physical memory reserved by graph memory nodes that is not actively in use. Allocations that have not been freed and graphs that are scheduled or running are considered to be actively using the physical memory and will not be impacted. Use of the trim API will make physical memory available to other allocation APIs and other applications or processes, but will cause CUDA to reallocate and remap memory when the trimmed graphs are next launched. Note that `cudaDeviceGraphMemTrim` operates on a different pool from `cudaMemPoolTrimTo()`. The graph memory pool is not exposed to the steam ordered memory allocator. CUDA

allows applications to query their graph memory footprint through the `cudaDeviceGetGraphMemAttribute` API. Querying the attribute `cudaGraphMemAttrReservedMemCurrent` returns the amount of physical memory reserved by the driver for graph allocations in the current process. Querying `cudaGraphMemAttrUsedMemCurrent` returns the amount of physical memory currently mapped by at least one graph. Either of these attributes can be used to track when new physical memory is acquired by CUDA for the sake of an allocating graph. Both of these attributes are useful for examining how much memory is saved by the sharing mechanism.

16.7. Peer Access

Graph allocations can be configured for access from multiple GPUs, in which case CUDA will map the allocations onto the peer GPUs as required. CUDA allows graph allocations requiring different mappings to reuse the same virtual address. When this occurs, the address range is mapped onto all GPUs required by the different allocations. This means an allocation may sometimes allow more peer access than was requested during its creation; however, relying on these extra mappings is still an error.

16.7.1. Peer Access with Graph Node APIs

The `cudaGraphAddMemAllocNode` API accepts mapping requests in the `accessDescs` array field of the node parameters structures. The `poolProps.location` embedded structure specifies the resident device for the allocation. Access from the allocating GPU is assumed to be needed, thus the application does not need to specify an entry for the resident device in the `accessDescs` array.

```
cudaMemAllocNodeParams params = {};
params.poolProps.allocType = cudaMemAllocationTypePinned;
params.poolProps.location.type = cudaMemLocationTypeDevice;
// specify device 1 as the resident device
params.poolProps.location.id = 1;
params.bytesize = size;

// allocate an allocation resident on device 1 accessible from device 1
cudaGraphAddMemAllocNode(&allocNode, graph, NULL, 0, &params);

accessDescs[2];
// boilerplate for the access descs (only ReadWrite and Device access supported by
↪the add node api)
accessDescs[0].flags = cudaMemAccessFlagsProtReadWrite;
accessDescs[0].location.type = cudaMemLocationTypeDevice;
accessDescs[1].flags = cudaMemAccessFlagsProtReadWrite;
accessDescs[1].location.type = cudaMemLocationTypeDevice;

// access being requested for device 0 & 2.  Device 1 access requirement left
↪implicit.
accessDescs[0].location.id = 0;
accessDescs[1].location.id = 2;

// access request array has 2 entries.
params.accessDescCount = 2;
params.accessDescs = accessDescs;
```

(continues on next page)

(continued from previous page)

```
// allocate an allocation resident on device 1 accessible from devices 0, 1 and 2. (0
↪& 2 from the descriptors, 1 from it being the resident device).
cudaGraphAddMemAllocNode(&allocNode, graph, NULL, 0, &params);
```

16.7.2. Peer Access with Stream Capture

For stream capture, the allocation node records the peer accessibility of the allocating pool at the time of the capture. Altering the peer accessibility of the allocating pool after a `cudaMallocFromPoolAsync` call is captured does not affect the mappings that the graph will make for the allocation.

```
// boilerplate for the access descs (only ReadWrite and Device access supported by
↪the add node api)
accessDesc.flags = cudaMemAccessFlagsProtReadWrite;
accessDesc.location.type = cudaMemLocationTypeDevice;
accessDesc.location.id = 1;

// let memPool be resident and accessible on device 0

cudaStreamBeginCapture(stream);
cudaMallocAsync(&dptr1, size, memPool, stream);
cudaStreamEndCapture(stream, &graph1);

cudaMemPoolSetAccess(memPool, &accessDesc, 1);

cudaStreamBeginCapture(stream);
cudaMallocAsync(&dptr2, size, memPool, stream);
cudaStreamEndCapture(stream, &graph2);

//The graph node allocating dptr1 would only have the device 0 accessibility even
↪though memPool now has device 1 accessibility.
//The graph node allocating dptr2 will have device 0 and device 1 accessibility,
↪since that was the pool accessibility at the time of the cudaMallocAsync call.
```

16.8. Memory Nodes in Child Graphs

CUDA 12.9 introduces the ability to move child graph ownership to a parent graph. Child graphs which are moved to the parent are allowed to contain memory allocation and free nodes. This allows a child graph containing allocation or free nodes to be independently constructed prior to its addition in a parent graph.

The following restrictions apply to child graphs after they have been moved:

- Cannot be independently instantiated or destroyed.
- Cannot be added as a child graph of a separate parent graph.
- Cannot be used as an argument to cuGraphExecUpdate.
- Cannot have additional memory allocation or free nodes added.

```
// Create the child graph
cudaGraphCreate(&child, 0);

// parameters for a basic allocation
cudaMemAllocNodeParams params = {};
params.poolProps.allocType = cudaMemAllocationTypePinned;
params.poolProps.location.type = cudaMemLocationTypeDevice;
// specify device 0 as the resident device
params.poolProps.location.id = 0;
params.bytesize = size;

cudaGraphAddMemAllocNode(&allocNode, graph, NULL, 0, &params);
// Additional nodes using the allocation could be added here
cudaGraphAddMemFreeNode(&freeNode, graph, &allocNode, 1, params.dptr);

// Create the parent graph
cudaGraphCreate(&parent, 0);

// Move the child graph to the parent graph
cudaGraphNodeParams childNodeParams = { cudaGraphNodeTypeGraph };
childNodeParams.graph.graph = child;
childNodeParams.graph.ownership = cudaGraphChildGraphOwnershipMove;
cudaGraphAddNode(&parentNode, parent, NULL, NULL, 0, &childNodeParams);
```

Chapter 17. Mathematical Functions

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

The reference manual lists, along with their description, all the functions of the C/C++ standard library mathematical functions that are supported in device code, as well as all intrinsic functions (that are only supported in device code).

This section provides accuracy information for some of these functions when applicable. It uses ULP for quantification. For further information on the definition of the Unit in the Last Place (ULP), please see Jean-Michel Muller's paper *On the definition of ulp(x)*, RR-5504, LIP RR-2005-09, INRIA, LIP. 2005, pp.16 at https://hal.inria.fr/inria-00070503/document.

Mathematical functions supported in device code do not set the global `errno` variable, nor report any floating-point exceptions to indicate errors; thus, if error diagnostic mechanisms are required, the user should implement additional screening for inputs and outputs of the functions. The user is responsible for the validity of pointer arguments. The user must not pass uninitialized parameters to the Mathematical functions as this may result in undefined behavior: functions are inlined in the user program and thus are subject to compiler optimizations.

17.1. Standard Functions

The functions from this section can be used in both host and device code.

This section specifies the error bounds of each function when executed on the device and also when executed on the host in the case where the host does not supply the function.

The error bounds are generated from extensive but not exhaustive tests, so they are not guaranteed bounds.

Single-Precision Floating-Point Functions

Addition and multiplication are IEEE-compliant, so have a maximum error of 0.5 ulp.

The recommended way to round a single-precision floating-point operand to an integer, with the result being a single-precision floating-point number is `rintf()`, not `roundf()`. The reason is that `roundf()` maps to a 4-instruction sequence on the device, whereas `rintf()` maps to a single instruction. `truncf()`, `ceilf()`, and `floorf()` each map to a single instruction as well.

Table 17: Single-Precision Mathematical Standard Library Functions with Maximum ULP Error. The maximum error is stated as the absolute value of the difference in ulps between the result returned by the CUDA library function and a correctly rounded single-precision result obtained according to the round-to-nearest ties-to-even rounding mode.

Function	Maximum ulp error
`x+y`	0 (IEEE-754 round-to-nearest-even)
`x*y`	0 (IEEE-754 round-to-nearest-even)
`x/y`	0 for compute capability ≥ 2 when compiled with `-prec-div=true` 2 (full range), otherwise
`1/x`	0 for compute capability ≥ 2 when compiled with `-prec-div=true` 1 (full range), otherwise
`rsqrtf(x)` `1/sqrtf(x)`	2 (full range) Applies to `1/sqrtf(x)` only when it is converted to `rsqrtf(x)` by the compiler.
`sqrtf(x)`	0 when compiled with `-prec-sqrt=true` Otherwise 1 for compute capability ≥ 5.2 and 3 for older architectures
`cbrtf(x)`	1 (full range)
`rcbrtf(x)`	1 (full range)
`hypotf(x,y)`	3 (full range)
`rhypotf(x,y)`	2 (full range)
`norm3df(x,y,z)`	3 (full range)
`rnorm3df(x,y,z)`	2 (full range)
`norm4df(x,y,z,t)`	3 (full range)
`rnorm4df(x,y,z,t)`	2 (full range)
`normf(dim,arr)`	An error bound cannot be provided because a fast algorithm is used with accuracy loss due to round-off. .
`rnormf(dim,arr)`	An error bound cannot be provided because a fast algorithm is used with accuracy loss due to round-off. .
`expf(x)`	2 (full range)
`exp2f(x)`	2 (full range)
`exp10f(x)`	2 (full range)
`expm1f(x)`	1 (full range)
`logf(x)`	1 (full range)

continues on next page

Table 17 – continued from previous page

Function	Maximum ulp error
`log2f(x)`	1 (full range)
`log10f(x)`	2 (full range)
`log1pf(x)`	1 (full range)
`sinf(x)`	2 (full range)
`cosf(x)`	2 (full range)
`tanf(x)`	4 (full range)
`sincosf(x,sptr,cptr)`	2 (full range)
`sinpif(x)`	1 (full range)
`cospif(x)`	1 (full range)
`sincospif(x,sptr,cptr)`	1 (full range)
`asinf(x)`	2 (full range)
`acosf(x)`	2 (full range)
`atanf(x)`	2 (full range)
`atan2f(y,x)`	3 (full range)
`sinhf(x)`	3 (full range)
`coshf(x)`	2 (full range)
`tanhf(x)`	2 (full range)
`asinhf(x)`	3 (full range)
`acoshf(x)`	4 (full range)
`atanhf(x)`	3 (full range)
`powf(x,y)`	4 (full range)
`erff(x)`	2 (full range)
`erfcf(x)`	4 (full range)
`erfinvf(x)`	2 (full range)
`erfcinvf(x)`	4 (full range)
`erfcxf(x)`	4 (full range)
`normcdff(x)`	5 (full range)
`normcdfinvf(x)`	5 (full range)
`lgammaf(x)`	6 (outside interval -10.001 ... -2.264; larger inside)
`tgammaf(x)`	5 (full range)
`fmaf(x,y,z)`	0 (full range)
`frexpf(x,exp)`	0 (full range)

continues on next page

Table 17 – continued from previous page

Function	Maximum ulp error
ldexpf(x,exp)	0 (full range)
scalbnf(x,n)	0 (full range)
scalblnf(x,l)	0 (full range)
logbf(x)	0 (full range)
ilogbf(x)	0 (full range)
j0f(x)	9 for |x| < 8 otherwise, the maximum absolute error is 2.2 x 10^{-6}
j1f(x)	9 for |x| < 8 otherwise, the maximum absolute error is 2.2 x 10^{-6}
jnf(n,x)	For n = 128, the maximum absolute error is 2.2 x 10^{-6}
y0f(x)	9 for |x| < 8 otherwise, the maximum absolute error is 2.2 x 10^{-6}
y1f(x)	9 for |x| < 8 otherwise, the maximum absolute error is 2.2 x 10^{-6}
ynf(n,x)	ceil(2 + 2.5n) for |x| < n otherwise, the maximum absolute error is 2.2 x 10^{-6}
cyl_bessel_i0f(x)	6 (full range)
cyl_bessel_i1f(x)	6 (full range)
fmodf(x,y)	0 (full range)
remainderf(x,y)	0 (full range)
remquof(x,y,iptr)	0 (full range)
modff(x,iptr)	0 (full range)
fdimf(x,y)	0 (full range)
truncf(x)	0 (full range)
roundf(x)	0 (full range)
rintf(x)	0 (full range)
nearbyintf(x)	0 (full range)
ceilf(x)	0 (full range)
floorf(x)	0 (full range)
lrintf(x)	0 (full range)

continues on next page

Table 17 – continued from previous page

Function	Maximum ulp error
`lroundf(x)`	0 (full range)
`llrintf(x)`	0 (full range)
`llroundf(x)`	0 (full range)

Double-Precision Floating-Point Functions

The recommended way to round a double-precision floating-point operand to an integer, with the result being a double-precision floating-point number is `rint()`, not `round()`. The reason is that `round()` maps to a 5-instruction sequence on the device, whereas `rint()` maps to a single instruction. `trunc()`, `ceil()`, and `floor()` each map to a single instruction as well.

Table 18: Double-Precision Mathematical Standard Library Functions with Maximum ULP Error. The maximum error is stated as the absolute value of the difference in ulps between the result returned by the CUDA library function and a correctly rounded double-precision result obtained according to the round-to-nearest ties-to-even rounding mode.

Function	Maximum ulp error
`x+y`	0 (IEEE-754 round-to-nearest-even)
`x*y`	0 (IEEE-754 round-to-nearest-even)
`x/y`	0 (IEEE-754 round-to-nearest-even)
`1/x`	0 (IEEE-754 round-to-nearest-even)
`sqrt(x)`	0 (IEEE-754 round-to-nearest-even)
`rsqrt(x)`	1 (full range)
`cbrt(x)`	1 (full range)
`rcbrt(x)`	1 (full range)
`hypot(x,y)`	2 (full range)
`rhypot(x,y)`	1 (full range)
`norm3d(x,y,z)`	2 (full range)
`rnorm3d(x,y,z)`	1 (full range)
`norm4d(x,y,z,t)`	2 (full range)
`rnorm4d(x,y,z,t)`	1 (full range)
`norm(dim,arr)`	An error bound cannot be provided because a fast algorithm is used with accuracy loss due to round-off.
`rnorm(dim,arr)`	An error bound cannot be provided because a fast algorithm is used with accuracy loss due to round-off.

continues on next page

Table 18 – continued from previous page

Function	Maximum ulp error
exp(x)	1 (full range)
exp2(x)	1 (full range)
exp10(x)	1 (full range)
expm1(x)	1 (full range)
log(x)	1 (full range)
log2(x)	1 (full range)
log10(x)	1 (full range)
log1p(x)	1 (full range)
sin(x)	2 (full range)
cos(x)	2 (full range)
tan(x)	2 (full range)
sincos(x,sptr,cptr)	2 (full range)
sinpi(x)	2 (full range)
cospi(x)	2 (full range)
sincospi(x,sptr,cptr)	2 (full range)
asin(x)	2 (full range)
acos(x)	2 (full range)
atan(x)	2 (full range)
atan2(y,x)	2 (full range)
sinh(x)	2 (full range)
cosh(x)	1 (full range)
tanh(x)	1 (full range)
asinh(x)	3 (full range)
acosh(x)	3 (full range)
atanh(x)	2 (full range)
pow(x,y)	2 (full range)
erf(x)	2 (full range)
erfc(x)	5 (full range)
erfinv(x)	5 (full range)
erfcinv(x)	6 (full range)
erfcx(x)	4 (full range)
normcdf(x)	5 (full range)

continues on next page

Table 18 – continued from previous page

Function	Maximum ulp error
`normcdfinv(x)`	8 (full range)
`lgamma(x)`	4 (outside interval -23.0001 ... -2.2637; larger inside)
`tgamma(x)`	10 (full range)
`fma(x,y,z)`	0 (IEEE-754 round-to-nearest-even)
`frexp(x,exp)`	0 (full range)
`ldexp(x,exp)`	0 (full range)
`scalbn(x,n)`	0 (full range)
`scalbln(x,l)`	0 (full range)
`logb(x)`	0 (full range)
`ilogb(x)`	0 (full range)
`j0(x)`	7 for |x| < 8 otherwise, the maximum absolute error is 5 x 10^{-12}
`j1(x)`	7 for |x| < 8 otherwise, the maximum absolute error is 5 x 10^{-12}
`jn(n,x)`	For n = 128, the maximum absolute error is 5 x 10^{-12}
`y0(x)`	7 for |x| < 8 otherwise, the maximum absolute error is 5 x 10^{-12}
`y1(x)`	7 for |x| < 8 otherwise, the maximum absolute error is 5 x 10^{-12}
`yn(n,x)`	For |x| > 1.5n, the maximum absolute error is 5 x 10^{-12}
`cyl_bessel_i0(x)`	6 (full range)
`cyl_bessel_i1(x)`	6 (full range)
`fmod(x,y)`	0 (full range)
`remainder(x,y)`	0 (full range)
`remquo(x,y,iptr)`	0 (full range)
`modf(x,iptr)`	0 (full range)
`fdim(x,y)`	0 (full range)
`trunc(x)`	0 (full range)
`round(x)`	0 (full range)

continues on next page

Table 18 – continued from previous page

Function	Maximum ulp error
`rint(x)`	0 (full range)
`nearbyint(x)`	0 (full range)
`ceil(x)`	0 (full range)
`floor(x)`	0 (full range)
`lrint(x)`	0 (full range)
`lround(x)`	0 (full range)
`llrint(x)`	0 (full range)
`llround(x)`	0 (full range)

Quad-Precision Floating-Point Functions

Note that the quad-precision mathematical functions are currently only available to devices with compute capability 10.0 and later. Due to the specifics of implementation, the support of `__float128` and `_Float128` types in device code is also limited to select combinations of host platforms, see also *Host Compiler Extensions*.

Table 19: Quad-Precision Mathematical Standard Library Functions with Maximum ULP Error. The maximum error is stated as the absolute value of the difference in ulps between the result returned by the CUDA library function and a correctly rounded quad-precision result obtained according to the round-to-nearest ties-to-even rounding mode.

Function	Maximum ulp error
`x+y __nv_fp128_add(x, y)`	0 (IEEE-754 round-to-nearest-even)
`x-y __nv_fp128_sub(x, y)`	0 (IEEE-754 round-to-nearest-even)
`x*y __nv_fp128_mul(x, y)`	0 (IEEE-754 round-to-nearest-even)
`x/y __nv_fp128_div(x, y)`	0 (IEEE-754 round-to-nearest-even)
`__nv_fp128_sqrt(x)`	0 (IEEE-754 round-to-nearest-even)
`__nv_fp128_fma(x, y, z)`	0 (IEEE-754 round-to-nearest-even)
`__nv_fp128_sin(x)`	1 (full range)
`__nv_fp128_cos(x)`	1 (full range)
`__nv_fp128_tan(x)`	1 (full range)
`__nv_fp128_asin(x)`	1 (full range)
`__nv_fp128_acos(x)`	1 (full range)
`__nv_fp128_atan(x)`	1 (full range)
`__nv_fp128_exp(x)`	1 (full range)
`__nv_fp128_exp2(x)`	1 (full range)

continues on next page

Table 19 – continued from previous page

Function	Maximum ulp error
`__nv_fp128_exp10(x)`	1 (full range)
`__nv_fp128_expm1(x)`	1 (full range)
`__nv_fp128_log(x)`	1 (full range)
`__nv_fp128_log2(x)`	1 (full range)
`__nv_fp128_log10(x)`	1 (full range)
`__nv_fp128_log1p(x)`	1 (full range)
`__nv_fp128_pow(x, y)`	1 (full range)
`__nv_fp128_sinh(x)`	1 (full range)
`__nv_fp128_cosh(x)`	1 (full range)
`__nv_fp128_tanh(x)`	1 (full range)
`__nv_fp128_asinh(x)`	1 (full range)
`__nv_fp128_acosh(x)`	1 (full range)
`__nv_fp128_atanh(x)`	1 (full range)
`__nv_fp128_hypot(x, y)`	1 (full range)
`__nv_fp128_ceil(x)`	0 (full range)
`__nv_fp128_trunc(x)`	0 (full range)
`__nv_fp128_floor(x)`	0 (full range)
`__nv_fp128_round(x)`	0 (full range)
`__nv_fp128_rint(x)`	0 (full range)
`__nv_fp128_fabs(x)`	0 (full range)
`__nv_fp128_copysign(x, y)`	0 (full range)
`__nv_fp128_fmax(x, y)`	0 (full range)
`__nv_fp128_fmin(x, y)`	0 (full range)
`__nv_fp128_fdim(x, y)`	0 (full range)
`__nv_fp128_fmod(x, y)`	0 (full range)
`__nv_fp128_remainder(x, y)`	0 (full range)
`__nv_fp128_frexp(x, nptr)`	0 (full range)
`__nv_fp128_modf(x, iptr)`	0 (full range)
`__nv_fp128_ldexp(x, exp)`	0 (full range)
`__nv_fp128_ilogb(x)`	0 (full range)

17.2. Intrinsic Functions

The functions from this section can only be used in device code.

Among these functions are the less accurate, but faster versions of some of the functions of *Standard Functions*. They have the same name prefixed with `__` (such as `__sinf(x)`). They are faster as they map to fewer native instructions. The compiler has an option (`-use_fast_math`) that forces each function in Table 20 to compile to its intrinsic counterpart. In addition to reducing the accuracy of the affected functions, it may also cause some differences in special case handling. A more robust approach is to selectively replace mathematical function calls by calls to intrinsic functions only where it is merited by the performance gains and where changed properties such as reduced accuracy and different special case handling can be tolerated.

Table 20: Functions Affected by -use_fast_math

Operator/Function	Device Function
`x/y`	`__fdividef(x,y)`
`sinf(x)`	`__sinf(x)`
`cosf(x)`	`__cosf(x)`
`tanf(x)`	`__tanf(x)`
`sincosf(x,sptr,cptr)`	`__sincosf(x,sptr,cptr)`
`logf(x)`	`__logf(x)`
`log2f(x)`	`__log2f(x)`
`log10f(x)`	`__log10f(x)`
`expf(x)`	`__expf(x)`
`exp10f(x)`	`__exp10f(x)`
`powf(x,y)`	`__powf(x,y)`
`tanhf(x)`	`__tanhf(x)`

Single-Precision Floating-Point Functions

`__fadd_[rn,rz,ru,rd]()` and `__fmul_[rn,rz,ru,rd]()` map to addition and multiplication operations that the compiler never merges into FMADs. By contrast, additions and multiplications generated from the '*' and '+' operators will frequently be combined into FMADs.

Functions suffixed with `_rn` operate using the round to nearest even rounding mode.

Functions suffixed with `_rz` operate using the round towards zero rounding mode.

Functions suffixed with `_ru` operate using the round up (to positive infinity) rounding mode.

Functions suffixed with `_rd` operate using the round down (to negative infinity) rounding mode.

The accuracy of floating-point division varies depending on whether the code is compiled with `-prec-div=false` or `-prec-div=true`. When the code is compiled with `-prec-div=false`, both the regular division `/` operator and `__fdividef(x,y)` have the same accuracy, but for $2^{126} < |y| < 2^{128}$, `__fdividef(x,y)` delivers a result of zero, whereas the `/` operator delivers the correct result to within the accuracy stated in Table 21. Also, for $2^{126} < |y| < 2^{128}$, if `x` is infinity, `__fdividef(x, y)` delivers a `NaN` (as a result of multiplying infinity by zero), while the `/` operator returns infinity. On

the other hand, the / operator is IEEE-compliant when the code is compiled with `-prec-div=true` or without any `-prec-div` option at all since its default value is true.

Table 21: Single-Precision Floating-Point Intrinsic Functions. (Supported by the CUDA Runtime Library with Respective Error Bounds)

Function	Error bounds
`__fadd_[rn,rz,ru,rd](x,y)`	IEEE-compliant.
`__fsub_[rn,rz,ru,rd](x,y)`	IEEE-compliant.
`__fmul_[rn,rz,ru,rd](x,y)`	IEEE-compliant.
`__fmaf_[rn,rz,ru,rd](x,y,z)`	IEEE-compliant.
`__frcp_[rn,rz,ru,rd](x)`	IEEE-compliant.
`__fsqrt_[rn,rz,ru,rd](x)`	IEEE-compliant.
`__frsqrt_rn(x)`	IEEE-compliant.
`__fdiv_[rn,rz,ru,rd](x,y)`	IEEE-compliant.
`__fdividef(x,y)`	For `|y|` in $[2^{-126}, 2^{126}]$, the maximum ulp error is 2.
`__expf(x)`	The maximum ulp error is `2 + floor(abs(1.173 * x))`.
`__exp10f(x)`	The maximum ulp error is `2 + floor(abs(2.97 * x))`.
`__logf(x)`	For x in [0.5, 2], the maximum absolute error is $2^{-21.41}$, otherwise, the maximum ulp error is 3.
`__log2f(x)`	For x in [0.5, 2], the maximum absolute error is 2^{-22}, otherwise, the maximum ulp error is 2.
`__log10f(x)`	For x in [0.5, 2], the maximum absolute error is 2^{-24}, otherwise, the maximum ulp error is 3.
`__sinf(x)`	For x in $[-\pi, \pi]$, the maximum absolute error is $2^{-21.41}$, and larger otherwise.
`__cosf(x)`	For x in $[-\pi, \pi]$, the maximum absolute error is $2^{-21.19}$, and larger otherwise.
`__sincosf(x,sptr,cptr)`	Same as `__sinf(x)` and `__cosf(x)`.
`__tanf(x)`	Derived from its implementation as `__sinf(x) * (1/__cosf(x))`.
`__powf(x, y)`	Derived from its implementation as `exp2f(y * __log2f(x))`.
`__tanhf(x)`	The maximum relative error of the current implementation is 2^{-11}. Subnormal results of this fast intrinsic are not flushed to zero even under `-ftz=true` compiler setting. Available for devices with compute capability of at least 7.5; and defaults to regular `tanhf()` function behavior on other devices.

Double-Precision Floating-Point Functions

`__dadd_rn()` and `__dmul_rn()` map to addition and multiplication operations that the compiler never merges into FMADs. By contrast, additions and multiplications generated from the '*' and '+' operators will frequently be combined into FMADs.

Table 22: Double-Precision Floating-Point Intrinsic Functions. (Supported by the CUDA Runtime Library with Respective Error Bounds)

Function	Error bounds
`__dadd_[rn,rz,ru,rd](x,y)`	IEEE-compliant.
`__dsub_[rn,rz,ru,rd](x,y)`	IEEE-compliant.
`__dmul_[rn,rz,ru,rd](x,y)`	IEEE-compliant.
`__fma_[rn,rz,ru,rd](x,y,z)`	IEEE-compliant.
`__ddiv_[rn,rz,ru,rd](x,y)(x,y)`	IEEE-compliant. Requires compute capability > 2.
`__drcp_[rn,rz,ru,rd](x)`	IEEE-compliant. Requires compute capability > 2.
`__dsqrt_[rn,rz,ru,rd](x)`	IEEE-compliant. Requires compute capability > 2.

Chapter 18. C++ Language Support

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

As described in *Compilation with NVCC*, CUDA source files compiled with `nvcc` can include a mix of host code and device code. The CUDA front-end compiler aims to emulate the host compiler behavior with respect to C++ input code. The input source code is processed according to the C++ ISO/IEC 14882:2003, C++ ISO/IEC 14882:2011, C++ ISO/IEC 14882:2014 or C++ ISO/IEC 14882:2017 specifications, and the CUDA front-end compiler aims to emulate any host compiler divergences from the ISO specification. In addition, the supported language is extended with CUDA-specific constructs described in this document[7], and is subject to the restrictions described below.

C++11 Language Features, *C++14 Language Features* and *C++17 Language Features* provide support matrices for the C++11, C++14, C++17 and C++20 features, respectively. *Restrictions* lists the language restrictions. *Polymorphic Function Wrappers* and *Extended Lambdas* describe additional features. *Code Samples* gives code samples.

18.1. C++11 Language Features

The following table lists new language features that have been accepted into the C++11 standard. The "Proposal" column provides a link to the ISO C++ committee proposal that describes the feature, while the "Available in nvcc (device code)" column indicates the first version of nvcc that contains an implementation of this feature (if it has been implemented) for device code.

Table 23: C++11 Language Features

Language Feature	C++11 Proposal	Available in nvcc (device code)
Rvalue references	N2118	7.0
Rvalue references for `*this`	N2439	7.0
Initialization of class objects by rvalues	N1610	7.0
Non-static data member initializers	N2756	7.0
Variadic templates	N2242	7.0

continues on next page

Table 23 – continued from previous page

Language Feature	C++11 Proposal	Available in nvcc (device code)
Extending variadic template template parameters	N2555	7.0
Initializer lists	N2672	7.0
Static assertions	N1720	7.0
`auto`-typed variables	N1984	7.0
Multi-declarator `auto`	N1737	7.0
Removal of auto as a storage-class specifier	N2546	7.0
New function declarator syntax	N2541	7.0
Lambda expressions	N2927	7.0
Declared type of an expression	N2343	7.0
Incomplete return types	N3276	7.0
Right angle brackets	N1757	7.0
Default template arguments for function templates	DR226	7.0
Solving the SFINAE problem for expressions	DR339	7.0
Alias templates	N2258	7.0
Extern templates	N1987	7.0
Null pointer constant	N2431	7.0
Strongly-typed enums	N2347	7.0
Forward declarations for enums	N2764 DR1206	7.0
Standardized attribute syntax	N2761	7.0
Generalized constant expressions	N2235	7.0
Alignment support	N2341	7.0
Conditionally-support behavior	N1627	7.0
Changing undefined behavior into diagnosable errors	N1727	7.0
Delegating constructors	N1986	7.0
Inheriting constructors	N2540	7.0
Explicit conversion operators	N2437	7.0
New character types	N2249	7.0
Unicode string literals	N2442	7.0
Raw string literals	N2442	7.0
Universal character names in literals	N2170	7.0
User-defined literals	N2765	7.0

continues on next page

Table 23 – continued from previous page

Language Feature	C++11 Proposal	Available in nvcc (device code)
Standard Layout Types	N2342	7.0
Defaulted functions	N2346	7.0
Deleted functions	N2346	7.0
Extended friend declarations	N1791	7.0
Extending `sizeof`	N2253 DR850	7.0
Inline namespaces	N2535	7.0
Unrestricted unions	N2544	7.0
Local and unnamed types as template arguments	N2657	7.0
Range-based for	N2930	7.0
Explicit virtual overrides	N2928 N3206 N3272	7.0
Minimal support for garbage collection and reachability-based leak detection	N2670	N/A (see *Restrictions*)
Allowing move constructors to throw [noexcept]	N3050	7.0
Defining move special member functions	N3053	7.0
Concurrency		
Sequence points	N2239	
Atomic operations	N2427	
Strong Compare and Exchange	N2748	
Bidirectional Fences	N2752	
Memory model	N2429	
Data-dependency ordering: atomics and memory model	N2664	
Propagating exceptions	N2179	
Allow atomics use in signal handlers	N2547	
Thread-local storage	N2659	
Dynamic initialization and destruction with concurrency	N2660	
C99 Features in C++11		
`__func__` predefined identifier	N2340	7.0
C99 preprocessor	N1653	7.0
`long long`	N1811	7.0
Extended integral types	N1988	

18.2. C++14 Language Features

The following table lists new language features that have been accepted into the C++14 standard.

Table 24: C++14 Language Features

Language Feature	C++14 Proposal	Available in nvcc (device code)
Tweak to certain C++ contextual conversions	N3323	9.0
Binary literals	N3472	9.0
Functions with deduced return type	N3638	9.0
Generalized lambda capture (init-capture)	N3648	9.0
Generic (polymorphic) lambda expressions	N3649	9.0
Variable templates	N3651	9.0
Relaxing requirements on constexpr functions	N3652	9.0
Member initializers and aggregates	N3653	9.0
Clarifying memory allocation	N3664	
Sized deallocation	N3778	
`[[deprecated]]` attribute	N3760	9.0
Single-quotation-mark as a digit separator	N3781	9.0

18.3. C++17 Language Features

All C++17 language features are supported in nvcc version 11.0 and later, subject to restrictions described *here*.

18.4. C++20 Language Features

All C++20 language features are supported in nvcc version 12.0 and later, subject to restrictions described *here*.

18.5. Restrictions

18.5.1. Host Compiler Extensions

Host compiler specific language extensions are not supported in device code.

`__Complex` types are only supported in host code.

`__int128` type is supported in device code when compiled in conjunction with a host compiler that supports it.

`__float128` type is supported for devices with compute capability 10.0 and later, when compiled in conjunction with a host compiler that supports the type. A constant expression of `__float128` type may be processed by the compiler in a floating point representation with lower precision.

18.5.2. Preprocessor Symbols

18.5.2.1 __CUDA_ARCH__

1. The type signature of the following entities shall not depend on whether `__CUDA_ARCH__` is defined or not, or on a particular value of `__CUDA_ARCH__`:
   - `__global__` functions and function templates
   - `__device__` and `__constant__` variables
   - textures and surfaces

   Example:

```
#if !defined(__CUDA_ARCH__)
typedef int mytype;
#else
typedef double mytype;
#endif

__device__ mytype xxx;         // error: xxx's type depends on __CUDA_ARCH__
__global__ void foo(mytype in, // error: foo's type depends on __CUDA_ARCH__
                    mytype *ptr)
{
  *ptr = in;
}
```

2. If a `__global__` function template is instantiated and launched from the host, then the function template must be instantiated with the same template arguments irrespective of whether `__CUDA_ARCH__` is defined and regardless of the value of `__CUDA_ARCH__`.

Example:

```
__device__ int result;
template <typename T>
__global__ void kern(T in)
{
  result = in;
}

__host__ __device__ void foo(void)
{
#if !defined(__CUDA_ARCH__)
  kern<<<1,1>>>(1);      // error: "kern<int>" instantiation only
                         // when __CUDA_ARCH__ is undefined!
#endif
}

int main(void)
{
  foo();
  cudaDeviceSynchronize();
  return 0;
}
```

3. In separate compilation mode, the presence or absence of a definition of a function or variable with external linkage shall not depend on whether `__CUDA_ARCH__` is defined or on a particular value of `__CUDA_ARCH__` Page 327, 7.

   Example:

```
#if !defined(__CUDA_ARCH__)
void foo(void) { }                  // error: The definition of foo()
                                    // is only present when __CUDA_ARCH__
                                    // is undefined
#endif
```

4. In separate compilation, `__CUDA_ARCH__` must not be used in headers such that different objects could contain different behavior. Or, it must be guaranteed that all objects will compile for the same compute_arch. If a weak function or template function is defined in a header and its behavior depends on `__CUDA_ARCH__`, then the instances of that function in the objects could conflict if the objects are compiled for different compute arch.

   For example, if an a.h contains:

```
template<typename T>
__device__ T* getptr(void)
{
#if __CUDA_ARCH__ == 700
  return NULL; /* no address */
#else
  __shared__ T arr[256];
  return arr;
#endif
}
```

   Then if `a.cu` and `b.cu` both include `a.h` and instantiate `getptr` for the same type, and `b.cu` expects a non-NULL address, and compile with:

```
nvcc -arch=compute_70 -dc a.cu
nvcc -arch=compute_80 -dc b.cu
nvcc -arch=sm_80 a.o b.o
```

At link time only one version of the `getptr` is used, so the behavior would depend on which version is chosen. To avoid this, either `a.cu` and `b.cu` must be compiled for the same compute arch, or `__CUDA_ARCH__` should not be used in the shared header function.

The compiler does not guarantee that a diagnostic will be generated for the unsupported uses of `__CUDA_ARCH__` described above.

18.5.3. Qualifiers

18.5.3.1 Device Memory Space Specifiers

The `__device__`, `__shared__`, `__managed__` and `__constant__` memory space specifiers are not allowed on:

- `class`, `struct`, and `union` data members,
- formal parameters,
- non-extern variable declarations within a function that executes on the host.

The `__device__`, `__constant__` and `__managed__` memory space specifiers are not allowed on variable declarations that are neither extern nor static within a function that executes on the device.

A `__device__`, `__constant__`, `__managed__` or `__shared__` variable definition cannot have a class type with a non-empty constructor or a non-empty destructor. A constructor for a class type is considered empty at a point in the translation unit, if it is either a trivial constructor or it satisfies all of the following conditions:

- The constructor function has been defined.
- The constructor function has no parameters, the initializer list is empty and the function body is an empty compound statement.
- Its class has no virtual functions, no virtual base classes and no non-static data member initializers.
- The default constructors of all base classes of its class can be considered empty.
- For all the nonstatic data members of its class that are of class type (or array thereof), the default constructors can be considered empty.

A destructor for a class is considered empty at a point in the translation unit, if it is either a trivial destructor or it satisfies all of the following conditions:

- The destructor function has been defined.
- The destructor function body is an empty compound statement.
- Its class has no virtual functions and no virtual base classes.
- The destructors of all base classes of its class can be considered empty.
- For all the nonstatic data members of its class that are of class type (or array thereof), the destructor can be considered empty.

When compiling in the whole program compilation mode (see the nvcc user manual for a description of this mode), `__device__`, `__shared__`, `__managed__` and `__constant__` variables cannot be defined as external using the `extern` keyword. The only exception is for dynamically allocated `__shared__` variables as described in *__shared__*.

When compiling in the separate compilation mode (see the nvcc user manual for a description of this mode), `__device__`, `__shared__`, `__managed__` and `__constant__` variables can be defined as external using the `extern` keyword. `nvlink` will generate an error when it cannot find a definition for an external variable (unless it is a dynamically allocated `__shared__` variable).

18.5.3.2 __managed__ Memory Space Specifier

Variables marked with the `__managed__` memory space specifier ("managed" variables) have the following restrictions:

- The address of a managed variable is not a constant expression.
- A managed variable shall not have a const qualified type.
- A managed variable shall not have a reference type.
- The address or value of a managed variable shall not be used when the CUDA runtime may not be in a valid state, including the following cases:
  - In static/dynamic initialization or destruction of an object with static or thread local storage duration.
  - In code that executes after exit() has been called (for example, a function marked with gcc's "`__attribute__((destructor))`").
  - In code that executes when CUDA runtime may not be initialized (for example, a function marked with gcc's "`__attribute__((constructor))`").
- A managed variable cannot be used as an unparenthesized id-expression argument to a `decltype()` expression.
- Managed variables have the same coherence and consistency behavior as specified for dynamically allocated managed memory.
- When a CUDA program containing managed variables is run on an execution platform with multiple GPUs, the variables are allocated only once, and not per GPU.
- A managed variable declaration without the extern linkage is not allowed within a function that executes on the host.
- A managed variable declaration without the extern or static linkage is not allowed within a function that executes on the device.

Here are examples of legal and illegal uses of managed variables:

```
__device__ __managed__ int xxx = 10;          // OK

int *ptr = &xxx;                              // error: use of managed variable
                                              // (xxx) in static initialization
struct S1_t {
  int field;
  S1_t(void) : field(xxx) { };
};
struct S2_t {
  ~S2_t(void) { xxx = 10; }
```

(continues on next page)

(continued from previous page)

```
};

S1_t temp1;                                 // error: use of managed variable
                                            // (xxx) in dynamic initialization

S2_t temp2;                                 // error: use of managed variable
                                            // (xxx) in the destructor of
                                            // object with static storage
                                            // duration

__device__ __managed__ const int yyy = 10;  // error: const qualified type

__device__ __managed__ int &zzz = xxx;      // error: reference type

template <int *addr> struct S3_t { };
S3_t<&xxx> temp;                            // error: address of managed
                                            // variable(xxx) not a
                                            // constant expression

__global__ void kern(int *ptr)
{
  assert(ptr == &xxx);                      // OK
  xxx = 20;                                 // OK
}
int main(void)
{
  int *ptr = &xxx;                          // OK
  kern<<<1,1>>>(ptr);
  cudaDeviceSynchronize();
  xxx++;                                    // OK
  decltype(xxx) qqq;                        // error: managed variable(xxx) used
                                            // as unparenthized argument to
                                            // decltype

  decltype((xxx)) zzz = yyy;                // OK
}
```

18.5.3.3 Volatile Qualifier

Note: The `volatile` keyword is supported to maintain compatibility with ISO C++; however, few if any of its remaining non-deprecated uses apply to GPUs.

Reads and writes to volatile qualified objects are not atomic, and are compiled to one or more .volatile instructions which do NOT guarantee:

- ordering of memory operations, or
- that the number of memory operations performed by the HW matches the number of PTX instructions.

That is, CUDA C++ volatile is not suitable for:

- **Inter-Thread Synchronization**: use atomic operations via cuda::atomic_ref, cuda::atomic, or *Atomic Functions* instead. Atomic memory operations provide inter-thread synchronization guarantees and deliver much better performance than volatile operations. CUDA C++ volatile opera-

tions do not provide any inter-thread synchronization guarantees and are therefore not correct for inter-thread synchronization. The following example shows how to pass a message across two threads using atomic operations.

cuda::atomic_ref

```
__global__ void kernel(int* flag, int* data) {
  cuda::atomic_ref<int, cuda::thread_scope_device> f{*flag};
  if (threadIdx.x == 0) {
    // Consumer: blocks until flag is set by producer, then reads data
    while(f.load(cuda::memory_order_acquire) == 0);
    if (*data != 42) __trap(); // Errors if wrong data read
  } else if (threadIdx.x == 1) {
    // Producer: writes data then sets flag
    *data = 42;
    f.store(1, cuda::memory_order_release);
  }
}
```

cuda::atomic

```
__global__ void kernel(cuda::atomic<int, cuda::thread_scope_device>*
↪flag, int* data) {
  if (threadIdx.x == 0) {
    // Consumer: blocks until flag is set by producer, then reads data
    while(flag->load(cuda::memory_order_acquire) == 0);
    if (*data != 42) __trap(); // Errors if wrong data read
  } else if (threadIdx.x == 1) {
    // Producer: writes data then sets flag
    *data = 42;
    flag->store(1, cuda::memory_order_release);
  }
}
```

Atomic Functions (`atomicAdd` and `atomicExch`)

```
__global__ void kernel(int* flag, int* data) {
  if (threadIdx.x == 0) {
    // Consumer: blocks until flag is set by producer, then reads data
    while(atomicAdd(flag, 0) == 0); // Load with Relaxed Read-Modify-
↪Write
    __threadfence();                // SequentiallyConsistent fence
    if (*data != 42) __trap();      // Errors if wrong data read
  } else if (threadIdx.x == 1) {
    // Producer: writes data then sets flag
    *data = 42;
    __threadfence();     // SequentiallyConsistent fence
    atomicExch(flag, 1); // Store with Relaxed Read-Modify-Write
  }
}
```

- **Memory Mapped IO** (MMIO): use PTX MMIO operations via inline PTX instead. PTX MMIO operations strictly preserve the number of memory accesses performed. CUDA C++ `volatile` operations do not preserve the number of memory accesses performed, and may perform more or less accesses than requested in a non-deterministic way, making them incorrect for MMIO. The following example shows how to read and write from a register using PTX mmio operations.

```
__global__ void kernel(int* mmio_reg0, int* mmio_reg1) {
  // Write to MMIO register:
  int value = 13;
  asm volatile("st.relaxed.mmio.sys.u32 [%0], %1;" :: "l"(mmio_reg0), "r
↪"(value) : "memory");

  // Read MMIO register:
  asm volatile("ld.relaxed.mmio.sys.u32 %0, [%1];" : "=r"(value) : "l
↪"(mmio_reg1) : "memory");

  if (value != 42) __trap(); // Errors if wrong data read
}
```

18.5.4. Pointers

Dereferencing a pointer either to global or shared memory in code that is executed on the host, or to host memory in code that is executed on the device results in an undefined behavior, most often in a segmentation fault and application termination.

The address obtained by taking the address of a `__device__`, `__shared__` or `__constant__` variable can only be used in device code. The address of a `__device__` or `__constant__` variable obtained through `cudaGetSymbolAddress()` as described in *Device Memory* can only be used in host code.

18.5.5. Operators

18.5.5.1 Assignment Operator

`__constant__` variables can only be assigned from the host code through runtime functions (*Device Memory*); they cannot be assigned from the device code.

`__shared__` variables cannot have an initialization as part of their declaration.

It is not allowed to assign values to any of the built-in variables defined in *Built-in Variables*.

18.5.5.2 Address Operator

It is not allowed to take the address of any of the built-in variables defined in *Built-in Variables*.

18.5.6. Run Time Type Information (RTTI)

The following RTTI-related features are supported in host code, but not in device code.

- `typeid` operator
- `std::type_info`
- `dynamic_cast` operator

18.5.7. Exception Handling

Exception handling is only supported in host code, but not in device code.

Exception specification is not supported for `__global__` functions.

18.5.8. Standard Library

Standard libraries are only supported in host code, but not in device code, unless specified otherwise.

18.5.9. Namespace Reservations

Unless an exception is otherwise noted, it is undefined behavior to add any declarations or definitions to `cuda::`, `nv::`, `cooperative_groups::` or any namespace nested within.

Examples:

```
namespace cuda{
   // Bad: class declaration added to namespace cuda
   struct foo{};

   // Bad: function definition added to namespace cuda
```

(continues on next page)

(continued from previous page)

```
   cudaStream_t make_stream(){
      cudaStream_t s;
      cudaStreamCreate(&s);
      return s;
   }
} // namespace cuda

namespace cuda{
   namespace utils{
      // Bad: function definition added to namespace nested within cuda
      cudaStream_t make_stream(){
          cudaStream_t s;
          cudaStreamCreate(&s);
          return s;
      }
   } // namespace utils
} // namespace cuda

namespace utils{
   namespace cuda{
     // Okay: namespace cuda may be used nested within a non-reserved namespace
     cudaStream_t make_stream(){
          cudaStream_t s;
          cudaStreamCreate(&s);
          return s;
      }
   } // namespace cuda
} // namespace utils

// Bad: Equivalent to adding symbols to namespace cuda at global scope
using namespace utils;
```

18.5.10. Functions

18.5.10.1 External Linkage

A call within some device code of a function declared with the extern qualifier is only allowed if the function is defined within the same compilation unit as the device code, i.e., a single file or several files linked together with relocatable device code and nvlink.

18.5.10.2 Implicitly-declared and non-virtual explicitly-defaulted functions

Let F denote a function that is either implicitly-declared or is a non-virtual function that is explicitly-defaulted on its first declaration. The execution space specifiers (`__host__`, `__device__`) for F are the union of the execution space specifiers of all the functions that invoke it (note that a `__global__` caller will be treated as a `__device__` caller for this analysis). For example:

```
class Base {
  int x;
public:
  __host__ __device__ Base(void) : x(10) {}
```

(continues on next page)

(continued from previous page)

```
};

class Derived : public Base {
  int y;
};

class Other: public Base {
  int z;
};

__device__ void foo(void)
{
  Derived D1;
  Other D2;
}

__host__ void bar(void)
{
  Other D3;
}
```

Here, the implicitly-declared constructor function “Derived::Derived” will be treated as a `__device__` function, since it is invoked only from the `__device__` function “foo”. The implicitly-declared constructor function “Other::Other” will be treated as a `__host__ __device__` function, since it is invoked both from a `__device__` function “foo” and a `__host__` function “bar”.

In addition, if F is an implicitly declared virtual function (e.g.,a virtual destructor), then the execution spaces of each virtual function D overridden by F are added to the set of execution spaces for F, if D is not implicitly declared.

For example:

```
struct Base1 { virtual __host__ __device__ ~Base1() { } };
struct Derived1 : Base1 { }; // implicitly-declared virtual destructor
                             // ~Derived1 has __host__ __device__
                             // execution space specifiers

struct Base2 { virtual __device__ ~Base2() = default; };
struct Derived2 : Base2 { }; // implicitly-declared virtual destructor
                             // ~Derived2 has __device__ execution
                             // space specifiers
```

18.5.10.3 Function Parameters

`__global__` function parameters are passed to the device via constant memory and are limited to 32,764 bytes starting with Volta, and 4 KB on older architectures.

`__global__` functions cannot have a variable number of arguments.

`__global__` function parameters cannot be pass-by-reference.

In separate compilation mode, if a `__device__` or `__global__` function is ODR-used in a particular translation unit, then the parameter and return types of the function must be complete in that translation unit.

Example:

```
//first.cu:
struct S;
__device__ void foo(S); // error: type 'S' is incomplete
__device__ auto *ptr = foo;

int main() { }

//second.cu:
struct S { int x; };
__device__ void foo(S) { }
```

```
//compiler invocation
$nvcc -std=c++14 -rdc=true first.cu second.cu -o first
nvlink error   : Prototype doesn't match for '_Z3foo1S' in '/tmp/tmpxft_00005c8c_
↪00000000-18_second.o', first defined in '/tmp/tmpxft_00005c8c_00000000-18_second.o'
nvlink fatal   : merge_elf failed
```

18.5.10.3.1 `__global__` Function Argument Processing

When a `__global__` function is launched from device code, each argument must be trivially copyable and trivially destructible.

When a `__global__` function is launched from host code, each argument type is allowed to be non-trivially copyable or non-trivially-destructible, but the processing for such types does not follow the standard C++ model, as described below. User code must ensure that this workflow does not affect program correctness. The workflow diverges from standard C++ in two areas:

1. **Memcpy instead of copy constructor invocation**

   When lowering a `__global__` function launch from host code, the compiler generates stub functions that copy the parameters one or more times by value, before eventually using `memcpy` to copy the arguments to the `__global__` function's parameter memory on the device. This occurs even if an argument was non-trivially-copyable, and therefore may break programs where the copy constructor has side effects.

   Example:

```
#include <cassert>
struct S {
 int x;
 int *ptr;
 __host__ __device__ S() { }
 __host__ __device__ S(const S &) { ptr = &x; }
};

__global__ void foo(S in) {
 // this assert may fail, because the compiler
 // generated code will memcpy the contents of "in"
 // from host to kernel parameter memory, so the
 // "in.ptr" is not initialized to "&in.x" because
 // the copy constructor is skipped.
 assert(in.ptr == &in.x);
}

int main() {
  S tmp;
```

(continues on next page)

(continued from previous page)

```
  foo<<<1,1>>>(tmp);
  cudaDeviceSynchronize();
}
```

Example:

```
#include <cassert>

__managed__ int counter;
struct S1 {
S1() { }
S1(const S1 &) { ++counter; }
};

__global__ void foo(S1) {

/* this assertion may fail, because
   the compiler generates stub
   functions on the host for a kernel
   launch, and they may copy the
   argument by value more than once.
*/
assert(counter == 1);
}

int main() {
S1 V;
foo<<<1,1>>>(V);
cudaDeviceSynchronize();
}
```

2. **Destructor may be invoked before the ``__global__`` function has finished**

   Kernel launches are asynchronous with host execution. As a result, if a `__global__` function argument has a non-trivial destructor, the destructor may execute in host code even before the `__global__` function has finished execution. This may break programs where the destructor has side effects.

   Example:

```
struct S {
 int *ptr;
 S() : ptr(nullptr) { }
 S(const S &) { cudaMallocManaged(&ptr, sizeof(int)); }
 ~S() { cudaFree(ptr); }
};

__global__ void foo(S in) {

  //error: This store may write to memory that has already been
  //       freed (see below).
  *(in.ptr) = 4;

}

int main() {
 S V;
```

(continues on next page)

(continued from previous page)

```
 /* The object 'V' is first copied by value to a compiler-generated
  * stub function that does the kernel launch, and the stub function
  * bitwise copies the contents of the argument to kernel parameter
  * memory.
  * However, GPU kernel execution is asynchronous with host
  * execution.
  * As a result, S::~S() will execute when the stub function   returns, releasing
↪allocated memory, even though the kernel may not have finished execution.
  */
 foo<<<1,1>>>(V);
 cudaDeviceSynchronize();
}
```

18.5.10.3.2 Toolkit and Driver Compatibility

Developers must use the 12.1 Toolkit and r530 driver or higher to compile, launch, and debug kernels that accept parameters larger than 4KB. If such kernels are launched on older drivers, CUDA will issue the error `CUDA_ERROR_NOT_SUPPORTED`.

18.5.10.3.3 Link Compatibility across Toolkit Revisions

When linking device objects, if at least one device object contains a kernel with a parameter larger than 4KB, the developer must recompile all objects from their respective device sources with the 12.1 toolkit or higher before linking them together. Failure to do so will result in a linker error.

18.5.10.4 Static Variables within Function

Variable memory space specifiers are allowed in the declaration of a static variable `V` within the immediate or nested block scope of a function `F` where:

- `F` is a `__global__` or `__device__`-only function.
- `F` is a `__host__ __device__` function and `__CUDA_ARCH__` is defined[11].

If no explicit memory space specifier is present in the declaration of `V`, an implicit `__device__` specifier is assumed during device compilation.

`V` has the same initialization restrictions as a variable with the same memory space specifiers declared in namespace scope for example a `__device__` variable cannot have a 'non-empty' constructor (see *Device Memory Space Specifiers*).

Examples of legal and illegal uses of function-scope static variables are shown below.

```
struct S1_t {
  int x;
};

struct S2_t {
  int x;
```

(continues on next page)

[11] The intent is to allow variable memory space specifiers for static variables in a `__host__ __device__` function during device compilation, but disallow it during host compilation

(continued from previous page)

```
  __device__ S2_t(void) { x = 10; }
};

struct S3_t {
  int x;
  __device__ S3_t(int p) : x(p) { }
};

__device__ void f1() {
  static int i1;              // OK, implicit __device__ memory space specifier
  static int i2 = 11;         // OK, implicit __device__ memory space specifier
  static __managed__ int m1;  // OK
  static __device__ int d1;   // OK
  static __constant__ int c1; // OK

  static S1_t i3;             // OK, implicit __device__ memory space specifier
  static S1_t i4 = {22};      // OK, implicit __device__ memory space specifier

  static __shared__ int i5;   // OK

  int x = 33;
  static int i6 = x;          // error: dynamic initialization is not allowed
  static S1_t i7 = {x};       // error: dynamic initialization is not allowed

  static S2_t i8;             // error: dynamic initialization is not allowed
  static S3_t i9(44);         // error: dynamic initialization is not allowed
}

__host__ __device__ void f2() {
  static int i1;              // OK, implicit __device__ memory space specifier
                              // during device compilation.
#ifdef __CUDA_ARCH__
  static __device__ int d1;   // OK, declaration is only visible during device
                              // compilation  (__CUDA_ARCH__ is defined)
#else
  static int d0;              // OK, declaration is only visible during host
                              // compilation (__CUDA_ARCH__ is not defined)
#endif

  static __device__ int d2;   // error: __device__ variable inside
                              // a host function during host compilation
                              // i.e. when __CUDA_ARCH__ is not defined

  static __shared__ int i2;  // error: __shared__ variable inside
                             // a host function during host compilation
                             // i.e. when __CUDA_ARCH__ is not defined
}
```

18.5.10.5 Function Pointers

The address of a `__global__` function taken in host code cannot be used in device code (e.g. to launch the kernel). Similarly, the address of a `__global__` function taken in device code cannot be used in host code.

It is not allowed to take the address of a `__device__` function in host code.

18.5.10.6 Function Recursion

`__global__` functions do not support recursion.

18.5.10.7 Friend Functions

A `__global__` function or function template cannot be defined in a friend declaration.

Example:

```
struct S1_t {
  friend __global__
  void foo1(void);  // OK: not a definition
  template<typename T>
  friend __global__
  void foo2(void); // OK: not a definition

  friend __global__
  void foo3(void) { } // error: definition in friend declaration

  template<typename T>
  friend __global__
  void foo4(void) { } // error: definition in friend declaration
};
```

18.5.10.8 Operator Function

An operator function cannot be a `__global__` function.

18.5.10.9 Allocation and Deallocation Functions

A user-defined `operator new`, `operator new[]`, `operator delete`, or `operator delete[]` cannot be used to replace the corresponding `__host__` or `__device__` builtins provided by the compiler.

18.5.11. Classes

18.5.11.1 Data Members

Static data members are not supported except for those that are also const-qualified (see *Const-qualified variables*).

18.5.11.2 Function Members

Static member functions cannot be `__global__` functions.

18.5.11.3 Virtual Functions

When a function in a derived class overrides a virtual function in a base class, the execution space specifiers (i.e., `__host__`, `__device__`) on the overridden and overriding functions must match.

It is not allowed to pass as an argument to a `__global__` function an object of a class with virtual functions.

If an object is created in host code, invoking a virtual function for that object in device code has undefined behavior.

If an object is created in device code, invoking a virtual function for that object in host code has undefined behavior.

See *Windows-Specific* for additional constraints when using the Microsoft host compiler.

Example:

```
struct S1 { virtual __host__ __device__ void foo() { } };

__managed__ S1 *ptr1, *ptr2;

__managed__ __align__(16) char buf1[128];
__global__ void kern() {
  ptr1->foo();     // error: virtual function call on a object
                   //        created in host code.
  ptr2 = new(buf1) S1();
}

int main(void) {
  void *buf;
  cudaMallocManaged(&buf, sizeof(S1), cudaMemAttachGlobal);
  ptr1 = new (buf) S1();
  kern<<<1,1>>>();
  cudaDeviceSynchronize();
  ptr2->foo();  // error: virtual function call on an object
                //        created in device code.
}
```

18.5.11.4 Virtual Base Classes

It is not allowed to pass as an argument to a `__global__` function an object of a class derived from virtual base classes.

See *Windows-Specific* for additional constraints when using the Microsoft host compiler.

18.5.11.5 Anonymous Unions

Member variables of a namespace scope anonymous union cannot be referenced in a `__global__` or `__device__` function.

18.5.11.6 Windows-Specific

The CUDA compiler follows the IA64 ABI for class layout, while the Microsoft host compiler does not. Let T denote a pointer to member type, or a class type that satisfies any of the following conditions:

- T has virtual functions.
- T has a virtual base class.
- T has multiple inheritance with more than one direct or indirect empty base class.
- All direct and indirect base classes B of T are empty and the type of the first field F of T uses B in its definition, such that B is laid out at offset 0 in the definition of F.

Let C denote T or a class type that has T as a field type or as a base class type. The CUDA compiler may compute the class layout and size differently than the Microsoft host compiler for the type C.

As long as the type C is used exclusively in host or device code, the program should work correctly.

Passing an object of type C between host and device code has undefined behavior, for example, as an argument to a `__global__` function or through `cudaMemcpy*()` calls.

Accessing an object of type C or any subobject in device code, or invoking a member function in device code, has undefined behavior if the object is created in host code.

Accessing an object of type C or any subobject in host code, or invoking a member function in host code, has undefined behavior if the object is created in device code[12].

18.5.12. Templates

A type or template cannot be used in the type, non-type or template template argument of a `__global__` function template instantiation or a `__device__/__constant__` variable instantiation if either:

- The type or template is defined within a `__host__` or `__host__ __device__`.
- The type or template is a class member with `private` or `protected` access and its parent class is not defined within a `__device__` or `__global__` function.
- The type is unnamed.
- The type is compounded from any of the types above.

[12] One way to debug suspected layout mismatch of a type C is to use `printf` to output the values of `sizeof(C)` and `offsetof(C, field)` in host and device code.

Example:

```
template <typename T>
__global__ void myKernel(void) { }

class myClass {
private:
    struct inner_t { };
public:
    static void launch(void)
    {
       // error: inner_t is used in template argument
       // but it is private
       myKernel<inner_t><<<1,1>>>();
    }
};

// C++14 only
template <typename T> __device__ T d1;

template <typename T1, typename T2> __device__ T1 d2;

void fn() {
  struct S1_t { };
  // error (C++14 only): S1_t is local to the function fn
  d1<S1_t> = {};

  auto lam1 = [] { };
  // error (C++14 only): a closure type cannot be used for
  // instantiating a variable template
  d2<int, decltype(lam1)> = 10;
}
```

18.5.13. Trigraphs and Digraphs

Trigraphs are not supported on any platform. Digraphs are not supported on Windows.

18.5.14. Const-qualified variables

Let 'V' denote a namespace scope variable or a class static member variable that has const qualified type and does not have execution space annotations (for example, `__device__`, `__constant__`, `__shared__`). V is considered to be a host code variable.

The value of V may be directly used in device code, if

- V has been initialized with a constant expression before the point of use,
- the type of V is not volatile-qualified, and
- it has one of the following types:
  - built-in floating point type except when the Microsoft compiler is used as the host compiler,
  - built-in integral type.

Device source code cannot contain a reference to V or take the address of V.

Example:

```
const int xxx = 10;
struct S1_t {  static const int yyy = 20; };

extern const int zzz;
const float www = 5.0;
__device__ void foo(void) {
  int local1[xxx];          // OK
  int local2[S1_t::yyy];    // OK

  int val1 = xxx;           // OK

  int val2 = S1_t::yyy;     // OK

  int val3 = zzz;           // error: zzz not initialized with constant
                            // expression at the point of use.

  const int &val3 = xxx;    // error: reference to host variable
  const int *val4 = &xxx;   // error: address of host variable
  const float val5 = www;   // OK except when the Microsoft compiler is used as
                            // the host compiler.
}
const int zzz = 20;
```

18.5.15. Long Double

The use of `long double` type is not supported in device code.

18.5.16. Deprecation Annotation

nvcc supports the use of `deprecated` attribute when using `gcc`, `clang`, `xlC`, `icc` or `pgcc` host compilers, and the use of `deprecated` declspec when using the `cl.exe` host compiler. It also supports the `[[deprecated]]` standard attribute when the C++14 dialect has been enabled. The CUDA frontend compiler will generate a deprecation diagnostic for a reference to a deprecated entity from within the body of a `__device__`, `__global__` or `__host__ __device__` function when `__CUDA_ARCH__` is defined (i.e., during device compilation phase). Other references to deprecated entities will be handled by the host compiler, e.g., a reference from within a `__host__` function.

The CUDA frontend compiler does not support the `#pragma gcc diagnostic` or `#pragma warning` mechanisms supported by various host compilers. Therefore, deprecation diagnostics generated by the CUDA frontend compiler are not affected by these pragmas, but diagnostics generated by the host compiler will be affected. To suppress the warning for device-code, user can use NVIDIA specific pragma *#pragma nv_diag_suppress*. The `nvcc` flag `-Wno-deprecated-declarations` can be used to suppress all deprecation warnings, and the flag `-Werror=deprecated-declarations` can be used to turn deprecation warnings into errors.

18.5.17. Noreturn Annotation

nvcc supports the use of `noreturn` attribute when using `gcc`, `clang`, `xlC`, `icc` or `pgcc` host compilers, and the use of `noreturn` declspec when using the `cl.exe` host compiler. It also supports the `[[noreturn]]` standard attribute when the C++11 dialect has been enabled.

The attribute/declspec can be used in both host and device code.

18.5.18. [[likely]] / [[unlikely]] Standard Attributes

These attributes are accepted in all configurations that support the C++ standard attribute syntax. The attributes can be used to hint to the device compiler optimizer whether a statement is more or less likely to be executed compared to any alternative path that does not include the statement.

Example:

```
__device__ int foo(int x) {

 if (i < 10) [[likely]] { // the 'if' block will likely be entered
  return 4;
 }
 if (i < 20) [[unlikely]] { // the 'if' block will not likely be entered
  return 1;
 }
 return 0;
}
```

If these attributes are used in host code when `__CUDA_ARCH__` is undefined, then they will be present in the code parsed by the host compiler, which may generate a warning if the attributes are not supported. For example, `clang11` host compiler will generate an 'unknown attribute' warning.

18.5.19. const and pure GNU Attributes

These attributes are supported for both host and device functions, when using a language dialect and host compiler that also supports these attributes e.g. with g++ host compiler.

For a device function annotated with the `pure` attribute, the device code optimizer assumes that the function does not change any mutable state visible to caller functions (e.g. memory).

For a device function annotated with the `const` attribute, the device code optimizer assumes that the function does not access or change any mutable state visible to caller functions (e.g. memory).

Example:

```
__attribute__((const)) __device__ int get(int in);

__device__ int doit(int in) {
int sum = 0;

//because 'get' is marked with 'const' attribute
//device code optimizer can recognize that the
//second call to get() can be commoned out.
sum = get(in);
```

(continues on next page)

(continued from previous page)

```
sum += get(in);

return sum;
}
```

18.5.20. __nv_pure__ Attribute

The `__nv_pure__` attributed is supported for both host and device functions. For host functions, when using a language dialect that supports the `pure` GNU attribute, the `__nv_pure__` attribute is translated to the `pure` GNU attribute. Similarly when using MSVC as the host compiler, the attribute is translated to the MSVC `noalias` attribute.

When a device function is annotated with the `__nv_pure__` attribute, the device code optimizer assumes that the function does not change any mutable state visible to caller functions (e.g. memory).

18.5.21. Intel Host Compiler Specific

The CUDA frontend compiler parser does not recognize some of the intrinsic functions supported by the Intel compiler (e.g. `icc`). When using the Intel compiler as a host compiler, `nvcc` will therefore enable the macro `__INTEL_COMPILER_USE_INTRINSIC_PROTOTYPES` during preprocessing. This macro enables explicit declarations of the Intel compiler intrinsic functions in the associated header files, allowing `nvcc` to support use of such functions in host code[13].

18.5.22. C++11 Features

C++11 features that are enabled by default by the host compiler are also supported by nvcc, subject to the restrictions described in this document. In addition, invoking nvcc with `-std=c++11` flag turns on all C++11 features and also invokes the host preprocessor, compiler and linker with the corresponding C++11 dialect option[14].

18.5.22.1 Lambda Expressions

The execution space specifiers for all member functions[15] of the closure class associated with a lambda expression are derived by the compiler as follows. As described in the C++11 standard, the compiler creates a closure type in the smallest block scope, class scope or namespace scope that contains the lambda expression. The innermost function scope enclosing the closure type is computed, and the corresponding function's execution space specifiers are assigned to the closure class member functions. If there is no enclosing function scope, the execution space specifier is `__host__`.

Examples of lambda expressions and computed execution space specifiers are shown below (in comments).

[13] Note that this may negatively impact compile time due to presence of extra declarations.

[14] At present, the `-std=c++11` flag is supported only for the following host compilers : gcc version >= 4.7, clang, icc >= 15, and xlc >= 13.1

[15] including `operator()`

```
auto globalVar = [] { return 0; }; // __host__

void f1(void) {
  auto l1 = [] { return 1; };      // __host__
}

__device__ void f2(void) {
  auto l2 = [] { return 2; };      // __device__
}

__host__ __device__ void f3(void) {
  auto l3 = [] { return 3; };      // __host__ __device__
}

__device__ void f4(int (*fp)() = [] { return 4; } /* __host__ */) {
}

__global__ void f5(void) {
  auto l5 = [] { return 5; };      // __device__
}

__device__ void f6(void) {
  struct S1_t {
    static void helper(int (*fp)() = [] {return 6; } /* __device__ */) {
    }
  };
}
```

The closure type of a lambda expression cannot be used in the type or non-type argument of a `__global__` function template instantiation, unless the lambda is defined within a `__device__` or `__global__` function.

Example:

```
template <typename T>
__global__ void foo(T in) { };

template <typename T>
struct S1_t { };

void bar(void) {
  auto temp1 = [] { };

  foo<<<1,1>>>(temp1);                    // error: lambda closure type used in
                                          // template type argument
  foo<<<1,1>>>( S1_t<decltype(temp1)>()); // error: lambda closure type used in
                                          // template type argument
}
```

18.5.22.2 std::initializer_list

By default, the CUDA compiler will implicitly consider the member functions of `std::initializer_list` to have `__host__ __device__` execution space specifiers, and therefore they can be invoked directly from device code. The nvcc flag `--no-host-device-initializer-list` will disable this behavior; member functions of `std::initializer_list` will then be considered as `__host__` functions and will not be directly invokable from device code.

Example:

```
#include <initializer_list>

__device__ int foo(std::initializer_list<int> in);

__device__ void bar(void)
  {
    foo({4,5,6});   // (a) initializer list containing only
                    // constant expressions.

    int i = 4;
    foo({i,5,6});   // (b) initializer list with at least one
                    // non-constant element.
                    // This form may have better performance than (a).
  }
```

18.5.22.3 Rvalue references

By default, the CUDA compiler will implicitly consider `std::move` and `std::forward` function templates to have `__host__ __device__` execution space specifiers, and therefore they can be invoked directly from device code. The nvcc flag `--no-host-device-move-forward` will disable this behavior; `std::move` and `std::forward` will then be considered as `__host__` functions and will not be directly invokable from device code.

18.5.22.4 Constexpr functions and function templates

By default, a constexpr function cannot be called from a function with incompatible execution space[16]. The experimental nvcc flag `--expt-relaxed-constexpr` removes this restriction[17]. When this flag is specified, host code can invoke a `__device__` constexpr function and device code can invoke a `__host__` constexpr function. nvcc will define the macro `__CUDACC_RELAXED_CONSTEXPR__` when `--expt-relaxed-constexpr` has been specified. Note that a function template instantiation may not be a constexpr function even if the corresponding template is marked with the keyword `constexpr` (C++11 Standard Section [`dcl.constexpr.p6`]).

[16] The restrictions are the same as with a non-constexpr callee function.
[17] Note that the behavior of experimental flags may change in future compiler releases.

18.5.22.5 Constexpr variables

Let 'V' denote a namespace scope variable or a class static member variable that has been marked constexpr and that does not have execution space annotations (e.g., `__device__`, `__constant__`, `__shared__`). V is considered to be a host code variable.

If V is of scalar type[18] other than `long double` and the type is not volatile-qualified, the value of V can be directly used in device code. In addition, if V is of a non-scalar type then scalar elements of V can be used inside a constexpr `__device__` or `__host__ __device__` function, if the call to the function is a constant expression[19]. Device source code cannot contain a reference to V or take the address of V.

Example:

```
constexpr int xxx = 10;
constexpr int yyy = xxx + 4;
struct S1_t { static constexpr int qqq = 100; };

constexpr int host_arr[] = { 1, 2, 3};
constexpr __device__ int get(int idx) { return host_arr[idx]; }

__device__ int foo(int idx) {
  int v1 = xxx + yyy + S1_t::qqq;  // OK
  const int &v2 = xxx;             // error: reference to host constexpr
                                   // variable
  const int *v3 = &xxx;            // error: address of host constexpr
                                   // variable
  const int &v4 = S1_t::qqq;       // error: reference to host constexpr
                                   // variable
  const int *v5 = &S1_t::qqq;      // error: address of host constexpr
                                   // variable

  v1 += get(2);                    // OK: 'get(2)' is a constant
                                   // expression.
  v1 += get(idx);                  // error: 'get(idx)' is not a constant
                                   // expression
  v1 += host_arr[2];               // error: 'host_arr' does not have
                                   // scalar type.
  return v1;
}
```

18.5.22.6 Inline namespaces

For an input CUDA translation unit, the CUDA compiler may invoke the host compiler for compiling the host code within the translation unit. In the code passed to the host compiler, the CUDA compiler will inject additional compiler generated code, if the input CUDA translation unit contained a definition of any of the following entities:

- `__global__` function or function template instantiation
- `__device__`, `__constant__`
- variables with surface or texture type

[18] C++ Standard Section [basic.types]
[19] C++ Standard Section [expr.const]

The compiler generated code contains a reference to the defined entity. If the entity is defined within an inline namespace and another entity of the same name and type signature is defined in an enclosing namespace, this reference may be considered ambiguous by the host compiler and host compilation will fail.

This limitation can be avoided by using unique names for such entities defined within an inline namespace.

Example:

```
__device__ int Gvar;
inline namespace N1 {
  __device__ int Gvar;
}

// <-- CUDA compiler inserts a reference to "Gvar" at this point in the
// translation unit. This reference will be considered ambiguous by the
// host compiler and compilation will fail.
```

Example:

```
inline namespace N1 {
  namespace N2 {
    __device__ int Gvar;
  }
}

namespace N2 {
  __device__ int Gvar;
}

// <-- CUDA compiler inserts reference to "::N2::Gvar" at this point in
// the translation unit. This reference will be considered ambiguous by
// the host compiler and compilation will fail.
```

18.5.22.6.1 Inline unnamed namespaces

The following entities cannot be declared in namespace scope within an inline unnamed namespace:

- `__managed__`, `__device__`, `__shared__` and `__constant__` variables
- `__global__` function and function templates
- variables with surface or texture type

Example:

```
inline namespace {
  namespace N2 {
    template <typename T>
    __global__ void foo(void);            // error

    __global__ void bar(void) { }         // error

    template <>
    __global__ void foo<int>(void) { }    // error

    __device__ int x1b;                   // error
```

(continues on next page)

(continued from previous page)

```
    __constant__ int x2b;                // error
    __shared__ int x3b;                  // error

    texture<int> q2;                     // error
    surface<int> s2;                     // error
  }
};
```

18.5.22.7 thread_local

The `thread_local` storage specifier is not allowed in device code.

18.5.22.8 __global__ functions and function templates

If the closure type associated with a lambda expression is used in a template argument of a `__global__` function template instantiation, the lambda expression must either be defined in the immediate or nested block scope of a `__device__` or `__global__` function, or must be an *extended lambda*.

Example:

```
template <typename T>
__global__ void kernel(T in) { }

__device__ void foo_device(void)
{
  // All kernel instantiations in this function
  // are valid, since the lambdas are defined inside
  // a __device__ function.

  kernel<<<1,1>>>( [] __device__ { } );
  kernel<<<1,1>>>( [] __host__ __device__ { } );
  kernel<<<1,1>>>( []  { } );
}

auto lam1 = [] { };

auto lam2 = [] __host__ __device__ { };

void foo_host(void)
{
   // OK: instantiated with closure type of an extended __device__ lambda
   kernel<<<1,1>>>( [] __device__ { } );

   // OK: instantiated with closure type of an extended __host__ __device__
   // lambda
   kernel<<<1,1>>>( [] __host__ __device__ { } );

   // error: unsupported: instantiated with closure type of a lambda
   // that is not an extended lambda
   kernel<<<1,1>>>( []  { } );

   // error: unsupported: instantiated with closure type of a lambda
```

(continues on next page)

(continued from previous page)

```
   // that is not an extended lambda
   kernel<<<1,1>>>( lam1);

   // error: unsupported: instantiated with closure type of a lambda
   // that is not an extended lambda
   kernel<<<1,1>>>( lam2);
}
```

A `__global__` function or function template cannot be declared as `constexpr`.

A `__global__` function or function template cannot have a parameter of type `std::initializer_list` or `va_list`.

A `__global__` function cannot have a parameter of rvalue reference type.

A variadic `__global__` function template has the following restrictions:

- Only a single pack parameter is allowed.
- The pack parameter must be listed last in the template parameter list.

Example:

```
// ok
template <template <typename...> class Wrapper, typename... Pack>
__global__ void foo1(Wrapper<Pack...>);

// error: pack parameter is not last in parameter list
template <typename... Pack, template <typename...> class Wrapper>
__global__ void foo2(Wrapper<Pack...>);

// error: multiple parameter packs
template <typename... Pack1, int...Pack2, template<typename...> class Wrapper1,
          template<int...> class Wrapper2>
__global__ void foo3(Wrapper1<Pack1...>, Wrapper2<Pack2...>);
```

18.5.22.9 __managed__ and __shared__ variables

`__managed__ and `__shared__` variables cannot be marked with the keyword `constexpr`.

18.5.22.10 Defaulted functions

Execution space specifiers on a non-virtual function that is explicitly-defaulted on its first declaration are ignored by the CUDA compiler. Instead, the CUDA compiler will infer the execution space specifiers as described in *Implicitly-declared and non-virtual explicitly-defaulted functions*.

Execution space specifiers are not ignored if the function is either:

- Explicitly-defaulted but not on its first declaration.
- Explicitly-defaulted and virtual.

Example:

```
struct S1 {
  // warning: __host__ annotation is ignored on a non-virtual function that
  //          is explicitly-defaulted on its first declaration
```

(continues on next page)

(continued from previous page)

```
  __host__ S1() = default;
};

__device__ void foo1() {
  //note: __device__ execution space is derived for S1::S1
  //       based on implicit call from within __device__ function
  //       foo1
  S1 s1;
}

struct S2 {
  __host__ S2();
};

//note: S2::S2 is not defaulted on its first declaration, and
//      its execution space is fixed to __host__  based on its
//      first declaration.
S2::S2() = default;

__device__ void foo2() {
   // error: call from __device__ function 'foo2' to
   //        __host__ function 'S2::S2'
   S2 s2;
}

struct S3 {
  //note: S3::~S3 has __host__ execution space
  virtual __host__ ~S3() = default;
};

__device__ void foo3() {
  S3 qqq;
}  /*(implicit destructor call for 'qqq'):
      error: call from a __device__ fuction 'foo3' to a
     __host__ function 'S3::~S3' */
```

18.5.23. C++14 Features

C++14 features enabled by default by the host compiler are also supported by nvcc. Passing nvcc `-std=c++14` flag turns on all C++14 features and also invokes the host preprocessor, compiler and linker with the corresponding C++14 dialect option[20]. This section describes the restrictions on the supported C++14 features.

[20] At present, the `-std=c++14` flag is supported only for the following host compilers : gcc version >= 5.1, clang version >= 3.7 and icc version >= 17

18.5.23.1 Functions with deduced return type

A `__global__` function cannot have a deduced return type.

If a `__device__` function has deduced return type, the CUDA frontend compiler will change the function declaration to have a `void` return type, before invoking the host compiler. This may cause issues for introspecting the deduced return type of the `__device__` function in host code. Thus, the CUDA compiler will issue compile-time errors for referencing such deduced return type outside device function bodies, except if the reference is absent when `__CUDA_ARCH__` is undefined.

Examples:

```
__device__ auto fn1(int x) {
  return x;
}

__device__ decltype(auto) fn2(int x) {
  return x;
}

__device__ void device_fn1() {
  // OK
  int (*p1)(int) = fn1;
}

// error: referenced outside device function bodies
decltype(fn1(10)) g1;

void host_fn1() {
  // error: referenced outside device function bodies
  int (*p1)(int) = fn1;

  struct S_local_t {
    // error: referenced outside device function bodies
    decltype(fn2(10)) m1;

    S_local_t() : m1(10) { }
  };
}

// error: referenced outside device function bodies
template <typename T = decltype(fn2)>
void host_fn2() { }

template<typename T> struct S1_t { };

// error: referenced outside device function bodies
struct S1_derived_t : S1_t<decltype(fn1)> { };
```

18.5.23.2 Variable templates

A `__device__/__constant__` variable template cannot have a const qualified type when using the Microsoft host compiler.

Examples:

```
// error: a __device__ variable template cannot
// have a const qualified type on Windows
template <typename T>
__device__ const T d1(2);

int *const x = nullptr;
// error: a __device__ variable template cannot
// have a const qualified type on Windows
template <typename T>
__device__ T *const d2(x);

// OK
template <typename T>
__device__ const T *d3;

__device__ void fn() {
  int t1 = d1<int>;

  int *const t2 = d2<int>;

  const int *t3 = d3<int>;
}
```

18.5.24. C++17 Features

C++17 features enabled by default by the host compiler are also supported by nvcc. Passing nvcc `-std=c++17` flag turns on all C++17 features and also invokes the host preprocessor, compiler and linker with the corresponding C++17 dialect option[21]. This section describes the restrictions on the supported C++17 features.

18.5.24.1 Inline Variable

- A namespace scope inline variable declared with `__device__` or `__constant__` or `__managed__` memory space specifier must have internal linkage, if the code is compiled with nvcc in whole program compilation mode.

  Examples:

  ```
  inline __device__ int xxx; //error when compiled with nvcc in
                             //whole program compilation mode.
                             //ok when compiled with nvcc in
                             //separate compilation mode.

  inline __shared__ int yyy0; // ok.
  ```

(continues on next page)

[21] At present, the `-std=c++17` flag is supported only for the following host compilers : gcc version >= 7.0, clang version >= 8.0, Visual Studio version >= 2017, pgi compiler version >= 19.0, icc compiler version >= 19.0

(continued from previous page)

```
static inline __device__ int yyy; // ok: internal linkage
namespace {
inline __device__ int zzz; // ok: internal linkage
}
```

- When using g++ host compiler, an inline variable declared with `__managed__` memory space specifier may not be visible to the debugger.

18.5.24.2 Structured Binding

A structured binding cannot be declared with a variable memory space specifier.

Example:

```
struct S { int x; int y; };
__device__ auto [a1, b1] = S{4,5}; // error
```

18.5.25. C++20 Features

C++20 features enabled by default by the host compiler are also supported by nvcc. Passing nvcc `-std=c++20` flag turns on all C++20 features and also invokes the host preprocessor, compiler and linker with the corresponding C++20 dialect option[22]. This section describes the restrictions on the supported C++20 features.

18.5.25.1 Module support

Modules are not supported in CUDA C++, in either host or device code. Uses of the `module`, `export` and `import` keywords are diagnosed as errors.

18.5.25.2 Coroutine support

Coroutines are not supported in device code. Uses of the `co_await`, `co_yield` and `co_return` keywords in the scope of a device function are diagnosed as error during device compilation.

18.5.25.3 Three-way comparison operator

The three-way comparison operator is supported in both host and device code, but some uses implicitly rely on functionality from the Standard Template Library provided by the host implementation. Uses of those operators may require specifying the flag `--expt-relaxed-constexpr` to silence warnings and the functionality requires that the host implementation satisfies the requirements of device code.

Example:

[22] At present, the `-std=c++20` flag is supported only for the following host compilers : gcc version >= 10.0, clang version >= 10.0, Visual Studio Version >= 2022 and nvc++ version >= 20.7.

```
#include<compare>
struct S {
  int x, y, z;
  auto operator<=>(const S& rhs) const = default;
  __host__ __device__ bool operator<=>(int rhs) const { return false; }
};
__host__ __device__ bool f(S a, S b) {
  if (a <=> 1) // ok, calls a user-defined host-device overload
    return true;
  return a < b; // call to an implicitly-declared function and requires
                // a device-compatible std::strong_ordering implementation
}
```

18.5.25.4 Consteval functions

Ordinarily, cross execution space calls are not allowed, and cause a compiler diagnostic (warning or error). This restriction does not apply when the called function is declared with the `consteval` specifier. Thus, a `__device__` or `__global__` function can call a `__host__consteval` function, and a `__host__` function can call a `__device__ consteval` function.

Example:

```
namespace N1 {
//consteval host function
consteval int hcallee() { return 10; }

__device__ int dfunc() { return hcallee(); /* OK */ }
__global__ void gfunc() { (void)hcallee(); /* OK */ }
__host__ __device__ int hdfunc() { return hcallee();  /* OK */ }
int hfunc() { return hcallee(); /* OK */ }
} // namespace N1


namespace N2 {
//consteval device function
consteval __device__ int dcallee() { return 10; }

__device__ int dfunc() { return dcallee(); /* OK */ }
__global__ void gfunc() { (void)dcallee(); /* OK */ }
__host__ __device__ int hdfunc() { return dcallee();  /* OK */ }
int hfunc() { return dcallee(); /* OK */ }
}
```

18.6. Polymorphic Function Wrappers

A polymorphic function wrapper class template `nvstd::function` is provided in the `nvfunctional` header. Instances of this class template can be used to store, copy and invoke any callable target, e.g., lambda expressions. `nvstd::function` can be used in both host and device code.

Example:

```
#include <nvfunctional>

__device__ int foo_d() { return 1; }
__host__ __device__ int foo_hd () { return 2; }
__host__ int foo_h() { return 3; }

__global__ void kernel(int *result) {
  nvstd::function<int()> fn1 = foo_d;
  nvstd::function<int()> fn2 = foo_hd;
  nvstd::function<int()> fn3 =  []() { return 10; };

  *result = fn1() + fn2() + fn3();
}

__host__ __device__ void hostdevice_func(int *result) {
  nvstd::function<int()> fn1 = foo_hd;
  nvstd::function<int()> fn2 =  []() { return 10; };

  *result = fn1() + fn2();
}

__host__ void host_func(int *result) {
  nvstd::function<int()> fn1 = foo_h;
  nvstd::function<int()> fn2 = foo_hd;
  nvstd::function<int()> fn3 =  []() { return 10; };

  *result = fn1() + fn2() + fn3();
}
```

Instances of `nvstd::function` in host code cannot be initialized with the address of a `__device__` function or with a functor whose `operator()` is a `__device__` function. Instances of `nvstd::function` in device code cannot be initialized with the address of a `__host__` function or with a functor whose `operator()` is a `__host__` function.

`nvstd::function` instances cannot be passed from host code to device code (and vice versa) at run time. `nvstd::function` cannot be used in the parameter type of a `__global__` function, if the `__global__` function is launched from host code.

Example:

```
#include <nvfunctional>

__device__ int foo_d() { return 1; }
__host__ int foo_h() { return 3; }
auto lam_h = [] { return 0; };

__global__ void k(void) {
  // error: initialized with address of __host__ function
  nvstd::function<int()> fn1 = foo_h;

  // error: initialized with address of functor with
  // __host__ operator() function
  nvstd::function<int()> fn2 = lam_h;
}

__global__ void kern(nvstd::function<int()> f1) { }

void foo(void) {
```

(continues on next page)

(continued from previous page)

```
  // error: initialized with address of __device__ function
  nvstd::function<int()> fn1 = foo_d;

  auto lam_d = [=] __device__ { return 1; };

  // error: initialized with address of functor with
  // __device__ operator() function
  nvstd::function<int()> fn2 = lam_d;

  // error: passing nvstd::function from host to device
  kern<<<1,1>>>(fn2);
}
```

nvstd::function is defined in the nvfunctional header as follows:

```
namespace nvstd {
  template <class _RetType, class ..._ArgTypes>
  class function<_RetType(_ArgTypes...)>
  {
    public:
      // constructors
      __device__ __host__  function() noexcept;
      __device__ __host__  function(nullptr_t) noexcept;
      __device__ __host__  function(const function &);
      __device__ __host__  function(function &&);

      template<class _F>
      __device__ __host__  function(_F);

      // destructor
      __device__ __host__  ~function();

      // assignment operators
      __device__ __host__  function& operator=(const function&);
      __device__ __host__  function& operator=(function&&);
      __device__ __host__  function& operator=(nullptr_t);
      __device__ __host__  function& operator=(_F&&);

      // swap
      __device__ __host__  void swap(function&) noexcept;

      // function capacity
      __device__ __host__  explicit operator bool() const noexcept;

      // function invocation
      __device__ _RetType operator()(_ArgTypes...) const;
  };

  // null pointer comparisons
  template <class _R, class... _ArgTypes>
  __device__ __host__
  bool operator==(const function<_R(_ArgTypes...)>&, nullptr_t) noexcept;

  template <class _R, class... _ArgTypes>
  __device__ __host__
  bool operator==(nullptr_t, const function<_R(_ArgTypes...)>&) noexcept;
```

(continues on next page)

(continued from previous page)

```
  template <class _R, class... _ArgTypes>
  __device__ __host__
  bool operator!=(const function<_R(_ArgTypes...)>&, nullptr_t) noexcept;

  template <class _R, class... _ArgTypes>
  __device__ __host__
  bool operator!=(nullptr_t, const function<_R(_ArgTypes...)>&) noexcept;

  // specialized algorithms
  template <class _R, class... _ArgTypes>
  __device__ __host__
  void swap(function<_R(_ArgTypes...)>&, function<_R(_ArgTypes...)>&);
}
```

18.7. Extended Lambdas

The nvcc flag '`--extended-lambda`' allows explicit execution space annotations in a lambda expression[23]. The execution space annotations should be present after the 'lambda-introducer' and before the optional 'lambda-declarator'. nvcc will define the macro `__CUDACC_EXTENDED_LAMBDA__` when the '`--extended-lambda`' flag has been specified.

An 'extended `__device__` lambda' is a lambda expression that is annotated explicitly with '`__device__`', and is defined within the immediate or nested block scope of a `__host__` or `__host__ __device__` function.

An 'extended `__host__ __device__` lambda' is a lambda expression that is annotated explicitly with both '`__host__`' and '`__device__`', and is defined within the immediate or nested block scope of a `__host__` or `__host__ __device__` function.

An 'extended lambda' denotes either an extended `__device__` lambda or an extended `__host__ __device__` lambda. Extended lambdas can be used in the type arguments of *__global__ function template instantiation*.

If the execution space annotations are not explicitly specified, they are computed based on the scopes enclosing the closure class associated with the lambda, as described in the section on C++11 support. The execution space annotations are applied to all methods of the closure class associated with the lambda.

Example:

```
void foo_host(void) {
  // not an extended lambda: no explicit execution space annotations
  auto lam1 = [] { };

  // extended __device__ lambda
  auto lam2 = [] __device__ { };

  // extended __host__ __device__ lambda
  auto lam3 = [] __host__ __device__ { };

  // not an extended lambda: explicitly annotated with only '__host__'
  auto lam4 = [] __host__ { };
```

(continues on next page)

[23] When using the icc host compiler, this flag is only supported for icc >= 1800.

(continued from previous page)

```
}

__host__ __device__ void foo_host_device(void) {
  // not an extended lambda: no explicit execution space annotations
  auto lam1 = [] { };

  // extended __device__ lambda
  auto lam2 = [] __device__ { };

  // extended __host__ __device__ lambda
  auto lam3 = [] __host__ __device__ { };

  // not an extended lambda: explicitly annotated with only '__host__'
  auto lam4 = [] __host__ { };
}

__device__ void foo_device(void) {
  // none of the lambdas within this function are extended lambdas,
  // because the enclosing function is not a __host__ or __host__ __device__
  // function.
  auto lam1 = [] { };
  auto lam2 = [] __device__ { };
  auto lam3 = [] __host__ __device__ { };
  auto lam4 = [] __host__ { };
}

// lam1 and lam2 are not extended lambdas because they are not defined
// within a __host__ or __host__ __device__ function.
auto lam1 = [] { };
auto lam2 = [] __host__ __device__ { };
```

18.7.1. Extended Lambda Type Traits

The compiler provides type traits to detect closure types for extended lambdas at compile time:

`__nv_is_extended_device_lambda_closure_type(type)`: If 'type' is the closure class created for an extended `__device__` lambda, then the trait is true, otherwise it is false.

`__nv_is_extended_device_lambda_with_preserved_return_type(type)`: If 'type' is the closure class created for an extended `__device__` lambda and the lambda is defined with trailing return type (with restriction), then the trait is true, otherwise it is false. If the trailing return type definition refers to any lambda parameter name, the return type is not preserved.

`__nv_is_extended_host_device_lambda_closure_type(type)`: If 'type' is the closure class created for an extended `__host__ __device__` lambda, then the trait is true, otherwise it is false.

These traits can be used in all compilation modes, irrespective of whether lambdas or extended lambdas are enabled[24].

Example:

```
#define IS_D_LAMBDA(X) __nv_is_extended_device_lambda_closure_type(X)
#define IS_DPRT_LAMBDA(X) __nv_is_extended_device_lambda_with_preserved_return_type(X)
```

(continues on next page)

[24] The traits will always return false if extended lambda mode is not active.

(continued from previous page)

```
#define IS_HD_LAMBDA(X) __nv_is_extended_host_device_lambda_closure_type(X)

auto lam0 = [] __host__ __device__ { };

void foo(void) {
  auto lam1 = [] { };
  auto lam2 = [] __device__ { };
  auto lam3 = [] __host__ __device__ { };
  auto lam4 = [] __device__ () --> double { return 3.14; }
  auto lam5 = [] __device__ (int x) --> decltype(&x) { return 0; }

  // lam0 is not an extended lambda (since defined outside function scope)
  static_assert(!IS_D_LAMBDA(decltype(lam0)), "");
  static_assert(!IS_DPRT_LAMBDA(decltype(lam0)), "");
  static_assert(!IS_HD_LAMBDA(decltype(lam0)), "");

  // lam1 is not an extended lambda (since no execution space annotations)
  static_assert(!IS_D_LAMBDA(decltype(lam1)), "");
  static_assert(!IS_DPRT_LAMBDA(decltype(lam1)), "");
  static_assert(!IS_HD_LAMBDA(decltype(lam1)), "");

  // lam2 is an extended __device__ lambda
  static_assert(IS_D_LAMBDA(decltype(lam2)), "");
  static_assert(!IS_DPRT_LAMBDA(decltype(lam2)), "");
  static_assert(!IS_HD_LAMBDA(decltype(lam2)), "");

  // lam3 is an extended __host__ __device__ lambda
  static_assert(!IS_D_LAMBDA(decltype(lam3)), "");
  static_assert(!IS_DPRT_LAMBDA(decltype(lam3)), "");
  static_assert(IS_HD_LAMBDA(decltype(lam3)), "");

  // lam4 is an extended __device__ lambda with preserved return type
  static_assert(IS_D_LAMBDA(decltype(lam4)), "");
  static_assert(IS_DPRT_LAMBDA(decltype(lam4)), "");
  static_assert(!IS_HD_LAMBDA(decltype(lam4)), "");

  // lam5 is not an extended __device__ lambda with preserved return type
  // because it references the operator()'s parameter types in the trailing return type.
  static_assert(IS_D_LAMBDA(decltype(lam5)), "");
  static_assert(!IS_DPRT_LAMBDA(decltype(lam5)), "");
  static_assert(!IS_HD_LAMBDA(decltype(lam5)), "");
}
```

18.7.2. Extended Lambda Restrictions

The CUDA compiler will replace an extended lambda expression with an instance of a placeholder type defined in namespace scope, before invoking the host compiler. The template argument of the placeholder type requires taking the address of a function enclosing the original extended lambda expression. This is required for the correct execution of any `__global__` function template whose template argument involves the closure type of an extended lambda. The *enclosing function* is computed as follows.

By definition, the extended lambda is present within the immediate or nested block scope of a `__host__` or `__host__ __device__` function. If this function is not the `operator()` of a lambda

expression, then it is considered the enclosing function for the extended lambda. Otherwise, the extended lambda is defined within the immediate or nested block scope of the `operator()` of one or more enclosing lambda expressions. If the outermost such lambda expression is defined in the immediate or nested block scope of a function F, then F is the computed enclosing function, else the enclosing function does not exist.

Example:

```
void foo(void) {
  // enclosing function for lam1 is "foo"
  auto lam1 = [] __device__ { };

  auto lam2 = [] {
     auto lam3 = [] {
        // enclosing function for lam4 is "foo"
        auto lam4 = [] __host__ __device__ { };
     };
  };
}

auto lam6 = [] {
  // enclosing function for lam7 does not exist
  auto lam7 = [] __host__ __device__ { };
};
```

Here are the restrictions on extended lambdas:

1. An extended lambda cannot be defined inside another extended lambda expression.

   Example:

   ```
   void foo(void) {
     auto lam1 = [] __host__ __device__  {
       // error: extended lambda defined within another extended lambda
       auto lam2 = [] __host__ __device__ { };
     };
   }
   ```

2. An extended lambda cannot be defined inside a generic lambda expression.

   Example:

   ```
   void foo(void) {
     auto lam1 = [] (auto) {
       // error: extended lambda defined within a generic lambda
       auto lam2 = [] __host__ __device__ { };
     };
   }
   ```

3. If an extended lambda is defined within the immediate or nested block scope of one or more nested lambda expression, the outermost such lambda expression must be defined inside the immediate or nested block scope of a function.

   Example:

   ```
   auto lam1 = []  {
     // error: outer enclosing lambda is not defined within a
     // non-lambda-operator() function.
     auto lam2 = [] __host__ __device__ { };
   };
   ```

4. The enclosing function for the extended lambda must be named and its address can be taken. If the enclosing function is a class member, then the following conditions must be satisfied:
   - All classes enclosing the member function must have a name.
   - The member function must not have private or protected access within its parent class.
   - All enclosing classes must not have private or protected access within their respective parent classes.

   Example:

```
void foo(void) {
  // OK
  auto lam1 = [] __device__ { return 0; };
  {
    // OK
    auto lam2 = [] __device__ { return 0; };
    // OK
    auto lam3 = [] __device__ __host__ { return 0; };
  }
}

struct S1_t {
  S1_t(void) {
    // Error: cannot take address of enclosing function
    auto lam4 = [] __device__ { return 0; };
  }
};

class C0_t {
  void foo(void) {
    // Error: enclosing function has private access in parent class
    auto temp1 = [] __device__ { return 10; };
  }
  struct S2_t {
    void foo(void) {
      // Error: enclosing class S2_t has private access in its
      // parent class
      auto temp1 = [] __device__ { return 10; };
    }
  };
};
```

5. It must be possible to take the address of the enclosing routine unambiguously, at the point where the extended lambda has been defined. This may not be feasible in some cases e.g. when a class typedef shadows a template type argument of the same name.

   Example:

```
template <typename> struct A {
  typedef void Bar;
  void test();
};

template<> struct A<void> { };

template <typename Bar>
void A<Bar>::test() {
```

(continues on next page)

(continued from previous page)

```
  /* In code sent to host compiler, nvcc will inject an
     address expression here, of the form:
     (void (A< Bar> ::*)(void))(&A::test))

     However, the class typedef 'Bar' (to void) shadows the
     template argument 'Bar', causing the address
     expression in A<int>::test to actually refer to:
     (void (A< void> ::*)(void))(&A::test))

     ..which doesn't take the address of the enclosing
     routine 'A<int>::test' correctly.
  */
  auto lam1 = [] __host__ __device__ { return 4; };
}

int main() {
  A<int> xxx;
  xxx.test();
}
```

6. An extended lambda cannot be defined in a class that is local to a function.

   Example:

```
void foo(void) {
  struct S1_t {
    void bar(void) {
      // Error: bar is member of a class that is local to a function.
      auto lam4 = [] __host__ __device__ { return 0; };
    }
  };
}
```

7. The enclosing function for an extended lambda cannot have deduced return type.

   Example:

```
auto foo(void) {
  // Error: the return type of foo is deduced.
  auto lam1 = [] __host__ __device__ { return 0; };
}
```

8. __host__ __device__ extended lambdas cannot be generic lambdas.

   Example:

```
void foo(void) {
  // Error: __host__ __device__ extended lambdas cannot be
  // generic lambdas.
  auto lam1 = [] __host__ __device__ (auto i) { return i; };

  // Error: __host__ __device__ extended lambdas cannot be
  // generic lambdas.
  auto lam2 = [] __host__ __device__ (auto ...i) {
               return sizeof...(i);
              };
}
```

9. If the enclosing function is an instantiation of a function template or a member function template, and/or the function is a member of a class template, the template(s) must satisfy the following constraints:

   - The template must have at most one variadic parameter, and it must be listed last in the template parameter list.
   - The template parameters must be named.
   - The template instantiation argument types cannot involve types that are either local to a function (except for closure types for extended lambdas), or are private or protected class members.

   Example:

```
template <typename T>
__global__ void kern(T in) { in(); }

template <typename... T>
struct foo {};

template < template <typename...> class T, typename... P1,
          typename... P2>
void bar1(const T<P1...>, const T<P2...>) {
  // Error: enclosing function has multiple parameter packs
  auto lam1 =  [] __device__ { return 10; };
}

template < template <typename...> class T, typename... P1,
          typename T2>
void bar2(const T<P1...>, T2) {
  // Error: for enclosing function, the
  // parameter pack is not last in the template parameter list.
  auto lam1 =  [] __device__ { return 10; };
}

template <typename T, T>
void bar3(void) {
  // Error: for enclosing function, the second template
  // parameter is not named.
  auto lam1 =  [] __device__ { return 10; };
}

int main() {
  foo<char, int, float> f1;
  foo<char, int> f2;
  bar1(f1, f2);
  bar2(f1, 10);
  bar3<int, 10>();
}
```

   Example:

```
template <typename T>
__global__ void kern(T in) { in(); }

template <typename T>
void bar4(void) {
  auto lam1 =  [] __device__ { return 10; };
```

(continues on next page)

(continued from previous page)

```
  kern<<<1,1>>>(lam1);
}

struct C1_t { struct S1_t { }; friend int main(void); };
int main() {
  struct S1_t { };
  // Error: enclosing function for device lambda in bar4
  // is instantiated with a type local to main.
  bar4<S1_t>();

  // Error: enclosing function for device lambda in bar4
  // is instantiated with a type that is a private member
  // of a class.
  bar4<C1_t::S1_t>();
}
```

10. With Visual Studio host compilers, the enclosing function must have external linkage. The restriction is present because this host compiler does not support using the address of non-extern linkage functions as template arguments, which is needed by the CUDA compiler transformations to support extended lambdas.
11. With Visual Studio host compilers, an extended lambda shall not be defined within the body of an 'if-constexpr' block.
12. An extended lambda has the following restrictions on captured variables:
    - In the code sent to the host compiler, the variable may be passed by value to a sequence of helper functions before being used to direct-initialize the field of the class type used to represent the closure type for the extended lambda[25].
    - A variable can only be captured by value.
    - A variable of array type cannot be captured if the number of array dimensions is greater than 7.
    - For a variable of array type, in the code sent to the host compiler, the closure type's array field is first default-initialized, and then each element of the array field is copy-assigned from the corresponding element of the captured array variable. Therefore, the array element type must be default-constructible and copy-assignable in host code.
    - A function parameter that is an element of a variadic argument pack cannot be captured.
    - The type of the captured variable cannot involve types that are either local to a function (except for closure types of extended lambdas), or are private or protected class members.
    - For a __host__ __device__ extended lambda, the types used in the return or parameter types of the lambda expression's `operator()` cannot involve types that are either local to a function (except for closure types of extended lambdas), or are private or protected class members.
    - Init-capture is not supported for __host__ __device__ extended lambdas. Init-capture is supported for __device__ extended lambdas, except when the init-capture is of array type or of type `std::initializer_list`.
    - The function call operator for an extended lambda is not constexpr. The closure type for an extended lambda is not a literal type. The constexpr and consteval specifier cannot be used in the declaration of an extended lambda.

[25] In contrast, the C++ standard specifies that the captured variable is used to direct-initialize the field of the closure type.

- A variable cannot be implicitly captured inside an if-constexpr block lexically nested inside an extended lambda, unless it has already been implicitly captured earlier outside the if-constexpr block or appears in the explicit capture list for the extended lambda (see example below).

Example

```
void foo(void) {
  // OK: an init-capture is allowed for an
  // extended __device__ lambda.
  auto lam1 = [x = 1] __device__ () { return x; };

  // Error: an init-capture is not allowed for
  // an extended __host__ __device__ lambda.
  auto lam2 = [x = 1] __host__ __device__ () { return x; };

  int a = 1;
  // Error: an extended __device__ lambda cannot capture
  // variables by reference.
  auto lam3 = [&a] __device__ () { return a; };

  // Error: by-reference capture is not allowed
  // for an extended __device__ lambda.
  auto lam4 = [&x = a] __device__ () { return x; };

  struct S1_t { };
  S1_t s1;
  // Error: a type local to a function cannot be used in the type
  // of a captured variable.
  auto lam6 = [s1] __device__ () { };

  // Error: an init-capture cannot be of type std::initializer_list.
  auto lam7 = [x = {11}] __device__ () { };

  std::initializer_list<int> b = {11,22,33};
  // Error: an init-capture cannot be of type std::initializer_list.
  auto lam8 = [x = b] __device__ () { };

  // Error scenario (lam9) and supported scenarios (lam10, lam11)
  // for capture within 'if-constexpr' block
  int yyy = 4;
  auto lam9 = [=] __device__ {
    int result = 0;
    if constexpr(false) {
      //Error: An extended __device__ lambda cannot first-capture
      //       'yyy' in constexpr-if context
      result += yyy;
    }
    return result;
  };

  auto lam10 = [yyy] __device__ {
    int result = 0;
    if constexpr(false) {
      //OK: 'yyy' already listed in explicit capture list for the extended lambda
      result += yyy;
    }
    return result;
```

(continues on next page)

(continued from previous page)

```
  };

  auto lam11 = [=] __device__ {
    int result = yyy;
    if constexpr(false) {
      //OK: 'yyy' already implicit captured outside the 'if-constexpr' block
      result += yyy;
    }
    return result;
  };
}
```

13. When parsing a function, the CUDA compiler assigns a counter value to each extended lambda within that function. This counter value is used in the substituted named type passed to the host compiler. Hence, whether or not an extended lambda is defined within a function should not depend on a particular value of `__CUDA_ARCH__`, or on `__CUDA_ARCH__` being undefined.

    Example

```
template <typename T>
__global__ void kernel(T in) { in(); }

__host__ __device__ void foo(void) {
  // Error: the number and relative declaration
  // order of extended lambdas depends on
  // __CUDA_ARCH__
#if defined(__CUDA_ARCH__)
  auto lam1 = [] __device__ { return 0; };
  auto lam1b = [] __host__ __device__ { return 10; };
#endif
  auto lam2 = [] __device__ { return 4; };
  kernel<<<1,1>>>(lam2);
}
```

14. As described above, the CUDA compiler replaces a `__device__` extended lambda defined in a host function with a placeholder type defined in namespace scope. Unless the trait `__nv_is_extended_device_lambda_with_preserved_return_type()` returns true for the closure type of the extended lambda, the placeholder type does not define a `operator()` function equivalent to the original lambda declaration. An attempt to determine the return type or parameter types of the `operator()` function of such a lambda may therefore work incorrectly in host code, as the code processed by the host compiler will be semantically different than the input code processed by the CUDA compiler. However, it is OK to introspect the return type or parameter types of the `operator()` function within device code. Note that this restriction does not apply to `__host__ __device__` extended lambdas, or to `__device__` extended lambdas for which the trait `__nv_is_extended_device_lambda_with_preserved_return_type()` returns true.

    Example

```
#include <type_traits>
const char& getRef(const char* p) { return *p; }

void foo(void) {
  auto lam1 = [] __device__ { return "10"; };

  // Error: attempt to extract the return type
```

(continues on next page)

(continued from previous page)

```
  // of a __device__ lambda in host code
  std::result_of<decltype(lam1)()>::type xx1 = "abc";


  auto lam2 = [] __host__ __device__  { return "10"; };

  // OK : lam2 represents a __host__ __device__ extended lambda
  std::result_of<decltype(lam2)()>::type xx2 = "abc";

  auto lam3 = []  __device__ () -> const char * { return "10"; };

  // OK : lam3 represents a __device__ extended lambda with preserved return type
  std::result_of<decltype(lam3)()>::type xx2 = "abc";
  static_assert( std::is_same_v< std::result_of<decltype(lam3)()>::type, const
↪char *>);

  auto lam4 = [] __device__ (char x) -> decltype(getRef(&x)) { return 0; };
  // lam4's return type is not preserved because it references the operator()'s
  // parameter types in the trailing return type.
  static_assert( ! __nv_is_extended_device_lambda_with_preserved_return_
↪type(decltype(lam4)), "" );
}
```

15. For an extended device lambda: - Introspecting the parameter type of operator() is only supported in device code. - Introspecting the return type of operator() is supported only in device code, unless the trait function __nv_is_extended_device_lambda_with_preserved_return_type() returns true.

16. If the functor object represented by an extended lambda is passed from host to device code (e.g., as the argument of a `__global__` function), then any expression in the body of the lambda expression that captures variables must be remain unchanged irrespective of whether the `__CUDA_ARCH__` macro is defined, and whether the macro has a particular value. This restriction arises because the lambda's closure class layout depends on the order in which captured variables are encountered when the compiler processes the lambda expression; the program may execute incorrectly if the closure class layout differs in device and host compilation.

    Example

```
__device__ int result;

template <typename T>
__global__ void kernel(T in) { result = in(); }

void foo(void) {
  int x1 = 1;
  auto lam1 = [=] __host__ __device__ {
    // Error: "x1" is only captured when __CUDA_ARCH__ is defined.
#ifdef __CUDA_ARCH__
    return x1 + 1;
#else
    return 10;
#endif
  };
  kernel<<<1,1>>>(lam1);
}
```

17. As described previously, the CUDA compiler replaces an extended `__device__` lambda expres-

sion with an instance of a placeholder type in the code sent to the host compiler. This placeholder type does not define a pointer-to-function conversion operator in host code, however the conversion operator is provided in device code. Note that this restriction does not apply to `__host__ __device__` extended lambdas.

Example

```
template <typename T>
__global__ void kern(T in) {
  int (*fp)(double) = in;

  // OK: conversion in device code is supported
  fp(0);
  auto lam1 = [](double) { return 1; };

  // OK: conversion in device code is supported
  fp = lam1;
  fp(0);
}

void foo(void) {
  auto lam_d = [] __device__ (double) { return 1; };
  auto lam_hd = [] __host__ __device__ (double) { return 1; };
  kern<<<1,1>>>(lam_d);
  kern<<<1,1>>>(lam_hd);

  // OK : conversion for __host__ __device__ lambda is supported
  // in host code
  int (*fp)(double) = lam_hd;

  // Error: conversion for __device__ lambda is not supported in
  // host code.
  int (*fp2)(double) = lam_d;
}
```

18. As described previously, the CUDA compiler replaces an extended `__device__` or `__host__ __device__` lambda expression with an instance of a placeholder type in the code sent to the host compiler. This placeholder type may define C++ special member functions (e.g. constructor, destructor). As a result, some standard C++ type traits may return different results for the closure type of the extended lambda, in the CUDA frontend compiler versus the host compiler. The following type traits are affected: `std::is_trivially_copyable`, `std::is_trivially_constructible`, `std::is_trivially_copy_constructible`, `std::is_trivially_move_constructible`, `std::is_trivially_destructible`.

    Care must be taken that the results of these type traits are not used in `__global__` function template instantiation or in `__device__ / __constant__ / __managed__` variable template instantiation.

    Example

```
template <bool b>
void __global__ foo() { printf("hi"); }

template <typename T>
void dolaunch() {

// ERROR: this kernel launch may fail, because CUDA frontend compiler
// and host compiler may disagree on the result of
```

(continues on next page)

(continued from previous page)

```
// std::is_trivially_copyable() trait on the closure type of the
// extended lambda
foo<std::is_trivially_copyable<T>::value><<<1,1>>>();
cudaDeviceSynchronize();
}

int main() {
int x = 0;
auto lam1 = [=] __host__ __device__ () { return x; };
dolaunch<decltype(lam1)>();
}
```

The CUDA compiler will generate compiler diagnostics for a subset of cases described in 1-12; no diagnostic will be generated for cases 13-17, but the host compiler may fail to compile the generated code.

18.7.3. Notes on __host__ __device__ lambdas

Unlike `__device__` lambdas, `__host__ __device__` lambdas can be called from host code. As described earlier, the CUDA compiler replaces an extended lambda expression defined in host code with an instance of a named placeholder type. The placeholder type for an extended `__host__ __device__` lambda invokes the original lambda's `operator()` with an indirect function call[Page 458, 24].

The presence of the indirect function call may cause an extended `__host__ __device__` lambda to be less optimized by the host compiler than lambdas that are implicitly or explicitly `__host__` only. In the latter case, the host compiler can easily inline the body of the lambda into the calling context. But in case of an extended `__host__ __device__` lambda, the host compiler encounters the indirect function call and may not be able to easily inline the original `__host__ __device__` lambda body.

18.7.4. *this Capture By Value

When a lambda is defined within a non-static class member function, and the body of the lambda refers to a class member variable, C++11/C++14 rules require that the `this` pointer of the class is captured by value, instead of the referenced member variable. If the lambda is an extended `__device__` or `__host____device__` lambda defined in a host function, and the lambda is executed on the GPU, accessing the referenced member variable on the GPU will cause a run time error if the `this` pointer points to host memory.

Example:

```
#include <cstdio>

template <typename T>
__global__ void foo(T in) { printf("\n value = %d", in()); }

struct S1_t {
  int xxx;
  __host__ __device__ S1_t(void) : xxx(10) { };

  void doit(void) {
```

(continues on next page)

(continued from previous page)

```
    auto lam1 = [=] __device__ {
       // reference to "xxx" causes
       // the 'this' pointer (S1_t*) to be captured by value
       return xxx + 1;

    };

    // Kernel launch fails at run time because 'this->xxx'
    // is not accessible from the GPU
    foo<<<1,1>>>(lam1);
    cudaDeviceSynchronize();
  }
};

int main(void) {
  S1_t s1;
  s1.doit();
}
```

C++17 solves this problem by adding a new "*this" capture mode. In this mode, the compiler makes a copy of the object denoted by "*this" instead of capturing the pointer `this` by value. The "*this" capture mode is described in more detail here: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0018r3.html` .

The CUDA compiler supports the "*this" capture mode for lambdas defined within `__device__` and `__global__` functions and for extended `__device__` lambdas defined in host code, when the `--extended-lambda` nvcc flag is used.

Here's the above example modified to use "*this" capture mode:

```
#include <cstdio>

template <typename T>
__global__ void foo(T in) { printf("\n value = %d", in()); }

struct S1_t {
  int xxx;
  __host__ __device__ S1_t(void) : xxx(10) { };

  void doit(void) {

    // note the "*this" capture specification
    auto lam1 = [=, *this] __device__ {

       // reference to "xxx" causes
       // the object denoted by '*this' to be captured by
       // value, and the GPU code will access copy_of_star_this->xxx
       return xxx + 1;

    };

    // Kernel launch succeeds
    foo<<<1,1>>>(lam1);
    cudaDeviceSynchronize();
  }
};
```

(continues on next page)

(continued from previous page)

```
int main(void) {
  S1_t s1;
  s1.doit();
}
```

“*this” capture mode is not allowed for unannotated lambdas defined in host code, or for extended `__host____device__` lambdas, unless “*this” capture is enabled by the selected language dialect. Examples of supported and unsupported usage:

```
struct S1_t {
  int xxx;
  __host__ __device__ S1_t(void) : xxx(10) { };

  void host_func(void) {

    // OK: use in an extended __device__ lambda
    auto lam1 = [=, *this] __device__ { return xxx; };

    // Use in an extended __host__ __device__ lambda
    // Error if *this capture not enabled by language dialect
    auto lam2 = [=, *this] __host__ __device__ { return xxx; };

    // Use in an unannotated lambda in host function
    // Error if *this capture not enabled by language dialect
    auto lam3 = [=, *this]  { return xxx; };
  }

  __device__ void device_func(void) {

    // OK: use in a lambda defined in a __device__ function
    auto lam1 = [=, *this] __device__ { return xxx; };

    // OK: use in a lambda defined in a __device__ function
    auto lam2 = [=, *this] __host__ __device__ { return xxx; };

    // OK: use in a lambda defined in a __device__ function
    auto lam3 = [=, *this]  { return xxx; };
  }

   __host__ __device__ void host_device_func(void) {

    // OK: use in an extended __device__ lambda
    auto lam1 = [=, *this] __device__ { return xxx; };

    // Use in an extended __host__ __device__ lambda
    // Error if *this capture not enabled by language dialect
    auto lam2 = [=, *this] __host__ __device__ { return xxx; };

    // Use in an unannotated lambda in a __host__ __device__ function
    // Error if *this capture not enabled by language dialect
    auto lam3 = [=, *this]  { return xxx; };
  }
};
```

18.7.5. Additional Notes

1. ADL Lookup: As described earlier, the CUDA compiler will replace an extended lambda expression with an instance of a placeholder type, before invoking the host compiler. One template argument of the placeholder type uses the address of the function enclosing the original lambda expression. This may cause additional namespaces to participate in argument dependent lookup (ADL), for any host function call whose argument types involve the closure type of the extended lambda expression. This may cause an incorrect function to be selected by the host compiler.

   Example:

```
namespace N1 {
  struct S1_t { };
  template <typename T>  void foo(T);
};

namespace N2 {
  template <typename T> int foo(T);

  template <typename T>  void doit(T in) {     foo(in);  }
}

void bar(N1::S1_t in) {
  /* extended __device__ lambda. In the code sent to the host compiler, this
     is replaced with the placeholder type instantiation expression
     ' __nv_dl_wrapper_t< __nv_dl_tag<void (*)(N1::S1_t in),(&bar),1> > { }'

     As a result, the namespace 'N1' participates in ADL lookup of the
     call to "foo" in the body of N2::doit, causing ambiguity.
  */
  auto lam1 = [=] __device__ { };
  N2::doit(lam1);
}
```

   In the example above, the CUDA compiler replaced the extended lambda with a placeholder type that involves the `N1` namespace. As a result, the namespace `N1` participates in the ADL lookup for `foo(in)` in the body of `N2::doit`, and host compilation fails because multiple overload candidates `N1::foo` and `N2::foo` are found.

18.8. Relaxed Constexpr (-expt-relaxed-constexpr)

By default, the following cross-execution space calls are not supported:

1. Calling a `__device__`-only `constexpr` function from a `__host__` function during host code generation phase (i.e, when `__CUDA_ARCH__` macro is undefined). Example:

```
constexpr __device__ int D() { return 0; }
int main() {
    int x = D();  //ERROR: calling a __device__-only constexpr function
↪from host code
}
```

2. Calling a `__host__`-only `constexpr` function from a `__device__` or `__global__` function, during device code generation phase (i.e. when `__CUDA_ARCH__` macro is defined). Example:

```
constexpr  int H() { return 0; }
__device__ void dmain()
{
    int x = H();  //ERROR: calling a __host__-only constexpr function from
↪device code
}
```

The experimental flag `-expt-relaxed-constexpr` can be used to relax this constraint. When this flag is specified, the compiler will support cross execution space calls described above, as follows:

1. A cross-execution space call to a constexpr function is supported if it occurs in a context that requires constant evaluation, e.g., in the initializer of a constexpr variable. Example:

```
constexpr __host__ int H(int x) { return x+1; };
__global__ void doit() {
constexpr int val = H(1); // OK: call is in a context that
                          // requires constant evaluation.
}

constexpr __device__ int D(int x) { return x+1; }
int main() {
constexpr int val = D(1); // OK: call is in a context that
                          // requires constant evaluation.
}
```

2. Otherwise:

   a. During device code generation, device code is generated for the body of a `__host__`-only constexpr function H, unless H is not used or is only called in a constexpr context. Example:

```
// NOTE: "H" is emitted in generated device code because it is
// called from device code in a non-constexpr context
constexpr __host__ int H(int x) { return x+1; }

__device__ int doit(int in) {
  in = H(in);  // OK, even though argument is not a constant expression
  return in;
}
```

   b. **All code restrictions applicable to a ``__device__`` function are also applicable to the ``constexpr host``-only function ``H`` that is called from device code. However, compiler may not emit any build time diagnostics for ``H`` for these restrictions**[8] .

      For example, the following code patterns are unsupported in the body of H (as with any `__device__` function), but no compiler diagnostic may be generated:

      - ODR-use of a host variable or `__host__`-only non-constexpr function. Example:

```
int qqq, www;

constexpr __host__ int* H(bool b) { return b ? &qqq : &www; };

__device__ int doit(bool flag) {
```

(continues on next page)

[8] Diagnostics are usually generated during parsing, but the host-only function H may already have been parsed before the call to H from device code is encountered later in the translation unit.

(continued from previous page)

```
  int *ptr;
  ptr = H(flag); // ERROR: H() attempts to refer to host
↪variables 'qqq' and 'www'.
                 // code will compile, but will NOT execute
↪correctly.
  return *ptr;
}
```

- Use of exceptions (`throw/catch`) and RTTI (`typeid, dynamic_cast`). Example:

```
struct Base { };
struct Derived : public Base { };

// NOTE: "H" is emitted in generated device code
constexpr int H(bool b, Base *ptr) {
  if (b) {
    return 1;
  } else if (typeid(ptr) == typeid(Derived)) { // ERROR: use of
↪typeid in code executing on the GPU
    return 2;
  } else {
    throw int{4}; // ERROR: use of throw in code executing on
↪the GPU
  }
}
__device__ void doit(bool flag) {
  int val;
  Derived d;
  val = H(flag, &d); //ERROR: H() attempts use typeid and
↪throw(), which are not allowed in code that executes on the GPU
}
```

c. During host code generation, the body of a `__device__`-only constexpr function D is preserved in the code sent to the host compiler. If the body of D attempts to ODR-use a namespace scope device variable or a `__device__`-only non-constexpr function, then the call to D from host code is not supported (code may build without compiler diagnostics, but may behave incorrectly at run time). Example:

```
__device__ int qqq, www;
constexpr __device__ int* D(bool b) { return b ? &qqq : &www; };

int doit(bool flag) {
  int *ptr;
  ptr = D(flag); // ERROR: D() attempts to refer to device variables
↪'qqq' and 'www'
                 // code will compile, but will NOT execute correctly.
  return *ptr;
}
```

d. **Note: Given above restrictions and lack of compiler diagnostics for incorrect usage, be careful when calling a constexpr __host__ function in the standard C++ headers from device code**, since the implementation of the function will vary depending on the host platform, e.g., based on the `libstdc++` version for gcc host compiler. Such code may break silently when being ported to a different platform or host compiler version (if the target C++ library implementation odr-uses a host code variable or function, as described earlier).

Example:

```
__device__ int get(int in) {
 int val = std::foo(in); // "std::foo" is constexpr function defined in
↪the host compiler's standard library header
                         // WARNING: if std::foo implementation ODR-
↪uses host variables or functions,
                         // code will not work correctly
}
```

18.9. Code Samples

18.9.1. Data Aggregation Class

```
class PixelRGBA {
public:
    __device__ PixelRGBA(): r_(0), g_(0), b_(0), a_(0) { }

    __device__ PixelRGBA(unsigned char r, unsigned char g,
                         unsigned char b, unsigned char a = 255):
                         r_(r), g_(g), b_(b), a_(a) { }

private:
    unsigned char r_, g_, b_, a_;

    friend PixelRGBA operator+(const PixelRGBA&, const PixelRGBA&);
};

__device__
PixelRGBA operator+(const PixelRGBA& p1, const PixelRGBA& p2)
{
    return PixelRGBA(p1.r_ + p2.r_, p1.g_ + p2.g_,
                     p1.b_ + p2.b_, p1.a_ + p2.a_);
}

__device__ void func(void)
{
    PixelRGBA p1, p2;
    // ...      // Initialization of p1 and p2 here
    PixelRGBA p3 = p1 + p2;
}
```

18.9.2. Derived Class

```
__device__ void* operator new(size_t bytes, MemoryPool& p);
__device__ void operator delete(void*, MemoryPool& p);
class Shape {
public:
    __device__ Shape(void) { }
    __device__ void putThis(PrintBuffer *p) const;
    __device__ virtual void Draw(PrintBuffer *p) const {
         p->put("Shapeless");
    }
    __device__ virtual ~Shape() {}
};
class Point : public Shape {
public:
    __device__ Point() : x(0), y(0) {}
    __device__ Point(int ix, int iy) : x(ix), y(iy) { }
    __device__ void PutCoord(PrintBuffer *p) const;
    __device__ void Draw(PrintBuffer *p) const;
    __device__ ~Point() {}
private:
    int x, y;
};
__device__ Shape* GetPointObj(MemoryPool& pool)
{
    Shape* shape = new(pool) Point(rand(-20,10), rand(-100,-20));
    return shape;
}
```

18.9.3. Class Template

```
template <class T>
class myValues {
    T values[MAX_VALUES];
public:
    __device__ myValues(T clear) { ... }
    __device__ void setValue(int Idx, T value) { ... }
    __device__ void putToMemory(T* valueLocation) { ... }
};

template <class T>
void __global__ useValues(T* memoryBuffer) {
    myValues<T> myLocation(0);
    ...
}

__device__ void* buffer;

int main()
{
    ...
    useValues<int><<<blocks, threads>>>(buffer);
    ...
}
```

18.9.4. Function Template

```
template <typename T>
__device__ bool func(T x)
{
   ...
   return (...);
}

template <>
__device__ bool func<int>(T x) // Specialization
{
   return true;
}

// Explicit argument specification
bool result = func<double>(0.5);

// Implicit argument deduction
int x = 1;
bool result = func(x);
```

18.9.5. Functor Class

```
class Add {
public:
    __device__  float operator() (float a, float b) const
    {
        return a + b;
    }
};

class Sub {
public:
    __device__  float operator() (float a, float b) const
    {
        return a - b;
    }
};

// Device code
template<class O> __global__
void VectorOperation(const float * A, const float * B, float * C,
                     unsigned int N, O op)
{
    unsigned int iElement = blockDim.x * blockIdx.x + threadIdx.x;
    if (iElement < N)
        C[iElement] = op(A[iElement], B[iElement]);
}

// Host code
int main()
{
    ...
```

(continues on next page)

(continued from previous page)

```
    VectorOperation<<<blocks, threads>>>(v1, v2, v3, N, Add());
    ...
}
```

Chapter 19. Texture Fetching

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

This section gives the formula used to compute the value returned by the texture functions of *Texture Functions* depending on the various attributes of the texture object (see *Texture and Surface Memory*).

The texture bound to the texture object is represented as an array T of

- N texels for a one-dimensional texture,
- $N \times M$ texels for a two-dimensional texture,
- $N \times M \times L$ texels for a three-dimensional texture.

It is fetched using non-normalized texture coordinates x, y, and z, or the normalized texture coordinates x/N, y/M, and z/L as described in *Texture Memory*. In this section, the coordinates are assumed to be in the valid range. *Texture Memory* explained how out-of-range coordinates are remapped to the valid range based on the addressing mode.

19.1. Nearest-Point Sampling

In this filtering mode, the value returned by the texture fetch is

- *tex(x)=T[i]* for a one-dimensional texture,
- *tex(x,y)=T[i,j]* for a two-dimensional texture,
- *tex(x,y,z)=T[i,j,k]* for a three-dimensional texture,

where *i=floor(x)*, *j=floor(y)*, and *k=floor(z)*.

Figure 36 illustrates nearest-point sampling for a one-dimensional texture with *N=4*.

For integer textures, the value returned by the texture fetch can be optionally remapped to [0.0, 1.0] (see *Texture Memory*).

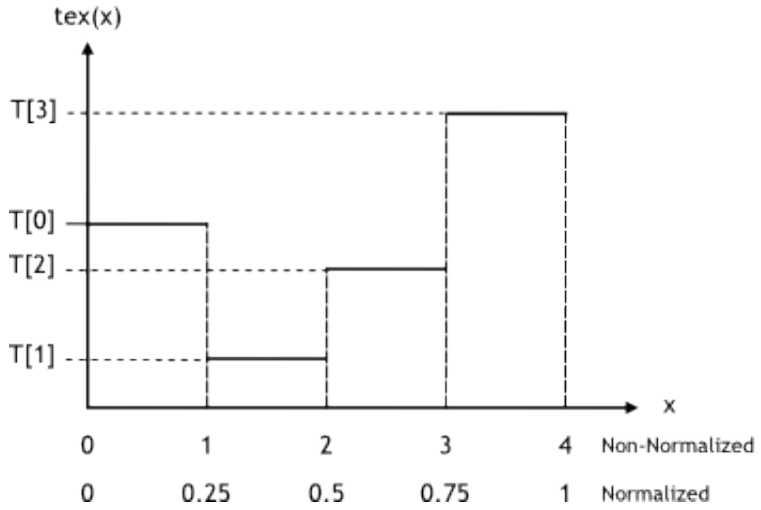


Figure 36: Nearest-Point Sampling Filtering Mode

19.2. Linear Filtering

In this filtering mode, which is only available for floating-point textures, the value returned by the texture fetch is

- $tex(x) = (1-\alpha)T[i] + \alpha T[i+1]$ for a one-dimensional texture,
- $tex(x) = (1-\alpha)T[i] + \alpha T[i+1]$ for a one-dimensional texture,
- $tex(x,y) = (1-\alpha)(1-\beta)T[i,j] + \alpha(1-\beta)T[i+1,j] + (1-\alpha)\beta T[i,j+1] + \alpha\beta T[i+1,j+1]$ for a two-dimensional texture,
- $tex(x,y,z)$ =

  $(1-\alpha)(1-\beta)(1-\gamma)T[i,j,k] + \alpha(1-\beta)(1-\gamma)T[i+1,j,k] +$

  $(1-\alpha)\beta(1-\gamma)T[i,j+1,k] + \alpha\beta(1-\gamma)T[i+1,j+1,k] +$

  $(1-\alpha)(1-\beta)\gamma T[i,j,k+1] + \alpha(1-\beta)\gamma T[i+1,j,k+1] +$

  $(1-\alpha)\beta\gamma T[i,j+1,k+1] + \alpha\beta\gamma T[i+1,j+1,k+1]$

  for a three-dimensional texture,

where:

- $i = floor(x\ B) *, \alpha = frac(x\ B) *, *x\ B\ = x - 0.5,$
- $j = floor(y\ B) *, \beta = frac(y\ B) *, *y\ B\ = y - 0.5,$
- $k = floor(z\ B) *, \gamma = frac(z\ B) *, *z\ B\ = z - 0.5,$

α, β, and γ are stored in 9-bit fixed point format with 8 bits of fractional value (so 1.0 is exactly represented).

Figure 37 illustrates linear filtering of a one-dimensional texture with *N=4*.

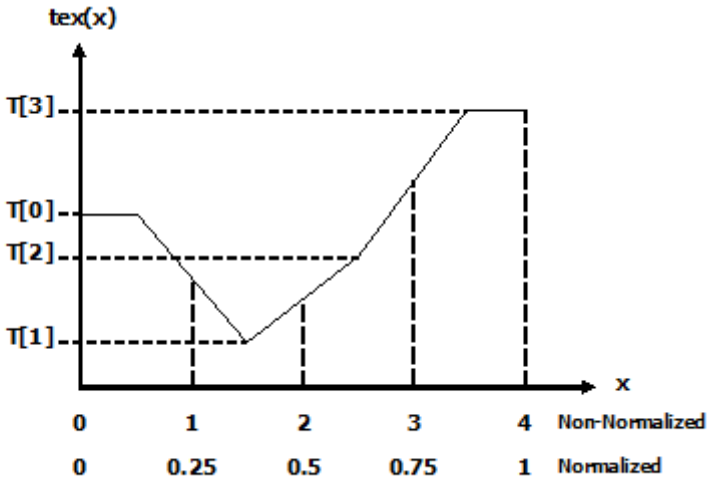


Figure 37: Linear Filtering Mode

19.3. Table Lookup

A table lookup *TL(x)* where *x* spans the interval *[0,R]* can be implemented as *TL(x)=tex((N-1)/R)x+0.5)* in order to ensure that *TL(0)=T[0]* and *TL(R)=T[N-1]*.

Figure 38 illustrates the use of texture filtering to implement a table lookup with *R=4* or *R=1* from a one-dimensional texture with *N=4*.

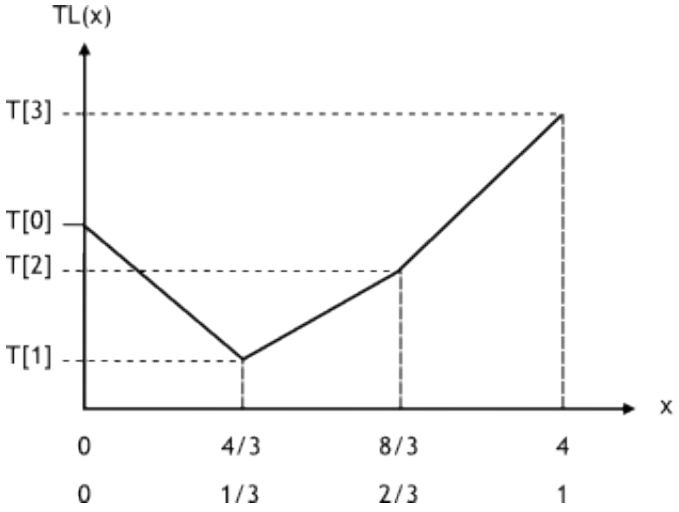


Figure 38: One-Dimensional Table Lookup Using Linear Filtering

Chapter 20. Compute Capabilities

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

The general specifications and features of a compute device depend on its compute capability (see *Compute Capability*).

Table 26 and Table 27 show the features and technical specifications associated with each compute capability that is currently supported.

Section *Floating-Point Standard* reviews compliance with the IEEE floating-point standard.

Sections *Compute Capability 5.x*, *Compute Capability 6.x*, *Compute Capability 7.x*, *Compute Capability 8.x*, *Compute Capability 9.0*, *Compute Capability 10.0*, and *Compute Capability 12.0* give more details on the architecture of devices with these respective compute capabilities.

20.1. Feature Availability

Most compute features introduced with a compute architecture are intended to be available on all subsequent architectures. This is shown in Table 26 by the “yes” for availability of a feature on compute capabilities subsequent to its introduction.

20.1.1. Architecture-Specific Features

Beginning with devices of Compute Capability 9.0, specialized compute features that are introduced with an architecture may not be guaranteed to be available on all subsequent compute capabilities. These features are called *architecture-specific* features and target acceleration of specialized operations, such as Tensor Core operations, which are not intended for all classes of compute capabilities or may significantly change on future generations. Code must be compiled with an architecture-specific compiler target (see *Feature Set Compiler Targets*) to enable architecture-specific features. Code compiled with an architecture-specific compiler target can only be run on the exact compute capability it was compiled for.

20.1.2. Family-Specific Features

Beginning with devices of Compute Capability 10.0, some architecture-specific features are common to devices of more than one compute capability. The devices that contain these features are part of the same family and these features can also be called *family-specific* features. Family-specific features are guaranteed to be available on all devices in the same family. A family-specific compiler target is required to enable family-specific features. See Section 20.1.3. Code compiled for a family-specific target can only be run on GPUs which are members of that family.

20.1.3. Feature Set Compiler Targets

There are three sets of compute features which the compiler can target:

Baseline Feature Set: The predominant set of compute features that are introduced with the intent to be available for subsequent compute architectures. These features and their availability are summarized in Table 26.

Architecture-Specific Feature Set: A small and highly specialized set of features called architecture-specific, that are introduced to accelerate specialized operations, which are not guaranteed to be available or might change significantly on subsequent compute architectures. These features are summarized in the respective "Compute Capability #.#" subsections. The architecture-specific feature set is a superset of the family-specific feature set. Architecture-specific compiler targets were introduced with Compute Capability 9.0 devices and are selected by using an **a** suffix in the compilation target, for example by specifying `compute_100a` or `compute_120a` as the compute target.

Family-Specific Feature Set: Some architecture-specific features are common to GPUs of more than one compute capability. These features are summarized in the respective "Compute Capability #.#" subsections. With a few exceptions, later generation devices with the same major compute capability are in the same family. Table 25 indicates the compatibility of family-specific targets with device compute capability, including exceptions. The family-specific feature set is a superset of the baseline feature set. Family-specific compiler targets were introduced with Compute Capability 10.0 devices and are selected by using a **f** suffix in the compilation target, for example by specifying `compute_100f` or `compute_120f` as the compute target.

All devices starting from compute capability 9.0 have a set of features that are architecture-specific. To utilize the complete set of these features on a specific GPU, the architecture-specific compiler target with the suffix **a** must be used. Additionally, starting from compute capability 10.0, there are sets of features that appear in multiple devices with different minor compute capability. These sets of instructions are called family-specific features, and the devices which share these features are said to be part of the same family. The family-specific features are a subset of the architecture-specific features that are shared by all members of that GPU family. The family-specific compiler target with the suffix **f** allows the compiler to generate code which uses this common subset of architecture-specific features.

For example:

- The `compute_100` compilation target does not allow use of architecture-specific features. This target will be compatible with all devices of compute capability 10.0 and later.
- The `compute_100f` *family-specific* compilation target allows the use of the subset of architecture-specific features that are common across the GPU family. This target will only be compatible with devices that are part of the GPU family. In this example it is compatible with devices of Compute Capability 10.0 and Compute Capability 10.3. The features available in the family-specific `compute_100f` target is a superset of the features available in the baseline `compute_100` target.

- The `compute_100a` *architecture-specific* compilation target allows use of the complete set of architecture-specific features in Compute Capability 10.0 devices. This target will only be compatible with devices of Compute Capability 10.0 and no others. The features available in the `compute_100a` target form a superset of the features available in the `compute_100f` target.

Table 25: Family-Specific Compatibility

Compilation Target	**Compatible with Compute Capability**	
`compute_100f`	10.0	10.3
`compute_103f`	10.3[26]	
`compute_110f`	11.0[26]	
`compute_120f`	12.0	12.1
`compute_121f`	12.1[26]	

20.2. Features and Technical Specifications

Table 26: Feature Support per Compute Capability

Feature Support	**Compute Capability**					
(Unlisted features are supported for all compute capabilities)	7.x	8.x	9.0	10.0	11.0	12.0
Atomic functions operating on 128-bit integer values in global memory (*Atomic Functions*)	No		Yes			

continues on next page

[26] Some families only contain a single member when they are created. They may be expanded in the future to include more devices.

Table 26 – continued from previous page

Feature Support	Compute Capability	
Atomic functions operating on 128-bit integer values in shared memory (*Atomic Functions*)	No	Yes
Atomic addition operating on float2 and float4 floating point vectors in global memory (*atomicAdd()*)	No	Yes
Bfloat16-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No	Yes
Hardware-accelerated `memcpy_async` (*Asynchronous Data Copies using cuda::pipeline*)	No	Yes

continues on next page

Table 26 – continued from previous page

Feature Support	Compute Capability		
Hardware-accelerated Split Arrive/Wait Barrier (*Asynchronous Barrier*)	No	Yes	
L2 Cache Residency Management (*Device Memory L2 Access Management*)	No	Yes	
DPX Instructions for Accelerated Dynamic Programming	No		Yes
Distributed Shared Memory	No		Yes
Thread Block Cluster	No		Yes
Tensor Memory Accelerator (TMA) unit	No		Yes

Note that the KB and K units used in the following table correspond to 1024 bytes (i.e., a KiB) and 1024 respectively.

Table 27: Technical Specifications per Compute Capability

	Compute Capability								
Technical Specifications	7.5	8.0	8.6	8.7	8.9	9.0	10.0	11.0	12.0

continues on next page

Table 27 – continued from previous page

	Compute Capability						
Maximum number of resident grids per device (Concurrent Kernel Execution)	128						
Maximum dimensionality of grid of thread blocks	3						
Maximum x -dimension of a grid of thread blocks	2^{31}-1						
Maximum y- or z-dimension of a grid of thread blocks	65535						
Maximum dimensionality of thread block	3						
Maximum x- or y-dimensionality of a block	1024						
Maximum z-dimension of a block	64						
Maximum number of threads per block	1024						
Warp size	32						
Maximum number of resident blocks per SM	16	32	16	24	32	24	
Maximum number of resident warps per SM	32	64	48		64	48	
Maximum number of resident threads per SM	1024	2048	1536		2048	1536	
Number of 32-bit registers per SM	64 K						
Maximum number of 32-bit registers per thread block	64 K						
Maximum number of 32-bit registers per thread	255						
Maximum amount of shared memory per SM	64 KB	164 KB	100 KB	164 KB	100 KB	228 KB	100 KB
Maximum amount of shared memory per thread block[27]	64 KB	163 KB	99 KB	163 KB	99 KB	227 KB	99 KB
Number of shared memory banks	32						
Maximum amount of local memory per thread	512 KB						
Constant memory size	64 KB						
Cache working set per SM for constant memory	8 KB						
Cache working set per SM for texture memory	32 or 64 KB	28 KB ~ 192 KB	28 KB ~ 128 KB	28 KB ~ 192 KB	28 KB ~ 128 KB	28 KB ~ 256 KB	28 KB ~ 128 KB

continues on next page

Table 27 – continued from previous page

	Compute Capability
Maximum width for a 1D texture object using a CUDA array	131072
Maximum width for a 1D texture object using linear memory	2^{28}
Maximum width and number of layers for a 1D layered texture object	32768 x 2048
Maximum width and height for a 2D texture object using a CUDA array	131072 x 65536
Maximum width and height for a 2D texture object using linear memory	131072 x 65000
Maximum width and height for a 2D texture object using a CUDA array supporting texture gather	32768 x 32768
Maximum width, height, and number of layers for a 2D layered texture object	32768 x 32768 x 2048
Maximum width, height, and depth for a 3D texture object using to a CUDA array	16384 x 16384 x 16384
Maximum width (and height) for a cubemap texture object	32768
Maximum width (and height) and number of layers for a cubemap layered texture object	32768 x 2046
Maximum number of textures that can be bound to a kernel	256
Maximum width for a 1D surface object using a CUDA array	32768
Maximum width and number of layers for a 1D layered surface object	32768 x 2048
Maximum width and height for a 2D surface object using a CUDA array	131072 x 65536
Maximum width, height, and number of layers for a 2D layered surface object	32768 x 32768 x 1048
Maximum width, height, and depth for a 3D surface object using a CUDA array	16384 x 16384 x 16384
Maximum width (and height) for a cubemap surface object using a CUDA array	32768
Maximum width (and height) and number of layers for a cubemap layered surface object	32768 x 2046
Maximum number of surfaces that can use a kernel	32

[27] above 48 KB requires dynamic shared memory

20.3. Floating-Point Standard

All compute devices follow the IEEE 754-2008 standard for binary floating-point arithmetic with the following deviations:

- There is no dynamically configurable rounding mode; however, most of the operations support multiple IEEE rounding modes, exposed via device intrinsics.
- There is no mechanism for detecting that a floating-point exception has occurred and all operations behave as if the IEEE-754 exceptions are always masked, and deliver the masked response as defined by IEEE-754 if there is an exceptional event. For the same reason, while SNaN encodings are supported, they are not signaling and are handled as quiet.
- The result of a single-precision floating-point operation involving one or more input NaNs is the quiet NaN of bit pattern 0x7fffffff.
- Double-precision floating-point absolute value and negation are not compliant with IEEE-754 with respect to NaNs; these are passed through unchanged.

Code must be compiled with `-ftz=false`, `-prec-div=true`, and `-prec-sqrt=true` to ensure IEEE compliance (this is the default setting; see the `nvcc` user manual for description of these compilation flags).

Regardless of the setting of the compiler flag `-ftz`,

- atomic single-precision floating-point adds on global memory always operate in flush-to-zero mode, i.e., behave equivalent to `FADD.F32.FTZ.RN`,
- atomic single-precision floating-point adds on shared memory always operate with denormal support, i.e., behave equivalent to `FADD.F32.RN`.

In accordance to the IEEE-754R standard, if one of the input parameters to `fminf()`, `fmin()`, `fmaxf()`, or `fmax()` is NaN, but not the other, the result is the non-NaN parameter.

The conversion of a floating-point value to an integer value in the case where the floating-point value falls outside the range of the integer format is left undefined by IEEE-754. For compute devices, the behavior is to clamp to the end of the supported range. This is unlike the x86 architecture behavior.

The behavior of integer division by zero and integer overflow is left undefined by IEEE-754. For compute devices, there is no mechanism for detecting that such integer operation exceptions have occurred. Integer division by zero yields an unspecified, machine-specific value.

https://developer.nvidia.com/content/precision-performance-floating-point-and-ieee-754-compliance-nvidia-gpus includes more information on the floating point accuracy and compliance of NVIDIA GPUs.

20.4. Compute Capability 5.x

20.4.1. Architecture

An SM consists of:

- 128 CUDA cores for arithmetic operations (see CUDA C++ Best Practices Guide for throughputs of arithmetic operations),
- 32 special function units for single-precision floating-point transcendental functions,

- 4 warp schedulers.

When an SM is given warps to execute, it first distributes them among the four schedulers. Then, at every instruction issue time, each scheduler issues one instruction for one of its assigned warps that is ready to execute, if any.

An SM has:

- a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- a unified L1/texture cache of 24 KB used to cache reads from global memory,
- 64 KB of shared memory for devices of compute capability 5.0 or 96 KB of shared memory for devices of compute capability 5.2.

The unified L1/texture cache is also used by the texture unit that implements the various addressing modes and data filtering mentioned in *Texture and Surface Memory*.

There is also an L2 cache shared by all SMs that is used to cache accesses to local or global memory, including temporary register spills. Applications may query the L2 cache size by checking the `l2CacheSize` device property (see *Device Enumeration*).

The cache behavior (e.g., whether reads are cached in both the unified L1/texture cache and L2 or in L2 only) can be partially configured on a per-access basis using modifiers to the load instruction.

20.4.2. Global Memory

Global memory accesses are always cached in L2.

Data that is read-only for the entire lifetime of the kernel can also be cached in the unified L1/texture cache described in the previous section by reading it using the `__ldg()` function (see *Read-Only Data Cache Load Function*). When the compiler detects that the read-only condition is satisfied for some data, it will use `__ldg()` to read it. The compiler might not always be able to detect that the read-only condition is satisfied for some data. Marking pointers used for loading such data with both the `const` and `__restrict__` qualifiers increases the likelihood that the compiler will detect the read-only condition.

Data that is not read-only for the entire lifetime of the kernel cannot be cached in the unified L1/texture cache for devices of compute capability 5.0. For devices of compute capability 5.2, it is, by default, not cached in the unified L1/texture cache, but caching may be enabled using the following mechanisms:

- Perform the read using inline assembly with the appropriate modifier as described in the PTX reference manual;
- Compile with the `-Xptxas -dlcm=ca` compilation flag, in which case all reads are cached, except reads that are performed using inline assembly with a modifier that disables caching;
- Compile with the `-Xptxas -fscm=ca` compilation flag, in which case all reads are cached, including reads that are performed using inline assembly regardless of the modifier used.

When caching is enabled using one of the three mechanisms listed above, devices of compute capability 5.2 will cache global memory reads in the unified L1/texture cache for all kernel launches except for the kernel launches for which thread blocks consume too much of the SM's register file. These exceptions are reported by the profiler.

20.4.3. Shared Memory

Shared memory has 32 banks that are organized such that successive 32-bit words map to successive banks. Each bank has a bandwidth of 32 bits per clock cycle.

A shared memory request for a warp does not generate a bank conflict between two threads that access any address within the same 32-bit word (even though the two addresses fall in the same bank). In that case, for read accesses, the word is broadcast to the requesting threads and for write accesses, each address is written by only one of the threads (which thread performs the write is undefined).

Figure 39 shows some examples of strided access.

Figure 40 shows some examples of memory read accesses that involve the broadcast mechanism.

20.5. Compute Capability 6.x

20.5.1. Architecture

An SM consists of:

- 64 (compute capability 6.0) or 128 (6.1 and 6.2) CUDA cores for arithmetic operations,
- 16 (6.0) or 32 (6.1 and 6.2) special function units for single-precision floating-point transcendental functions,
- 2 (6.0) or 4 (6.1 and 6.2) warp schedulers.

When an SM is given warps to execute, it first distributes them among its schedulers. Then, at every instruction issue time, each scheduler issues one instruction for one of its assigned warps that is ready to execute, if any.

An SM has:

- a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- a unified L1/texture cache for reads from global memory of size 24 KB (6.0 and 6.2) or 48 KB (6.1),
- a shared memory of size 64 KB (6.0 and 6.2) or 96 KB (6.1).

The unified L1/texture cache is also used by the texture unit that implements the various addressing modes and data filtering mentioned in *Texture and Surface Memory*.

There is also an L2 cache shared by all SMs that is used to cache accesses to local or global memory, including temporary register spills. Applications may query the L2 cache size by checking the `l2CacheSize` device property (see *Device Enumeration*).

The cache behavior (for example, whether reads are cached in both the unified L1/texture cache and L2 or in L2 only) can be partially configured on a per-access basis using modifiers to the load instruction.

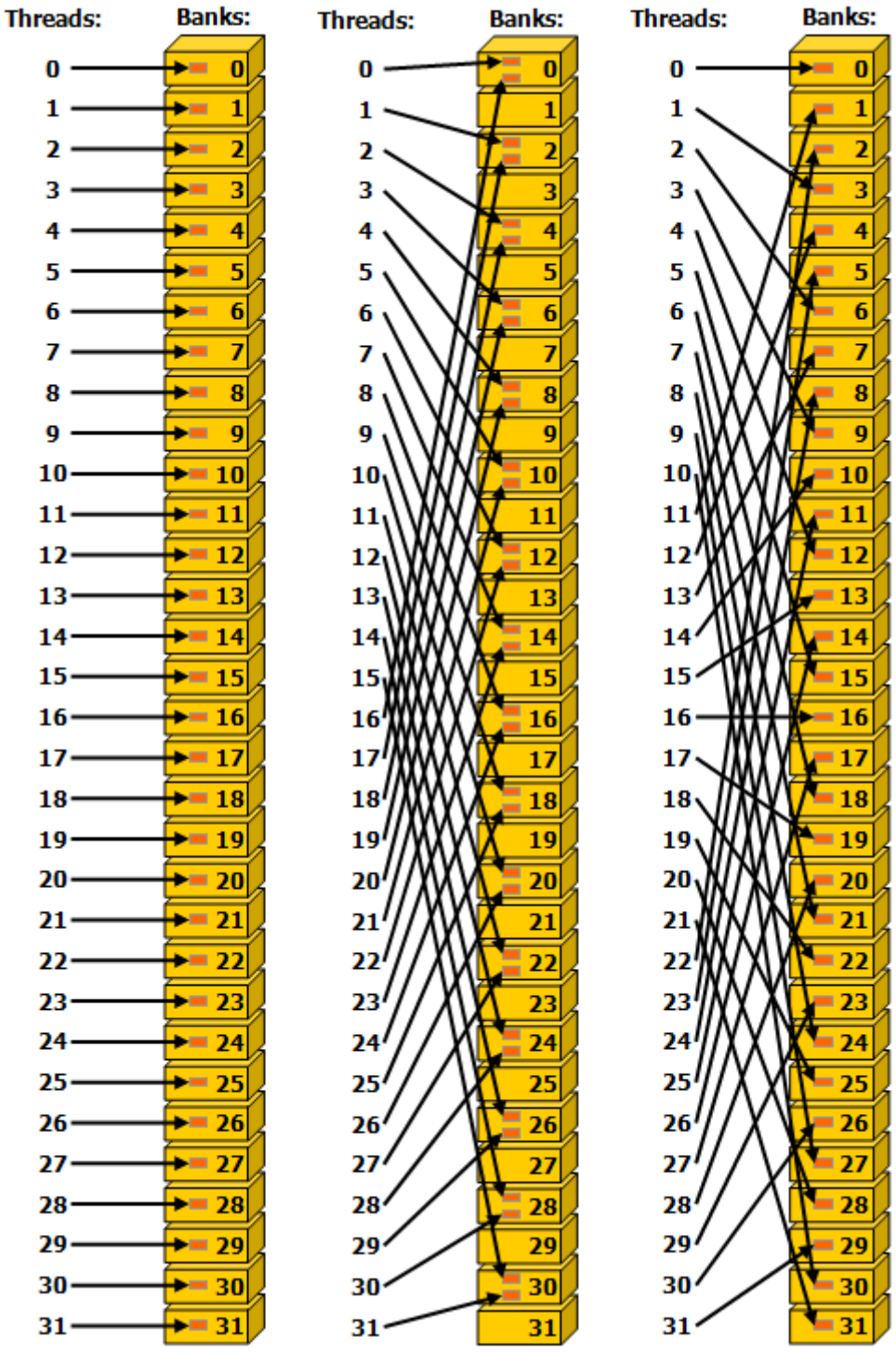


Figure 39: Strided Shared Memory Accesses in 32 bit bank size mode.

Left Linear addressing with a stride of one 32-bit word (no bank conflict).

Middle
Linear addressing with a stride of two 32-bit words (two-way bank conflict).

Right
Linear addressing with a stride of three 32-bit words (no bank conflict).

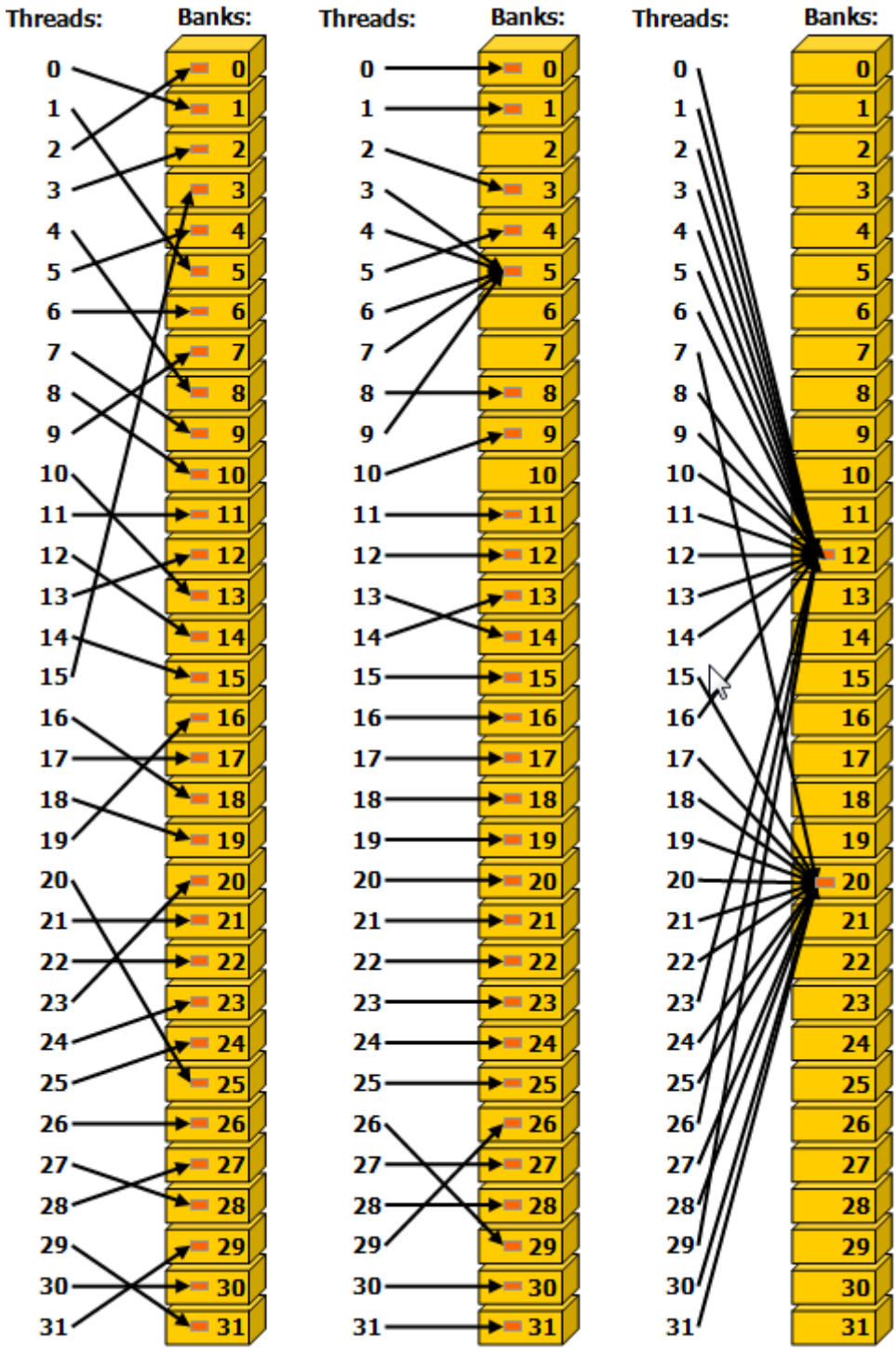


Figure 40: Irregular Shared Memory Accesses.

Left Conflict-free access via random permutation.

Middle
Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.

Right
Conflict-free broadcast access (threads access the same word within a bank).

20.5.2. Global Memory

Global memory behaves the same way as in devices of compute capability 5.x (See *Global Memory*).

20.5.3. Shared Memory

Shared memory behaves the same way as in devices of compute capability 5.x (See *Shared Memory*).

20.6. Compute Capability 7.x

20.6.1. Architecture

An SM consists of:

- 64 FP32 cores for single-precision arithmetic operations,
- 32 FP64 cores for double-precision arithmetic operations,[28]
- 64 INT32 cores for integer math,
- 8 mixed-precision Tensor Cores for deep learning matrix arithmetic
- 16 special function units for single-precision floating-point transcendental functions,
- 4 warp schedulers.

An SM statically distributes its warps among its schedulers. Then, at every instruction issue time, each scheduler issues one instruction for one of its assigned warps that is ready to execute, if any.

An SM has:

- a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- a unified data cache and shared memory with a total size of 128 KB (*Volta*) or 96 KB (*Turing*).

Shared memory is partitioned out of unified data cache, and can be configured to various sizes (See *Shared Memory*.) The remaining data cache serves as an L1 cache and is also used by the texture unit that implements the various addressing and data filtering modes mentioned in *Texture and Surface Memory*.

[28] 2 FP64 cores for double-precision arithmetic operations for devices of compute capabilities 7.5

20.6.2. Independent Thread Scheduling

The **NVIDIA Volta GPU Architecture** introduces *Independent Thread Scheduling* among threads in a warp, enabling intra-warp synchronization patterns previously unavailable and simplifying code changes when porting CPU code. However, this can lead to a rather different set of threads participating in the executed code than intended if the developer made assumptions about warp-synchronicity of previous hardware architectures.

Below are code patterns of concern and suggested corrective actions for Volta-safe code.

1. For applications using warp intrinsics (`__shfl*`, `__any`, `__all`, `__ballot`), it is necessary that developers port their code to the new, safe, synchronizing counterpart, with the `*_sync` suffix. The new warp intrinsics take in a mask of threads that explicitly define which lanes (threads of a warp) must participate in the warp intrinsic. See *Warp Vote Functions* and *Warp Shuffle Functions* for details.

Since the intrinsics are available with CUDA 9.0+, (if necessary) code can be executed conditionally with the following preprocessor macro:

```
#if defined(CUDART_VERSION) && CUDART_VERSION >= 9000
// *_sync intrinsic
#endif
```

These intrinsics are available on all architectures, not just **NVIDIA Volta GPU Architecture** or **NVIDIA Turing GPU Architecture**, and in most cases a single code-base will suffice for all architectures. Note, however, that for *Pascal* and earlier architectures, all threads in mask must execute the same warp intrinsic instruction in convergence, and the union of all values in mask must be equal to the warp's active mask. The following code pattern is valid on **NVIDIA Volta GPU Architecture**, but not on *Pascal* or earlier architectures.

```
if (tid % warpSize < 16) {
    ...
    float swapped = __shfl_xor_sync(0xffffffff, val, 16);
    ...
} else {
    ...
    float swapped = __shfl_xor_sync(0xffffffff, val, 16);
    ...
}
```

The replacement for `__ballot(1)` is `__activemask()`. Note that threads within a warp can diverge even within a single code path. As a result, `__activemask()` and `__ballot(1)` may return only a subset of the threads on the current code path. The following invalid code example sets bit `i` of `output` to 1 when `data[i]` is greater than `threshold`. `__activemask()` is used in an attempt to enable cases where `dataLen` is not a multiple of 32.

```
// Sets bit in output[] to 1 if the correspond element in data[i]
// is greater than 'threshold', using 32 threads in a warp.

for (int i = warpLane; i < dataLen; i += warpSize) {
    unsigned active = __activemask();
    unsigned bitPack = __ballot_sync(active, data[i] > threshold);
    if (warpLane == 0) {
        output[i / 32] = bitPack;
    }
}
```

This code is invalid because CUDA does not guarantee that the warp will diverge ONLY at the loop condition. When divergence happens for other reasons, conflicting results will be computed for the same 32-bit output element by different subsets of threads in the warp. A correct code might use a non-divergent loop condition together with `__ballot_sync()` to safely enumerate the set of threads in the warp participating in the threshold calculation as follows.

```
for (int i = warpLane; i - warpLane < dataLen; i += warpSize) {
    unsigned active = __ballot_sync(0xFFFFFFFF, i < dataLen);
    if (i < dataLen) {
        unsigned bitPack = __ballot_sync(active, data[i] > threshold);
        if (warpLane == 0) {
            output[i / 32] = bitPack;
        }
    }
}
```

Discovery Pattern demonstrates a valid use case for `__activemask()`.

1. If applications have warp-synchronous codes, they will need to insert the new `__syncwarp()` warp-wide barrier synchronization instruction between any steps where data is exchanged between threads via global or shared memory. Assumptions that code is executed in lockstep or that reads/writes from separate threads are visible across a warp without synchronization are invalid.

```
__shared__ float s_buff[BLOCK_SIZE];
s_buff[tid] = val;
__syncthreads();

// Inter-warp reduction
for (int i = BLOCK_SIZE / 2; i >= 32; i /= 2) {
    if (tid < i) {
        s_buff[tid] += s_buff[tid+i];
    }
    __syncthreads();
}

// Intra-warp reduction
// Butterfly reduction simplifies syncwarp mask
if (tid < 32) {
    float temp;
    temp = s_buff[tid ^ 16]; __syncwarp();
    s_buff[tid] += temp;     __syncwarp();
    temp = s_buff[tid ^ 8];  __syncwarp();
    s_buff[tid] += temp;     __syncwarp();
    temp = s_buff[tid ^ 4];  __syncwarp();
    s_buff[tid] += temp;     __syncwarp();
    temp = s_buff[tid ^ 2];  __syncwarp();
    s_buff[tid] += temp;     __syncwarp();
}

if (tid == 0) {
    *output = s_buff[0] + s_buff[1];
}
__syncthreads();
```

2. Although `__syncthreads()` has been consistently documented as synchronizing all threads in the thread block, *Pascal* and prior architectures could only enforce synchronization at the warp level. In certain cases, this allowed a barrier to succeed without being executed by every thread as

long as at least some thread in every warp reached the barrier. Starting with **NVIDIA Volta GPU Architecture**, the CUDA built-in `__syncthreads()` and PTX instruction `bar.sync` (and their derivatives) are enforced per thread and thus will not succeed until reached by all non-exited threads in the block. Code exploiting the previous behavior will likely deadlock and must be modified to ensure that all non-exited threads reach the barrier.

The `racecheck` and `synccheck` tools provided by `compute-saniter` can help with locating violations.

To aid migration while implementing the above-mentioned corrective actions, developers can opt-in to the Pascal scheduling model that does not support independent thread scheduling. See *Application Compatibility* for details.

20.6.3. Global Memory

Global memory behaves the same way as in devices of compute capability 5.x (See *Global Memory*).

20.6.4. Shared Memory

The amount of the unified data cache reserved for shared memory is configurable on a per kernel basis. For the *Volta* architecture (compute capability 7.0), the unified data cache has a size of 128 KB, and the shared memory capacity can be set to 0, 8, 16, 32, 64 or 96 KB. For the *Turing* architecture (compute capability 7.5), the unified data cache has a size of 96 KB, and the shared memory capacity can be set to either 32 KB or 64 KB. Unlike Kepler, the driver automatically configures the shared memory capacity for each kernel to avoid shared memory occupancy bottlenecks while also allowing concurrent execution with already launched kernels where possible. In most cases, the driver's default behavior should provide optimal performance.

Because the driver is not always aware of the full workload, it is sometimes useful for applications to provide additional hints regarding the desired shared memory configuration. For example, a kernel with little or no shared memory use may request a larger carveout in order to encourage concurrent execution with later kernels that require more shared memory. The new `cudaFuncSetAttribute()` API allows applications to set a preferred shared memory capacity, or `carveout`, as a percentage of the maximum supported shared memory capacity (96 KB for *Volta*, and 64 KB for *Turing*).

`cudaFuncSetAttribute()` relaxes enforcement of the preferred shared capacity compared to the legacy `cudaFuncSetCacheConfig()` API introduced with Kepler. The legacy API treated shared memory capacities as hard requirements for kernel launch. As a result, interleaving kernels with different shared memory configurations would needlessly serialize launches behind shared memory reconfigurations. With the new API, the carveout is treated as a hint. The driver may choose a different configuration if required to execute the function or to avoid thrashing.

```
// Device code
__global__ void MyKernel(...)
{
    __shared__ float buffer[BLOCK_DIM];
    ...
}

// Host code
int carveout = 50; // prefer shared memory capacity 50% of maximum
// Named Carveout Values:
// carveout = cudaSharedmemCarveoutDefault;   //  (-1)
```

(continues on next page)

(continued from previous page)

```
// carveout = cudaSharedmemCarveoutMaxL1;     //   (0)
// carveout = cudaSharedmemCarveoutMaxShared; // (100)
cudaFuncSetAttribute(MyKernel, cudaFuncAttributePreferredSharedMemoryCarveout,
↪carveout);
MyKernel <<<gridDim, BLOCK_DIM>>>(...);
```

In addition to an integer percentage, several convenience enums are provided as listed in the code comments above. Where a chosen integer percentage does not map exactly to a supported capacity (SM 7.0 devices support shared capacities of 0, 8, 16, 32, 64, or 96 KB), the next larger capacity is used. For instance, in the example above, 50% of the 96 KB maximum is 48 KB, which is not a supported shared memory capacity. Thus, the preference is rounded up to 64 KB.

Compute capability 7.x devices allow a single thread block to address the full capacity of shared memory: 96 KB on *Volta*, 64 KB on *Turing*. Kernels relying on shared memory allocations over 48 KB per block are architecture-specific, as such they must use dynamic shared memory (rather than statically sized arrays) and require an explicit opt-in using `cudaFuncSetAttribute()` as follows.

```
// Device code
__global__ void MyKernel(...)
{
    extern __shared__ float buffer[];
    ...
}

// Host code
int maxbytes = 98304; // 96 KB
cudaFuncSetAttribute(MyKernel, cudaFuncAttributeMaxDynamicSharedMemorySize, maxbytes);
MyKernel <<<gridDim, blockDim, maxbytes>>>(...);
```

Otherwise, shared memory behaves the same way as for devices of compute capability 5.x (See *Shared Memory*).

20.7. Compute Capability 8.x

20.7.1. Architecture

A Streaming Multiprocessor (SM) consists of:

- 64 FP32 cores for single-precision arithmetic operations in devices of compute capability 8.0 and 128 FP32 cores in devices of compute capability 8.6, 8.7 and 8.9,
- 32 FP64 cores for double-precision arithmetic operations in devices of compute capability 8.0 and 2 FP64 cores in devices of compute capability 8.6, 8.7 and 8.9
- 64 INT32 cores for integer math,
- 4 mixed-precision Third-Generation Tensor Cores supporting half-precision (fp16), `__nv_bfloat16`, `tf32`, sub-byte and double precision (fp64) matrix arithmetic for compute capabilities 8.0, 8.6 and 8.7 (see *Warp Matrix Functions* for details),
- 4 mixed-precision Fourth-Generation Tensor Cores supporting `fp8`, `fp16`, `__nv_bfloat16`, `tf32`, sub-byte and `fp64` for compute capability 8.9 (see *Warp Matrix Functions* for details),
- 16 special function units for single-precision floating-point transcendental functions,

- 4 warp schedulers.

An SM statically distributes its warps among its schedulers. Then, at every instruction issue time, each scheduler issues one instruction for one of its assigned warps that is ready to execute, if any.

An SM has:

- a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- a unified data cache and shared memory with a total size of 192 KB for devices of compute capability 8.0 and 8.7 (1.5x *Volta*'s 128 KB capacity) and 128 KB for devices of compute capabilities 8.6 and 8.9.

Shared memory is partitioned out of the unified data cache, and can be configured to various sizes (see *Shared Memory*). The remaining data cache serves as an L1 cache and is also used by the texture unit that implements the various addressing and data filtering modes mentioned in *Texture and Surface Memory*.

20.7.2. Global Memory

Global memory behaves the same way as for devices of compute capability 5.x (See *Global Memory*).

20.7.3. Shared Memory

Similar to the *Volta architecture*, the amount of the unified data cache reserved for shared memory is configurable on a per kernel basis. For the **NVIDIA Ampere GPU Architecture**, the unified data cache has a size of 192 KB for devices of compute capability 8.0 and 8.7 and 128 KB for devices of compute capabilities 8.6 and 8.9. The shared memory capacity can be set to 0, 8, 16, 32, 64, 100, 132 or 164 KB for devices of compute capability 8.0 and 8.7, and to 0, 8, 16, 32, 64 or 100 KB for devices of compute capabilities 8.6 and 8.9.

An application can set the `carveout`, i.e., the preferred shared memory capacity, with the `cudaFuncSetAttribute()`.

```
cudaFuncSetAttribute(kernel_name, cudaFuncAttributePreferredSharedMemoryCarveout,
↪carveout);
```

The API can specify the carveout either as an integer percentage of the maximum supported shared memory capacity of 164 KB for devices of compute capability 8.0 and 8.7 and 100 KB for devices of compute capabilities 8.6 and 8.9 respectively, or as one of the following values: {`cudaSharedmemCarveoutDefault`, `cudaSharedmemCarveoutMaxL1`, or `cudaSharedmemCarveoutMaxShared`. When using a percentage, the carveout is rounded up to the nearest supported shared memory capacity. For example, for devices of compute capability 8.0, 50% will map to a 100 KB carveout instead of an 82 KB one. Setting the `cudaFuncAttributePreferredSharedMemoryCarveout` is considered a hint by the driver; the driver may choose a different configuration, if needed.

Devices of compute capability 8.0 and 8.7 allow a single thread block to address up to 163 KB of shared memory, while devices of compute capabilities 8.6 and 8.9 allow up to 99 KB of shared memory. Kernels relying on shared memory allocations over 48 KB per block are architecture-specific, and must use dynamic shared memory rather than statically sized shared memory arrays. These kernels require an explicit opt-in by using `cudaFuncSetAttribute()` to set the `cudaFuncAttributeMaxDynamicSharedMemorySize`; see *Shared Memory* for the **NVIDIA Volta GPU Architecture**.

Note that the maximum amount of shared memory per thread block is smaller than the maximum shared memory partition available per SM. The 1 KB of shared memory not made available to a thread block is reserved for system use.

20.8. Compute Capability 9.0

20.8.1. Architecture

A Streaming Multiprocessor (SM) consists of:

- 128 FP32 cores for single-precision arithmetic operations,
- 64 FP64 cores for double-precision arithmetic operations,
- 64 INT32 cores for integer math,
- 4 mixed-precision fourth-generation Tensor Cores supporting the new FP8 input type in either `E4M3` or `E5M2` for exponent (E) and mantissa (M), half-precision (fp16), `__nv_bfloat16`, `tf32`, INT8 and double precision (fp64) matrix arithmetic (see *Warp Matrix Functions* for details) with sparsity support,
- 16 special function units for single-precision floating-point transcendental functions,
- 4 warp schedulers.

An SM statically distributes its warps among its schedulers. Then, at every instruction issue time, each scheduler issues one instruction for one of its assigned warps that is ready to execute, if any.

An SM has:

- a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- a unified data cache and shared memory with a total size of 256 KB for devices of compute capability 9.0 (1.33x **NVIDIA Ampere GPU Architecture's** 192 KB capacity).

Shared memory is partitioned out of the unified data cache, and can be configured to various sizes (see *Shared Memory*). The remaining data cache serves as an L1 cache and is also used by the texture unit that implements the various addressing and data filtering modes mentioned in *Texture and Surface Memory*.

20.8.2. Global Memory

Global memory behaves the same way as for devices of compute capability 5.x (See *Global Memory*).

20.8.3. Shared Memory

Similar to the *NVIDIA Ampere GPU architecture*, the amount of the unified data cache reserved for shared memory is configurable on a per kernel basis. For the *NVIDIA H100 Tensor Core GPU architecture*, the unified data cache has a size of 256 KB for devices of compute capability 9.0. The shared memory capacity can be set to 0, 8, 16, 32, 64, 100, 132, 164, 196 or 228 KB.

As with the *NVIDIA Ampere GPU architecture*, an application can configure its preferred shared memory capacity, i.e., the `carveout`. Devices of compute capability 9.0 allow a single thread block to address up to 227 KB of shared memory. Kernels relying on shared memory allocations over 48 KB per block are architecture-specific, and must use dynamic shared memory rather than statically sized shared memory arrays. These kernels require an explicit opt-in by using `cudaFuncSetAttribute()` to set the `cudaFuncAttributeMaxDynamicSharedMemorySize`; see *Shared Memory* for the **NVIDIA Volta GPU Architecture**.

Note that the maximum amount of shared memory per thread block is smaller than the maximum shared memory partition available per SM. The 1 KB of shared memory not made available to a thread block is reserved for system use.

20.8.4. Features Accelerating Specialized Computations

The NVIDIA Hopper GPU architecture includes features to accelerate matrix multiply-accumulate (MMA) computations with:

- asynchronous execution of MMA instructions
- MMA instructions acting on large matrices spanning a warp-group
- dynamic reassignment of register capacity among warp-groups to support even larger matrices, and
- operand matrices accessed directly from shared memory

See the PTX ISA for more details.

This feature set is only available within the CUDA compilation toolchain through inline PTX.

It is strongly recommended that applications utilize this complex feature set through CUDA-X libraries such as cuBLAS, cuDNN, or cuFFT.

It is strongly recommended that device kernels utilize this complex feature set through CUTLASS, a collection of CUDA C++ template abstractions for implementing high-performance matrix-multiplication (GEMM) and related computations at all levels and scales within CUDA.

20.9. Compute Capability 10.0

20.9.1. Architecture

A Streaming Multiprocessor (SM) consists of:

- 128 FP32 cores for single-precision arithmetic operations,
- 64 FP64 cores for double-precision arithmetic operations,

- 64 INT32 cores for integer math,
- 4 mixed-precision fifth-generation Tensor Cores supporting FP8 input type in either E4M3 or E5M2 for exponent (E) and mantissa (M), half-precision (fp16), `__nv_bfloat16`, `tf32`, INT8 and double precision (fp64) matrix arithmetic (see *Warp Matrix Functions* for details) with sparsity support,
- 16 special function units for single-precision floating-point transcendental functions,
- 4 warp schedulers.

An SM statically distributes its warps among its schedulers. Then, at every instruction issue time, each scheduler issues one instruction for one of its assigned warps that is ready to execute, if any.

An SM has:

- a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- a unified data cache and shared memory with a total size of 256 KB for devices of compute capability 10.0

Shared memory is partitioned out of the unified data cache, and can be configured to various sizes (see *Shared Memory*). The remaining data cache serves as an L1 cache and is also used by the texture unit that implements the various addressing and data filtering modes mentioned in *Texture and Surface Memory*.

20.9.2. Global Memory

Global memory behaves the same way as for devices of compute capability 5.x (See *Global Memory*).

20.9.3. Shared Memory

The amount of the unified data cache reserved for shared memory is configurable on a per kernel basis and is identical to *compute capability 9.0*. The unified data cache has a size of 256 KB for devices of compute capability 10.0. The shared memory capacity can be set to 0, 8, 16, 32, 64, 100, 132, 164, 196 or 228 KB.

As with the *NVIDIA Ampere GPU architecture*, an application can configure its preferred shared memory capacity, i.e., the `carveout`. Devices of compute capability 10.0 allow a single thread block to address up to 227 KB of shared memory. Kernels relying on shared memory allocations over 48 KB per block are architecture-specific, and must use dynamic shared memory rather than statically sized shared memory arrays. These kernels require an explicit opt-in by using `cudaFuncSetAttribute()` to set the `cudaFuncAttributeMaxDynamicSharedMemorySize`; see *Shared Memory* for the Volta architecture.

Note that the maximum amount of shared memory per thread block is smaller than the maximum shared memory partition available per SM. The 1 KB of shared memory not made available to a thread block is reserved for system use.

20.9.4. Features Accelerating Specialized Computations

The NVIDIA Blackwell GPU architecture extends features to accelerate matrix multiply-accumulate (MMA) from the NVIDIA Hopper GPU architecture.

See the PTX ISA for more details.

This feature set is only available within the CUDA compilation toolchain through inline PTX.

It is strongly recommended that applications utilize this complex feature set through CUDA-X libraries such as cuBLAS, cuDNN, or cuFFT.

It is strongly recommended that device kernels utilize this complex feature set through CUTLASS, a collection of CUDA C++ template abstractions for implementing high-performance matrix-multiplication (GEMM) and related computations at all levels and scales within CUDA.

20.10. Compute Capability 12.0

20.10.1. Architecture

A Streaming Multiprocessor (SM) consists of:

- 128 FP32 cores for single-precision arithmetic operations,
- 2 FP64 cores for double-precision arithmetic operations,
- 64 INT32 cores for integer math,
- Mixed-precision fifth-generation Tensor Core(s) supporting FP8 input type in either `E4M3` or `E5M2` for exponent (E) and mantissa (M), half-precision (fp16), `__nv_bfloat16`, `tf32`, INT8 and double precision (fp64) matrix arithmetic (see *Warp Matrix Functions* for details) with sparsity support,
- 16 special function units for single-precision floating-point transcendental functions,
- 4 warp schedulers.

An SM statically distributes its warps among its schedulers. Then, at every instruction issue time, each scheduler issues one instruction for one of its assigned warps that is ready to execute, if any.

An SM has:

- a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- a unified data cache and shared memory with a total size of 100 KB for devices of compute capability 12.0

Shared memory is partitioned out of the unified data cache, and can be configured to various sizes (see *Shared Memory*). The remaining data cache serves as an L1 cache and is also used by the texture unit that implements the various addressing and data filtering modes mentioned in *Texture and Surface Memory*.

20.10.2. Global Memory

Global memory behaves the same way as for devices of compute capability 5.x (See *Global Memory*).

20.10.3. Shared Memory

The amount of the unified data cache reserved for shared memory is configurable on a per kernel basis. The unified data cache has a size of 100 KB for devices of compute capability 12.0. The shared memory capacity can be set to 0, 8, 16, 32, 64, or 100 KB.

As with the *NVIDIA Ampere GPU architecture*, an application can configure its preferred shared memory capacity, i.e., the `carveout`. Devices of compute capability 12.0 allow a single thread block to address up to 99 KB of shared memory. Kernels relying on shared memory allocations over 48 KB per block are architecture-specific, and must use dynamic shared memory rather than statically sized shared memory arrays. These kernels require an explicit opt-in by using `cudaFuncSetAttribute()` to set the `cudaFuncAttributeMaxDynamicSharedMemorySize`; see *Shared Memory* for the Volta architecture.

Note that the maximum amount of shared memory per thread block is smaller than the maximum shared memory partition available per SM. The 1 KB of shared memory not made available to a thread block is reserved for system use.

20.10.4. Features Accelerating Specialized Computations

The NVIDIA Blackwell GPU architecture extends features to accelerate matrix multiply-accumulate (MMA) from the NVIDIA Hopper GPU architecture.

See the PTX ISA for more details.

This feature set is only available within the CUDA compilation toolchain through inline PTX.

It is strongly recommended that applications utilize this complex feature set through CUDA-X libraries such as cuBLAS, cuDNN, or cuFFT.

It is strongly recommended that device kernels utilize this complex feature set through CUTLASS, a collection of CUDA C++ template abstractions for implementing high-performance matrix-multiplication (GEMM) and related computations at all levels and scales within CUDA.

Chapter 21. Driver API

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

This section assumes knowledge of the concepts described in *CUDA Runtime*.

The driver API is implemented in the `cuda` dynamic library (`cuda.dll` or `cuda.so`) which is copied on the system during the installation of the device driver. All its entry points are prefixed with cu.

It is a handle-based, imperative API: Most objects are referenced by opaque handles that may be specified to functions to manipulate the objects.

The objects available in the driver API are summarized in Table 28.

Table 28: Objects Available in the CUDA Driver API

Object	Handle	Description
Device	CUdevice	CUDA-enabled device
Context	CUcontext	Roughly equivalent to a CPU process
Module	CUmodule	Roughly equivalent to a dynamic library
Function	CUfunction	Kernel
Heap memory	CUdeviceptr	Pointer to device memory
CUDA array	CUarray	Opaque container for one-dimensional or two-dimensional data on the device, readable via texture or surface references
Texture object	CUtexref	Object that describes how to interpret texture memory data
Surface reference	CUsurfref	Object that describes how to read or write CUDA arrays
Stream	CUstream	Object that describes a CUDA stream
Event	CUevent	Object that describes a CUDA event

The driver API must be initialized with `cuInit()` before any function from the driver API is called. A CUDA context must then be created that is attached to a specific device and made current to the calling host thread as detailed in *Context*.

Within a CUDA context, kernels are explicitly loaded as PTX or binary objects by the host code as described in *Module*. Kernels written in C++ must therefore be compiled separately into *PTX* or binary objects. Kernels are launched using API entry points as described in *Kernel Execution*.

Any application that wants to run on future device architectures must load *PTX*, not binary code. This is because binary code is architecture-specific and therefore incompatible with future architectures, whereas *PTX* code is compiled to binary code at load time by the device driver.

Here is the host code of the sample from *Kernels* written using the driver API:

```
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Initialize
    cuInit(0);

    // Get number of devices supporting CUDA
    int deviceCount = 0;
    cuDeviceGetCount(&deviceCount);
    if (deviceCount == 0) {
        printf("There is no device supporting CUDA.\n");
        exit (0);
    }

    // Get handle for device 0
    CUdevice cuDevice;
    cuDeviceGet(&cuDevice, 0);

    // Create context
    CUcontext cuContext;
    cuCtxCreate(&cuContext, NULL, 0, cuDevice);

    // Create module from binary file
    CUmodule cuModule;
    cuModuleLoad(&cuModule, "VecAdd.ptx");

    // Allocate vectors in device memory
    CUdeviceptr d_A;
    cuMemAlloc(&d_A, size);
    CUdeviceptr d_B;
    cuMemAlloc(&d_B, size);
    CUdeviceptr d_C;
    cuMemAlloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cuMemcpyHtoD(d_A, h_A, size);
```

(continues on next page)

(continued from previous page)

```
    cuMemcpyHtoD(d_B, h_B, size);

    // Get function handle from module
    CUfunction vecAdd;
    cuModuleGetFunction(&vecAdd, cuModule, "VecAdd");

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    void* args[] = { &d_A, &d_B, &d_C, &N };
    cuLaunchKernel(vecAdd,
                   blocksPerGrid, 1, 1, threadsPerBlock, 1, 1,
                   0, 0, args, 0);

    ...
}
```

Full code can be found in the `vectorAddDrv` CUDA sample.

21.1. Context

A CUDA context is analogous to a CPU process. All resources and actions performed within the driver API are encapsulated inside a CUDA context, and the system automatically cleans up these resources when the context is destroyed. Besides objects such as modules and texture or surface references, each context has its own distinct address space. As a result, `CUdeviceptr` values from different contexts reference different memory locations.

A host thread may have only one device context current at a time. When a context is created with `cuCtxCreate()`, it is made current to the calling host thread. CUDA functions that operate in a context (most functions that do not involve device enumeration or context management) will return `CUDA_ERROR_INVALID_CONTEXT` if a valid context is not current to the thread.

Each host thread has a stack of current contexts. `cuCtxCreate()` pushes the new context onto the top of the stack. `cuCtxPopCurrent()` may be called to detach the context from the host thread. The context is then "floating" and may be pushed as the current context for any host thread. `cuCtxPopCurrent()` also restores the previous current context, if any.

A usage count is also maintained for each context. `cuCtxCreate()` creates a context with a usage count of 1. `cuCtxAttach()` increments the usage count and `cuCtxDetach()` decrements it. A context is destroyed when the usage count goes to 0 when calling `cuCtxDetach()` or `cuCtxDestroy()`.

The driver API is interoperable with the runtime and it is possible to access the *primary context* (see *Initialization*) managed by the runtime from the driver API via `cuDevicePrimaryCtxRetain()`.

Usage count facilitates interoperability between third party authored code operating in the same context. For example, if three libraries are loaded to use the same context, each library would call `cuCtxAttach()` to increment the usage count and `cuCtxDetach()` to decrement the usage count when the library is done using the context. For most libraries, it is expected that the application will have created a context before loading or initializing the library; that way, the application can create the context using its own heuristics, and the library simply operates on the context handed to it. Libraries that wish to create their own contexts - unbeknownst to their API clients who may or may not have created contexts of their own - would use `cuCtxPushCurrent()` and `cuCtxPopCurrent()` as illustrated in the following figure.

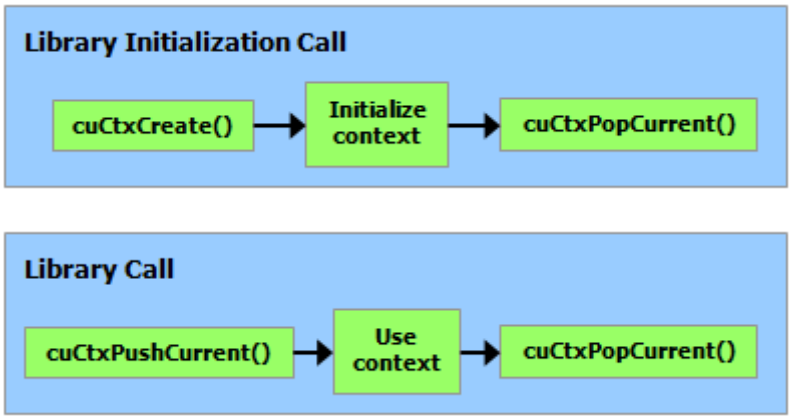


Figure 41: Library Context Management

21.2. Module

Modules are dynamically loadable packages of device code and data, akin to DLLs in Windows, that are output by nvcc (see *Compilation with NVCC*). The names for all symbols, including functions, global variables, and texture or surface references, are maintained at module scope so that modules written by independent third parties may interoperate in the same CUDA context.

This code sample loads a module and retrieves a handle to some kernel:

```
CUmodule cuModule;
cuModuleLoad(&cuModule, "myModule.ptx");
CUfunction myKernel;
cuModuleGetFunction(&myKernel, cuModule, "MyKernel");
```

This code sample compiles and loads a new module from PTX code and parses compilation errors:

```
#define BUFFER_SIZE 8192
CUmodule cuModule;
CUjit_option options[3];
void* values[3];
char* PTXCode = "some PTX code";
char error_log[BUFFER_SIZE];
int err;
options[0] = CU_JIT_ERROR_LOG_BUFFER;
values[0]  = (void*)error_log;
options[1] = CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES;
values[1]  = (void*)BUFFER_SIZE;
options[2] = CU_JIT_TARGET_FROM_CUCONTEXT;
values[2]  = 0;
err = cuModuleLoadDataEx(&cuModule, PTXCode, 3, options, values);
if (err != CUDA_SUCCESS)
    printf("Link error:\n%s\n", error_log);
```

This code sample compiles, links, and loads a new module from multiple PTX codes and parses link and compilation errors:

```
#define BUFFER_SIZE 8192
CUmodule cuModule;
```

(continues on next page)

(continued from previous page)

```
CUjit_option options[6];
void* values[6];
float walltime;
char error_log[BUFFER_SIZE], info_log[BUFFER_SIZE];
char* PTXCode0 = "some PTX code";
char* PTXCode1 = "some other PTX code";
CUlinkState linkState;
int err;
void* cubin;
size_t cubinSize;
options[0] = CU_JIT_WALL_TIME;
values[0] = (void*)&walltime;
options[1] = CU_JIT_INFO_LOG_BUFFER;
values[1] = (void*)info_log;
options[2] = CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES;
values[2] = (void*)BUFFER_SIZE;
options[3] = CU_JIT_ERROR_LOG_BUFFER;
values[3] = (void*)error_log;
options[4] = CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES;
values[4] = (void*)BUFFER_SIZE;
options[5] = CU_JIT_LOG_VERBOSE;
values[5] = (void*)1;
cuLinkCreate(6, options, values, &linkState);
err = cuLinkAddData(linkState, CU_JIT_INPUT_PTX,
                    (void*)PTXCode0, strlen(PTXCode0) + 1, 0, 0, 0, 0);
if (err != CUDA_SUCCESS)
    printf("Link error:\n%s\n", error_log);
err = cuLinkAddData(linkState, CU_JIT_INPUT_PTX,
                    (void*)PTXCode1, strlen(PTXCode1) + 1, 0, 0, 0, 0);
if (err != CUDA_SUCCESS)
    printf("Link error:\n%s\n", error_log);
cuLinkComplete(linkState, &cubin, &cubinSize);
printf("Link completed in %fms. Linker Output:\n%s\n", walltime, info_log);
cuModuleLoadData(cuModule, cubin);
cuLinkDestroy(linkState);
```

Full code can be found in the `ptxjit` CUDA sample.

21.3. Kernel Execution

`cuLaunchKernel()` launches a kernel with a given execution configuration.

Parameters are passed either as an array of pointers (next to last parameter of `cuLaunchKernel()`) where the nth pointer corresponds to the nth parameter and points to a region of memory from which the parameter is copied, or as one of the extra options (last parameter of `cuLaunchKernel()`).

When parameters are passed as an extra option (the `CU_LAUNCH_PARAM_BUFFER_POINTER` option), they are passed as a pointer to a single buffer where parameters are assumed to be properly offset with respect to each other by matching the alignment requirement for each parameter type in device code.

Alignment requirements in device code for the built-in vector types are listed in Table 7. For all other basic types, the alignment requirement in device code matches the alignment requirement in host code and can therefore be obtained using `__alignof()`. The only exception is when the host compiler

aligns `double` and `long long` (and `long` on a 64-bit system) on a one-word boundary instead of a two-word boundary (for example, using `gcc`'s compilation flag `-mno-align-double`) since in device code these types are always aligned on a two-word boundary.

`CUdeviceptr` is an integer, but represents a pointer, so its alignment requirement is `__alignof(void*)`.

The following code sample uses a macro (`ALIGN_UP()`) to adjust the offset of each parameter to meet its alignment requirement and another macro (`ADD_TO_PARAM_BUFFER()`) to add each parameter to the parameter buffer passed to the `CU_LAUNCH_PARAM_BUFFER_POINTER` option.

```
#define ALIGN_UP(offset, alignment) \
      (offset) = ((offset) + (alignment) - 1) & ~((alignment) - 1)

char paramBuffer[1024];
size_t paramBufferSize = 0;

#define ADD_TO_PARAM_BUFFER(value, alignment)                   \
    do {                                                        \
        paramBufferSize = ALIGN_UP(paramBufferSize, alignment); \
        memcpy(paramBuffer + paramBufferSize,                   \
               &(value), sizeof(value));                        \
        paramBufferSize += sizeof(value);                       \
    } while (0)

int i;
ADD_TO_PARAM_BUFFER(i, __alignof(i));
float4 f4;
ADD_TO_PARAM_BUFFER(f4, 16); // float4's alignment is 16
char c;
ADD_TO_PARAM_BUFFER(c, __alignof(c));
float f;
ADD_TO_PARAM_BUFFER(f, __alignof(f));
CUdeviceptr devPtr;
ADD_TO_PARAM_BUFFER(devPtr, __alignof(devPtr));
float2 f2;
ADD_TO_PARAM_BUFFER(f2, 8); // float2's alignment is 8

void* extra[] = {
    CU_LAUNCH_PARAM_BUFFER_POINTER, paramBuffer,
    CU_LAUNCH_PARAM_BUFFER_SIZE,    &paramBufferSize,
    CU_LAUNCH_PARAM_END
};
cuLaunchKernel(cuFunction,
               blockWidth, blockHeight, blockDepth,
               gridWidth, gridHeight, gridDepth,
               0, 0, 0, extra);
```

The alignment requirement of a structure is equal to the maximum of the alignment requirements of its fields. The alignment requirement of a structure that contains built-in vector types, `CUdeviceptr`, or non-aligned `double` and `long long`, might therefore differ between device code and host code. Such a structure might also be padded differently. The following structure, for example, is not padded at all in host code, but it is padded in device code with 12 bytes after field `f` since the alignment requirement for field `f4` is 16.

```
typedef struct {
    float  f;
    float4 f4;
```

(continues on next page)

(continued from previous page)

```
} myStruct;
```

21.4. Interoperability between Runtime and Driver APIs

An application can mix runtime API code with driver API code.

If a context is created and made current via the driver API, subsequent runtime calls will pick up this context instead of creating a new one.

If the runtime is initialized (implicitly as mentioned in *CUDA Runtime*), `cuCtxGetCurrent()` can be used to retrieve the context created during initialization. This context can be used by subsequent driver API calls.

The implicitly created context from the runtime is called the *primary context* (see *Initialization*). It can be managed from the driver API with the Primary Context Management functions.

Device memory can be allocated and freed using either API. `CUdeviceptr` can be cast to regular pointers and vice-versa:

```
CUdeviceptr devPtr;
float* d_data;

// Allocation using driver API
cuMemAlloc(&devPtr, size);
d_data = (float*)devPtr;

// Allocation using runtime API
cudaMalloc(&d_data, size);
devPtr = (CUdeviceptr)d_data;
```

In particular, this means that applications written using the driver API can invoke libraries written using the runtime API (such as cuFFT, cuBLAS, ...).

All functions from the device and version management sections of the reference manual can be used interchangeably.

21.5. Driver Entry Point Access

21.5.1. Introduction

The `Driver Entry Point Access APIs` provide a way to retrieve the address of a CUDA driver function. Starting from CUDA 11.3, users can call into available CUDA driver APIs using function pointers obtained from these APIs.

These APIs provide functionality similar to their counterparts, dlsym on POSIX platforms and GetProcAddress on Windows. The provided APIs will let users:

- Retrieve the address of a driver function using the `CUDA Driver API.`

- Retrieve the address of a driver function using the CUDA Runtime API.
- Request *per-thread default stream* version of a CUDA driver function. For more details, see *Retrieve Per-thread Default Stream Versions*.
- Access new CUDA features on older toolkits but with a newer driver.

21.5.2. Driver Function Typedefs

To help retrieve the CUDA Driver API entry points, the CUDA Toolkit provides access to headers containing the function pointer definitions for all CUDA driver APIs. These headers are installed with the CUDA Toolkit and are made available in the toolkit's `include/` directory. The table below summarizes the header files containing the `typedefs` for each CUDA API header file.

Table 29: Typedefs header files for CUDA driver APIs

API header file	API Typedef header file
cuda.h	cudaTypedefs.h
cudaGL.h	cudaGLTypedefs.h
cudaProfiler.h	cudaProfilerTypedefs.h
cudaVDPAU.h	cudaVDPAUTypedefs.h
cudaEGL.h	cudaEGLTypedefs.h
cudaD3D9.h	cudaD3D9Typedefs.h
cudaD3D10.h	cudaD3D10Typedefs.h
cudaD3D11.h	cudaD3D11Typedefs.h

The above headers do not define actual function pointers themselves; they define the typedefs for function pointers. For example, `cudaTypedefs.h` has the below typedefs for the driver API `cuMemAlloc`:

```
typedef CUresult (CUDAAPI *PFN_cuMemAlloc_v3020)(CUdeviceptr_v2 *dptr, size_t
↪bytesize);
typedef CUresult (CUDAAPI *PFN_cuMemAlloc_v2000)(CUdeviceptr_v1 *dptr, unsigned int
↪bytesize);
```

CUDA driver symbols have a version based naming scheme with a `_v*` extension in its name except for the first version. When the signature or the semantics of a specific CUDA driver API changes, we increment the version number of the corresponding driver symbol. In the case of the `cuMemAlloc` driver API, the first driver symbol name is `cuMemAlloc` and the next symbol name is `cuMemAlloc_v2`. The typedef for the first version which was introduced in CUDA 2.0 (2000) is `PFN_cuMemAlloc_v2000`. The typedef for the next version which was introduced in CUDA 3.2 (3020) is `PFN_cuMemAlloc_v3020`.

The `typedefs` can be used to more easily define a function pointer of the appropriate type in code:

```
PFN_cuMemAlloc_v3020 pfn_cuMemAlloc_v2;
PFN_cuMemAlloc_v2000 pfn_cuMemAlloc_v1;
```

21.5.3. Driver Function Retrieval

Using the Driver Entry Point Access APIs and the appropriate typedef, we can get the function pointer to any CUDA driver API.

21.5.3.1 Using the Driver API

The driver API requires CUDA version as an argument to get the ABI compatible version for the requested driver symbol. CUDA Driver APIs have a per-function ABI denoted with a `_v*` extension. For example, consider the versions of `cuStreamBeginCapture` and their corresponding `typedefs` from `cudaTypedefs.h`:

```
// cuda.h
CUresult CUDAAPI cuStreamBeginCapture(CUstream hStream);
CUresult CUDAAPI cuStreamBeginCapture_v2(CUstream hStream, CUstreamCaptureMode mode);

// cudaTypedefs.h
typedef CUresult (CUDAAPI *PFN_cuStreamBeginCapture_v10000)(CUstream hStream);
typedef CUresult (CUDAAPI *PFN_cuStreamBeginCapture_v10010)(CUstream hStream,
↪CUstreamCaptureMode mode);
```

From the above `typedefs` in the code snippet, version suffixes `_v10000` and `_v10010` indicate that the above APIs were introduced in CUDA 10.0 and CUDA 10.1 respectively.

```
#include <cudaTypedefs.h>

// Declare the entry points for cuStreamBeginCapture
PFN_cuStreamBeginCapture_v10000 pfn_cuStreamBeginCapture_v1;
PFN_cuStreamBeginCapture_v10010 pfn_cuStreamBeginCapture_v2;

// Get the function pointer to the cuStreamBeginCapture driver symbol
cuGetProcAddress("cuStreamBeginCapture", &pfn_cuStreamBeginCapture_v1, 10000, CU_GET_
↪PROC_ADDRESS_DEFAULT, &driverStatus);
// Get the function pointer to the cuStreamBeginCapture_v2 driver symbol
cuGetProcAddress("cuStreamBeginCapture", &pfn_cuStreamBeginCapture_v2, 10010, CU_GET_
↪PROC_ADDRESS_DEFAULT, &driverStatus);
```

Referring to the code snippet above, to retrieve the address to the `_v1` version of the driver API `cuStreamBeginCapture`, the CUDA version argument should be exactly 10.0 (10000). Similarly, the CUDA version for retrieving the address to the `_v2` version of the API should be 10.1 (10010). Specifying a higher CUDA version for retrieving a specific version of a driver API might not always be portable. For example, using 11030 here would still return the `_v2` symbol, but if a hypothetical `_v3` version is released in CUDA 11.3, the `cuGetProcAddress` API would start returning the newer `_v3` symbol instead when paired with a CUDA 11.3 driver. Since the ABI and function signatures of the `_v2` and `_v3` symbols might differ, calling the `_v3` function using the `_v10010` typedef intended for the `_v2` symbol would exhibit undefined behavior.

Note that requesting a driver API with an invalid CUDA version will return an error `CUDA_ERROR_NOT_FOUND`. In the above code examples, passing in a version less than 10000 (CUDA 10.0) would be invalid.

21.5.3.2 Using the Runtime API

The runtime API `cudaGetDriverEntryPointByVersion` uses the provided CUDA version to get the ABI compatible version for the requested driver symbol in the same way `cuGetProcAddress` does. In the below code snippet, the minimum CUDA version required would be CUDA 11.2 as `cuMemAllocAsync` was introduced then.

```
#include <cudaTypedefs.h>

int cudaVersion;
// Ensure a CUDA driver >= 11.2 is installed or we will get an error from
↪cuGetProcAddress
status = cuDriverGetVersion(&cudaVersion);
if (cudaVersion >= 11020) {

   // Declare the entry point
   PFN_cuMemAllocAsync_v11020 pfn_cuMemAllocAsync;

   // Intialize the entry point
   cudaGetDriverEntryPointByVersion("cuMemAllocAsync", &pfn_cuMemAllocAsync, 11020,
↪cudaEnableDefault, &driverStatus);

   // Call the entry point
   if(driverStatus == cudaDriverEntryPointSuccess && pfn_cuMemAllocAsync) {
       pfn_cuMemAllocAsync(...);
   }
}
```

21.5.3.3 Retrieve Per-thread Default Stream Versions

Some CUDA driver APIs can be configured to have *default stream* or *per-thread default stream* semantics. Driver APIs having *per-thread default stream* semantics are suffixed with *_ptsz* or *_ptds* in their name. For example, `cuLaunchKernel` has a *per-thread default stream* variant named `cuLaunchKernel_ptsz`. With the Driver Entry Point Access APIs, users can request for the *per-thread default stream* version of the driver API `cuLaunchKernel` instead of the *default stream* version. Configuring the CUDA driver APIs for *default stream* or *per-thread default stream* semantics affects the synchronization behavior. More details can be found here.

The *default stream* or *per-thread default stream* versions of a driver API can be obtained by one of the following ways:

- Use the compilation flag `--default-stream per-thread` or define the macro `CUDA_API_PER_THREAD_DEFAULT_STREAM` to get *per-thread default stream* behavior.
- Force *default stream* or *per-thread default stream* behavior using the flags `CU_GET_PROC_ADDRESS_LEGACY_STREAM/cudaEnableLegacyStream` or `CU_GET_PROC_ADDRESS_PER_THREAD_DEFAULT_STREAM/cudaEnablePerThreadDefaultStream` respectively.

21.5.3.4 Access New CUDA features

It is always recommended to install the latest CUDA toolkit to access new CUDA driver features, but if for some reason, a user does not want to update or does not have access to the latest toolkit, the API can be used to access new CUDA features with only an updated CUDA driver. For discussion, let us assume the user is on CUDA 12.3 and wants to use a new driver API `cuFoo` available in the CUDA 12.5 driver. The below code snippet illustrates this use-case:

```
int main()
{
    // Manually define the prototype as cudaTypedefs.h in CUDA 12.3 does not have the
↪cuFoo typedef
    typedef CUresult (CUDAAPI *PFN_cuFoo_v12050)(...);
    PFN_cuFoo_v12050 pfn_cuFoo = NULL;
    CUdriverProcAddressQueryResult driverStatus;
    int cudaVersion;

    // Ensure a CUDA driver >= 12.5 is installed or we will get an error from
↪cuGetProcAddress
    status = cuDriverGetVersion(&cudaVersion);
    if (cudaVersion >= 12050) {
        // Get the address for cuFoo API using cuGetProcAddress. Specify CUDA version as
        // 12050 since cuFoo was introduced then
        CUresult status = cuGetProcAddress("cuFoo", &pfn_cuFoo, 12050, CU_GET_PROC_
↪ADDRESS_DEFAULT, &driverStatus);

        if (status == CUDA_SUCCESS && pfn_cuFoo) {
            pfn_cuFoo(...);
        }
        else {
            printf("Cannot retrieve the address to cuFoo - driverStatus = %d\n",
↪driverStatus);
            assert(0);
        }
    }

    // rest of code here
}
```

In the next example, we discuss how to get a new version of an API released in a minor version of the CUDA Toolkit. Note that in the cuda.h header the version macro that would bump `cuDeviceGetUuid` to _v2 is not done until a major boundary. So during the 11.4+ releases the following example illustrates how to get the _v2 version.

Note in this case the original (not the _v2 version) typedef looks like:

```
typedef CUresult (CUDAAPI *PFN_cuDeviceGetUuid_v9020)(CUuuid *uuid, CUdevice_v1 dev);
```

But the _v2 version typedef looks like:

```
typedef CUresult (CUDAAPI *PFN_cuDeviceGetUuid_v11040)(CUuuid *uuid, CUdevice_v1 dev);
```

```
#include <cudaTypedefs.h>

CUuuid uuid;
CUdevice dev;
CUresult status;
```

(continues on next page)

(continued from previous page)

```
int cudaVersion;
CUdriverProcAddressQueryResult driverStatus;

status = cuDeviceGet(&dev, 0); // Get device 0
// handle status

status = cuDriverGetVersion(&cudaVersion);
// handle status

// Ensure a CUDA driver >= 11.4 is installed or we will get an error from
↪cuGetProcAddress
status = cuDriverGetVersion(&cudaVersion);
if (cudaVersion >= 11040) {
   PFN_cuDeviceGetUuid_v11040 pfn_cuDeviceGetUuid;
   status = cuGetProcAddress("cuDeviceGetUuid", &pfn_cuDeviceGetUuid, 11040, CU_GET_
↪PROC_ADDRESS_DEFAULT, &driverStatus);
   if(CUDA_SUCCESS == status && pfn_cuDeviceGetUuid) {
      pfn_cuDeviceGetUuid(&uuid, dev);
   }
}
```

21.5.4. Guidelines for cuGetProcAddress

Below are guidelines to keep in mind when using `cuGetProcAddress`.

- Code the CUDA version passed to `cuGetProcAddress` to match the typedef version (do not use a compile time constant such as `CUDA_VERSION` or a dynamic version such as returned from `cuDriverGetVersion`)
- Check the current driver version (such as from `cuDriverGetVersion`) is sufficient before calling `cuGetProcAddress` or an error is expected or an unexpected symbol may be returned

21.5.4.1 Guidelines for Runtime API Usage

Unless specified otherwise, the CUDA runtime API `cudaGetDriverEntryPointByVersion` will have similar guidelines as the driver entry point `cuGetProcAddress` since it allows for the user to request a specific CUDA driver version.

21.5.5. Determining cuGetProcAddress Failure Reasons

There are two types of errors with cuGetProcAddress. Those are (1) API/usage errors and (2) inability to find the driver API requested. The first error type will return error codes from the API via the CUresult return value. Things like passing NULL as the `pfn` variable or passing invalid `flags`.

The second error type encodes in the `CUdriverProcAddressQueryResult *symbolStatus` and can be used to help distinguish potential issues with the driver not being able to find the symbol requested. Take the following example:

```
// cuDeviceGetExecAffinitySupport was introduced in release CUDA 11.4
#include <cuda.h>
CUdriverProcAddressQueryResult driverStatus;
cudaVersion = ...;
status = cuGetProcAddress("cuDeviceGetExecAffinitySupport", &pfn, cudaVersion, 0, &
↪driverStatus);
if (CUDA_SUCCESS == status) {
    if (CU_GET_PROC_ADDRESS_VERSION_NOT_SUFFICIENT == driverStatus) {
        printf("We can use the new feature when you upgrade cudaVersion to 11.4, but
↪CUDA driver is good to go!\n");
        // Indicating cudaVersion was < 11.4 but run against a CUDA driver >= 11.4
    }
    else if (CU_GET_PROC_ADDRESS_SYMBOL_NOT_FOUND == driverStatus) {
        printf("Please update both CUDA driver and cudaVersion to at least 11.4 to
↪use the new feature!\n");
        // Indicating driver is < 11.4 since string not found, doesn't matter what
↪cudaVersion was
    }
    else if (CU_GET_PROC_ADDRESS_SUCCESS == driverStatus && pfn) {
        printf("You're using cudaVersion and CUDA driver >= 11.4, using new feature!\n
↪");
        pfn();
    }
}
```

The first case with the return code `CU_GET_PROC_ADDRESS_VERSION_NOT_SUFFICIENT` indicates that the `symbol` was found when searching in the CUDA driver but it was added later than the `cudaVersion` supplied. In the example, specifying `cudaVersion` as anything 11030 or less and when running against a CUDA driver >= CUDA 11.4 would give this result of `CU_GET_PROC_ADDRESS_VERSION_NOT_SUFFICIENT`. This is because `cuDeviceGetExecAffinitySupport` was added in CUDA 11.4 (11040).

The second case with the return code `CU_GET_PROC_ADDRESS_SYMBOL_NOT_FOUND` indicates that the `symbol` was not found when searching in the CUDA driver. This can be due to a few reasons such as unsupported CUDA function due to older driver as well as just having a typo. In the latter, similar to the last example if the user had put `symbol` as CUDeviceGetExecAffinitySupport - notice the capital CU to start the string - `cuGetProcAddress` would not be able to find the API because the string doesn't match. In the former case an example might be the user developing an application against a CUDA driver supporting the new API, and deploying the application against an older CUDA driver. Using the last example, if the developer developed against CUDA 11.4 or later but was deployed against a CUDA 11.3 driver, during their development they may have had a succesful `cuGetProcAddress`, but when deploying an application running against a CUDA 11.3 driver the call would no longer work with the `CU_GET_PROC_ADDRESS_SYMBOL_NOT_FOUND` returned in `driverStatus`.

Chapter 22. CUDA Environment Variables

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

The following table lists the CUDA environment variables. Environment variables related to the Multi-Process Service are documented in the Multi-Process Service section of the GPU Deployment and Management guide.

Table 30: CUDA Environment Variables

Variable	Values	Description
Device Enumeration and Properties		

continues on next page

Table 30 – continued from previous page

Variable	Values	Description
CUDA_VISIBLE_DEVICES	A comma-separated sequence of GPU identifiers MIG support: `MIG-<GPU-UUID>/<GPU instance ID>/<compute instance ID>`	GPU identifiers are given as integer indices or as UUID strings. GPU UUID strings should follow the same format as given by *nvidia-smi*, such as GPU-8932f937-d72c-4106-c12f-20bd9faed9f6. However, for convenience, abbreviated forms are allowed; simply specify enough digits from the beginning of the GPU UUID to uniquely identify that GPU in the target system. For example, CUDA_VISIBLE_DEVICES=GPU-8932f937 may be a valid way to refer to the above GPU UUID, assuming no other GPU in the system shares this prefix. Only the devices whose index is present in the sequence are visible to CUDA applications and they are enumerated in the order of the sequence. If one of the indices is invalid, only the devices whose index precedes the invalid index are visible to CUDA applications. For example, setting CUDA_VISIBLE_DEVICES to 2,1 causes device 0 to be invisible and device 2 to be enumerated before device 1. Setting CUDA_VISIBLE_DEVICES to 0,2,-1,1 causes devices 0 and 2 to be visible and device 1 to be invisible. MIG format starts with MIG keyword and GPU UUID should follow the same format as given by *nvidia-smi*. For example, MIG-GPU-8932f937-d72c-4106-c12f-20bd9faed9f6/1/2. Only single MIG instance enumeration is supported.
CUDA_MANAGED_FORCE_DEVICE_ALLOC	0 or 1 (default is 0)	Forces the driver to place all managed allocations in device memory.

continues on next page

Table 30 – continued from previous page

Variable	Values	Description
CUDA_DEVICE_ORDER	FASTEST_FIRST, PCI_BUS_ID, (default is FASTEST_FIRST)	FASTEST_FIRST causes CUDA to enumerate the available devices in fastest to slowest order using a simple heuristic. PCI_BUS_ID orders devices by PCI bus ID in ascending order.
Compilation		
CUDA_CACHE_DISABLE	0 or 1 (default is 0)	Disables caching (when set to 1) or enables caching (when set to 0) for just-in-time-compilation. When disabled, no binary code is added to or retrieved from the cache.
CUDA_CACHE_PATH	filepath	Specifies the folder where the just-in-time compiler caches binary codes; the default values are: ► on Windows, `%APPDATA%\NVIDIA\ComputeCache` ► on Linux, `~/.nv/ComputeCache`
CUDA_CACHE_MAXSIZE	integer (default is 1073741824 (1 GiB) for desktop/server platforms and 268435456 (256 MiB) for embedded platforms and the maximum is 4294967296 (4 GiB))	Specifies the size in bytes of the cache used by the just-in-time compiler. Binary codes whose size exceeds the cache size are not cached. Older binary codes are evicted from the cache to make room for newer binary codes if needed.

continues on next page

Table 30 – continued from previous page

Variable	Values	Description
CUDA_FORCE_PTX_JIT	0 or 1 (default is 0)	When set to 1, forces the device driver to ignore any binary code embedded in an application (see *Application Compatibility*) and to just-in-time compile embedded PTX code instead. If a kernel does not have embedded PTX code, it will fail to load. This environment variable can be used to validate that PTX code is embedded in an application and that its just-in-time compilation works as expected to guarantee application forward compatibility with future architectures (see *Just-in-Time Compilation*).
CUDA_DISABLE_PTX_JIT	0 or 1 (default is 0)	When set to 1, disables the just-in-time compilation of embedded PTX code and use the compatible binary code embedded in an application (see *Application Compatibility*). If a kernel does not have embedded binary code or the embedded binary was compiled for an incompatible architecture, then it will fail to load. This environment variable can be used to validate that an application has the compatible *SASS* code generated for each kernel.(see *Binary Compatibility*).

continues on next page

Table 30 – continued from previous page

Variable	Values	Description
CUDA_FORCE_JIT	0 or 1 (default is 0)	When set to 1, forces the device driver to ignore any binary code embedded in an application (see *Application Compatibility*) and to just-in-time compile embedded PTX code instead. If a kernel does not have embedded PTX code, it will fail to load. This environment variable can be used to validate that PTX code is embedded in an application and that its just-in-time compilation works as expected to guarantee application forward compatibility with future architectures (see *Just-in-Time Compilation*). The behavior can be overridden for embedded PTX by setting `CUDA_FORCE_PTX_JIT=0`.
CUDA_DISABLE_JIT	0 or 1 (default is 0)	When set to 1, disables the just-in-time compilation of embedded PTX code and use the compatible binary code embedded in an application (see *Application Compatibility*). If a kernel does not have embedded binary code or the embedded binary was compiled for an incompatible architecture, then it will fail to load. This environment variable can be used to validate that an application has the compatible SASS code generated for each kernel.(see *Binary Compatibility*). The behavior can be overridden for embedded PTX by setting `CUDA_DISABLE_PTX_JIT=0`.
Execution		
CUDA_LAUNCH_BLOCKING	0 or 1 (default is 0)	Disables (when set to 1) or enables (when set to 0) asynchronous kernel launches.

continues on next page

Table 30 – continued from previous page

Variable	Values	Description
CUDA_DEVICE_MAX_CONNECTIONS	1 to 32 (default is 8)	Sets the number of compute and copy engine concurrent connections (work queues) from the host to each device of compute capability 3.5 and above.
CUDA_DEVICE_MAX_COPY_CONNECTIONS	1 to 32 (default is 8)	Sets the number of copy engine concurrent connections (work queues) per async copy engine from the host to each device of compute capability 8.0 and above. When both CUDA_DEVICE_MAX_CONNECTIONS and CUDA_DEVICE_MAX_COPY_CONNECTIONS are set, only the number of copy connections set by CUDA_DEVICE_MAX_CONNECTIONS will be overwritten.
CUDA_AUTO_BOOST	0 or 1	Overrides the autoboost behavior set by the –auto-boost-default option of nvidia-smi. If an application requests via this environment variable a behavior that is different from nvidia-smi's, its request is honored if there is no other application currently running on the same GPU that successfully requested a different behavior, otherwise it is ignored.
CUDA_SCALE_LAUNCH_QUEUES	"0.25x", "0.5x", "2x" or "4x"	Scales the size of the queues available for launching work by a fixed multiplier.
cuda-gdb (on Linux platform)		
CUDA_DEVICE_WAITS_ON_EXCEPTION	0 or 1 (default is 0)	When set to 1, a CUDA application will halt when a device exception occurs, allowing a debugger to be attached for further debugging.
MPS service (on Linux platform)		

continues on next page

Table 30 – continued from previous page

Variable	Values	Description
CUDA_DEVICE_DEFAULT_PERSISTING_L2_CACHE_PERCENTAGE_LIMIT	Integer value (between 0 and 100, default is 0)	Devices of compute capability 8.x allow, a portion of L2 cache to be set-aside for persisting data accesses to global memory. When using CUDA MPS service, the set-aside size can only be controlled using this environment variable, before starting CUDA MPS control daemon. I.e., the environment variable should be set before running the command `nvidia-cuda-mps-control -d`.
Module loading		
CUDA_MODULE_LOADING	DEFAULT, LAZY, EAGER (default is LAZY)	Specifies the module loading mode for the application. When set to EAGER, all kernels and data from a cubin, fatbin or a PTX file are fully loaded upon corresponding `cuModuleLoad*` and `cuLibraryLoad*` API call. When set to LAZY, loading of specific kernels is delayed to the point a CUfunc handle is extracted with `cuModuleGetFunction` or `cuKernelGetFunction` API calls and data from the cubin is loaded at load of first kernel in the cubin or at first access of variables in the cubin. Default behavior may change in future CUDA releases.

continues on next page

Table 30 – continued from previous page

Variable	Values	Description
CUDA_MODULE_DATA_LOADING	DEFAULT, LAZY, EAGER (default is LAZY)	Specifies the data loading mode for the application. When set to EAGER, all data from a cubin, fatbin or a PTX file are fully loaded to memory upon corresponding `cuLibraryLoad*`. This doesn't affect the LAZY or EAGER loading of kernels. When set to LAZY, loading of data is delayed to the point at which a handle is required. Default behavior may change in future CUDA releases. Data loading behavior is inherited from `CUDA_MODULE_LOADING` if this environment variable is not set.
CUDA_BINARY_LOADER_THREAD_COUNT	integer (default is 0)	Sets the number of CPU threads to use when loading device binaries. When set to 0, the number of CPU threads used is set to a default value of 1. When this is set, CU_JIT_BINARY_LOADER_THREAD_COUNT is ignored.
Pre-loading dependent libraries		
CUDA_FORCE_PRELOAD_LIBRARIES	0 or 1 (default is 0)	When set to 1, forces the driver to preload the libraries required for NVVM and PTX just-in-time compilation during driver initialization. This will increase the memory footprint and the time taken for CUDA driver initialization. This environment variable needs to be set to avoid certain deadlock situations involving multiple CUDA threads.
CUDA Graphs		
CUDA_GRAPHS_USE_NODE_PRIORITY	0 or 1	Overrides the cudaGraphInstantiateFlagUseNodePriority flag on graph instantiation. When set to 1, the flag will be set for all graphs and when set to 0, the flag will be cleared for all graphs.

continues on next page

Table 30 – continued from previous page

Variable	Values	Description
CUDA Error Log Management		
CUDA_LOG_FILE	stdout, stderr, or valid file path	Provides a location for printing error logs as they occur. See the `Error Log Management` section for more details.

Chapter 23. Error Log Management

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

The *Error Log Management* mechanism allows for CUDA API errors to be reported to developers in a plain-English format that describes the cause of the issue.

23.1. Background

Traditionally, the only indication of a failed CUDA API call is the return of a non-zero code. As of CUDA Toolkit 12.9, the CUDA Runtime defines over 100 different return codes for error conditions, but many of them are generic and give the developer no assistance with debugging the cause.

23.2. Activation

Set the *CUDA_LOG_FILE* environment variable. Acceptable values are *stdout*, *stderr*, or a valid path on the system to write a file. The log buffer can be dumped via API even if *CUDA_LOG_FILE* was not set before program execution. NOTE: An error-free execution may not print any logs.

23.3. Output

Logs are output in the following format:

```
[Time][TID][Source][Severity][API Entry Point] Message
```

The following line is an actual error message that is generated if the developer tries to dump the Error Log Management logs to an unallocated buffer:

```
[22:21:32.099][25642][CUDA][E][cuLogsDumpToMemory] buffer cannot be NULL
```

Where before, all the developer would have gotten is *CUDA_ERROR_INVALID_VALUE* in the return code and possibly “invalid argument” if *cuGetErrorString* is called.

23.4. API Description

The CUDA Driver provides APIs in two categories for interacting with the Error Log Management feature.

This feature allows developers to register callback functions to be used whenever an error log is generated, where the callback signature is:

```
void callbackFunc(void *data, CUlogLevel logLevel, char *message, size_t length)
```

Callbacks are registered with this API:

```
CUresult cuLogsRegisterCallback(CUlogsCallback callbackFunc, void *userData,
↪CUlogsCallbackHandle *callback_out)
```

Where *userData* is passed to the callback function without modifications. *callback_out* should be stored by the caller for use in *cuLogsUnregisterCallback*.

```
CUresult cuLogsUnregisterCallback(CUlogsCallbackHandle callback)
```

The other set of API functions are for managing the output of logs. An important concept is the log iterator, which points to the current end of the buffer:

```
CUresult cuLogsCurrent(CUlogIterator *iterator_out, unsigned int flags)
```

The iterator position can be kept by the calling software in situations where a dump of the entire log buffer is not desired. Currently, the flags parameter must be 0, with additional options reserved for future CUDA releases.

At any time, the error log buffer can be dumped to either a file or memory with these functions:

```
CUresult cuLogsDumpToFile(CUlogIterator *iterator, const char *pathToFile, unsigned
↪int flags)
CUresult cuLogsDumpToMemory(CUlogIterator *iterator, char *buffer, size_t *size,
↪unsigned int flags)
```

If *iterator* is NULL, the entire buffer will be dumped, up to the maximum of 100 entries. If *iterator* is not NULL, logs will be dumped starting from that entry and the value of *iterator* will be updated to the current end of the logs, as if *cuLogsCurrent* had been called. If there have been more than 100 log entries into the buffer, a note will be added at the start of the dump noting this.

The flags parameter must be 0, with additional options reserved for future CUDA releases.

The *cuLogsDumpToMemory* function has additional considerations:

1. The buffer itself will be null-terminated, but each individual log entry will only be separated by a newline (n) character.
2. The maximum size of the buffer is 25600 bytes.
3. If the value provided in *size* is not sufficient to store all desired logs, a note will be added as the first entry and the oldest entries that do not fit will not be dumped.
4. After returning, *size* will contain the actual number of bytes written to the provided buffer.

23.5. Limitations and Known Issues

1. The log buffer is limited to 100 entries. After this limit is reached, the oldest entries will be replaced and log dumps will contain a line noting the rollover.
2. Not all CUDA APIs are covered yet. This is an ongoing project to provide better usage error reporting for all APIs.
3. The Error Log Management log location (if given) will not be tested for validity until/unless a log is generated.
4. The Error Log Management APIs are currently only available via the CUDA Driver. Equivalent APIs will be added to the CUDA Runtime in a future release.
5. The log messages are not localized to any language and all provided logs are in US English.

Chapter 24. Unified Memory Programming

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

Note: This chapter applies to devices with compute capability 5.0 or higher unless stated otherwise. For devices with compute capability lower than 5.0, refer to the CUDA toolkit documentation for CUDA 11.8.

This documentation on Unified Memory is divided into 3 parts:

- *General description of unified memory*
- *Unified Memory on devices with full CUDA Unified Memory support*
- *Unified Memory on devices without full CUDA Unified Memory support*

24.1. Unified Memory Introduction

CUDA Unified Memory provides all processors with:

- A single *unified* memory pool, that is, a single pointer value enables all processors in the system (all CPUs, all GPUs, etc.) to access this memory with all of their native memory operations (pointer dereferences, atomics, etc.).
- Concurrent access to the unified memory pool from all processors in the system.

Unified Memory improves GPU programming in several ways:

- **Productivity**: GPU programs may access Unified Memory from GPU and CPU threads concurrently without needing to create separate allocations (`cudaMalloc()`) and copy memory manually back and forth (`cudaMemcpy*()`).
- **Performance**:
  - Data access speed may be maximized by migrating data towards processors that access it most frequently. Applications can trigger manual migration of data and may use hints to control migration heuristics.

- Total system memory usage may be reduced by avoiding duplicating memory on both CPUs and GPUs.

- **Functionality**: It enables GPU programs to work on data that exceeds the GPU memory's capacity.

With CUDA Unified Memory, data movement still takes place, and hints may improve performance. These hints are not required for correctness or functionality, that is, programmers may focus on parallelizing their applications across GPUs and CPUs first, and worry about data-movement later in the development cycle as a performance optimization. Note that the physical location of data is invisible to a program and may be changed at any time, but accesses to the data's virtual address will remain valid and coherent from any processor regardless of locality.

There are two main ways to obtain CUDA Unified Memory:

- *System-Allocated Memory*: memory allocated on the host with system APIs: stack variables, global-/file-scope variables, `malloc()` / `mmap()` (see *System-Allocated Memory: in-depth examples* for in-depth examples), thread locals, etc.
- *CUDA APIs that explicitly allocate Unified Memory*: memory allocated with, for example, `cudaMallocManaged()`, are available on more systems and may perform better than System-Allocated Memory.

24.1.1. System Requirements for Unified Memory

The following table shows the different levels of support for CUDA Unified Memory, the device properties required to detect these levels of support and links to the documentation specific to each level of support:

Table 31: Overview of levels of unified memory support

Unified Memory Support Level	System device properties	Further documentation
Full CUDA Unified Memory: all memory has full support. This includes System-Allocated and CUDA Managed Memory.	Set to 1: `pageableMemoryAccess` *Systems with hardware acceleration* also have the following properties set to 1: `hostNativeAtomicSupported`, `pageableMemoryAccessUsesHostPageTables`, `directManagedMemAccessFromHost`	*Unified Memory on devices with full CUDA Unified Memory support*
Only CUDA Managed Memory has full support.	Set to 1: `concurrentManagedAccess` Set to 0: `pageableMemoryAccess`	*Unified Memory on devices with only CUDA Managed Memory support*
CUDA Managed Memory without full support: unified addressing but no concurrent access.	Set to 1: `managedMemory` Set to 0: `concurrentManagedAccess`	*Unified Memory on Windows or devices with compute capability 5.x* CUDA for Tegra Memory Management Unified Memory on Tegra
No Unified Memory support.	Set to 0: `managedMemory`	CUDA for Tegra Memory Management

The behavior of an application that attempts to use Unified Memory on a system that does not support it is undefined. The following properties enable CUDA applications to check the level of system support for Unified Memory, and to be portable between systems with different levels of support:

- `pageableMemoryAccess`: This property is set to 1 on systems with CUDA Unified Memory support where all threads may access System-Allocated Memory and CUDA Managed Memory. These systems include NVIDIA Grace Hopper, IBM Power9 + Volta, and modern Linux systems with HMM enabled (see next bullet), among others.
  - Linux HMM requires Linux kernel version 6.1.24+, 6.2.11+ or 6.3+, devices with compute capability 7.5 or higher and a CUDA driver version 535+ installed with Open Kernel Modules.
- `concurrentManagedAccess`: This property is set to 1 on systems with full CUDA Managed Memory support. When this property is set to 0, there is only partial support for Unified Memory in CUDA Managed Memory. For Tegra support of Unified Memory, see CUDA for Tegra Memory Management.

A program may query the level of GPU support for CUDA Unified Memory, by querying the attributes

in Table 31 using `cudaGetDeviceProperties()`.

24.1.2. Programming Model

With CUDA Unified Memory, separate allocations between host and device, and explicit memory transfers between them, are no longer required. Programs may allocate Unified Memory in the following ways:

- *System-Allocation APIs*: on *systems with full CUDA Unified Memory support* via any system allocation of the host process (C's `malloc()`, C++'s `new` operator, POSIX's `mmap` and so on).
- *CUDA Managed Memory Allocation APIs*: via the `cudaMallocManaged()` API which is syntactically similar to `cudaMalloc()`.
- *CUDA Managed Variables*: variables declared with `__managed__`, which are semantically similar to a `__device__` variable.

Most examples in this chapter provide at least two versions, one using CUDA Managed Memory and one using System-Allocated Memory. Tabs allow you to choose between them. The following samples illustrate how Unified Memory simplifies CUDA programs:

System (`malloc()`)

```
__global__ void write_value(int* ptr,
↪int v) {
  *ptr = v;
}

int main() {
  int* d_ptr = nullptr;
  // Does not require any unified memory
↪support
  cudaMalloc(&d_ptr, sizeof(int));
  write_value<<<1, 1>>>(d_ptr, 1);
  int h_value;
  // Copy memory back to the host and
↪synchronize
  cudaMemcpy(&h_value, d_ptr,
↪sizeof(int),
             cudaMemcpyDefault);
  printf("value = %d\n", h_value);
  cudaFree(d_ptr);
  return 0;
}
```

```
__global__ void write_value(int* ptr,
↪int v) {
  *ptr = v;
}

int main() {
  // Requires System-Allocated Memory
↪support
  int* ptr = (int*)malloc(sizeof(int));
  write_value<<<1, 1>>>(ptr, 1);
  // Synchronize required
  // (before, cudaMemcpy was
↪synchronizing)
  cudaDeviceSynchronize();
  printf("value = %d\n", *ptr);
  free(ptr);
  return 0;
}
```

System (Stack)

```
__global__ void write_value(int* ptr,
↪int v) {
  *ptr = v;
}

int main() {
  int* d_ptr = nullptr;
  // Does not require any unified memory
↪support
  cudaMalloc(&d_ptr, sizeof(int));
  write_value<<<1, 1>>>(d_ptr, 1);
  int h_value;
  // Copy memory back to the host and
↪synchronize
  cudaMemcpy(&h_value, d_ptr,
↪sizeof(int),
             cudaMemcpyDefault);
  printf("value = %d\n", h_value);
  cudaFree(d_ptr);
  return 0;
}
```

```
__global__ void write_value(int* ptr,
↪int v) {
  *ptr = v;
}

int main() {
  // Requires System-Allocated Memory
↪support
  int value;
  write_value<<<1, 1>>>(&value, 1);
  // Synchronize required
  // (before, cudaMemcpy was
↪synchronizing)
  cudaDeviceSynchronize();
  printf("value = %d\n", value);
  return 0;
}
```

Managed (cudaMallocManaged())

```
__global__ void write_value(int* ptr,
↪int v) {
  *ptr = v;
}

int main() {
  int* d_ptr = nullptr;
  // Does not require any unified memory
↪support
  cudaMalloc(&d_ptr, sizeof(int));
  write_value<<<1, 1>>>(d_ptr, 1);
  int h_value;
  // Copy memory back to the host and
↪synchronize
  cudaMemcpy(&h_value, d_ptr,
↪sizeof(int),
             cudaMemcpyDefault);
  printf("value = %d\n", h_value);
  cudaFree(d_ptr);
  return 0;
}
```

```
__global__ void write_value(int* ptr,
↪int v) {
  *ptr = v;
}

int main() {
  int* ptr = nullptr;
  // Requires CUDA Managed Memory support
  cudaMallocManaged(&ptr, sizeof(int));
  write_value<<<1, 1>>>(ptr, 1);
  // Synchronize required
  // (before, cudaMemcpy was
↪synchronizing)
  cudaDeviceSynchronize();
  printf("value = %d\n", *ptr);
  cudaFree(ptr);
  return 0;
}
```

Managed (__managed__)

```
__global__ void write_value(int* ptr,
↪int v) {
  *ptr = v;
}

int main() {
  int* d_ptr = nullptr;
  // Does not require any unified memory
↪support
  cudaMalloc(&d_ptr, sizeof(int));
  write_value<<<1, 1>>>(d_ptr, 1);
  int h_value;
  // Copy memory back to the host and
↪synchronize
  cudaMemcpy(&h_value, d_ptr,
↪sizeof(int),
             cudaMemcpyDefault);
  printf("value = %d\n", h_value);
  cudaFree(d_ptr);
  return 0;
}
```

```
__global__ void write_value(int* ptr,
↪int v) {
  *ptr = v;
}

// Requires CUDA Managed Memory support
__managed__ int value;

int main() {
  write_value<<<1, 1>>>(&value, 1);
  // Synchronize required
  // (before, cudaMemcpy was
↪synchronizing)
  cudaDeviceSynchronize();
  printf("value = %d\n", value);
  return 0;
}
```

In the example above, the device writes a value which is then read by the host:

- **Without Unified Memory**: both host- and device-side storage for the written value is required (`h_value` and `d_ptr` in the example), as is an explicit copy between the two using `cudaMemcpy()`.
- **With Unified Memory**: device accesses data directly from the host. `ptr` / `value` may be used without a separate `h_value` / `d_ptr` allocation and no copy routine is required, greatly simplifying and reducing the size of the program. With:
  - **System Allocated**: no other changes required.
  - **Managed Memory**: data allocation changed to use `cudaMallocManaged()`, which returns a pointer valid from both host and device code.

24.1.2.1 Allocation APIs for System-Allocated Memory

On *systems with full CUDA Unified Memory support*, all memory is unified memory. This includes memory allocated with system allocation APIs, such as `malloc()`, `mmap()`, C++ `new()` operator, and also automatic variables on CPU thread stacks, thread locals, global variables, and so on.

System-Allocated Memory may be populated on first touch, depending on the API and system settings used. First touch means that:

- The allocation APIs allocate virtual memory and return immediately, and
- physical memory is populated when a thread accesses the memory for the first time.

Usually, the physical memory will be chosen "close" to the processor that thread is running on. For example,

- GPU thread accesses it first: physical GPU memory of GPU that thread runs on is chosen.

- CPU thread accesses it first: physical CPU memory in the memory NUMA node of the CPU core that thread runs on is chosen.

CUDA Unified Memory Hint and Prefetch APIs, `cudaMemAdvise` and `cudaMemPreftchAsync`, may be used on System-Allocated Memory. These APIs are covered below in the *Data Usage Hints* section.

```
__global__ void printme(char *str) {
  printf(str);
}

int main() {
  // Allocate 100 bytes of memory, accessible to both Host and Device code
  char *s = (char*)malloc(100);
  // Physical allocation placed in CPU memory because host accesses "s" first
  strncpy(s, "Hello Unified Memory\n", 99);
  // Here we pass "s" to a kernel without explicitly copying
  printme<<< 1, 1 >>>(s);
  cudaDeviceSynchronize();
  // Free as for normal CUDA allocations
  cudaFree(s);
  return  0;
}
```

24.1.2.2 Allocation API for CUDA Managed Memory: `cudaMallocManaged()`

On systems with CUDA Managed Memory support, unified memory may be allocated using:

```
__host__ cudaError_t cudaMallocManaged(void **devPtr, size_t size);
```

This API is syntactically identical to `cudaMalloc()`: it allocates `size` bytes of managed memory and sets `devPtr` to refer to the allocation. CUDA Managed Memory is also deallocated with `cudaFree()`.

On *systems with full CUDA Managed Memory support*, managed memory allocations may be accessed concurrently by all CPUs and GPUs in the system. Replacing host calls to `cudaMalloc()` with `cudaMallocManaged()` does not impact program semantics on these systems; device code is not able to call `cudaMallocManaged()`.

The following example shows the use of `cudaMallocManaged()`:

```
__global__ void printme(char *str) {
  printf(str);
}

int main() {
  // Allocate 100 bytes of memory, accessible to both Host and Device code
  char *s;
  cudaMallocManaged(&s, 100);
  // Note direct Host-code use of "s"
  strncpy(s, "Hello Unified Memory\n", 99);
  // Here we pass "s" to a kernel without explicitly copying
  printme<<< 1, 1 >>>(s);
  cudaDeviceSynchronize();
  // Free as for normal CUDA allocations
  cudaFree(s);
  return  0;
}
```

Note: For systems that support CUDA Managed Memory allocations, but do not provide full support, see *Coherency and Concurrency*. Implementation details (may change any time):

- Devices of compute capability 5.x allocate CUDA Managed Memory on the GPU.
- Devices of compute capability 6.x and greater populate the memory on first touch, just like System-Allocated Memory APIs.

24.1.2.3 Global-Scope Managed Variables Using `__managed__`

CUDA `__managed__` variables behave as if they were allocated via `cudaMallocManaged()` (see *Allocation API for CUDA Managed Memory: cudaMallocManaged()*). They simplify programs with global variables, making it particularly easy to exchange data between host and device without manual allocations or copying.

On *systems with full CUDA Unified Memory support*, file-scope or global-scope variables cannot be directly accessed by device code. But a pointer to these variables may be passed to the kernel as an argument, see *System-Allocated Memory: in-depth examples* for examples.

System Allocator

```
__global__ void write_value(int* ptr, int v) {
  *ptr = v;
}

int main() {
  // Requires System-Allocated Memory support
  int value;
  write_value<<<1, 1>>>(&value, 1);
  // Synchronize required
  // (before, cudaMemcpy was synchronizing)
  cudaDeviceSynchronize();
  printf("value = %d\n", value);
  return 0;
}
```

Managed

```
__global__ void write_value(int* ptr, int v) {
  *ptr = v;
}

// Requires CUDA Managed Memory support
__managed__ int value;

int main() {
  write_value<<<1, 1>>>(&value, 1);
  // Synchronize required
  // (before, cudaMemcpy was synchronizing)
  cudaDeviceSynchronize();
  printf("value = %d\n", value);
```

(continues on next page)

(continued from previous page)

```
  return 0;
}
```

Note the absence of explicit `cudaMemcpy()` commands and the fact that the written value `value` is visible on both CPU and GPU.

CUDA `__managed__` variable implies `__device__` and is equivalent to `__managed__ __device__`, which is also allowed. Variables marked `__constant__` may not be marked as `__managed__`.

A valid CUDA context is necessary for the correct operation of `__managed__` variables. Accessing `__managed__` variables can trigger CUDA context creation if a context for the current device hasn't already been created. In the example above, accessing `value` before the kernel launch triggers context creation on the default device. In the absence of that access, the kernel launch would have triggered context creation.

C++ objects declared as `__managed__` are subject to certain specific constraints, particularly where static initializers are concerned. Please refer to *C++ Language Support* for a list of these constraints.

Note: For *devices with CUDA Managed Memory without full support*, visibility of `__managed__` variables for asynchronous operations executing in CUDA streams is discussed in the section on *Managing Data Visibility and Concurrent CPU + GPU Access with Streams*.

24.1.2.4 Difference between Unified Memory and Mapped Memory

The main difference between Unified Memory and *Mapped Memory* is that CUDA Mapped Memory does not guarantee that all kinds of memory accesses (for example atomics) are supported on all systems, while Unified Memory does. The limited set of memory operations that are guaranteed to be portably supported by CUDA Mapped Memory is available on more systems than Unified Memory.

24.1.2.5 Pointer Attributes

CUDA Programs may check whether a pointer addresses a CUDA Managed Memory allocation by calling `cudaPointerGetAttributes()` and testing whether the pointer attribute `value` is `cudaMemoryTypeManaged`.

This API returns `cudaMemoryTypeHost` for System-Allocated Memory that has been registered with `cudaHostRegister()` and `cudaMemoryTypeUnregistered` for System-Allocated Memory that CUDA is unaware of.

Pointer attributes do not state where the memory resides, they state how the memory was allocated or registered.

The following example shows how to detect the type of pointer at runtime:

```
char const* kind(cudaPointerAttributes a, bool pma, bool cma) {
    switch(a.type) {
    case cudaMemoryTypeHost: return pma?
      "Unified: CUDA Host or Registered Memory" :
      "Not Unified: CUDA Host or Registered Memory";
    case cudaMemoryTypeDevice: return "Not Unified: CUDA Device Memory";
    case cudaMemoryTypeManaged: return cma?
      "Unified: CUDA Managed Memory" : "Not Unified: CUDA Managed Memory";
```

(continues on next page)

(continued from previous page)

```
    case cudaMemoryTypeUnregistered: return pma?
      "Unified: System-Allocated Memory" :
      "Not Unified: System-Allocated Memory";
    default: return "unknown";
    }
}

void check_pointer(int i, void* ptr) {
  cudaPointerAttributes attr;
  cudaPointerGetAttributes(&attr, ptr);
  int pma = 0, cma = 0, device = 0;
  cudaGetDevice(&device);
  cudaDeviceGetAttribute(&pma, cudaDevAttrPageableMemoryAccess, device);
  cudaDeviceGetAttribute(&cma, cudaDevAttrConcurrentManagedAccess, device);
  printf("Pointer %d: memory is %s\n", i, kind(attr, pma, cma));
}

__managed__ int managed_var = 5;

int main() {
  int* ptr[5];
  ptr[0] = (int*)malloc(sizeof(int));
  cudaMallocManaged(&ptr[1], sizeof(int));
  cudaMallocHost(&ptr[2], sizeof(int));
  cudaMalloc(&ptr[3], sizeof(int));
  ptr[4] = &managed_var;

  for (int i = 0; i < 5; ++i) check_pointer(i, ptr[i]);

  cudaFree(ptr[3]);
  cudaFreeHost(ptr[2]);
  cudaFree(ptr[1]);
  free(ptr[0]);
  return 0;
}
```

24.1.2.6 Runtime detection of Unified Memory Support Level

The following example shows how to detect the Unified Memory support level at runtime:

```
int main() {
  int d;
  cudaGetDevice(&d);

  int pma = 0;
  cudaDeviceGetAttribute(&pma, cudaDevAttrPageableMemoryAccess, d);
  printf("Full Unified Memory Support: %s\n", pma == 1? "YES" : "NO");

  int cma = 0;
  cudaDeviceGetAttribute(&cma, cudaDevAttrConcurrentManagedAccess, d);
  printf("CUDA Managed Memory with full support: %s\n", cma == 1? "YES" : "NO");

  return 0;
}
```

24.1.2.7 GPU Memory Oversubscription

Unified Memory enables applications to *oversubscribe* the memory of any individual processor: in other words they can allocate and share arrays larger than the memory capacity of any individual processor in the system, enabling among others out-of-core processing of datasets that do not fit within a single GPU, without adding significant complexity to the programming model.

24.1.2.8 Performance Hints

The following sections describes the available unified memory performance hints, which may be used on all Unified Memory, for example, CUDA Managed memory or, on *systems with full CUDA Unified Memory support*, also all System-Allocated Memory. These APIs are hints, that is, they do not impact the semantics of applications, only their peformance. That is, they can be added or removed anywhere on any application without impacting its results.

CUDA Unified Memory may not always have all the information necessary to make the best performance decisions related to unified memory. These performance hints enable the application to provide CUDA with more information.

Note that applications should only use these hints if they improve their performance.

24.1.2.8.1 Data Prefetching

The `cudaMemPrefetchAsync` API is an asynchronous stream-ordered API that may migrate data to reside closer to the specified processor. The data may be accessed while it is being prefetched. The migration does not begin until all prior operations in the stream have completed, and completes before any subsequent operation in the stream.

```
cudaError_t cudaMemPrefetchAsync(const void *devPtr,
                                 size_t count,
                                 struct cudaMemLocation location,
                                 unsigned int flags,
                                 cudaStream_t stream);
```

A memory region containing `[devPtr, devPtr + count)` may be migrated to the destination device `location.id` if `location.type` is `cudaMemLocationTypeDevice` - or CPU if `location.type` is `cudaMemLocationTypeHost` - when the prefetch task is executed in the given `stream`. For details on `flags`, see the current CUDA Runtime API documentation.

Consider a simple code example below:

System Allocator

```
void test_prefetch_sam(cudaStream_t s) {
  char *data = (char*)malloc(N);
  init_data(data, N);                                          // execute on CPU
  cudaMemLocation location = {.type = cudaMemLocationTypeDevice, .id = myGpuId};
  cudaMemPrefetchAsync(data, N, location, s, 0 /* flags */);  // prefetch to GPU
  mykernel<<<(N + TPB - 1) / TPB, TPB, 0, s>>>(data, N);      // execute on GPU
  location = {.type = cudaMemLocationTypeHost};
  cudaMemPrefetchAsync(data, N, location, s, 0 /* flags */);  // prefetch to CPU
  cudaStreamSynchronize(s);
```

(continues on next page)

(continued from previous page)

```
  use_data(data, N);
  free(data);
}
```

Managed

```
void test_prefetch_managed(cudaStream_t s) {
  char *data;
  cudaMallocManaged(&data, N);
  init_data(data, N);                                          // execute on CPU
  cudaMemLocation location = {.type = cudaMemLocationTypeDevice, .id = myGpuId};
  cudaMemPrefetchAsync(data, N, location, s, 0 /* flags */);  // prefetch to GPU
  mykernel<<<(N + TPB - 1) / TPB, TPB, 0, s>>>(data, N);      // execute on GPU
  location = {.type = cudaMemLocationTypeHost};
  cudaMemPrefetchAsync(data, N, location, s, 0 /* flags */);  // prefetch to CPU
  cudaStreamSynchronize(s);
  use_data(data, N);
  cudaFree(data);
}
```

24.1.2.8.2 Memory Discarding

The `cudaMemDiscardBatchAsync` API allows applications to inform the CUDA runtime that the contents of specified memory ranges are no longer useful. The Unified Memory driver performs automatic memory transfers due to fault-based migration or memory evictions to support device memory oversubscription. These automatic memory transfers can sometimes be redundant, which severely decreases performance. Marking an address range as 'discard' will inform the Unified Memory driver that the application has consumed the contents in the range and there is no need to migrate this data on prefetches or page evictions in order to make room for other allocations. Reading a discarded page without a subsequent write access or prefetch will yield an indeterminate value. Whereas any new writes after the discard operation is guaranteed to be seen by a subsequent read access. Concurrent accesses or prefetches to address ranges being discarded will result in undefined behavior.

```
cudaError_t cudaMemDiscardBatchAsync(void **dptrs,
                                     size_t *sizes,
                                     size_t count,
                                     unsigned long long flags,
                                     cudaStream_t stream);
```

The function performs a batch of memory discards on address ranges specified in `dptrs` and `sizes` arrays. Both arrays must be of the same length as specified by `count`. Each memory range must refer to managed memory allocated via `cudaMallocManaged` or declared via `__managed__` variables.

The `cudaMemDiscardAndPrefetchBatchAsync` API combines both discard and prefetch operations. Calling `cudaMemDiscardAndPrefetchBatchAsync` is semantically equivalent to calling `cudaMemDiscardBatchAsync` followed by `cudaMemPrefetchBatchAsync`, but is more optimal. This is useful when the application needs the memory to be on the target location but does not need the contents of the memory.

```
cudaError_t cudaMemDiscardAndPrefetchBatchAsync(void **dptrs,
                                                size_t *sizes,
```

(continues on next page)

(continued from previous page)

```
                                                size_t count,
                                                struct cudaMemLocation *prefetchLocs,
                                                size_t *prefetchLocIdxs,
                                                size_t numPrefetchLocs,
                                                unsigned long long flags,
                                                cudaStream_t stream);
```

The `prefetchLocs` array specifies the destinations for prefetching, while `prefetchLocIdxs` indicates which operations each prefetch location applies to. For example, if a batch has 10 operations and the first 6 should be prefetched to one location while the remaining 4 to another, then `numPrefetchLocs` would be 2, `prefetchLocIdxs` would be {0, 6}, and `prefetchLocs` would contain the two destination locations.

Important considerations:

- Reading from a discarded range without a subsequent write or prefetch will return an indeterminate value
- The discard operation can be undone by writing to the range or prefetching it via `cudaMemPrefetchAsync`
- Any reads, writes, or prefetches that occur simultaneously with the discard operation result in undefined behavior
- All devices must have a non-zero value for `cudaDevAttrConcurrentManagedAccess`

24.1.2.8.3 Data Usage Hints

When multiple processors simultaneously access the same data, `cudaMemAdvise` may be used to hint how the data at `[devPtr, devPtr + count)` will be accessed:

```
cudaError_t cudaMemAdvise(const void *devPtr,
                          size_t count,
                          enum cudaMemoryAdvise advice,
                          struct cudaMemLocation location);
```

Where `advice` may take the following values:

- `cudaMemAdviseSetReadMostly`: This implies that the data is mostly going to be read from and only occasionally written to. In general, it allows trading off read bandwidth for write bandwidth on this region. Example:

```
void test_advise_managed(cudaStream_t stream) {
  char *dataPtr;
  size_t dataSize = 64 * TPB;  // 16 KiB
  // Allocate memory using cudaMallocManaged
  // (malloc may be used on systems with full CUDA Unified memory support)
  cudaMallocManaged(&dataPtr, dataSize);
  // Set the advice on the memory region
  cudaMemLocation loc = {.type = cudaMemLocationTypeDevice, .id = myGpuId};
  cudaMemAdvise(dataPtr, dataSize, cudaMemAdviseSetReadMostly, loc);
  int outerLoopIter = 0;
  while (outerLoopIter < maxOuterLoopIter) {
    // The data is written to in the outer loop on the CPU
    init_data(dataPtr, dataSize);
    // The data is made available to all GPUs by prefetching.
```

(continues on next page)

(continued from previous page)

```
    // Prefetching here causes read duplication of data instead
    // of data migration
    cudaMemLocation location;
    location.type = cudaMemLocationTypeDevice;
    for (int device = 0; device < maxDevices; device++) {
      location.id = device;
      cudaMemPrefetchAsync(dataPtr, dataSize, location, 0 /* flags */, stream);
    }
    // The kernel only reads this data in the inner loop
    int innerLoopIter = 0;
    while (innerLoopIter < maxInnerLoopIter) {
      mykernel<<<32, TPB, 0, stream>>>((const char *)dataPtr, dataSize);
      innerLoopIter++;
    }
    outerLoopIter++;
  }
  cudaFree(dataPtr);
}
```

- `cudaMemAdviseSetPreferredLocation`: In general, any memory may be migrated at any time to any location, for example, when a given processor is running out of physical memory. This hint tells the system that migrating this memory region away from its preferred location is undesired, by setting the preferred location for the data to be the physical memory belonging to device. Passing in a value of `cudaMemLocationTypeHost` for location.type sets the preferred location as CPU memory. Other hints, like `cudaMemPrefetchAsync`, may override this hint, leading the memory to be migrated away from its preferred location.
- `cudaMemAdviseSetAccessedBy`: In some systems, it may be beneficial for performance to establish a mapping into memory before accessing the data from a given processor. This hint tells the system that the data will be frequently accessed by `location.id` when `location.type` is `cudaMemLocationTypeDevice`, enabling the system to assume that creating these mappings pays off. This hint does not imply where the data should reside, but it can be combined with `cudaMemAdviseSetPreferredLocation` to specify that.

Each advice can be also unset by using one of the following values: `cudaMemAdviseUnsetReadMostly`, `cudaMemAdviseUnsetPreferredLocation` and `cudaMemAdviseUnsetAccessedBy`.

24.1.2.8.4 Querying Data Usage Attributes on Managed Memory

A program can query memory range attributes assigned through `cudaMemAdvise` or `cudaMemPrefetchAsync` on CUDA Managed Memory by using the following API:

```
cudaMemRangeGetAttribute(void *data,
                         size_t dataSize,
                         enum cudaMemRangeAttribute attribute,
                         const void *devPtr,
                         size_t count);
```

This function queries an attribute of the memory range starting at `devPtr` with a size of `count` bytes. The memory range must refer to managed memory allocated via `cudaMallocManaged` or declared via `__managed__` variables. It is possible to query the following attributes:

- `cudaMemRangeAttributeReadMostly`: the result returned will be 1 if the entire memory range has the `cudaMemAdviseSetReadMostly` attribute set, or 0 otherwise.

- `cudaMemRangeAttributePreferredLocation`: the result returned will be a GPU device id or `cudaCpuDeviceId` if the entire memory range has the corresponding processor as preferred location, otherwise `cudaInvalidDeviceId` will be returned. An application can use this query API to make decision about staging data through CPU or GPU depending on the preferred location attribute of the managed pointer. Note that the actual location of the memory range at the time of the query may be different from the preferred location.
- `cudaMemRangeAttributeAccessedBy`: will return the list of devices that have that advise set for that memory range.
- `cudaMemRangeAttributeLastPrefetchLocation`: will return the last location to which the memory range was prefetched explicitly using `cudaMemPrefetchAsync`. Note that this simply returns the last location that the application requested to prefetch the memory range to. It gives no indication as to whether the prefetch operation to that location has completed or even begun.
- **`cudaMemRangeAttributePreferredLocationType`:**
  will return the location type of the preferred location which will be `cudaMemLocationTypeDevice` if all pages in the memory range have the same GPU as their preferred location, or will be `cudaMemLocationTypeHost` if all pages in the memory range have the CPU as their preferred location, or it will be `cudaMemLocationTypeHostNuma` if all the pages in the memory range have the same host NUMA node ID as their preferred location or it will be `cudaMemLocationTypeInvalid` if either all the pages don't have the same preferred location or some of the pages don't have a preferred location at all.
- **`cudaMemRangeAttributePreferredLocationId`:**
  If the `cudaMemRangeAttributePreferredLocationType` query for the same address range returns `cudaMemLocationTypeDevice`, it will be a valid device ordinal or if it returns `cudaMemLocationTypeHostNuma`, it will be a valid host NUMA node ID or if it returns any other location type, the id should be ignored.
- **`cudaMemRangeAttributeLastPrefetchLocationType`:**
  will be the last location type to which all pages in the memory range were prefetched explicitly via `cudaMemPrefetchAsync` which will be `cudaMemLocationTypeDevice` if all pages in the memory range were prefetched to the same GPU, or will be `cudaMemLocationTypeHost` if all pages in the memory range were prefetched to the CPU or it will be `cudaMemLocationTypeHostNuma` if all the pages in the memory range were prefetched to the same host NUMA node ID or it will be `cudaMemLocationTypeInvalid` if either all the pages were not prefetched to the same location or some of the pages were never prefetched at all.
- **`cudaMemRangeAttributeLastPrefetchLocationId`:**
  If the `cudaMemRangeAttributeLastPrefetchLocationType` query for the same address range returns `cudaMemLocationTypeDevice`, it will be a valid device ordinal or if it returns `cudaMemLocationTypeHostNuma`, it will be a valid host NUMA node ID or if it returns any other location type, the id should be ignored.

Additionally, multiple attributes can be queried by using corresponding `cudaMemRangeGetAttributes` function.

24.2. Unified memory on devices with full CUDA Unified Memory support

24.2.1. System-Allocated Memory: in-depth examples

Systems with full CUDA Unified Memory support allow the device to access any memory owned by the host process interacting with the device. This section shows a few advanced use-cases, using a kernel that simply prints the first 8 characters of an input character array to the standard output stream:

```
__global__ void kernel(const char* type, const char* data) {
  static const int n_char = 8;
  printf("%s - first %d characters: '", type, n_char);
  for (int i = 0; i < n_char; ++i) printf("%c", data[i]);
  printf("'\n");
}
```

The following tabs show various ways of how this kernel may be called:

Malloc

```
void test_malloc() {
  const char test_string[] = "Hello World";
  char* heap_data = (char*)malloc(sizeof(test_string));
  strncpy(heap_data, test_string, sizeof(test_string));
  kernel<<<1, 1>>>("malloc", heap_data);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
  free(heap_data);
}
```

Managed

```
void test_managed() {
  const char test_string[] = "Hello World";
  char* data;
  cudaMallocManaged(&data, sizeof(test_string));
  strncpy(data, test_string, sizeof(test_string));
  kernel<<<1, 1>>>("managed", data);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
  cudaFree(data);
}
```

Stack variable

```
void test_stack() {
  const char test_string[] = "Hello World";
  kernel<<<1, 1>>>("stack", test_string);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
}
```

File-scope static variable

```
void test_static() {
  static const char test_string[] = "Hello World";
  kernel<<<1, 1>>>("static", test_string);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
}
```

Global-scope variable

```
const char global_string[] = "Hello World";

void test_global() {
  kernel<<<1, 1>>>("global", global_string);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
}
```

Global-scope extern variable

```
// declared in separate file, see below
extern char* ext_data;

void test_extern() {
  kernel<<<1, 1>>>("extern", ext_data);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
}
```

```
/** This may be a non-CUDA file */
char* ext_data;
static const char global_string[] = "Hello World";

void __attribute__ ((constructor)) setup(void) {
  ext_data = (char*)malloc(sizeof(global_string));
  strncpy(ext_data, global_string, sizeof(global_string));
}

void __attribute__ ((destructor)) tear_down(void) {
  free(ext_data);
}
```

The first three tabs above show the example as already detailed in the *Programming Model section*. The next three tabs show various ways a file-scope or global-scope variable can be accessed from the device.

Note that for the extern variable, it could be declared and its memory owned and managed by a third-party library, which does not interact with CUDA at all.

Also note that stack variables as well as file-scope and global-scope variables can only be accessed through a pointer by the GPU. In this specific example, this is convenient because the character array is already declared as a pointer: `const char*`. However, consider the following example with a global-scope integer:

```
// this variable is declared at global scope
int global_variable;

__global__ void kernel_uncompilable() {
  // this causes a compilation error: global (__host__) variables must not
  // be accessed from __device__ / __global__ code
  printf("%d\n", global_variable);
}

// On systems with pageableMemoryAccess set to 1, we can access the address
// of a global variable. The below kernel takes that address as an argument
__global__ void kernel(int* global_variable_addr) {
  printf("%d\n", *global_variable_addr);
}
int main() {
  kernel<<<1, 1>>>(&global_variable);
  ...
  return 0;
}
```

In the example above, we need to ensure to pass a *pointer* to the global variable to the kernel instead of directly accessing the global variable in the kernel. This is because global variables without the `__managed__` specifier are declared as `__host__`-only by default, thus most compilers won't allow using these variables directly in device code as of now.

24.2.1.1 File-backed Unified Memory

Since *systems with full CUDA Unified Memory support* allow the device to access any memory owned by the host process, they can directly access file-backed memory.

Here, we show a modified version of the initial example shown in the previous section to use file-backed memory in order to print a string from the GPU, read directly from an input file. In the following example, the memory is backed by a physical file, but the example applies to memory-backed files, too, as detailed in the section on *Inter-Process Communication (IPC) with Unified Memory*.

```
__global__ void kernel(const char* type, const char* data) {
  static const int n_char = 8;
  printf("%s - first %d characters: '", type, n_char);
  for (int i = 0; i < n_char; ++i) printf("%c", data[i]);
  printf("'\n");
}
```

```
void test_file_backed() {
  int fd = open(INPUT_FILE_NAME, O_RDONLY);
```

(continues on next page)

(continued from previous page)

```
  ASSERT(fd >= 0, "Invalid file handle");
  struct stat file_stat;
  int status = fstat(fd, &file_stat);
  ASSERT(status >= 0, "Invalid file stats");
  char* mapped = (char*)mmap(0, file_stat.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
  ASSERT(mapped != MAP_FAILED, "Cannot map file into memory");
  kernel<<<1, 1>>>("file-backed", mapped);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
  ASSERT(munmap(mapped, file_stat.st_size) == 0, "Cannot unmap file");
  ASSERT(close(fd) == 0, "Cannot close file");
}
```

Note that on systems without the `hostNativeAtomicSupported` property, including *systems with Linux HMM enabled*, atomic accesses to file-backed memory are not supported.

24.2.1.2 Inter-Process Communication (IPC) with Unified Memory

Note: As of now, using IPC with Unified Memory can have significant performance implications.

Many applications prefer to manage one GPU per process, but still need to use Unified Memory, for example for over-subscription, and access it from multiple GPUs.

CUDA IPC (see *Interprocess Communication*) does not support Managed Memory: handles to this type of memory may not be shared through any of the mechanisms discussed in this section. On *systems with full CUDA Unified Memory support*, System-Allocated Memory is Inter-Process Communication (IPC) capable. Once access to System-Allocated Memory has been shared with other processes, the same *Programming Model* applies, similar to *File-backed Unified Memory*.

See the following references for more information on various ways of creating IPC-capable System-Allocated Memory under Linux:

- mmap with MAP_SHARED
- POSIX IPC APIs
- Linux memfd_create

Note that it is not possible to share memory between different hosts and their devices using this technique.

24.2.2. Performance Tuning

In order to achieve good performance with Unified Memory, it is important to:

- Understand how paging works on your system, and how to avoid unnecessary page faults.
- Understand the various mechanisms allowing you to keep data local to the accessing processor.
- Consider tuning your application for the granularity of memory transfers of your system.

As general advice, *Performance Hints* might provide improved performance, but using them incorrectly might degrade performance compared to the default behavior. Also note that any hint has a performance cost associated with it on the host, thus useful hints must at the very least improve performance enough to overcome this cost.

24.2.2.1 Memory Paging and Page Sizes

Many of the sections for unified memory performance tuning assume prior knowledge on virtual addressing, memory pages and page sizes. This section attempts to define all necessary terms and explain why paging matters for performance.

All currently supported systems for Unified Memory use a virtual address space: this means that memory addresses used by an application represent a *virtual* location which might be *mapped* to a physical location where the memory actually resides.

All currently supported processors, including both CPUs and GPUs, additionally use memory *paging*. Because all systems use a virtual address space, there are two types of memory pages:

- Virtual pages: this represents a fixed-size contiguous chunk of virtual memory per process tracked by the operating system, which can be *mapped* into physical memory. Note that the virtual page is linked to the *mapping*: for example, a single virtual address might be mapped into physical memory using different page sizes.
- Physical pages: this represents a fixed-size contiguous chunk of memory the processor's main Memory Management Unit (MMU) supports and into which a virtual page can be mapped.

Currently, all x86_64 CPUs use 4KiB physical pages. Arm CPUs support multiple physical page sizes - 4KiB, 16KiB, 32KiB and 64KiB - depending on the exact CPU. Finally, NVIDIA GPUs support multiple physical page sizes, but prefer 2MiB physical pages or larger. Note that these sizes are subject to change in future hardware.

The default page size of virtual pages usually corresponds to the physical page size, but an application may use different page sizes as long as they are supported by the operating system and the hardware. Typically, supported virtual page sizes must be powers of 2 and multiples of the physical page size.

The logical entity tracking the mapping of virtual pages into physical pages will be referred to as a *page table*, and each mapping of a given virtual page with a given virtual size to physical pages is called a *page table entry (PTE)*. All supported processors provide specific caches for the page table to speed up the translation of virtual addresses to physical addresses. These caches are called *translation lookaside buffers (TLBs)*.

There are two important aspects for performance tuning of applications:

- the choice of virtual page size,
- whether the system offers a combined page table used by both CPUs and GPUs, or separate page tables for each CPU and GPU individually.

24.2.2.1.1 Choosing the right page size

In general, small page sizes lead to less (virtual) memory fragmentation but more TLB misses, whereas larger page sizes lead to more memory fragmentation but less TLB misses. Additionally, memory migration is generally more expensive with larger page sizes compared to smaller page sizes, because we typically migrate full memory pages. This can cause larger latency spikes in an application using large page sizes. See also the next section for more details on page faults.

One important aspect for performance tuning is that TLB misses are generally significantly more expensive on the GPU compared to the CPU. This means that if a GPU thread frequently accesses random locations of Unified Memory mapped using a small enough page size, it might be significantly slower compared to the same accesses to Unified Memory mapped using a large enough page size. While a similar effect might occur for a CPU thread randomly accessing a large area of memory mapped using a small page size, the slowdown is less pronounced, meaning that the application might want to trade-off this slowdown with having less memory fragmentation.

Note that in general, applications should not tune their performance to the physical page size of a given processor, since physical page sizes are subject to change depending on the hardware. The advice above only applies to virtual page sizes.

24.2.2.1.2 CPU and GPU page tables: hardware coherency vs. software coherency

Note: In the remainder of the performance tuning documentation, we will refer to systems with a combined page table for both CPUs and GPUs as *hardware coherent* systems. Systems with separate page tables for CPUs and GPUs are referred to as *software coherent*.

Hardware coherent systems such as NVIDIA Grace Hopper offer a logically combined page table for both CPUs and GPUs. This is important because in order to access *System-Allocated Memory from the GPU*, the GPU uses whichever page table entry was created by the CPU for the requested memory. If that page table entry uses the default CPU page size of 4KiB or 64KiB, accesses to large virtual memory areas will cause significant TLB misses, thus significant slowdowns.

See the section on *configuring huge pages* for examples on how to ensure System-Allocated Memory uses large enough page sizes to avoid this type of issue.

On the other hand, on systems where the CPUs and GPUs each have their own logical page table, different performance tuning aspects should be considered: in order to *guarantee coherency*, these systems usually use *page faults* in case a processor accesses a memory address mapped into the physical memory of a different processor. Such a page fault means that:

- it needs to be ensured that the currently owning processor (where the physical page currently resides) cannot access this page anymore, either by deleting the page table entry or updating it.
- it needs to be ensured that the processor requesting access can access this page, either by creating a new page table entry or updating and existing entry, such that it becomes valid/active.
- the physical page backing this virtual page must be moved/migrated to the processor requesting access: this can be an expensive operation, and the amount of work is proportional to the page size.

Overall, hardware coherent systems provide significant performance benefits compared to software coherent systems in cases where frequent concurrent accesses to the same memory page are made by both CPU and GPU threads:

- less page-faults: these systems do not need to use page-faults for emulating coherency or migrating memory,
- less contention: these systems are coherent at cache-line granularity instead of page-size granularity, that is, when there is contention from multiple processors within a cache line, only the cache line is exchanged which is much smaller than the smallest page-size, and when the different processors access different cache-lines within a page, then there is no contention.

This impacts the performance of the following scenarios:

- Atomic updates to the same address concurrently from both CPUs and GPUs.
- Signaling a GPU thread from a CPU thread or vice-versa.

24.2.2.2 Direct Unified Memory Access from host

Some devices have hardware support for coherent reads, stores and atomic accesses from the host on GPU-resident unified memory. These devices have the attribute `cudaDevAttrDirectManagedMemAccessFromHost` set to 1. Note that all *hardware coherent systems* have this attribute set for NVLink-connected devices. On these systems, the host has direct access to GPU-resident memory without page faults and data migration (see *Data Usage Hints* for more details on memory usage hints). Note that with CUDA Managed Memory, the `cudaMemAdviseSetAccessedBy` hint with location type `cudaMemLocationTypeHost` is necessary to enable this direct access without page faults.

Consider an example code below:

System Allocator

```
__global__ void write(int *ret, int a, int b) {
  ret[threadIdx.x] = a + b + threadIdx.x;
}

__global__ void append(int *ret, int a, int b) {
  ret[threadIdx.x] += a + b + threadIdx.x;
}
void test_malloc() {
  int *ret = (int*)malloc(1000 * sizeof(int));
  // for shared page table systems, the following hint is not necessary
  cudaMemLocation location = {.type = cudaMemLocationTypeHost};
  cudaMemAdvise(ret, 1000 * sizeof(int), cudaMemAdviseSetAccessedBy, location);

  write<<< 1, 1000 >>>(ret, 10, 100);            // pages populated in GPU memory
  cudaDeviceSynchronize();
  for(int i = 0; i < 1000; i++)
      printf("%d: A+B = %d\n", i, ret[i]);        // directManagedMemAccessFromHost=1:
↪CPU accesses GPU memory directly without migrations
                                                  // directManagedMemAccessFromHost=0:
↪CPU faults and triggers device-to-host migrations
  append<<< 1, 1000 >>>(ret, 10, 100);            // directManagedMemAccessFromHost=1:
↪GPU accesses GPU memory without migrations
  cudaDeviceSynchronize();                        // directManagedMemAccessFromHost=0:
↪GPU faults and triggers host-to-device migrations
  free(ret);
}
```

Managed

```
__global__ void write(int *ret, int a, int b) {
  ret[threadIdx.x] = a + b + threadIdx.x;
}

__global__ void append(int *ret, int a, int b) {
  ret[threadIdx.x] += a + b + threadIdx.x;
}

void test_managed() {
  int *ret;
  cudaMallocManaged(&ret, 1000 * sizeof(int));
  cudaMemLocation location = {.type = cudaMemLocationTypeHost};
  cudaMemAdvise(ret, 1000 * sizeof(int), cudaMemAdviseSetAccessedBy, location);  //
↪set direct access hint

  write<<< 1, 1000 >>>(ret, 10, 100);            // pages populated in GPU memory
  cudaDeviceSynchronize();
  for(int i = 0; i < 1000; i++)
      printf("%d: A+B = %d\n", i, ret[i]);        // directManagedMemAccessFromHost=1:
↪CPU accesses GPU memory directly without migrations
                                                  // directManagedMemAccessFromHost=0:
↪CPU faults and triggers device-to-host migrations
  append<<< 1, 1000 >>>(ret, 10, 100);            // directManagedMemAccessFromHost=1:
↪GPU accesses GPU memory without migrations
  cudaDeviceSynchronize();                        // directManagedMemAccessFromHost=0:
↪GPU faults and triggers host-to-device migrations
  cudaFree(ret);
}
```

After `write` kernel is completed, `ret` will be created and initialized in GPU memory. Next, the CPU will access `ret` followed by `append` kernel using the same `ret` memory again. This code will show different behavior depending on the system architecture and support of hardware coherency:

- On systems with `directManagedMemAccessFromHost=1`: CPU accesses to the managed buffer will not trigger any migrations; the data will remain resident in GPU memory and any subsequent GPU kernels can continue to access it directly without inflicting faults or migrations.
- On systems with `directManagedMemAccessFromHost=0`: CPU accesses to the managed buffer will page fault and initiate data migration; any GPU kernel trying to access the same data first time will page fault and migrate pages back to GPU memory.

24.2.2.3 Host Native Atomics

Some devices, including NVLink-connected devices in *hardware coherent systems*, support hardware-accelerated atomic accesses to CPU-resident memory. This implies that atomic accesses to host memory do not have to be emulated with a page fault. For these devices, the attribute `cudaDevAttrHostNativeAtomicSupported` is set to 1.

24.2.2.4 Atomic accesses & synchronization primitives

CUDA Unified Memory supports all atomic operations available to host and device threads, enabling all threads to cooperate by concurrently accessing the same shared memory location. The CUDA C++ standard library provides many heterogeneous synchronization primitives tuned for concurrent use between host and device threads, including `cuda::atomic`, `cuda::atomic_ref`, `cuda::barrier`, `cuda::semaphore`, among many others.

On systems without *CPU and GPU page tables: hardware coherency vs. software coherency*, atomic accesses from the device to file-backed host memory are not supported. The following example code is valid on systems with *CPU and GPU page tables: hardware coherency vs. software coherency* but exhibits undefined behavior on other systems:

```
#include <cuda/atomic>

#include <cstdio>
#include <fcntl.h>
#include <sys/mman.h>

#define ERR(msg, ...) { fprintf(stderr, msg, ##__VA_ARGS__); return EXIT_FAILURE; }

__global__ void kernel(int* ptr) {
  cuda::atomic_ref{*ptr}.store(2);
}

int main() {
  // this will be closed/deleted by default on exit
  FILE* tmp_file = tmpfile64();
  // need to allocate space in the file, we do this with posix_fallocate here
  int status = posix_fallocate(fileno(tmp_file), 0, 4096);
  if (status != 0) ERR("Failed to allocate space in temp file\n");
  int* ptr = (int*)mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_PRIVATE, fileno(tmp_
↪file), 0);
  if (ptr == MAP_FAILED) ERR("Failed to map temp file\n");

  // initialize the value in our file-backed memory
  *ptr = 1;
  printf("Atom value: %d\n", *ptr);

  // device and host thread access ptr concurrently, using cuda::atomic_ref
  kernel<<<1, 1>>>(ptr);
  while (cuda::atomic_ref{*ptr}.load() != 2);
  // this will always be 2
  printf("Atom value: %d\n", *ptr);

  return EXIT_SUCCESS;
}
```

On systems without *CPU and GPU page tables: hardware coherency vs. software coherency*, atomic ac-

cesses to unified memory may incur page faults which can lead to significant latencies. Note that this is not the case for all GPU atomics to CPU memory on these systems: operations listed by `nvidia-smi -q | grep "Atomic Caps Outbound"` may avoid page faults.

On systems with *CPU and GPU page tables: hardware coherency vs. software coherency*, atomics between host and device do not require page faults, but may still fault for other reasons that any memory access can fault for.

24.2.2.5 Memcpy()/Memset() Behavior With Unified Memory

`cudaMemcpy*()` and `cudaMemset*()` accept any unified memory pointer as arguments.

For `cudaMemcpy*()`, the direction specified as `cudaMemcpyKind` is a performance hint, which can have a higher performance impact if any of the arguments is a unified memory pointer.

Thus, it is recommended to follow the following performance advice:

- When the physical location of unified memory is known, use an accurate `cudaMemcpyKind` hint.
- Prefer `cudaMemcpyDefault` over an inaccurate `cudaMemcpyKind` hint.
- Always use populated (initialized) buffers: avoid using these APIs to initialize memory.
- Avoid using `cudaMemcpy*()` if both pointers point to System-Allocated Memory: launch a kernel or use a CPU memory copy algorithm such as `std::memcpy` instead.

24.3. Unified memory on devices without full CUDA Unified Memory support

24.3.1. Unified memory on devices with only CUDA Managed Memory support

For devices with compute capability 6.x or higher but without *pageable memory access*, CUDA Managed Memory is fully supported and coherent. The programming model and performance tuning of unified memory is largely similar to the model as described in *Unified memory on devices with full CUDA Unified Memory support*, with the notable exception that system allocators cannot be used to allocate memory. Thus, the following list of sub-sections do not apply:

- *System-Allocated Memory: in-depth examples*
- *Hardware/Software Coherency*

24.3.2. Unified memory on Windows or devices with compute capability 5.x

Devices with compute capability lower than 6.0 or Windows platforms support CUDA Managed Memory v1.0 with limited support for data migration and coherency as well as memory oversubscription. The following sub-sections describe in more detail how to use and optimize Managed Memory on these platforms.

24.3.2.1 Data Migration and Coherency

GPU architectures of compute capability lower than 6.0 do not support fine-grained movement of the managed data to GPU on-demand. Whenever a GPU kernel is launched all managed memory generally has to be transferred to GPU memory to avoid faulting on memory access. With compute capability 6.x a new GPU page faulting mechanism is introduced that provides more seamless Unified Memory functionality. Combined with the system-wide virtual address space, page faulting provides several benefits. First, page faulting means that the CUDA system software doesn't need to synchronize all managed memory allocations to the GPU before each kernel launch. If a kernel running on the GPU accesses a page that is not resident in its memory, it faults, allowing the page to be automatically migrated to the GPU memory on-demand. Alternatively, the page may be mapped into the GPU address space for access over the PCIe or NVLink interconnects (mapping on access can sometimes be faster than migration). Note that Unified Memory is system-wide: GPUs (and CPUs) can fault on and migrate memory pages either from CPU memory or from the memory of other GPUs in the system.

24.3.2.2 GPU Memory Oversubscription

Devices of compute capability lower than 6.0 cannot allocate more managed memory than the physical size of GPU memory.

24.3.2.3 Multi-GPU

On systems with devices of compute capabilities lower than 6.0 managed allocations are automatically visible to all GPUs in a system via the peer-to-peer capabilities of the GPUs. Managed memory allocations behave similar to unmanaged memory allocated using `cudaMalloc()`: the current active device is the home for the physical allocation but other GPUs in the system will access the memory at reduced bandwidth over the PCIe bus.

On Linux the managed memory is allocated in GPU memory as long as all GPUs that are actively being used by a program have the peer-to-peer support. If at any time the application starts using a GPU that doesn't have peer-to-peer support with any of the other GPUs that have managed allocations on them, then the driver will migrate all managed allocations to system memory. In this case, all GPUs experience PCIe bandwidth restrictions.

On Windows, if peer mappings are not available (for example, between GPUs of different architectures), then the system will automatically fall back to using zero-copy memory, regardless of whether both GPUs are actually used by a program. If only one GPU is actually going to be used, it is necessary to set the `CUDA_VISIBLE_DEVICES` environment variable before launching the program. This constrains which GPUs are visible and allows managed memory to be allocated in GPU memory.

Alternatively, on Windows users can also set `CUDA_MANAGED_FORCE_DEVICE_ALLOC` to a non-zero value to force the driver to always use device memory for physical storage. When this environment variable is set to a non-zero value, all devices used in that process that support managed memory have to

be peer-to-peer compatible with each other. The error `::cudaErrorInvalidDevice` will be returned if a device that supports managed memory is used and it is not peer-to-peer compatible with any of the other managed memory supporting devices that were previously used in that process, even if `::cudaDeviceReset` has been called on those devices. These environment variables are described in *CUDA Environment Variables*. Note that starting from CUDA 8.0 `CUDA_MANAGED_FORCE_DEVICE_ALLOC` has no effect on Linux operating systems.

24.3.2.4 Coherency and Concurrency

Simultaneous access to managed memory on devices of compute capability lower than 6.0 is not possible, because coherence could not be guaranteed if the CPU accessed a Unified Memory allocation while a GPU kernel was active.

24.3.2.4.1 GPU Exclusive Access To Managed Memory

To ensure coherency on pre-6.x GPU architectures, the Unified Memory programming model puts constraints on data accesses while both the CPU and GPU are executing concurrently. In effect, the GPU has exclusive access to all managed data while any kernel operation is executing, regardless of whether the specific kernel is actively using the data. When managed data is used with `cudaMemcpy*()` or `cudaMemset*()`, the system may choose to access the source or destination from the host or the device, which will put constraints on concurrent CPU access to that data while the `cudaMemcpy*()` or `cudaMemset*()` is executing. See *Memcpy()/Memset() Behavior With Unified Memory* for further details.

It is not permitted for the CPU to access any managed allocations or variables while the GPU is active for devices with `concurrentManagedAccess` property set to 0. On these systems concurrent CPU/GPU accesses, even to different managed memory allocations, will cause a segmentation fault because the page is considered inaccessible to the CPU.

```
__device__ __managed__ int x, y=2;
__global__  void  kernel() {
    x = 10;
}
int main() {
    kernel<<< 1, 1 >>>();
    y = 20;            // Error on GPUs not supporting concurrent access

    cudaDeviceSynchronize();
    return  0;
}
```

In example above, the GPU program `kernel` is still active when the CPU touches `y`. (Note how it occurs before `cudaDeviceSynchronize()`.) The code runs successfully on devices of compute capability 6.x due to the GPU page faulting capability which lifts all restrictions on simultaneous access. However, such memory access is invalid on pre-6.x architectures even though the CPU is accessing different data than the GPU. The program must explicitly synchronize with the GPU before accessing `y`:

```
__device__ __managed__ int x, y=2;
__global__  void  kernel() {
    x = 10;
}
int main() {
    kernel<<< 1, 1 >>>();
```

(continues on next page)

(continued from previous page)

```
    cudaDeviceSynchronize();
    y = 20;            //  Success on GPUs not supporing concurrent access
    return  0;
}
```

As this example shows, on systems with pre-6.x GPU architectures, a CPU thread may not access any managed data in between performing a kernel launch and a subsequent synchronization call, regardless of whether the GPU kernel actually touches that same data (or any managed data at all). The mere potential for concurrent CPU and GPU access is sufficient for a process-level exception to be raised.

Note that if memory is dynamically allocated with `cudaMallocManaged()` or `cuMemAllocManaged()` while the GPU is active, the behavior of the memory is unspecified until additional work is launched or the GPU is synchronized. Attempting to access the memory on the CPU during this time may or may not cause a segmentation fault. This does not apply to memory allocated using the flag `cudaMemAttachHost` or `CU_MEM_ATTACH_HOST`.

24.3.2.4.2 Explicit Synchronization and Logical GPU Activity

Note that explicit synchronization is required even if `kernel` runs quickly and finishes before the CPU touches `y` in the above example. Unified Memory uses logical activity to determine whether the GPU is idle. This aligns with the CUDA programming model, which specifies that a kernel can run at any time following a launch and is not guaranteed to have finished until the host issues a synchronization call.

Any function call that logically guarantees the GPU completes its work is valid. This includes `cudaDeviceSynchronize()`; `cudaStreamSynchronize()` and `cudaStreamQuery()` (provided it returns `cudaSuccess` and not `cudaErrorNotReady`) where the specified stream is the only stream still executing on the GPU; `cudaEventSynchronize()` and `cudaEventQuery()` in cases where the specified event is not followed by any device work; as well as uses of `cudaMemcpy()` and `cudaMemset()` that are documented as being fully synchronous with respect to the host.

Dependencies created between streams will be followed to infer completion of other streams by synchronizing on a stream or event. Dependencies can be created via `cudaStreamWaitEvent()` or implicitly when using the default (NULL) stream.

It is legal for the CPU to access managed data from within a stream callback, provided no other stream that could potentially be accessing managed data is active on the GPU. In addition, a callback that is not followed by any device work can be used for synchronization: for example, by signaling a condition variable from inside the callback; otherwise, CPU access is valid only for the duration of the callback(s).

There are several important points of note:

- It is always permitted for the CPU to access non-managed zero-copy data while the GPU is active.
- The GPU is considered active when it is running any kernel, even if that kernel does not make use of managed data. If a kernel might use data, then access is forbidden, unless device property `concurrentManagedAccess` is `1`.
- There are no constraints on concurrent inter-GPU access of managed memory, other than those that apply to multi-GPU access of non-managed memory.
- There are no constraints on concurrent GPU kernels accessing managed data.

Note how the last point allows for races between GPU kernels, as is currently the case for non-managed GPU memory. As mentioned previously, managed memory functions identically to non-managed memory from the perspective of the GPU. The following code example illustrates these points:

```
int main() {
    cudaStream_t stream1, stream2;
    cudaStreamCreate(&stream1);
    cudaStreamCreate(&stream2);
    int *non_managed, *managed, *also_managed;
    cudaMallocHost(&non_managed, 4);    // Non-managed, CPU-accessible memory
    cudaMallocManaged(&managed, 4);
    cudaMallocManaged(&also_managed, 4);
    // Point 1: CPU can access non-managed data.
    kernel<<< 1, 1, 0, stream1 >>>(managed);
    *non_managed = 1;
    // Point 2: CPU cannot access any managed data while GPU is busy,
    //          unless concurrentManagedAccess = 1
    // Note we have not yet synchronized, so "kernel" is still active.
    *also_managed = 2;      // Will issue segmentation fault
    // Point 3: Concurrent GPU kernels can access the same data.
    kernel<<< 1, 1, 0, stream2 >>>(managed);
    // Point 4: Multi-GPU concurrent access is also permitted.
    cudaSetDevice(1);
    kernel<<< 1, 1 >>>(managed);
    return  0;
}
```

24.3.2.4.3 Managing Data Visibility and Concurrent CPU + GPU Access with Streams

Until now it was assumed that for SM architectures before 6.x: 1) any active kernel may use any managed memory, and 2) it was invalid to use managed memory from the CPU while a kernel is active. Here we present a system for finer-grained control of managed memory designed to work on all devices supporting managed memory, including older architectures with `concurrentManagedAccess` equal to 0.

The CUDA programming model provides streams as a mechanism for programs to indicate dependence and independence among kernel launches. Kernels launched into the same stream are guaranteed to execute consecutively, while kernels launched into different streams are permitted to execute concurrently. Streams describe independence between work items and hence allow potentially greater efficiency through concurrency.

Unified Memory builds upon the stream-independence model by allowing a CUDA program to explicitly associate managed allocations with a CUDA stream. In this way, the programmer indicates the use of data by kernels based on whether they are launched into a specified stream or not. This enables opportunities for concurrency based on program-specific data access patterns. The function to control this behavior is:

```
cudaError_t cudaStreamAttachMemAsync(cudaStream_t stream,
                                     void *ptr,
                                     size_t length=0,
                                     unsigned int flags=0);
```

The `cudaStreamAttachMemAsync()` function associates `length` bytes of memory starting from `ptr` with the specified `stream`. (Currently, `length` must always be 0 to indicate that the entire region should be attached.) Because of this association, the Unified Memory system allows CPU access to this memory region so long as all operations in `stream` have completed, regardless of whether other streams are active. In effect, this constrains exclusive ownership of the managed memory region by an active GPU to per-stream activity instead of whole-GPU activity.

Most importantly, if an allocation is not associated with a specific stream, it is visible to all running

kernels regardless of their stream. This is the default visibility for a `cudaMallocManaged()` allocation or a `__managed__` variable; hence, the simple-case rule that the CPU may not touch the data while any kernel is running.

By associating an allocation with a specific stream, the program makes a guarantee that only kernels launched into that stream will touch that data. No error checking is performed by the Unified Memory system: it is the programmer's responsibility to ensure that guarantee is honored.

In addition to allowing greater concurrency, the use of `cudaStreamAttachMemAsync()` can (and typically does) enable data transfer optimizations within the Unified Memory system that may affect latencies and other overhead.

24.3.2.4.4 Stream Association Examples

Associating data with a stream allows fine-grained control over CPU + GPU concurrency, but what data is visible to which streams must be kept in mind when using devices of compute capability lower than 6.0. Looking at the earlier synchronization example:

```
__device__ __managed__ int x, y=2;
__global__  void  kernel() {
    x = 10;
}
int main() {
    cudaStream_t stream1;
    cudaStreamCreate(&stream1);
    cudaStreamAttachMemAsync(stream1, &y, 0, cudaMemAttachHost);
    cudaDeviceSynchronize();          // Wait for Host attachment to occur.
    kernel<<< 1, 1, 0, stream1 >>>(); // Note: Launches into stream1.
    y = 20;                           // Success – a kernel is running but "y"
                                      // has been associated with no stream.
    return  0;
}
```

Here we explicitly associate `y` with host accessibility, thus enabling access at all times from the CPU. (As before, note the absence of `cudaDeviceSynchronize()` before the access.) Accesses to `y` by the GPU running `kernel` will now produce undefined results.

Note that associating a variable with a stream does not change the associating of any other variable. For example, associating `x` with `stream1` does not ensure that only `x` is accessed by kernels launched in `stream1`, thus an error is caused by this code:

```
__device__ __managed__ int x, y=2;
__global__  void  kernel() {
    x = 10;
}
int main() {
    cudaStream_t stream1;
    cudaStreamCreate(&stream1);
    cudaStreamAttachMemAsync(stream1, &x);// Associate "x" with stream1.
    cudaDeviceSynchronize();              // Wait for "x" attachment to occur.
    kernel<<< 1, 1, 0, stream1 >>>();     // Note: Launches into stream1.
    y = 20;                               // ERROR: "y" is still associated globally
                                          // with all streams by default
    return  0;
}
```

Note how the access to y will cause an error because, even though x has been associated with a stream, we have told the system nothing about who can see y. The system therefore conservatively assumes that `kernel` might access it and prevents the CPU from doing so.

24.3.2.4.5 Stream Attach With Multithreaded Host Programs

The primary use for `cudaStreamAttachMemAsync()` is to enable independent task parallelism using CPU threads. Typically in such a program, a CPU thread creates its own stream for all work that it generates because using CUDA's NULL stream would cause dependencies between threads.

The default global visibility of managed data to any GPU stream can make it difficult to avoid interactions between CPU threads in a multi-threaded program. Function `cudaStreamAttachMemAsync()` is therefore used to associate a thread's managed allocations with that thread's own stream, and the association is typically not changed for the life of the thread.

Such a program would simply add a single call to `cudaStreamAttachMemAsync()` to use unified memory for its data accesses:

```
// This function performs some task, in its own private stream.
void run_task(int *in, int *out, int length) {
    // Create a stream for us to use.
    cudaStream_t stream;
    cudaStreamCreate(&stream);
    // Allocate some managed data and associate with our stream.
    // Note the use of the host-attach flag to cudaMallocManaged();
    // we then associate the allocation with our stream so that
    // our GPU kernel launches can access it.
    int *data;
    cudaMallocManaged((void **)&data, length, cudaMemAttachHost);
    cudaStreamAttachMemAsync(stream, data);
    cudaStreamSynchronize(stream);
    // Iterate on the data in some way, using both Host & Device.
    for(int i=0; i<N; i++) {
        transform<<< 100, 256, 0, stream >>>(in, data, length);
        cudaStreamSynchronize(stream);
        host_process(data, length);    // CPU uses managed data.
        convert<<< 100, 256, 0, stream >>>(out, data, length);
    }
    cudaStreamSynchronize(stream);
    cudaStreamDestroy(stream);
    cudaFree(data);
}
```

In this example, the allocation-stream association is established just once, and then `data` is used repeatedly by both the host and device. The result is much simpler code than occurs with explicitly copying data between host and device, although the result is the same.

24.3.2.4.6 Advanced Topic: Modular Programs and Data Access Constraints

In the previous example `cudaMallocManaged()` specifies the `cudaMemAttachHost` flag, which creates an allocation that is initially invisible to device-side execution. (The default allocation would be visible to all GPU kernels on all streams.) This ensures that there is no accidental interaction with another thread's execution in the interval between the data allocation and when the data is acquired for a specific stream.

Without this flag, a new allocation would be considered in-use on the GPU if a kernel launched by another thread happens to be running. This might impact the thread's ability to access the newly allocated data from the CPU (for example, within a base-class constructor) before it is able to explicitly attach it to a private stream. To enable safe independence between threads, therefore, allocations should be made specifying this flag.

Note: An alternative would be to place a process-wide barrier across all threads after the allocation has been attached to the stream. This would ensure that all threads complete their data/stream associations before any kernels are launched, avoiding the hazard. A second barrier would be needed before the stream is destroyed because stream destruction causes allocations to revert to their default visibility. The `cudaMemAttachHost` flag exists both to simplify this process, and because it is not always possible to insert global barriers where required.

24.3.2.4.7 Memcpy()/Memset() Behavior With Stream-associated Unified Memory

See *Memcpy()/Memset() Behavior With Unified Memory* for a general overview of `cudaMemcpy*` / `cudaMemset*` behavior on devices with `concurrentManagedAccess` set. On devices where `concurrentManagedAccess` is not set, the following rules apply:

If `cudaMemcpyHostTo*` is specified and the source data is unified memory, then it will be accessed from the host if it is coherently accessible from the host in the copy stream *(1)*; otherwise it will be accessed from the device. Similar rules apply to the destination when `cudaMemcpy*ToHost` is specified and the destination is unified memory.

If `cudaMemcpyDeviceTo*` is specified and the source data is unified memory, then it will be accessed from the device. The source must be coherently accessible from the device in the copy stream *(2)*; otherwise, an error is returned. Similar rules apply to the destination when `cudaMemcpy*ToDevice` is specified and the destination is unified memory.

If `cudaMemcpyDefault` is specified, then unified memory will be accessed from the host either if it cannot be coherently accessed from the device in the copy stream *(2)* or if the preferred location for the data is `cudaCpuDeviceId` and it can be coherently accessed from the host in the copy stream *(1)*; otherwise, it will be accessed from the device.

When using `cudaMemset*()` with unified memory, the data must be coherently accessible from the device in the stream being used for the `cudaMemset()` operation *(2)*; otherwise, an error is returned.

When data is accessed from the device either by `cudaMemcpy*` or `cudaMemset*`, the stream of operation is considered to be active on the GPU. During this time, any CPU access of data that is associated with that stream or data that has global visibility, will result in a segmentation fault if the GPU has a zero value for the device attribute `concurrentManagedAccess`. The program must synchronize appropriately to ensure the operation has completed before accessing any associated data from the CPU.

1. Coherently accessible from the host in a given stream means that the memory neither has global visibility nor is it associated with the given stream.

2. Coherently accessible from the device in a given stream means that the memory either has global visibility or is associated with the given stream.

Chapter 25. Lazy Loading

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

25.1. What is Lazy Loading?

Lazy Loading delays loading of CUDA modules and kernels from program initialization closer to kernels execution. If a program does not use every single kernel it has included, then some kernels will be loaded unneccessarily. This is very common, especially if you include any libraries. Most of the time, programs only use a small amount of kernels from libraries they include.

Thanks to Lazy Loading, programs are able to only load kernels they are actually going to use, saving time on initialization. This reduces memory overhead, both on GPU memory and host memory.

Lazy Loading is enabled by setting the `CUDA_MODULE_LOADING` environment variable to `LAZY`.

Firstly, CUDA Runtime will no longer load all modules during program initialization, with the exception of modules containing managed variables. Each module will be loaded on first usage of a variable or a kernel from that module. This optimization is only relevant to CUDA Runtime users, CUDA Driver users who use `cuModuleLoad` are unaffected. This optimization shipped in CUDA 11.8. The behavior for CUDA Driver users who use `cuLibraryLoad` to load module data into memory can be changed by setting the `CUDA_MODULE_DATA_LOADING` environment variable.

Secondly, loading a module (`cuModuleLoad*()` family of functions) will not be loading kernels immediately, instead it will delay loading of a kernel until `cuModuleGetFunction()` is called. There are certain exceptions here, some kernels have to be loaded during `cuModuleLoad*()`, such as kernels of which pointers are stored in global variables. This optimization is relevant to both CUDA Runtime and CUDA Driver users. CUDA Runtime will only call `cuModuleGetFunction()` when a kernel is used/referenced for the first time. This optimization shipped in CUDA 11.7.

Both of these optimizations are designed to be invisible to the user, assuming CUDA Programming Model is followed.

25.2. Lazy Loading version support

Lazy Loading is a CUDA Runtime and CUDA Driver feature. Upgrades to both might be necessary to utilize the feature.

25.2.1. Driver

Lazy Loading requires R515+ user-mode library, but it supports Forward Compatibility, meaning it can run on top of older kernel mode drivers.

Without R515+ user-mode library, Lazy Loading is not available in any shape or form, even if toolkit version is 11.7+.

25.2.2. Toolkit

Lazy Loading was introduced in CUDA 11.7, and received a significant upgrade in CUDA 11.8.

If your application uses CUDA Runtime, then in order to see benefits from Lazy Loading your application must use 11.7+ CUDA Runtime.

As CUDA Runtime is usually linked statically into programs and libraries, this means that you have to recompile your program with CUDA 11.7+ toolkit and use CUDA 11.7+ libraries.

Otherwise you will not see the benefits of Lazy Loading, even if your driver version supports it.

If only some of your libraries are 11.7+, you will only see benefits of Lazy Loading in those libraries. Other libraries will still load everything eagerly.

25.2.3. Compiler

Lazy Loading does not require any compiler support. Both SASS and PTX compiled with pre-11.7 compilers can be loaded with Lazy Loading enabled, and will see full benefits of the feature. However, 11.7+ CUDA Runtime is still required, as described above.

25.3. Triggering loading of kernels in lazy mode

Loading kernels and variables happens automatically, without any need for explicit loading. Simply launching a kernel or referencing a variable or a kernel will automatically load relevant modules and kernels.

However, if for any reason you wish to load a kernel without executing it or modifying it in any way, we recommend the following.

25.3.1. CUDA Driver API

Loading of kernels happens during `cuModuleGetFunction()` call. This call is necessary even without Lazy Loading, as it is the only way to obtain a kernel handle.

However, you can also use this API to control with finer granularity when kernels are loaded.

25.3.2. CUDA Runtime API

CUDA Runtime API manages module management automatically, so we recommend simply using `cudaFuncGetAttributes()` to reference the kernel.

This will ensure that the kernel is loaded without changing the state.

25.4. Querying whether Lazy Loading is Turned On

In order to check whether user enabled Lazy Loading, `CUresult cuModuleGetLoadingMode ( CUmoduleLoadingMode* mode )` can be used.

It's important to note that CUDA must be initialized before running this function. Sample usage can be seen in the snippet below.

```
#include "cuda.h"
#include "assert.h"
#include "iostream"

int main() {
        CUmoduleLoadingMode mode;

        assert(CUDA_SUCCESS == cuInit(0));
        assert(CUDA_SUCCESS == cuModuleGetLoadingMode(&mode));

        std::cout << "CUDA Module Loading Mode is " << ((mode == CU_MODULE_LAZY_
↪LOADING) ? "lazy" : "eager") << std::endl;

        return 0;
}
```

25.5. Possible Issues when Adopting Lazy Loading

Lazy Loading is designed so that it should not require any modifications to applications to use it. That said, there are some caveats, especially when applications are not fully compliant with CUDA Programming Model.

25.5.1. Concurrent Execution

Loading kernels might require context synchronization. Some programs incorrectly treat the possibility of concurrent execution of kernels as a guarantee. In such cases, if program assumes that two kernels will be able to execute concurrently, and one of the kernels will not return without the other kernel executing, there is a possibility of a deadlock.

If kernel A will be spinning in an infinite loop until kernel B is executing. In such case launching kernel B will trigger lazy loading of kernel B. If this loading will require context synchronization, then we have a deadlock: kernel A is waiting for kernel B, but loading kernel B is stuck waiting for kernel A to finish to synchronize the context.

Such program is an anti-pattern, but if for any reason you want to keep it you can do the following:

- preload all kernels that you hope to execute concurrently prior to launching them
- run application with `CUDA_MODULE_DATA_LOADING=EAGER` to force loading data eagerly without forcing each function to load eagerly

25.5.2. Allocators

Lazy Loading delays loading code from initialization phase of the program closer to execution phase. Loading code onto the GPU requires memory allocation.

If your application tries to allocate the entire VRAM on startup, for example, to use it for its own allocator, then it might turn out that there will be no more memory left to load the kernels. This is despite the fact that overall Lazy Loading frees up more memory for the user. CUDA will need to allocate some memory to load each kernel, which usually happens at first launch time of each kernel. If your application allocator greedily allocated everything, CUDA will fail to allocate memory.

Possible solutions:

- use `cudaMallocAsync()` instead of an allocator that allocates the entire VRAM on startup
- add some buffer to compensate for the delayed loading of kernels
- preload all kernels that will be used in the program before trying to initialize your allocator

25.5.3. Autotuning

Some applications launch several kernels implementing the same functionality to determine which one is the fastest. While it is overall advisable to run at least one warmup iteration, it becomes especially important with Lazy Loading. After all, including time taken to load the kernel will skew your results.

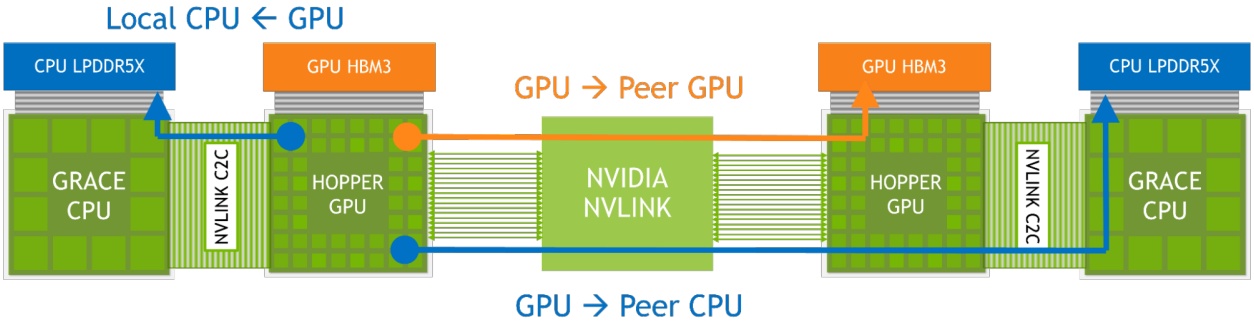
Possible solutions:

- do at least one warmup interaction prior to measurement
- preload the benchmarked kernel prior to launching it

Chapter 26. Extended GPU Memory

Warning: This document has been replaced by a new CUDA Programming Guide. The information in this document should be considered legacy, and this document is no longer being updated as of CUDA 13.0. Please refer to the CUDA Programming Guide for up-to-date information on CUDA.

The Extended GPU Memory (EGM) feature, utilizing the high-bandwidth NVLink-C2C, facilitates efficient access to all system memory by GPUs, in a single-node system. EGM applies to integrated CPU-GPU NVIDIA systems by allowing physical memory allocation that can be accessed from any GPU thread within the setup. EGM ensures that all GPUs can access its resources at the speed of either GPU-GPU NVLink or NVLink-C2C.



In this setup, memory accesses occur via the local high-bandwidth NVLink-C2C. For remote memory accesses, GPU NVLink and, in some cases, NVLink-C2C are used. With EGM, GPU threads gain the capability to access all available memory resources, including CPU attached memory and HBM3, over the NVSwitch fabric.

26.1. Preliminaries

Before diving into API changes for EGM functionalities, we are going to cover currently supported topologies, identifier assignment, prerequisites for virtual memory management, and CUDA types for EGM.

26.1.1. EGM Platforms: System topology

Currently, EGM can be enabled in three platforms: **(1) Single-Node, Single-GPU**: Consists of an Arm-based CPU, CPU attached memory, and a GPU. Between the CPU and the GPU there is a high bandwidth C2C (Chip-to-Chip) interconnect. **(2) Single-Node, Multi-GPU**: Consists of fully connected four single-node, single-GPU platforms. **(3) Multi-Node, Single-GPU**: Two or more single-node multi-socket systems.

Note: Using `cgroups` to limit available devices will block routing over EGM and cause performance issues. Use `CUDA_VISIBLE_DEVICES` instead.

26.1.2. Socket Identifiers: What are they? How to access them?

NUMA (Non-Uniform Memory Access) is a memory architecture used in multi-processor computer systems such that the memory is divided into multiple nodes. Each node has its own processors and memory. In such a system, NUMA divides the system into nodes and assigns a unique identifier (*numaID*) to every node.

EGM uses the NUMA node identifier which is assigned by the operating system. Note that, this identifier is different from the ordinal of a device and it is associated with the closest host node. In addition to the existing methods, the user can obtain the identifier of the host node (*numaID*) by calling cuDeviceGetAttribute with `CU_DEVICE_ATTRIBUTE_HOST_NUMA_ID` attribute type as follows:

```
int numaId;
cuDeviceGetAttribute(&numaId, CU_DEVICE_ATTRIBUTE_HOST_NUMA_ID, deviceOrdinal);
```

26.1.3. Allocators and EGM support

Mapping system memory as EGM does not cause any performance issues. In fact, accessing a remote socket's system memory mapped as EGM is going to be faster. Because, with EGM traffic is guaranteed to be routed over NVLinks. Currently, `cuMemCreate` and `cudaMemPoolCreate` allocators are supported with appropriate location type and NUMA identifiers.

26.1.4. Memory management extensions to current APIs

Currently, EGM memory can be mapped with Virtual Memory (`cuMemCreate`) or Stream Ordered Memory (`cudaMemPoolCreate`) allocators. The user is responsible for allocating physical memory and mapping it to a virtual memory address space on all sockets.

Note: Multi-node, single-GPU platforms require interprocess communication. Therefore we encourage the reader to see *Chapter 3*

Note: We encourage readers to read CUDA Programming Guide's *Chapter 10* and *Chapter 11* for a better understanding.

New CUDA property types have been added to APIs for allowing those approaches to understand allocation locations using NUMA-like node identifiers:

CUDA Type	Used with
`CU_MEM_LOCATION_TYPE_HOST_NUMA`	`CUmemAllocationProp` for `cuMemCreate`
`cudaMemLocationTypeHostNuma`	`cudaMemPoolProps` for `cudaMemPoolCreate`

Note: Please see CUDA Driver API and CUDA Runtime Data Types to find more about NUMA specific CUDA types.

26.2. Using the EGM Interface

26.2.1. Single-Node, Single-GPU

Any of the existing CUDA host allocators as well as system allocated memory can be used to benefit from high-bandwidth C2C. To the user, local access is what a host allocation is today.

Note: Refer to the tuning guide for more information about memory allocators and page sizes.

26.2.2. Single-Node, Multi-GPU

In a multi-GPU system, the user has to provide host information for the placement. As we mentioned, a natural way to express that information would be by using NUMA node IDs and EGM follows this approach. Therefore, using the `cuDeviceGetAttribute` function the user should be able to learn the closest NUMA node id. (See *Socket Identifiers: What are they? How to access them?*). Then the user can allocate and manage EGM memory using VMM (Virtual Memory Management) API or CUDA Memory Pool.

26.2.2.1 Using VMM APIs

The first step in memory allocation using Virtual Memory Management APIs is to create a physical memory chunk that will provide a backing for the allocation. See CUDA Programming Guide's *Virtual Memory Management section* for more details. In EGM allocations the user has to explicitly provide `CU_MEM_LOCATION_TYPE_HOST_NUMA` as the location type and *numaID* as the location identifier. Also in EGM, allocations must be aligned to appropriate granularity of the platform. The following code snippet shows allocating physical memory with `cuMemCreate`:

```
CUmemAllocationProp prop{};
prop.type = CU_MEM_ALLOCATION_TYPE_PINNED;
prop.location.type = CU_MEM_LOCATION_TYPE_HOST_NUMA;
prop.location.id = numaId;
size_t granularity = 0;
cuMemGetAllocationGranularity(&granularity, &prop, MEM_ALLOC_GRANULARITY_MINIMUM);
size_t padded_size = ROUND_UP(size, granularity);
CUmemGenericAllocationHandle allocHandle;
cuMemCreate(&allocHandle, padded_size, &prop, 0);
```

After physical memory allocation, we have to reserve an address space and map it to a pointer. These procedures do not have EGM-specific changes:

```
CUdeviceptr dptr;
cuMemAddressReserve(&dptr, padded_size, 0, 0, 0);
cuMemMap(dptr, padded_size, 0, allocHandle, 0);
```

Finally, the user has to explicitly protect mapped virtual address ranges. Otherwise access to the mapped space would result in a crash. Similar to the memory allocation, the user has to provide `CU_MEM_LOCATION_TYPE_HOST_NUMA` as the location type and *numaId* as the location identifier. Following code snippet create an access descriptors for the host node and the GPU to give read and write access for the mapped memory to both of them:

```
CUmemAccessDesc accessDesc[2]{{}};
accessDesc[0].location.type = CU_MEM_LOCATION_TYPE_HOST_NUMA;
accessDesc[0].location.id = numaId;
accessDesc[0].flags = CU_MEM_ACCESS_FLAGS_PROT_READWRITE;
accessDesc[1].location.type = CU_MEM_LOCATION_TYPE_DEVICE;
accessDesc[1].location.id = currentDev;
accessDesc[1].flags = CU_MEM_ACCESS_FLAGS_PROT_READWRITE;
cuMemSetAccess(dptr, size, accessDesc, 2);
```

26.2.2.2 Using CUDA Memory Pool

To define EGM, the user can create a memory pool on a node and give access to peers. In this case, the user has to explicitly define `cudaMemLocationTypeHostNuma` as the location type and *numaId* as the location identifier. The following code snippet shows creating a memory pool `cudaMemPoolCreate`:

```
cudaSetDevice(homeDevice);
cudaMemPoolProps props{};
props.allocType = cudaMemAllocationTypePinned;
props.location.type = cudaMemLocationTypeHostNuma;
props.location.id = numaId;
cudaMemPoolCreate(&memPool, &props);
```

Additionally, for direct connect peer access, it is also possible to use the existing peer access API, `cudaMemPoolSetAccess`. An example for an *accessingDevice* is shown in the following code snippet:

```
cudaMemAccessDesc desc{};
desc.flags = cudaMemAccessFlagsProtReadWrite;
desc.location.type = cudaMemLocationTypeDevice;
desc.location.id = accessingDevice;
cudaMemPoolSetAccess(memPool, &desc, 1);
```

When the memory pool is created, and accesses are given, the user can set created memory pool to the *residentDevice* and start allocating memory using `cudaMallocAsync`:

```
cudaDeviceSetMemPool(residentDevice, memPool);
cudaMallocAsync(&ptr, size, memPool, stream);
```

Note: EGM is mapped with 2MB pages. Therefore, users may encounter more TLB misses when accessing very large allocations.

26.2.3. Multi-Node, Single-GPU

Beyond memory allocation, remote peer access does not have EGM-specific modification and it follows CUDA inter process (IPC) protocol. See CUDA Programming Guide for more details in IPC.

The user should allocate memory using `cuMemCreate` and again the user has to explicitly provide `CU_MEM_LOCATION_TYPE_HOST_NUMA` as the location type and *numaID* as the location identifier. In addition `CU_MEM_HANDLE_TYPE_FABRIC` should be defined as the requested handle type. The following code snippet shows allocating physical memory on Node A:

```
CUmemAllocationProp prop{};
prop.type = CU_MEM_ALLOCATION_TYPE_PINNED;
prop.requestedHandleTypes = CU_MEM_HANDLE_TYPE_FABRIC;
prop.location.type = CU_MEM_LOCATION_TYPE_HOST_NUMA;
prop.location.id = numaId;
size_t granularity = 0;
cuMemGetAllocationGranularity(&granularity, &prop,
                              MEM_ALLOC_GRANULARITY_MINIMUM);
size_t padded_size = ROUND_UP(size, granularity);
size_t page_size = ...;
assert(padded_size % page_size == 0);
CUmemGenericAllocationHandle allocHandle;
cuMemCreate(&allocHandle, padded_size, &prop, 0);
```

After creating allocation handle using `cuMemCreate` the user can export that handle to the other node, Node B, calling `cuMemExportToShareableHandle`:

```
cuMemExportToShareableHandle(&fabricHandle, allocHandle,
                             CU_MEM_HANDLE_TYPE_FABRIC, 0);
// At this point, fabricHandle should be sent to Node B via TCP/IP.
```

On Node B, the handle can be imported using `cuMemImportFromShareableHandle` and treated as any other fabric handle

```
// At this point, fabricHandle should be received from Node A via TCP/IP.
CUmemGenericAllocationHandle allocHandle;
cuMemImportFromShareableHandle(&allocHandle, &fabricHandle,
                               CU_MEM_HANDLE_TYPE_FABRIC);
```

When handle is imported at Node B, then the user can reserve an address space and map it locally in a regular fashion:

```
size_t granularity = 0;
cuMemGetAllocationGranularity(&granularity, &prop,
                              MEM_ALLOC_GRANULARITY_MINIMUM);
size_t padded_size = ROUND_UP(size, granularity);
size_t page_size = ...;
assert(padded_size % page_size == 0);
CUdeviceptr dptr;
cuMemAddressReserve(&dptr, padded_size, 0, 0, 0);
cuMemMap(dptr, padded_size, 0, allocHandle, 0);
```

As the final step, the user should give appropriate accesses to each of the local GPUs at Node B. An example code snippet that gives read and write access to eight local GPUs:

```
// Give all 8 local  GPUS access to exported EGM memory located on Node A.
↪                                                    |
CUmemAccessDesc accessDesc[8];
for (int i = 0; i < 8; i++) {
   accessDesc[i].location.type = CU_MEM_LOCATION_TYPE_DEVICE;
   accessDesc[i].location.id = i;
   accessDesc[i].flags = CU_MEM_ACCESS_FLAGS_PROT_READWRITE;
}
cuMemSetAccess(dptr, size, accessDesc, 8);
```

Chapter 27. Notices

27.1. Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation (“NVIDIA”) makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or

services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

27.2. OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

27.3. Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

©2007-2026, NVIDIA Corporation & affiliates. All rights reserved