CUDA Runtime API

API Reference Manual
Table of Contents

Chapter 1. Difference between the driver and runtime APIs...........................................1
Chapter 2. API synchronization behavior............................................................................3
Chapter 3. Stream synchronization behavior..................................................................... 5
Chapter 4. Graph object thread safety................................................................................7
Chapter 5. Rules for version mixing...................................................................................8
Chapter 6. Modules.............................................................................................................. 9
  6.1. Device Management....................................................................................................10
    cudaChooseDevice.............................................................................................................. 10
    cudaDeviceFlushGPUDirectRDMAWrites........................................................................ 11
    cudaDeviceGetAttribute.................................................................................................... 12
    cudaDeviceGetByPCIBusId............................................................................................... 18
    cudaDeviceGetCacheConfig............................................................................................. 19
    cudaDeviceGetDefaultMemPool....................................................................................... 20
    cudaDeviceGetLimit.......................................................................................................... 20
    cudaDeviceGetMemPool..................................................................................................... 22
    cudaDeviceGetNvSciSyncAttributes............................................................................... 22
    cudaDeviceGetP2PAttribute............................................................................................. 24
    cudaDeviceGetPCIBusId.................................................................................................... 25
    cudaDeviceGetSharedMemConfig.................................................................................. 26
    cudaDeviceGetStreamPriorityRange............................................................................. 27
    cudaDeviceGetTexture1DLinearMaxWidth..................................................................... 28
    cudaDeviceReset............................................................................................................ 29
    cudaDeviceSetCacheConfig............................................................................................ 30
    cudaDeviceSetLimit......................................................................................................... 31
    cudaDeviceSetMemPool................................................................................................... 33
    cudaDeviceSetSharedMemConfig.................................................................................. 34
    cudaDeviceSynchronize................................................................................................. 35
    cudaGetDevice............................................................................................................... 36
    cudaGetDeviceCount...................................................................................................... 36
    cudaGetDeviceFlags....................................................................................................... 37
    cudaGetDeviceProperties............................................................................................... 38
    cudaMemcpyCloseMemHandle......................................................................................... 44
    cudaMemcpyGetEventHandle......................................................................................... 45
    cudaMemcpyGetMemHandle............................................................................................ 46

CUDA Runtime API

vRelease Version | ii
cudaIpcOpenEventHandle........................................................................................................... 47
cudaIpcOpenMemHandle........................................................................................................... 48
cudaSetDevice............................................................................................................................. 49
cudaSetDeviceFlags.................................................................................................................... 50
cudaSetValidDevices................................................................................................................... 52

6.2. Thread Management [DEPRECATED].................................................................................... 53
cudaThreadExit........................................................................................................................... 53
cudaThreadGetCacheConfig....................................................................................................... 54
cudaThreadGetLimit................................................................................................................... 55
cudaThreadSetCacheConfig....................................................................................................... 56
cudaThreadSetLimit.................................................................................................................... 57
cudaThreadSynchronize............................................................................................................. 59

6.3. Error Handling......................................................................................................................... 59
cudaGetErrorName..................................................................................................................... 60
cudaGetErrorString..................................................................................................................... 60
cudaGetLastError....................................................................................................................... 61
cudaPeekAtLastError................................................................................................................. 62

6.4. Stream Management.............................................................................................................. 63
cudaStreamCallback_t............................................................................................................... 63
cudaCtxResetPersistingL2Cache............................................................................................... 63
cudaStreamAddCallback............................................................................................................ 63
cudaStreamAttachMemAsync.................................................................................................... 65
cudaStreamBeginCapture.......................................................................................................... 67
cudaStreamCopyAttributes........................................................................................................ 68
cudaStreamCreate...................................................................................................................... 69
cudaStreamCreateWithFlags..................................................................................................... 70
cudaStreamCreateWithPriority................................................................................................. 71
cudaStreamDestroy.................................................................................................................... 72
cudaStreamEndCapture............................................................................................................. 73
cudaStreamGetAttribute............................................................................................................. 74
cudaStreamGetCaptureInfo........................................................................................................ 74
cudaStreamGetCaptureInfo_v2................................................................................................. 75
cudaStreamGetFlags.................................................................................................................. 77
cudaStreamGetPriority............................................................................................................... 78
cudaStreamIsCapturing.............................................................................................................. 79
cudaStreamQuery....................................................................................................................... 80
cudaStreamSetAttribute............................................................................................................. 81
cudaStreamSynchronize............................................................................................................. 81
cudaStreamUpdateCaptureDependencies................................................................................ 82
cudaStreamWaitEvent................................................................................................................ 83
cudaThreadExchangeStreamCaptureMode............................................................................... 84

6.5. Event Management................................................................................................................. 85
cudaEventCreate.........................................................................................................................86
cudaEventCreateWithFlags........................................................................................................ 86
cudaEventDestroy....................................................................................................................... 88
cudaEventElapsedTime............................................................................................................... 89
cudaEventQuery.......................................................................................................................... 90
cudaEventRecord........................................................................................................................ 91
cudaEventRecordWithFlags........................................................................................................ 92
cudaEventSynchronize................................................................................................................93

6.6. External Resource Interoperability........................................................................................ 94
cudaDestroyExternalMemory.....................................................................................................95
cudaDestroyExternalSemaphore............................................................................................... 96
cudaExternalMemoryGetMappedBuffer.................................................................................... 97
cudaExternalMemoryGetMappedMipmappedArray.................................................................. 98
cudalImportExternalMemory.....................................................................................................100
cudalImportExternalSemaphore............................................................................................... 103
cudaSignalExternalSemaphoresAsync.................................................................................... 106
cudaWaitExternalSemaphoresAsync....................................................................................... 108

6.7. Execution Control..................................................................................................................110
cudaFuncGetAttributes.............................................................................................................110
cudaFuncSetAttribute............................................................................................................... 111
cudaFuncSetCacheConfig.........................................................................................................112
cudaFuncSetSharedMemConfig...............................................................................................113
cudaGetParameterBuffer......................................................................................................... 115
cudaGetParameterBufferV2..................................................................................................... 116
cudaLaunchCooperativeKernel................................................................................................ 117
cudaLaunchCooperativeKernelMultiDevice.............................................................................118
cudaLaunchHostFunc..................................................................................................................121
cudaLaunchKernel.................................................................................................................... 122
cudaSetDoubleForDevice......................................................................................................... 124
cudaSetDoubleForHost.............................................................................................................125

6.8. Occupancy..............................................................................................................................125
cudaOccupancyAvailableDynamicSMemPerBlock..................................................................126
cudaOccupancyMaxActiveBlocksPerMultiprocessor..................................................................127
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags.............................................128
6.9. Memory Management

cudaArrayGetInfo ........................................................................................................... 130

cudaArrayGetPlane ........................................................................................................ 131

cudaArrayGetSparseProperties .................................................................................... 132

cudaFree ....................................................................................................................... 132

cudaFreeArray .............................................................................................................. 133

cudaFreeHost ................................................................................................................ 134

cudaFreeMipmappedArray ............................................................................................. 135

cudaGetMipmappedArrayLevel ....................................................................................... 136

cudaGetSymbolAddress ................................................................................................. 137

cudaGetSymbolSize ....................................................................................................... 138

cudaHostAlloc ................................................................................................................ 139

cudaHostGetDevicePointer ............................................................................................ 140

cudaHostGetFlags ......................................................................................................... 142

cudaHostRegister .......................................................................................................... 145

cudaHostUnregister ....................................................................................................... 145

cudaMalloc .................................................................................................................... 145

cudaMalloc3D .................................................................................................................. 146

cudaMalloc3DArray ......................................................................................................... 147

cudaMallocArray ............................................................................................................. 150

cudaMallocHost ............................................................................................................. 152

cudaMallocManaged ....................................................................................................... 153

cudaMallocPitch ............................................................................................................ 156

cudaMallocMipmappedArray ............................................................................................ 156

cudaMemAdvise .............................................................................................................. 159

cudaMemcpy .................................................................................................................... 163

cudaMemcpy2D ................................................................................................................ 164

cudaMemcpy2DArrayToArray ........................................................................................... 166

cudaMemcpy2DAsync ....................................................................................................... 168

cudaMemcpy2DFromArray ............................................................................................... 170

cudaMemcpy2DFromArrayAsync ..................................................................................... 171

cudaMemcpy2DToArray .................................................................................................... 173

cudaMemcpy2DToArrayAsync ......................................................................................... 175

cudaMemcpy3D ................................................................................................................ 176

cudaMemcpy3DAsync ....................................................................................................... 178

cudaMemcpy3DPeer ......................................................................................................... 181

cudaMemcpy3DPeerAsync ............................................................................................... 182

cudaMemcpyAsync .......................................................................................................... 183
cudaMemcpyFromSymbol........................................................................................................ 185
cudaMemcpyFromSymbolAsync.............................................................................................. 186
cudaMemcpyPeer......................................................................................................................187
cudaMemcpyPeerAsync............................................................................................................189
cudaMemcpyToSymbol............................................................................................................. 190
cudaMemcpyToSymbolAsync................................................................................................... 191
cudaMemGetInfo.......................................................................................................................193
cudaMemPrefetchAsync...........................................................................................................194
cudaMemRangeGetAttribute.................................................................................................... 196
cudaMemRangeGetAttributes................................................................................................. 198
cudaMemset..............................................................................................................................199
cudaMemset2D..........................................................................................................................200
cudaMemset2DAsync................................................................................................................201
cudaMemset3D..........................................................................................................................202
cudaMemset3DAsync................................................................................................................204
cudaMemsetAsync.................................................................................................................... 205
cudaMipmappedArrayGetSparseProperties............................................................................ 206
make_cudaExtent......................................................................................................................207
make_cudaPitchedPtr.............................................................................................................. 208
make_cudaPos..........................................................................................................................208
6.10. Memory Management [DEPRECATED] .............................................................................. 209
cudaMemcpyArrayToArray....................................................................................................... 209
cudaMemcpyFromArray........................................................................................................... 211
cudaMemcpyFromArrayAsync..................................................................................................212
cudaMemcpyToArray................................................................................................................ 214
cudaMemcpyToArrayAsync.......................................................................................................215
6.11. Stream Ordered Memory Allocator....................................................................................216
cudaFreeAsync..........................................................................................................................217
cudaMallocAsync...................................................................................................................... 218
cudaMallocFromPoolAsync......................................................................................................219
cudaMemPoolCreate................................................................................................................ 220
cudaMemPoolDestroy...............................................................................................................221
cudaMemPoolExportPointer.................................................................................................... 222
cudaMemPoolExportToShareableHandle.............................................................................. 222
cudaMemPoolGetAccess.......................................................................................................... 223
cudaMemPoolGetAttribute....................................................................................................... 224
cudaMemPoolImportFromShareableHandle.......................................................................... 225
cudaMemPoolImportPointer.................................................................................................... 226
CUDA Runtime API

6.12. Unified Addressing

6.13. Peer Device Memory Access


6.15. OpenGL Interoperability [DEPRECATED]

6.16. Direct3D 9 Interoperability

6.17. Direct3D 9 Interoperability [DEPRECATED]
cudaD3D9ResourceGetMappedPitch....................................................................................... 257
cudaD3D9ResourceGetMappedPointer................................................................................... 258
cudaD3D9ResourceGetMappedSize.........................................................................................260
cudaD3D9ResourceGetSurfaceDimensions............................................................................ 261
cudaD3D9ResourceSetMapFlags.............................................................................................262
cudaD3D9UnmapResources.....................................................................................................263
cudaD3D9UnregisterResource.................................................................................................264

6.18. Direct3D 10 Interoperability..........................................................................................264
cudaD3D10DeviceList............................................................................................................... 264
cudaD3D10GetDevice................................................................................................................265
cudaD3D10GetDevices..............................................................................................................266
cudaGraphicsD3D10RegisterResource....................................................................................267

6.19. Direct3D 10 Interoperability [DEPRECATED].................................................................269
cudaD3D10MapFlags................................................................................................................ 269
cudaD3D10RegisterFlags......................................................................................................... 269
cudaD3D10GetDirect3DDevice................................................................................................. 270
cudaD3D10MapResources........................................................................................................270
cudaD3D10RegisterResource.................................................................................................. 271
cudaD3D10ResourceGetMappedArray.....................................................................................273
cudaD3D10ResourceGetMappedPitch..................................................................................... 274
cudaD3D10ResourceGetMappedPointer................................................................................. 275
cudaD3D10ResourceGetMappedSize.......................................................................................276
cudaD3D10ResourceGetSurfaceDimensions.......................................................................... 277
cudaD3D10ResourceSetMapFlags...........................................................................................278
cudaD3D10SetDirect3DDevice................................................................................................. 279
cudaD3D10UnmapResources...................................................................................................280
cudaD3D10UnregisterResource...............................................................................................281

6.20. Direct3D 11 Interoperability..........................................................................................282
cudaD3D11DeviceList............................................................................................................... 282
cudaD3D11GetDevice................................................................................................................282
cudaD3D11GetDevices..............................................................................................................283
cudaGraphicsD3D11RegisterResource....................................................................................284

6.21. Direct3D 11 Interoperability [DEPRECATED].................................................................286
cudaD3D11GetDirect3DDevice................................................................................................. 286
cudaD3D11SetDirect3DDevice................................................................................................. 287

6.22. VDPAU Interoperability..................................................................................................287
cudaGraphicsVDPAURegisterOutputSurface...........................................................................288
cudaGraphicsVDPAURegisterVideoSurface.............................................................................289
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>cudaVDPAUGetDevice</td>
<td>290</td>
</tr>
<tr>
<td>cudaVDPAUSetVDPAUDevice</td>
<td>290</td>
</tr>
<tr>
<td>cudaEGLStreamConsumerAcquireFrame</td>
<td>291</td>
</tr>
<tr>
<td>cudaEGLStreamConsumerConnect</td>
<td>292</td>
</tr>
<tr>
<td>cudaEGLStreamConsumerConnectWithFlags</td>
<td>293</td>
</tr>
<tr>
<td>cudaEGLStreamConsumerDisconnect</td>
<td>294</td>
</tr>
<tr>
<td>cudaEGLStreamConsumerReleaseFrame</td>
<td>294</td>
</tr>
<tr>
<td>cudaEGLStreamProducerConnect</td>
<td>295</td>
</tr>
<tr>
<td>cudaEGLStreamProducerDisconnect</td>
<td>296</td>
</tr>
<tr>
<td>cudaEGLStreamProducerPresentFrame</td>
<td>296</td>
</tr>
<tr>
<td>cudaEGLStreamProducerReturnFrame</td>
<td>297</td>
</tr>
<tr>
<td>cudaEventCreateFromEGLSync</td>
<td>298</td>
</tr>
<tr>
<td>cudaGraphicsEGLRegisterImage</td>
<td>299</td>
</tr>
<tr>
<td>cudaGraphicsResourceGetMappedEglFrame</td>
<td>300</td>
</tr>
<tr>
<td>cudaGraphicsMapResources</td>
<td>301</td>
</tr>
<tr>
<td>cudaGraphicsResourceGetMappedMipmappedArray</td>
<td>302</td>
</tr>
<tr>
<td>cudaGraphicsResourceGetMappedPointer</td>
<td>303</td>
</tr>
<tr>
<td>cudaGraphicsResourceSetMapFlags</td>
<td>304</td>
</tr>
<tr>
<td>cudaGraphicsSubResourceGetMappedArray</td>
<td>306</td>
</tr>
<tr>
<td>cudaGraphicsUnmapResources</td>
<td>307</td>
</tr>
<tr>
<td>cudaGraphicsUnregisterResource</td>
<td>308</td>
</tr>
<tr>
<td>cudaBindTexture</td>
<td>309</td>
</tr>
<tr>
<td>cudaBindTexture2D</td>
<td>310</td>
</tr>
<tr>
<td>cudaBindTextureToArray</td>
<td>312</td>
</tr>
<tr>
<td>cudaBindTextureToMipmappedArray</td>
<td>313</td>
</tr>
<tr>
<td>cudaGetTextureAlignmentOffset</td>
<td>314</td>
</tr>
<tr>
<td>cudaGetTextureReference</td>
<td>315</td>
</tr>
<tr>
<td>cudaUnbindTexture</td>
<td>316</td>
</tr>
<tr>
<td>cudaBindSurfaceToArray</td>
<td>317</td>
</tr>
<tr>
<td>cudaGetSurfaceReference</td>
<td>318</td>
</tr>
<tr>
<td>cudaCreateChannelDesc</td>
<td>319</td>
</tr>
<tr>
<td>cudaCreateTextureObject</td>
<td>320</td>
</tr>
<tr>
<td>cudaDestroyTextureObject</td>
<td>325</td>
</tr>
</tbody>
</table>
cudaGetChannelDesc.................................................................................................326
cudaGetTextureObjectResourceDesc...........................................................................327
cudaGetTextureObjectResourceViewDesc.....................................................................328
cudaGetTextureObjectTextureDesc..............................................................................329
6.28. Surface Object Management..................................................................................329
   cudaCreateSurfaceObject.........................................................................................330
   cudaDestroySurfaceObject.........................................................................................331
   cudaGetSurfaceObjectResourceDesc.........................................................................332
6.29. Version Management...............................................................................................332
   cudaDriverGetVersion.................................................................................................333
   cudaRuntimeGetVersion...............................................................................................333
6.30. Graph Management..................................................................................................334
   cudaDeviceGetGraphMemAttribute..............................................................................334
   cudaDeviceGraphMemTrim............................................................................................336
   cudaDeviceSetGraphMemAttribute..............................................................................337
   cudaGraphAddChildGraphNode......................................................................................338
   cudaGraphAddDependencies.........................................................................................339
   cudaGraphAddEmptyNode.............................................................................................340
   cudaGraphAddEventRecordNode...................................................................................341
   cudaGraphAddEventWaitNode.......................................................................................343
   cudaGraphAddExternalSemaphoresSignalNode............................................................344
   cudaGraphAddExternalSemaphoresWaitNode..................................................................346
   cudaGraphAddHostNode...............................................................................................347
   cudaGraphAddKernelNode............................................................................................349
   cudaGraphAddMemAllocNode.......................................................................................351
   cudaGraphAddMemcpyNode...........................................................................................353
   cudaGraphAddMemcpyNode1D.......................................................................................354
   cudaGraphAddMemcpyNodeFromSymbol.........................................................................356
   cudaGraphAddMemcpyNodeToSymbol............................................................................358
   cudaGraphAddMemFreeNode..........................................................................................360
   cudaGraphAddMemsetNode............................................................................................361
   cudaGraphChildGraphNodeGetGraph............................................................................362
   cudaGraphClone............................................................................................................363
   cudaGraphCreate..........................................................................................................364
   cudaGraphDebugDotPrint...............................................................................................365
   cudaGraphDestroy........................................................................................................366
   cudaGraphDestroyNode...............................................................................................367
   cudaGraphNodeEventRecordNodeGetEvent....................................................................368
CUDA Runtime API

vRelease Version

xi
cudaGraphMemcpyNodeSetParamsFromSymbol...................................................... 412
cudaGraphMemcpyNodeSetParamsToSymbol.......................................................... 413
cudaGraphMemFreeNodeGetParams..................................................................... 415
cudaGraphMemsetNodeGetParams....................................................................... 416
cudaGraphMemsetNodeSetParams....................................................................... 417
cudaGraphNodeFindInClone.................................................................................. 418
cudaGraphNodeGetDependencies......................................................................... 419
cudaGraphNodeGetDependentNodes..................................................................... 420
cudaGraphNodeGetType....................................................................................... 421
cudaGraphReleaseUserObject.............................................................................. 422
cudaGraphRemoveDependencies......................................................................... 423
cudaGraphRetainUserObject............................................................................... 424
cudaGraphUpload............................................................................................... 425
cudaUserObjectCreate......................................................................................... 425
cudaUserObjectRelease....................................................................................... 426
cudaUserObjectRetain........................................................................................ 427
6.31. Driver Entry Point Access.......................................................................... 428
cudaGetDriverEntryPoint................................................................................... 428
6.32. C++ API Routines.......................................................................................... 429
    _cudaOccupancyB2DHelper............................................................................... 429
cudaBindSurfaceToArray..................................................................................... 430
cudaBindSurfaceToArray..................................................................................... 431
cudaBindTexture................................................................................................ 432
cudaBindTexture................................................................................................ 433
cudaBindTexture2D............................................................................................. 434
cudaBindTexture2D............................................................................................. 436
cudaBindTextureToArray..................................................................................... 437
cudaBindTextureToArray..................................................................................... 438
cudaBindTextureToMipmappedArray.................................................................... 439
cudaBindTextureToMipmappedArray.................................................................... 441
cudaCreateChannelDesc..................................................................................... 442
cudaEventCreate............................................................................................... 442
cudaFuncGetAttributes....................................................................................... 444
cudaFuncSetAttribute......................................................................................... 445
cudaFuncSetCacheConfig.................................................................................... 446
cudaGetSymbolAddress....................................................................................... 447
cudaGetSymbolSize............................................................................................ 448
cudaGetTextureAlignmentOffset........................................................................ 449
cudaGraphAddMemcpyNodeFromSymbol...............................................................................450
cudaGraphAddMemcpyNodeToSymbol...............................................................................452
cudaGraphExecMemcpyNodeSetParamsFromSymbol...........................................................454
cudaGraphExecMemcpyNodeSetParamsToSymbol................................................................455
cudaGraphMemcpyNodeSetParamsFromSymbol...................................................................457
cudaGraphMemcpyNodeSetParamsToSymbol........................................................................458
cudaLaunchCooperativeKernel..........................................................................................459
cudaLaunchKernel.............................................................................................................461
cudaMallocAsync.............................................................................................................462
cudaMallocHost...............................................................................................................462
cudaMallocManaged.........................................................................................................464
cudaMemcpyFromSymbol....................................................................................................467
cudaMemcpyFromSymbolAsync..........................................................................................468
cudaMemcpyToSymbol.......................................................................................................469
cudaMemcpyToSymbolAsync..............................................................................................471
cudaOccupancyAvailableDynamicSMemPerBlock................................................................472
cudaOccupancyMaxActiveBlocksPerMultiprocessor.........................................................473
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags..........................................474
cudaOccupancyMaxPotentialBlockSize............................................................................476
cudaOccupancyMaxPotentialBlockSizeVariableSMem.....................................................477
cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags.......................................479
cudaOccupancyMaxPotentialBlockSizeWithFlags............................................................480
cudaStreamAttachMemAsync.............................................................................................482
cudaUnbindTexture..........................................................................................................484

6.33. Interactions with the CUDA Driver API......................................................................485
cudaGetFuncBySymbol......................................................................................................488

6.34. Profiler Control [DEPRECATED]..............................................................................488
cudaProfilerInitialize......................................................................................................488

6.35. Profiler Control..........................................................................................................489
cudaProfilerStart.............................................................................................................489
cudaProfilerStop..............................................................................................................490

6.36. Data types used by CUDA Runtime.........................................................................490
cudaAccessPolicyWindow.................................................................................................491
cudaArraySparseProperties...............................................................................................491
cudaChannelFormatDesc..................................................................................................491
cudaDeviceProp...............................................................................................................491
cudaEglFrame....................................................................................................................491
cudaEglPlaneDesc.............................................................................................................491
cudaExtent........................................................................................................................................491
cudaExternalMemoryBufferDesc............................................................................................491
cudaExternalMemoryHandleDesc............................................................................................491
cudaExternalMemoryMipmappedArrayDesc...........................................................................491
cudaExternalSemaphoreHandleDesc......................................................................................491
cudaExternalSemaphoreSignalNodeParams..........................................................................491
cudaExternalSemaphoreSignalParams...................................................................................491
cudaExternalSemaphoreWaitNodeParams.............................................................................491
cudaExternalSemaphoreWaitParams.......................................................................................491
cudaFuncAttributes..............................................................................................................491
cudaHostNodeParams............................................................................................................491
cudalpcEventHandle_t............................................................................................................492
cudalpcMemHandle_t...............................................................................................................492
cudaKernelNodeAttrValue........................................................................................................492
cudaKernelNodeParams..........................................................................................................492
cudaLaunchParams..................................................................................................................492
cudaMemAccessDesc...............................................................................................................492
cudaMemAllocNodeParams.....................................................................................................492
cudacopy3DParms....................................................................................................................492
cudacopy3DPeerParms.............................................................................................................492
cudaMemLocation.....................................................................................................................492
cudaMemPoolProps..................................................................................................................492
cudaMemPoolPtrExportData....................................................................................................492
cudaMemsetParams.................................................................................................................492
cudaPitchedPtr..........................................................................................................................492
cudaPointerAttributes............................................................................................................492
cudaPos.....................................................................................................................................492
cudaResourceDesc...................................................................................................................492
cudaResourceViewDesc............................................................................................................492
cudaStreamAttrValue................................................................................................................493
cudaTextureDesc......................................................................................................................493
CUuuid_st..................................................................................................................................493
surfaceReference....................................................................................................................493
textureReference......................................................................................................................493
cudaAccessProperty................................................................................................................493
cudaCGScope............................................................................................................................493
cudaChannelFormatKind..........................................................................................................493
cudaComputeMode...................................................................................................................494
cudaInvalidDeviceId................................................................. 541
cudaIpcMemLazyEnablePeerAccess............................................ 541
cudaMemAttachGlobal............................................................ 541
cudaMemAttachHost............................................................... 542
cudaMemAttachSingle............................................................ 542
cudaNvSciSyncAttrSignal....................................................... 542
cudaNvSciSyncAttrWait........................................................ 542
cudaOccupancyDefault.......................................................... 542
cudaOccupancyDisableCachingOverride.................................... 542
cudaPeerAccessDefault.......................................................... 542
cudaStreamDefault............................................................... 542
cudaStreamLegacy............................................................... 542
cudaStreamNonBlocking......................................................... 543
cudaStreamPerThread............................................................ 543

Chapter 7. Data Structures.................................................. 544
__cudaOccupancyB2DHelper................................................... 545
cudaAccessPolicyWindow...................................................... 545
  base_ptr........................................................................... 545
  hitProp........................................................................... 545
  hitRatio........................................................................... 545
  missProp........................................................................ 546
  num_bytes...................................................................... 546
cudaArraySparseProperties.................................................. 546
  depth............................................................................. 546
  flags............................................................................. 546
  height............................................................................ 546
  miptailFirstLevel............................................................ 546
  miptailSize.................................................................. 546
  width............................................................................ 546
cudaChannelFormatDesc....................................................... 547
  f.................................................................................. 547
  w................................................................................. 547
  x................................................................................ 547
  y................................................................................ 547
  z................................................................................ 547
cudaDeviceProp................................................................. 547
  accessPolicyMaxWindowSize........................................... 547
  asyncEngineCount......................................................... 547
canMapHostMemory.................................................................................................................548
canUseHostPointerForRegisteredMem...................................................................................548
clockRate...................................................................................................................................548
computeMode............................................................................................................................548
computePreemptionSupported................................................................................................548
concurrentKernels....................................................................................................................548
concurrentManagedAccess......................................................................................................548
cooperativeLaunch....................................................................................................................548
cooperativeMultiDeviceLaunch...............................................................................................548
deviceOverlap............................................................................................................................548
directManagedMemAccessFromHost........................................................................................549
ECCEnabled...............................................................................................................................549
globalL1CacheSupported.........................................................................................................549
hostNativeAtomicSupported.....................................................................................................549
integrated..................................................................................................................................549
isMultiGpuBoard.......................................................................................................................549
kernelExecTimeoutEnabled......................................................................................................549
l2CacheSize...............................................................................................................................549
localL1CacheSupported............................................................................................................549
luid.............................................................................................................................................549
luidDeviceNodeMask................................................................................................................550
major..........................................................................................................................................550
managedMemory......................................................................................................................550
maxBlocksPerMultiProcessor..................................................................................................550
maxGridSize..............................................................................................................................550
maxSurface1D...........................................................................................................................550
maxSurface1DLayered..............................................................................................................550
maxSurface2D...........................................................................................................................550
maxSurface2DLayered..............................................................................................................550
maxSurface3D...........................................................................................................................550
maxSurfaceCubemap................................................................................................................550
maxSurfaceCubemapLayered..................................................................................................551
maxTexture1D...........................................................................................................................551
maxTexture1DLayered..............................................................................................................551
maxTexture1DLinear................................................................................................................551
maxTexture1DMipmap..............................................................................................................551
maxTexture2D...........................................................................................................................551
maxTexture2D...........................................................................................................................551
maxTexture2DGather................................................................................................................551
<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>uuid</td>
<td>555</td>
</tr>
<tr>
<td>warpSize</td>
<td>555</td>
</tr>
<tr>
<td>cudaEglFrame</td>
<td>555</td>
</tr>
<tr>
<td>eglColorFormat</td>
<td>555</td>
</tr>
<tr>
<td>frameType</td>
<td>556</td>
</tr>
<tr>
<td>pArray</td>
<td>556</td>
</tr>
<tr>
<td>planeCount</td>
<td>556</td>
</tr>
<tr>
<td>planeDesc</td>
<td>556</td>
</tr>
<tr>
<td>pPitch</td>
<td>556</td>
</tr>
<tr>
<td>cudaEglPlaneDesc</td>
<td>556</td>
</tr>
<tr>
<td>channelDesc</td>
<td>556</td>
</tr>
<tr>
<td>depth</td>
<td>556</td>
</tr>
<tr>
<td>height</td>
<td>556</td>
</tr>
<tr>
<td>numChannels</td>
<td>556</td>
</tr>
<tr>
<td>pitch</td>
<td>557</td>
</tr>
<tr>
<td>reserved</td>
<td>557</td>
</tr>
<tr>
<td>width</td>
<td>557</td>
</tr>
<tr>
<td>cudaExtent</td>
<td>557</td>
</tr>
<tr>
<td>depth</td>
<td>557</td>
</tr>
<tr>
<td>height</td>
<td>557</td>
</tr>
<tr>
<td>width</td>
<td>557</td>
</tr>
<tr>
<td>cudaExternalMemoryBufferDesc</td>
<td>557</td>
</tr>
<tr>
<td>flags</td>
<td>557</td>
</tr>
<tr>
<td>offset</td>
<td>558</td>
</tr>
<tr>
<td>size</td>
<td>558</td>
</tr>
<tr>
<td>cudaExternalMemoryHandleDesc</td>
<td>558</td>
</tr>
<tr>
<td>fd</td>
<td>558</td>
</tr>
<tr>
<td>flags</td>
<td>558</td>
</tr>
<tr>
<td>handle</td>
<td>558</td>
</tr>
<tr>
<td>name</td>
<td>558</td>
</tr>
<tr>
<td>nvSciBufObject</td>
<td>558</td>
</tr>
<tr>
<td>size</td>
<td>559</td>
</tr>
<tr>
<td>type</td>
<td>559</td>
</tr>
<tr>
<td>win32</td>
<td>559</td>
</tr>
<tr>
<td>cudaExternalMemoryMipmappedArrayDesc</td>
<td>559</td>
</tr>
<tr>
<td>extent</td>
<td>559</td>
</tr>
<tr>
<td>flags</td>
<td>560</td>
</tr>
<tr>
<td>formatDesc</td>
<td>560</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>numLevels</td>
<td>560</td>
</tr>
<tr>
<td>offset</td>
<td>560</td>
</tr>
<tr>
<td>cudaExternalSemaphoreHandleDesc</td>
<td>560</td>
</tr>
<tr>
<td>fd</td>
<td>560</td>
</tr>
<tr>
<td>flags</td>
<td>560</td>
</tr>
<tr>
<td>handle</td>
<td>561</td>
</tr>
<tr>
<td>name</td>
<td>561</td>
</tr>
<tr>
<td>nvSciSyncObj</td>
<td>561</td>
</tr>
<tr>
<td>type</td>
<td>561</td>
</tr>
<tr>
<td>win32</td>
<td>561</td>
</tr>
<tr>
<td>cudaExternalSemaphoreSignalNodeParams</td>
<td>562</td>
</tr>
<tr>
<td>extSemArray</td>
<td>562</td>
</tr>
<tr>
<td>numExtSems</td>
<td>562</td>
</tr>
<tr>
<td>paramsArray</td>
<td>562</td>
</tr>
<tr>
<td>cudaExternalSemaphoreSignalParams</td>
<td>562</td>
</tr>
<tr>
<td>fence</td>
<td>562</td>
</tr>
<tr>
<td>flags</td>
<td>562</td>
</tr>
<tr>
<td>keyedMutex</td>
<td>563</td>
</tr>
<tr>
<td>value</td>
<td>563</td>
</tr>
<tr>
<td>cudaExternalSemaphoreWaitNodeParams</td>
<td>563</td>
</tr>
<tr>
<td>extSemArray</td>
<td>563</td>
</tr>
<tr>
<td>numExtSems</td>
<td>563</td>
</tr>
<tr>
<td>paramsArray</td>
<td>564</td>
</tr>
<tr>
<td>cudaExternalSemaphoreWaitParams</td>
<td>564</td>
</tr>
<tr>
<td>fence</td>
<td>564</td>
</tr>
<tr>
<td>flags</td>
<td>564</td>
</tr>
<tr>
<td>key</td>
<td>564</td>
</tr>
<tr>
<td>keyedMutex</td>
<td>565</td>
</tr>
<tr>
<td>timeoutMs</td>
<td>565</td>
</tr>
<tr>
<td>value</td>
<td>565</td>
</tr>
<tr>
<td>cudaFuncAttributes</td>
<td>565</td>
</tr>
<tr>
<td>binaryVersion</td>
<td>565</td>
</tr>
<tr>
<td>cacheModeCA</td>
<td>565</td>
</tr>
<tr>
<td>constSizeBytes</td>
<td>565</td>
</tr>
<tr>
<td>localSizeBytes</td>
<td>565</td>
</tr>
<tr>
<td>maxDynamicSharedSizeBytes</td>
<td>566</td>
</tr>
</tbody>
</table>
maxThreadsPerBlock................................................................. 566
numRegs......................................................................................... 566
preferredShmemCarveout.......................................................... 566
ptxVersion................................................................................. 566
sharedSizeBytes.................................................................... 566
cudaHostNodeParams............................................................... 566
   fn......................................................................................... 566
   userData............................................................................. 567
cudalpcEventHandle_t.............................................................. 567
cudalpcMemHandle_t................................................................. 567
cudaKernelNodeAttrValue......................................................... 567
   accessPolicyWindow.......................................................... 567
cudaKernelNodeParams............................................................ 567
   blockDim............................................................................ 567
   extra................................................................................. 568
   func............................................................................... 568
   gridDim............................................................................. 568
   kernelParams..................................................................... 568
   sharedMemBytes................................................................. 568
cudaLaunchParams................................................................. 568
   args................................................................................. 568
   blockDim............................................................................ 568
   func............................................................................... 568
   gridDim............................................................................. 568
   sharedMem........................................................................ 569
   stream............................................................................ 569
cudaMemAccessDesc............................................................... 569
   flags............................................................................... 569
   location............................................................................ 569
cudaMemAllocNodeParams....................................................... 569
   accessDescCount............................................................... 569
   accessDescs.................................................................... 569
   bytesize........................................................................... 570
   dptr............................................................................... 570
   poolProps......................................................................... 570
cudaMemcpy3DParms................................................................. 570
   dstArray.......................................................................... 570
   dstPos............................................................................. 570
dstPtr......................................................................................................................................... 570
extent......................................................................................................................................... 570
kind............................................................................................................................................ 570
srcArray..................................................................................................................................... 571
srcPos........................................................................................................................................571
srcPtr.........................................................................................................................................571
cudaMemcpy3DPeerParms.............................................................................................................571
dstArray..................................................................................................................................... 571
dstDevice................................................................................................................................... 571
dstPos........................................................................................................................................571
dstPtr......................................................................................................................................... 571
extent......................................................................................................................................... 571
srcArray..................................................................................................................................... 571
srcDevice................................................................................................................................... 572
srcPos........................................................................................................................................572
srcPtr.........................................................................................................................................572
cudaMemLocation........................................................................................................................ 572
id................................................................................................................................................ 572
type............................................................................................................................................ 572
cudaMemPoolProps..................................................................................................................... 572
allocType....................................................................................................................................572
handleTypes.............................................................................................................................. 572
location...................................................................................................................................... 573
reserved.....................................................................................................................................573
win32SecurityAttributes............................................................................................................573
cudaMemPoolPtrExportData....................................................................................................... 573
cudaMemsetParams.................................................................................................................... 573
dst.............................................................................................................................................. 573
eleme...
Chapter 8. Data Fields.................................................................................................................. 582
Chapter 9. Deprecated List........................................................................................................... 595
Chapter 1. Difference between the driver and runtime APIs

The driver and runtime APIs are very similar and can for the most part be used interchangeably. However, there are some key differences worth noting between the two.

**Complexity vs. control**

The runtime API eases device code management by providing implicit initialization, context management, and module management. This leads to simpler code, but it also lacks the level of control that the driver API has.

In comparison, the driver API offers more fine-grained control, especially over contexts and module loading. Kernel launches are much more complex to implement, as the execution configuration and kernel parameters must be specified with explicit function calls. However, unlike the runtime, where all the kernels are automatically loaded during initialization and stay loaded for as long as the program runs, with the driver API it is possible to only keep the modules that are currently needed loaded, or even dynamically reload modules. The driver API is also language-independent as it only deals with cubin objects.

**Context management**

Context management can be done through the driver API, but is not exposed in the runtime API. Instead, the runtime API decides itself which context to use for a thread: if a context has been made current to the calling thread through the driver API, the runtime will use that, but if there is no such context, it uses a “primary context.” Primary contexts are created as needed, one per device per process, are reference-counted, and are then destroyed when there are no more references to them. Within one process, all users of the runtime API will share the primary context, unless a context has been made current to each thread. The context that the runtime uses, i.e., either the current context or primary context, can be synchronized with `cudaDeviceSynchronize()`, and destroyed with `cudaDeviceReset()`.

Using the runtime API with primary contexts has its tradeoffs, however. It can cause trouble for users writing plug-ins for larger software packages, for example, because if all plug-ins run in the same process, they will all share a context but will likely have no way to communicate with each other. So, if one of them calls `cudaDeviceReset()` after finishing all its CUDA work, the other plug-ins will fail because the context they were using was destroyed.
without their knowledge. To avoid this issue, CUDA clients can use the driver API to create and set the current context, and then use the runtime API to work with it. However, contexts may consume significant resources, such as device memory, extra host threads, and performance costs of context switching on the device. This runtime-driver context sharing is important when using the driver API in conjunction with libraries built on the runtime API, such as cuBLAS or cuFFT.
Chapter 2. API synchronization behavior

The API provides memcpy/memset functions in both synchronous and asynchronous forms, the latter having an “Async” suffix. This is a misnomer as each function may exhibit synchronous or asynchronous behavior depending on the arguments passed to the function.

Memcpy

In the reference documentation, each memcpy function is categorized as synchronous or asynchronous, corresponding to the definitions below.

Synchronous

1. All transfers involving Unified Memory regions are fully synchronous with respect to the host.

2. For transfers from pageable host memory to device memory, a stream sync is performed before the copy is initiated. The function will return once the pageable buffer has been copied to the staging memory for DMA transfer to device memory, but the DMA to final destination may not have completed.

3. For transfers from pinned host memory to device memory, the function is synchronous with respect to the host.

4. For transfers from device to either pageable or pinned host memory, the function returns only once the copy has completed.

5. For transfers from device memory to device memory, no host-side synchronization is performed.

6. For transfers from any host memory to any host memory, the function is fully synchronous with respect to the host.

Asynchronous

1. For transfers from device memory to pageable host memory, the function will return only once the copy has completed.
2. For transfers from any host memory to any host memory, the function is fully synchronous with respect to the host.

3. For all other transfers, the function is fully asynchronous. If pageable memory must first be staged to pinned memory, this will be handled asynchronously with a worker thread.

Memset

The synchronous memset functions are asynchronous with respect to the host except when the target is pinned host memory or a Unified Memory region, in which case they are fully synchronous. The Async versions are always asynchronous with respect to the host.

Kernel Launches

Kernel launches are asynchronous with respect to the host. Details of concurrent kernel execution and data transfers can be found in the CUDA Programmers Guide.
Chapter 3.  Stream synchronization behavior

Default stream

The default stream, used when 0 is passed as a cudaStream_t or by APIs that operate on a stream implicitly, can be configured to have either legacy or per-thread synchronization behavior as described below.

The behavior can be controlled per compilation unit with the --default-stream nvcc option. Alternatively, per-thread behavior can be enabled by defining the CUDA_API_PER_THREAD_DEFAULT_STREAM macro before including any CUDA headers. Either way, the CUDA_API_PER_THREAD_DEFAULT_STREAM macro will be defined in compilation units using per-thread synchronization behavior.

Legacy default stream

The legacy default stream is an implicit stream which synchronizes with all other streams in the same CUcontext except for non-blocking streams, described below. (For applications using the runtime APIs only, there will be one context per device.) When an action is taken in the legacy stream such as a kernel launch or cudaStreamWaitEvent(), the legacy stream first waits on all blocking streams, the action is queued in the legacy stream, and then all blocking streams wait on the legacy stream.

For example, the following code launches a kernel k_1 in stream s, then k_2 in the legacy stream, then k_3 in stream s:

```c
k_1<<<1, 1, 0, s>>>();
k_2<<<1, 1>>>();
k_3<<<1, 1, 0, s>>>();
```

The resulting behavior is that k_2 will block on k_1 and k_3 will block on k_2.

Non-blocking streams which do not synchronize with the legacy stream can be created using the cudaStreamNonBlocking flag with the stream creation APIs.

The legacy default stream can be used explicitly with the CUSstream (cudaStream_t) handle CU_STREAM_LEGACY (cudaStreamLegacy).
Per-thread default stream

The per-thread default stream is an implicit stream local to both the thread and the 
CUcontext, and which does not synchronize with other streams (just like explicitly created 
streams). The per-thread default stream is not a non-blocking stream and will synchronize 
with the legacy default stream if both are used in a program.

The per-thread default stream can be used explicitly with the CUSstream [cudaStream_t] 
handle CU_STREAM_PER_THREAD [cudaStreamPerThread].
Graph objects (cudaGraph_t, CUgraph) are not internally synchronized and must not be accessed concurrently from multiple threads. API calls accessing the same graph object must be serialized externally.

Note that this includes APIs which may appear to be read-only, such as cudaGraphClone() (cuGraphClone()) and cudaGraphInstantiate() (cuGraphInstantiate()). No API or pair of APIs is guaranteed to be safe to call on the same graph object from two different threads without serialization.
Chapter 5. Rules for version mixing

1. Starting with CUDA 11.0, the ABI version for the CUDA runtime is bumped every major release. CUDA-defined types, whether opaque handles or structures like cudaDeviceProp, have their ABI tied to the major release of the CUDA runtime. It is unsafe to pass them from function A to function B if those functions have been compiled with different major versions of the toolkit and linked together into the same device executable.

2. The CUDA Driver API has a per-function ABI denoted with a _v* extension. CUDA-defined types (e.g. structs) should not be passed across different ABI versions. For example, an application calling cuMemcpy2D_v2(const CUDA_MEMCPY2D_v2 *pCopy) and using the older version of the struct CUDA_MEMCPY2D_v1 instead of CUDA_MEMCPY2D_v2.

3. Users should not arbitrarily mix different API versions during the lifetime of a resource. These resources include IPC handles, memory, streams, contexts, events, etc. For example, a user who wants to allocate CUDA memory using cuMemAlloc_v2 should free the memory using cuMemFree_v2 and not cuMemFree.
Chapter 6. Modules

Here is a list of all modules:

- Device Management
- Thread Management [DEPRECATED]
- Error Handling
- Stream Management
- Event Management
- External Resource Interoperability
- Execution Control
- Occupancy
- Memory Management
- Memory Management [DEPRECATED]
- Stream Ordered Memory Allocator
- Unified Addressing
- Peer Device Memory Access
- OpenGL Interoperability
- OpenGL Interoperability [DEPRECATED]
- Direct3D 9 Interoperability
- Direct3D 9 Interoperability [DEPRECATED]
- Direct3D 10 Interoperability
- Direct3D 10 Interoperability [DEPRECATED]
- Direct3D 11 Interoperability
- Direct3D 11 Interoperability [DEPRECATED]
- VDPAU Interoperability
6.1. Device Management

This section describes the device management functions of the CUDA runtime application programming interface.

```c
__host__cudaError_t cudaChooseDevice (int *device, const cudaDeviceProp *prop)
```

Select compute-device which best matches criteria.

**Parameters**

- **device**
  - Device with best match

- **prop**
  - Desired device properties

**Returns**

- cudaSuccess
- cudaErrorInvalidValue
Description

Returns in *device the device which has properties that best match *prop.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaGetDeviceCount, cudaGetDevice, cudaSetDevice, cudaGetDeviceProperties

__host__cudaError_t
cudaDeviceFlushGPUDirectRDMAWrites (cudaFlushGPUDirectRDMAWritesTarget target, cudaFlushGPUDirectRDMAWritesScope scope)

Blocks until remote writes are visible to the specified scope.

Parameters

target
  - The target of the operation, see cudaFlushGPUDirectRDMAWritesTarget

scope
  - The scope of the operation, see cudaFlushGPUDirectRDMAWritesScope

Returns

cudaSuccess, cudaErrorNotSupported.

Description

Blocks until remote writes to the target context via mappings created through GPUDirect RDMA APIs, like nvidia_p2p_get_pages (see https://docs.nvidia.com/cuda/gpudirect-rdma for more information), are visible to the specified scope.
If the scope equals or lies within the scope indicated by `cudaDevAttrGPUDirectRDMAResOrdering`, the call will be a no-op and can be safely omitted for performance. This can be determined by comparing the numerical values between the two enums, with smaller scopes having smaller values.

Users may query support for this API via `cudaDevAttrGPUDirectRDMAPushWritesOptions`.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

`cuFlushGPUDirectRDMAPushes`

```c
__host__ __device__ cudaError_t
cudaDeviceGetAttribute (int *value, cudaDeviceAttr attr, int device)
```

Returns information about the device.

**Parameters**

- **value**
  - Returned device attribute value
- **attr**
  - Device attribute to query
- **device**
  - Device number to query

**Returns**

`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorInvalidValue`

**Description**

Returns in *value the integer value of the attribute `attr` on device `device`. The supported attributes are:
- `cudaDevAttrMaxThreadsPerBlock`: Maximum number of threads per block;
- `cudaDevAttrMaxBlockDimX`: Maximum x-dimension of a block;
- `cudaDevAttrMaxBlockDimY`: Maximum y-dimension of a block;
- `cudaDevAttrMaxBlockDimZ`: Maximum z-dimension of a block;
- `cudaDevAttrMaxGridDimX`: Maximum x-dimension of a grid;
- `cudaDevAttrMaxGridDimY`: Maximum y-dimension of a grid;
- `cudaDevAttrMaxGridDimZ`: Maximum z-dimension of a grid;
- `cudaDevAttrMaxSharedMemoryPerBlock`: Maximum amount of shared memory available to a thread block in bytes;
- `cudaDevAttrTotalConstantMemory`: Memory available on device for __constant__ variables in a CUDA C kernel in bytes;
- `cudaDevAttrWarpSize`: Warp size in threads;
- `cudaDevAttrMaxPitch`: Maximum pitch in bytes allowed by the memory copy functions that involve memory regions allocated through `cudaMallocPitch()`;
- `cudaDevAttrMaxTexture1DWidth`: Maximum 1D texture width;
- `cudaDevAttrMaxTexture1DLinearWidth`: Maximum width for a 1D texture bound to linear memory;
- `cudaDevAttrMaxTexture1DMipmappedWidth`: Maximum mipmapped 1D texture width;
- `cudaDevAttrMaxTexture2DWidth`: Maximum 2D texture width;
- `cudaDevAttrMaxTexture2DHeight`: Maximum 2D texture height;
- `cudaDevAttrMaxTexture2DLinearWidth`: Maximum width for a 2D texture bound to linear memory;
- `cudaDevAttrMaxTexture2DLinearHeight`: Maximum height for a 2D texture bound to linear memory;
- `cudaDevAttrMaxTexture2DLinearPitch`: Maximum pitch in bytes for a 2D texture bound to linear memory;
- `cudaDevAttrMaxTexture2DMipmappedWidth`: Maximum mipmapped 2D texture width;
- `cudaDevAttrMaxTexture2DMipmappedHeight`: Maximum mipmapped 2D texture height;
- `cudaDevAttrMaxTexture3DWidth`: Maximum 3D texture width;
- `cudaDevAttrMaxTexture3DHeight`: Maximum 3D texture height;
- `cudaDevAttrMaxTexture3DDepth`: Maximum 3D texture depth;
- `cudaDevAttrMaxTexture3DWidthAlt`: Alternate maximum 3D texture width, 0 if no alternate maximum 3D texture size is supported;
- `cudaDevAttrMaxTexture3DHeightAlt`: Alternate maximum 3D texture height, 0 if no alternate maximum 3D texture size is supported;
- `cudaDevAttrMaxTexture3DDepthAlt`: Alternate maximum 3D texture depth, 0 if no alternate maximum 3D texture size is supported;
- `cudaDevAttrMaxTextureCubemapWidth`: Maximum cubemap texture width or height;
- `cudaDevAttrMaxTexture1DLayeredWidth`: Maximum 1D layered texture width;
- `cudaDevAttrMaxTexture1DLayeredLayers`: Maximum layers in a 1D layered texture;
- `cudaDevAttrMaxTexture2DLayeredWidth`: Maximum 2D layered texture width;
- `cudaDevAttrMaxTexture2DLayeredHeight`: Maximum 2D layered texture height;
- `cudaDevAttrMaxTexture2DLayeredLayers`: Maximum layers in a 2D layered texture;
- `cudaDevAttrMaxTextureCubemapLayeredWidth`: Maximum cubemap layered texture width or height;
- `cudaDevAttrMaxTextureCubemapLayeredLayers`: Maximum layers in a cubemap layered texture;
- `cudaDevAttrMaxSurface1DWidth`: Maximum 1D surface width;
- `cudaDevAttrMaxSurface2DWidth`: Maximum 2D surface width;
- `cudaDevAttrMaxSurface2DHeight`: Maximum 2D surface height;
- `cudaDevAttrMaxSurface3DWidth`: Maximum 3D surface width;
- `cudaDevAttrMaxSurface3DHeight`: Maximum 3D surface height;
- `cudaDevAttrMaxSurface3DDepth`: Maximum 3D surface depth;
- `cudaDevAttrMaxSurface1DLayeredWidth`: Maximum 1D layered surface width;
- `cudaDevAttrMaxSurface1DLayeredLayers`: Maximum layers in a 1D layered surface;
- `cudaDevAttrMaxSurface2DLayeredWidth`: Maximum 2D layered surface width;
- `cudaDevAttrMaxSurface2DLayeredHeight`: Maximum 2D layered surface height;
- `cudaDevAttrMaxSurface2DLayeredLayers`: Maximum layers in a 2D layered surface;
- `cudaDevAttrMaxSurfaceCubemapWidth`: Maximum cubemap surface width;
- `cudaDevAttrMaxSurfaceCubemapLayeredWidth`: Maximum cubemap layered surface width;
- `cudaDevAttrMaxSurfaceCubemapLayeredLayers`: Maximum layers in a cubemap layered surface;
- `cudaDevAttrMaxRegistersPerBlock`: Maximum number of 32-bit registers available to a thread block;
- `cudaDevAttrClockRate`: Peak clock frequency in kilohertz;
- `cudaDevAttrTextureAlignment`: Alignment requirement; texture base addresses aligned to `textureAlign` bytes do not need an offset applied to texture fetches;
- `cudaDevAttrTexturePitchAlignment`: Pitch alignment requirement for 2D texture references bound to pitched memory;
- `cudaDevAttrGpuOverlap`: 1 if the device can concurrently copy memory between host and device while executing a kernel, or 0 if not;
- `cudaDevAttrMultiProcessorCount`: Number of multiprocessors on the device;
- `cudaDevAttrKernelExecTimeout`: 1 if there is a run time limit for kernels executed on the device, or 0 if not;
- `cudaDevAttrIntegrated`: 1 if the device is integrated with the memory subsystem, or 0 if not;
- `cudaDevAttrCanMapHostMemory`: 1 if the device can map host memory into the CUDA address space, or 0 if not;
- `cudaDevAttrComputeMode`: Compute mode is the compute mode that the device is currently in. Available modes are as follows:
  - `cudaComputeModeDefault`: Default mode - Device is not restricted and multiple threads can use `cudaSetDevice()` with this device.
  - `cudaComputeModeExclusive`: Compute-exclusive mode - Only one thread will be able to use `cudaSetDevice()` with this device.
  - `cudaComputeModeProhibited`: Compute-prohibited mode - No threads can use `cudaSetDevice()` with this device.
  - `cudaComputeModeExclusiveProcess`: Compute-exclusive-process mode - Many threads in one process will be able to use `cudaSetDevice()` with this device.
- `cudaDevAttrConcurrentKernels`: 1 if the device supports executing multiple kernels within the same context simultaneously, or 0 if not. It is not guaranteed that multiple kernels will be resident on the device concurrently so this feature should not be relied upon for correctness;
- `cudaDevAttrEccEnabled`: 1 if error correction is enabled on the device, 0 if error correction is disabled or not supported by the device;
- `cudaDevAttrPciBusId`: PCI bus identifier of the device;
- `cudaDevAttrPciDeviceId`: PCI device (also known as slot) identifier of the device;
- `cudaDevAttrTccDriver`: 1 if the device is using a TCC driver. TCC is only available on Tesla hardware running Windows Vista or later;
- `cudaDevAttrMemoryClockRate`: Peak memory clock frequency in kilohertz;
- `cudaDevAttrGlobalMemoryBusWidth`: Global memory bus width in bits;
- `cudaDevAttrL2CacheSize`: Size of L2 cache in bytes. 0 if the device doesn’t have L2 cache;
- `cudaDevAttrMaxThreadsPerMultiProcessor`: Maximum resident threads per multiprocessor;
- `cudaDevAttrUnifiedAddressing`: 1 if the device shares a unified address space with the host, or 0 if not;
- `cudaDevAttrComputeCapabilityMajor`: Major compute capability version number;
- `cudaDevAttrComputeCapabilityMinor`: Minor compute capability version number;
- `cudaDevAttrStreamPrioritiesSupported`: 1 if the device supports stream priorities, or 0 if not;
- `cudaDevAttrGlobalL1CacheSupported`: 1 if device supports caching globals in L1 cache, 0 if not;
- `cudaDevAttrLocalL1CacheSupported`: 1 if device supports caching locals in L1 cache, 0 if not;
- `cudaDevAttrMaxSharedMemoryPerMultiprocessor`: Maximum amount of shared memory available to a multiprocessor in bytes; this amount is shared by all thread blocks simultaneously resident on a multiprocessor;
- `cudaDevAttrMaxRegistersPerMultiprocessor`: Maximum number of 32-bit registers available to a multiprocessor; this number is shared by all thread blocks simultaneously resident on a multiprocessor;
- `cudaDevAttrManagedMemory`: 1 if device supports allocating managed memory, 0 if not;
- `cudaDevAttrIsMultiGpuBoard`: 1 if device is on a multi-GPU board, 0 if not;
- `cudaDevAttrMultiGpuBoardGroupId`: Unique identifier for a group of devices on the same multi-GPU board;
- `cudaDevAttrHostNativeAtomicSupported`: 1 if the link between the device and the host supports native atomic operations;
- `cudaDevAttrSingleToDoublePrecisionPerfRatio`: Ratio of single precision performance [in floating-point operations per second] to double precision performance;
- `cudaDevAttrPageableMemoryAccess`: 1 if the device supports coherently accessing pageable memory without calling cudaHostRegister on it, and 0 otherwise.
- `cudaDevAttrConcurrentManagedAccess`: 1 if the device can coherently access managed memory concurrently with the CPU, and 0 otherwise.
- `cudaDevAttrComputePreemptionSupported`: 1 if the device supports Compute Preemption, 0 if not.
- `cudaDevAttrCanUseHostPointerForRegisteredMem`: 1 if the device can access host registered memory at the same virtual address as the CPU, and 0 otherwise.
- `cudaDevAttrCooperativeLaunch`: 1 if the device supports launching cooperative kernels via `cudaLaunchCooperativeKernel`, and 0 otherwise.
- `cudaDevAttrCooperativeMultiDeviceLaunch`: 1 if the device supports launching cooperative kernels via `cudaLaunchCooperativeKernelMultiDevice`, and 0 otherwise.
- `cudaDevAttrCanFlushRemoteWrites`: 1 if the device supports flushing of outstanding remote writes, and 0 otherwise.
- `cudaDevAttrHostRegisterSupported`: 1 if the device supports host memory registration via `cudaHostRegister`, and 0 otherwise.
- `cudaDevAttrPageableMemoryAccessUsesHostPageTables`: 1 if the device accesses pageable memory via the host’s page tables, and 0 otherwise.
- `cudaDevAttrDirectManagedMemAccessFromHost`: 1 if the host can directly access managed memory on the device without migration, and 0 otherwise.
- `cudaDevAttrMaxSharedMemoryPerBlockOptin`: Maximum per block shared memory size on the device. This value can be opted into when using `cudaFuncSetAttribute`.
- `cudaDevAttrMaxBlocksPerMultiprocessor`: Maximum number of thread blocks that can reside on a multiprocessor.
- `cudaDevAttrMaxPersistingL2CacheSize`: Maximum L2 persisting lines capacity setting in bytes.
- `cudaDevAttrHostRegisterReadOnly`: Device supports using the `cudaHostRegister` flag `cudaHostRegisterReadOnly` to register memory that must be mapped as read-only to the GPU
- `cudaDevAttrSparseCudaArraySupported`: 1 if the device supports sparse CUDA arrays and sparse CUDA mipmapped arrays.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
`cudaGetDeviceCount`, `cudaGetDevice`, `cudaSetDevice`, `cudaChooseDevice`, `cudaGetDeviceProperties`, `cuDeviceGetAttribute`

```c
__host__cudaError_t cudaDeviceGetByPCIBusId (int *device, const char *pciBusId)
```

Returns a handle to a compute device.

**Parameters**

`device`
- Returned device ordinal

`pciBusId`
- String in one of the following forms: `[domain]`:`[bus]`:`[device]`:`[function]`
  - `[domain]`:`[bus]`:`[device]`
  - `[bus]`:`[device]`:`[function]`
  where `domain`, `bus`, `device`, and `function` are all hexadecimal values

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`

**Description**

Returns in `*device` a device ordinal given a PCI bus ID string.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
`cudaDeviceGetPCIBusId`, `cuDeviceGetByPCIBusId`
__host__ __device__ cudaError_t
cudaDeviceGetCacheConfig (cudaFuncCache *pCacheConfig)

Returns the preferred cache configuration for the current device.

Parameters

pCacheConfig
- Returned cache configuration

Returns

cudaSuccess

Description

On devices where the L1 cache and shared memory use the same hardware resources, this returns through pCacheConfig the preferred cache configuration for the current device. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute functions.

This will return a pCacheConfig of cudaFuncCachePreferNone on devices where the size of the L1 cache and shared memory are fixed.

The supported cache configurations are:

- cudaFuncCachePreferNone: no preference for shared memory or L1 (default)
- cudaFuncCachePreferShared: prefer larger shared memory and smaller L1 cache
- cudaFuncCachePreferL1: prefer larger L1 cache and smaller shared memory
- cudaFuncCachePreferEqual: prefer equal size L1 cache and shared memory

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaDeviceSetCacheConfig, cudaFuncSetCacheConfig [ C API], cudaFuncSetCacheConfig [ C++ API], cuCtxGetCacheConfig

__host__cudaError_t cudaDeviceGetDefaultMemPool (cudaMemPool_t *memPool, int device)
Returns the default mempool of a device.

Returns
cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue cudaErrorNotSupported

Description
The default mempool of a device contains device memory from that device.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cuDeviceGetDefaultMemPool, cudaMallocAsync, cudaMemPoolTrimTo,
cudaMemPoolGetAttribute, cudaDeviceSetMemPool, cudaMemPoolSetAttribute,
cudaMemPoolSetAccess

__host____device__cudaError_t cudaDeviceGetLimit (size_t *pValue, cudaLimit limit)
Returns resource limits.

Parameters
pValue
  - Returned size of the limit
limit
  - Limit to query
Returns

- `cudaSuccess`
- `cudaErrorUnsupportedLimit`
- `cudaErrorInvalidValue`

Description

Returns in `*pValue` the current size of `limit`. The supported `cudaLimit` values are:

- `cudaLimitStackSize`: stack size in bytes of each GPU thread;
- `cudaLimitPrintFifoSize`: size in bytes of the shared FIFO used by the printf() device system call;
- `cudaLimitMallocHeapSize`: size in bytes of the heap used by the malloc() and free() device system calls;
- `cudaLimitDevRuntimeSyncDepth`: maximum grid depth at which a thread can issue the device runtime call `cudaDeviceSynchronize()` to wait on child grid launches to complete;
- `cudaLimitDevRuntimePendingLaunchCount`: maximum number of outstanding device runtime launches;
- `cudaLimitMaxL2FetchGranularity`: L2 cache fetch granularity;
- `cudaLimitPersistingL2CacheSize`: Persisting L2 cache size in bytes

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotAllowed` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

- `cudaDeviceSetLimit`
- `cuCtxGetLimit`
__host__cudaError_t cudaDeviceGetMemPool(cuMemPool_t *memPool, int device)

Gets the current mempool for a device.

Returns
cudaSuccess, cudaMemcpyInvalidValue cudaErrorNotSupported

Description

Returns the last pool provided to `cudaDeviceSetMemPool` for this device or the device’s default memory pool if `cudaDeviceSetMemPool` has never been called. By default the current mempool is the default mempool for a device, otherwise the returned pool must have been set with `cuDeviceSetMemPool` or `cudaDeviceSetMemPool`.

Note:

‣ Note that this function may also return error codes from previous, asynchronous launches.
‣ Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
‣ Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cuDeviceGetMemPool, cudaDeviceGetDefaultMemPool, cudaDeviceSetMemPool

__host__cudaError_t 
 cudaDeviceGetNvSciSyncAttributes (void *nvSciSyncAttrList, int device, int flags)

Return NvSciSync attributes that this device can support.

Parameters

nvSciSyncAttrList
  - Return NvSciSync attributes supported.
device
  - Valid Cuda Device to get NvSciSync attributes for.
**flags**
- flags describing NvSciSync usage.

**Description**

Returns in `nvSciSyncAttrList`, the properties of NvSciSync that this CUDA device, `dev` can support. The returned `nvSciSyncAttrList` can be used to create an NvSciSync that matches this device's capabilities.

If `NvSciSyncAttrKey_RequiredPerm` field in `nvSciSyncAttrList` is already set this API will return `cudaErrorInvalidValue`.

The applications should set `nvSciSyncAttrList` to a valid `NvSciSyncAttrList` failing which this API will return `cudaErrorInvalidHandle`.

The `flags` controls how applications intends to use the NvSciSync created from the `nvSciSyncAttrList`. The valid flags are:

- `cudaNvSciSyncAttrSignal`, specifies that the applications intends to signal an NvSciSync on this CUDA device.
- `cudaNvSciSyncAttrWait`, specifies that the applications intends to wait on an NvSciSync on this CUDA device.

At least one of these flags must be set, failing which the API returns `cudaErrorInvalidValue`. Both the flags are orthogonal to one another: a developer may set both these flags that allows to set both wait and signal specific attributes in the same `nvSciSyncAttrList`.

Note that this API updates the input `nvSciSyncAttrList` with values equivalent to the following public attribute key-values: `NvSciSyncAttrKey_RequiredPerm` is set to


- `NvSciSyncAttrValPrimitiveType_SysmemSemaphore` on any valid device.
- `NvSciSyncAttrValPrimitiveType_Syncpoint` if `device` is a Tegra device.
- `NvSciSyncAttrValPrimitiveType_SysmemSemaphorePayload64b` if `device` is GA10X+. `NvSciSyncAttrKey_GpuId` is set to the same UUID that is returned in `cudaDeviceProp.uuid` from `cudaDeviceGetProperties` for this device.

`cudaSuccess`, `cudaErrorDeviceUninitialized`, `cudaErrorInvalidValue`, `cudaErrorInvalidHandle`, `cudaErrorInvalidDevice`, `cudaErrorNotSupported`, `cudaErrorMemoryAllocation`  

See also:
__host__ cudaError_t cudaDeviceGetP2PAttribute(
    int *value, cudaDeviceP2PAttr attr, int srcDevice, int dstDevice)

Queries attributes of the link between two devices.

Parameters

value
- Returned value of the requested attribute

attr

srcDevice
- The source device of the target link.

dstDevice
- The destination device of the target link.

Returns

cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue

Description

Returns in *value the value of the requested attribute attr of the link between srcDevice and dstDevice. The supported attributes are:

- cudaDevP2PAttrPerformanceRank: A relative value indicating the performance of the link between two devices. Lower value means better performance (0 being the value used for most performant link).
- cudaDevP2PAttrAccessSupported: 1 if peer access is enabled.
- cudaDevP2PAttrNativeAtomicSupported: 1 if native atomic operations over the link are supported.
- cudaDevP2PAttrCudaArrayAccessSupported: 1 if accessing CUDA arrays over the link is supported.

Returns cudaErrorInvalidDevice if srcDevice or dstDevice are not valid or if they represent the same device.

Returns cudaErrorInvalidValue if attr is not valid or if value is a null pointer.

Note:
Note that this function may also return error codes from previous, asynchronous launches.

Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.

Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaCtxEnablePeerAccess, cudaCtxDisablePeerAccess, cudaCtxCanAccessPeer, cuDeviceGetP2PAttribute

```c
__host__ cudaError_t cudaMemcpy (char *pciBusId, int len, int device)
```

Returns a PCI Bus Id string for the device.

**Parameters**

- `pciBusId` - Returned identifier string for the device in the following format `[domain]:[bus]:[device]:[function]` where `domain`, `bus`, `device`, and `function` are all hexadecimal values. `pciBusId` should be large enough to store 13 characters including the NULL-terminator.

- `len` - Maximum length of string to store in `name`

- `device` - Device to get identifier string for

**Returns**
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevice

**Description**

Returns an ASCII string identifying the device `dev` in the NULL-terminated string pointed to by `pciBusId`. `len` specifies the maximum length of the string that may be returned.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.

- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
`cudaDeviceGetByPCIBusId`, `cuDeviceGetPCIBusId`

```c
__host__ __device__ cudaError_t
cudaDeviceGetSharedMemConfig(cudaSharedMemConfig *pConfig)
```

Returns the shared memory configuration for the current device.

**Parameters**

**pConfig**
- Returned cache configuration

**Returns**
`cudaSuccess`, `cudaErrorInvalidValue`

**Description**

This function will return in `pConfig` the current size of shared memory banks on the current device. On devices with configurable shared memory banks, `cudaDeviceSetSharedMemConfig` can be used to change this setting, so that all subsequent kernel launches will by default use the new bank size. When `cudaDeviceGetSharedMemConfig` is called on devices without configurable shared memory, it will return the fixed bank size of the hardware.

The returned bank configurations can be either:

- `cudaSharedMemBankSizeFourByte` - shared memory bank width is four bytes.
- `cudaSharedMemBankSizeEightByte` - shared memory bank width is eight bytes.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
`cudaDeviceSetCacheConfig`, `cudaDeviceGetCacheConfig`, `cudaDeviceSetSharedMemConfig`, `cudaFuncSetCacheConfig`, `cuCtxGetSharedMemConfig`

__host__ __cudaError_t
cudaDeviceGetStreamPriorityRange (int *leastPriority, int *greatestPriority)
Returns numerical values that correspond to the least and greatest stream priorities.

Parameters

**leastPriority**
- Pointer to an int in which the numerical value for least stream priority is returned

**greatestPriority**
- Pointer to an int in which the numerical value for greatest stream priority is returned

Returns
`cudaSuccess`

Description

Returns in `*leastPriority` and `*greatestPriority` the numerical values that correspond to the least and greatest stream priorities respectively. Stream priorities follow a convention where lower numbers imply greater priorities. The range of meaningful stream priorities is given by `[*greatestPriority, *leastPriority]`. If the user attempts to create a stream with a priority value that is outside the the meaningful range as specified by this API, the priority is automatically clamped down or up to either `*leastPriority` or `*greatestPriority` respectively. See `cudaStreamCreateWithPriority` for details on creating a priority stream. A NULL may be passed in for `*leastPriority` or `*greatestPriority` if the value is not desired.

This function will return ‘0’ in both `*leastPriority` and `*greatestPriority` if the current context’s device does not support stream priorities (see `cudaDeviceGetAttribute`).

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
__host__ cudaError_t
cudaDeviceGetTexture1DLinearMaxWidth
(size_t *maxWidthInElements, const
cudaChannelFormatDesc *fmtDesc, int device)

Returns the maximum number of elements allocatable in a 1D linear texture for a given element size.

Parameters

maxWidthInElements
  - Returns maximum number of texture elements allocatable for given fmtDesc.

fmtDesc
  - Texture format description.

device

Returns

cudaSuccess, cudaErrorUnsupportedLimit, cudaErrorInvalidValue

Description

Returns in maxWidthInElements the maximum number of elements allocatable in a 1D linear texture for given format descriptor fmtDesc.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
See also:
cuDeviceGetMaxTexture1DLinear,

__host__cudaError_t cudaDeviceReset (void)
Destroy all allocations and reset all state on the current device in the current process.

Returns
cudaSuccess

Description
Explicitly destroys and cleans up all resources associated with the current device in the current process. Any subsequent API call to this device will reinitialize the device.

Note that this function will reset the device immediately. It is the caller’s responsibility to ensure that the device is not being accessed by any other host threads from the process when this function is called.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaDeviceSynchronize
__host__ cudaError_t cudaDeviceSetCacheConfig (cudaFuncCache cacheConfig)
Sets the preferred cache configuration for the current device.

Parameters

**cacheConfig**
- Requested cache configuration

Returns
cudaSuccess

Description
On devices where the L1 cache and shared memory use the same hardware resources, this sets through cacheConfig the preferred cache configuration for the current device. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute the function. Any function preference set via cudaFuncSetCacheConfig [ C API] or cudaFuncSetCacheConfig [ C++ API] will be preferred over this device-wide setting. Setting the device-wide cache configuration to cudaFuncCachePreferNone will cause subsequent kernel launches to prefer to not change the cache configuration unless required to launch the kernel.

This setting does nothing on devices where the size of the L1 cache and shared memory are fixed.

Launching a kernel with a different preference than the most recent preference setting may insert a device-side synchronization point.

The supported cache configurations are:

- **cudaFuncCachePreferNone**: no preference for shared memory or L1 (default)
- **cudaFuncCachePreferShared**: prefer larger shared memory and smaller L1 cache
- **cudaFuncCachePreferL1**: prefer larger L1 cache and smaller shared memory
- **cudaFuncCachePreferEqual**: prefer equal size L1 cache and shared memory

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
`cudaDeviceGetCacheConfig`, `cudaFuncSetCacheConfig` [C API], `cudaFuncSetCacheConfig` [C++ API], `cuCtxSetCacheConfig`

```c
__host__ cudaError_t cudaDeviceSetLimit (cudaLimit limit, size_t value)
```

Set resource limits.

Parameters

- **limit**
  - Limit to set

- **value**
  - Size of limit

Returns

- `cudaSuccess`
- `cudaErrorUnsupportedLimit`
- `cudaErrorInvalidValue`
- `cudaErrorMemoryAllocation`

Description

Setting `limit` to `value` is a request by the application to update the current limit maintained by the device. The driver is free to modify the requested value to meet h/w requirements (this could be clamping to minimum or maximum values, rounding up to nearest element size, etc). The application can use `cudaDeviceGetLimit()` to find out exactly what the limit has been set to.

Setting each `cudaLimit` has its own specific restrictions, so each is discussed here.

- **`cudaLimitStackSize`** controls the stack size in bytes of each GPU thread.
- **`cudaLimitPrintfFifoSize`** controls the size in bytes of the shared FIFO used by the printf() device system call. Setting `cudaLimitPrintfFifoSize` must not be performed after launching any kernel that uses the printf() device system call - in such case `cudaErrorInvalidValue` will be returned.
- **`cudaLimitMallocHeapSize`** controls the size in bytes of the heap used by the malloc() and free() device system calls. Setting `cudaLimitMallocHeapSize` must not be performed after launching any kernel that uses the malloc() or free() device system calls - in such case `cudaErrorInvalidValue` will be returned.
CUDA Runtime API

• **cudaLimitDevRuntimeSyncDepth** controls the maximum nesting depth of a grid at which a thread can safely call **cudaDeviceSynchronize**. Setting this limit must be performed before any launch of a kernel that uses the device runtime and calls **cudaDeviceSynchronize** above the default sync depth, two levels of grids. Calls to **cudaDeviceSynchronize** will fail with error code **cudaErrorSyncDepthExceeded** if the limitation is violated. This limit can be set smaller than the default or up the maximum launch depth of 24. When setting this limit, keep in mind that additional levels of sync depth require the runtime to reserve large amounts of device memory which can no longer be used for user allocations. If these reservations of device memory fail, **cudaDeviceSetLimit** will return **cudaErrorMemoryAllocation**, and the limit can be reset to a lower value. This limit is only applicable to devices of compute capability 3.5 and higher. Attempting to set this limit on devices of compute capability less than 3.5 will result in the error **cudaErrorUnsupportedLimit** being returned.

• **cudaLimitDevRuntimePendingLaunchCount** controls the maximum number of outstanding device runtime launches that can be made from the current device. A grid is outstanding from the point of launch up until the grid is known to have been completed. Device runtime launches which violate this limitation fail and return **cudaErrorLaunchPendingCountExceeded** when **cudaGetLastError** is called after launch. If more pending launches than the default (2048 launches) are needed for a module using the device runtime, this limit can be increased. Keep in mind that being able to sustain additional pending launches will require the runtime to reserve larger amounts of device memory upfront which can no longer be used for allocations. If these reservations fail, **cudaDeviceSetLimit** will return **cudaErrorMemoryAllocation**, and the limit can be reset to a lower value. This limit is only applicable to devices of compute capability 3.5 and higher. Attempting to set this limit on devices of compute capability less than 3.5 will result in the error **cudaErrorUnsupportedLimit** being returned.

• **cudaLimitMaxL2FetchGranularity** controls the L2 cache fetch granularity. Values can range from 0B to 128B. This is purely a performance hint and it can be ignored or clamped depending on the platform.

• **cudaLimitPersistingL2CacheSize** controls size in bytes available for persisting L2 cache. This is purely a performance hint and it can be ignored or clamped depending on the platform.

**Note:**

• Note that this function may also return error codes from previous, asynchronous launches.

• Note that this function may also return **cudaErrorInitializationError**, **cudaErrorInsufficientDriver** or **cudaErrorNoDevice** if this call tries to initialize internal CUDA RT state.
__host__ cudaError_t cudaDeviceSetMemPool (int device, cudaMemPool_t memPool)

Sets the current memory pool of a device.

**Returns**

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevice, cudaErrorNotSupported

**Description**

The memory pool must be local to the specified device. Unless a mempool is specified in the cudaMallocAsync call, cudaMallocAsync allocates from the current mempool of the provided stream’s device. By default, a device’s current memory pool is its default memory pool.

**Note:**

Use cudaMallocFromPoolAsync to specify asynchronous allocations from a device different than the one the stream runs on.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**

\_\_host\_\_cudaError\_t
cudaDeviceSetSharedMemConfig
(cudaSharedMemConfig config)

Sets the shared memory configuration for the current device.

Parameters

cfg
- Requested cache configuration

Returns
cudaSuccess, cudaErrorInvalidValue

Description

On devices with configurable shared memory banks, this function will set the shared memory
bank size which is used for all subsequent kernel launches. Any per-function setting of shared
memory set via cudaFuncSetSharedMemConfig will override the device wide setting.

Changing the shared memory configuration between launches may introduce a device side
synchronization point.

Changing the shared memory bank size will not increase shared memory usage or affect
occupancy of kernels, but may have major effects on performance. Larger bank sizes will
allow for greater potential bandwidth to shared memory, but will change what kinds of
accesses to shared memory will result in bank conflicts.

This function will do nothing on devices with fixed shared memory bank size.

The supported bank configurations are:

- cudaSharedMemBankSizeDefault: set bank width the device default (currently, four bytes)
- cudaSharedMemBankSizeFourByte: set shared memory bank width to be four bytes
  natively.
- cudaSharedMemBankSizeEightByte: set shared memory bank width to be eight bytes
  natively.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal
  CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**
- `cudaDeviceSetCacheConfig`, `cudaDeviceGetCacheConfig`, `cudaDeviceGetSharedMemConfig`, `cudaFuncSetCacheConfig`, `cuCtxSetSharedMemConfig`

```c
__host__ __device__ cudaError_t cudaDeviceSynchronize (void)
```

Wait for compute device to finish.

**Returns**
- `cudaSuccess`

**Description**

Blocks until the device has completed all preceding requested tasks. `cudaDeviceSynchronize()` returns an error if one of the preceding tasks has failed. If the `cudaDeviceScheduleBlockingSync` flag was set for this device, the host thread will block until the device has finished its work.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**
- `cudaDeviceReset`, `cuCtxSynchronize`
\texttt{\_\_host\_\_device\_\_cudaError\_t cudaGetDevice (int *device)}

Returns which device is currently being used.

**Parameters**

\textbf{device}
- Returns the device on which the active host thread executes the device code.

**Returns**

\texttt{cudaSuccess, cudaErrorInvalidValue}

**Description**

Returns in \texttt{*device} the current device for the calling host thread.

\begin{itemize}
  \item Note that this function may also return error codes from previous, asynchronous launches.
  \item Note that this function may also return \texttt{cudaErrorInitializationError, cudaErrorInsufficientDriver} or \texttt{cudaErrorNoDevice} if this call tries to initialize internal CUDA RT state.
  \item Note that as specified by \texttt{cudaStreamAddCallback} no CUDA function may be called from callback. \texttt{cudaErrorNotPermitted} may, but is not guaranteed to, be returned as a diagnostic in such case.
\end{itemize}

**See also:**

\texttt{cudaGetDeviceCount, cudaSetDevice, cudaGetDeviceProperties, cudaChooseDevice, cuCtxGetCurrent}

\texttt{\_\_host\_\_device\_\_cudaError\_t cudaGetDeviceCount (int *count)}

Returns the number of compute-capable devices.

**Parameters**

\textbf{count}
- Returns the number of devices with compute capability greater or equal to 2.0
Returns

cudaSuccess

Description

Returns in *count the number of devices with compute capability greater or equal to 2.0 that are available for execution.

Note:

‣ Note that this function may also return error codes from previous, asynchronous launches.
‣ Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
‣ Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGetDevice, cudaSetDevice, cudaGetDeviceProperties, cudaChooseDevice, cuDeviceGetCount

__host__cudaError_t cudaGetDeviceFlags (unsigned int *flags)

Gets the flags for the current device.

Parameters

flags
  - Pointer to store the device flags

Returns

cudaSuccess, cudaErrorInvalidDevice

Description

Returns in flags the flags for the current device. If there is a current device for the calling thread, the flags for the device are returned. If there is no current device, the flags for the first device are returned, which may be the default flags. Compare to the behavior of cudaSetDeviceFlags.
Typically, the flags returned should match the behavior that will be seen if the calling thread uses a device after this call, without any change to the flags or current device inbetween by this or another thread. Note that if the device is not initialized, it is possible for another thread to change the flags for the current device before it is initialized. Additionally, when using exclusive mode, if this thread has not requested a specific device, it may use a device other than the first device, contrary to the assumption made by this function.

If a context has been created via the driver API and is current to the calling thread, the flags for that context are always returned.

Flags returned by this function may specifically include `cudaDeviceMapHost` even though it is not accepted by `cudaSetDeviceFlags` because it is implicit in runtime API flags. The reason for this is that the current context may have been created via the driver API in which case the flag is not implicit and may be unset.

---

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

---

**See also:**

`cudaGetDevice`, `cudaGetDeviceProperties`, `cudaSetDevice`, `cudaSetDeviceFlags`, `cuCtxGetFlags`, `cuDevicePrimaryCtxGetState`

---

```c
__host__ cudaError_t cudaGetDeviceProperties (cudaDeviceProp *prop, int device)
```

Returns information about the compute-device.

**Parameters**

- **prop**
  - Properties for the specified device
- **device**
  - Device number to get properties for

**Returns**

`cudaSuccess`, `cudaErrorInvalidDevice`
Description

Returns in *prop the properties of device dev. The cudaDeviceProp structure is defined as:

```
struct cudaDeviceProp {
    char name[256];
    cudaUUID_t uuid;
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int regsPerBlock;
    int warpSize;
    size_t memPitch;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int gridSize[3];
    int clockRate;
    size_t totalConstMem;
    int major;
    int minor;
    size_t textureAlignment;
    size_t texturePitchAlignment;
    int deviceOverlap;
    int multiProcessorCount;
    int kernelExecTimeoutEnabled;
    int integrated;
    int canMapHostMemory;
    int computeMode;
    int maxTexture1D;
    int maxTexture1DMipmap;
    int maxTexture1DLinear;
    int maxTexture2D[2];
    int maxTexture2DMipmap[2];
    int maxTexture2DLinear[3];
    int maxTexture2Dgather[2];
    int maxTexture3D[3];
    int maxTexture3DAlt[3];
    int maxTextureCubemap;
    int maxTexture1DLayered[2];
    int maxTexture2DLayered[3];
    int maxTextureCubemapLayered[2];
    int maxSurface1D;
    int maxSurface2D[2];
    int maxSurface3D[3];
    int maxSurface1DLayered[2];
    int maxSurface2DLayered[3];
    int maxSurfaceCubemap;
    int maxSurfaceCubemapLayered[2];
    size_t surfaceAlignment;
    int ConcurrentKernels;
    int ECCEnabled;
    int pciBusID;
    int pciDeviceID;
    int pciDomainID;
    int tccDriver;
    int asyncEngineCount;
    int unifiedAddressing;
    int memoryClockRate;
    int memoryBusWidth;
    int l2CacheSize;
    int persistingL2CacheMaxSize;
    int maxThreadsPerMultiProcessor;
    int streamPrioritiesSupported;
    int globalL1CacheSupported;
    int localL1CacheSupported;
    size_t sharedMemPerMultiprocessor;
};
```
```c
int regsPerMultiprocessor;
int managedMemory;
int lMultiGpuBoard;
int multiGpuBoardGroupId;
int singleToDoublePrecisionPerfRatio;
int pageableMemoryAccess;
int concurrentManagedAccess;
int computePreemptionSupported;
int canUseHostPointerForRegisteredMem;
int cooperativeLaunch;
int cooperativeMultiDeviceLaunch;
int pageableMemoryAccessUsesHostPageTables;
int directManagedMemAccessFromHost;
int accessPolicyMaxWindowSize;
```

where:

- **name[256]** is an ASCII string identifying the device;
- **uuid** is a 16-byte unique identifier.
- **totalGlobalMem** is the total amount of global memory available on the device in bytes;
- **sharedMemPerBlock** is the maximum amount of shared memory available to a thread block in bytes;
- **regsPerBlock** is the maximum number of 32-bit registers available to a thread block;
- **warpSize** is the warp size in threads;
- **memPitch** is the maximum pitch in bytes allowed by the memory copy functions that involve memory regions allocated through `cudaMallocPitch()`;
- **maxThreadsPerBlock** is the maximum number of threads per block;
- **maxThreadsDim[3]** contains the maximum size of each dimension of a block;
- **maxGridSize[3]** contains the maximum size of each dimension of a grid;
- **clockRate** is the clock frequency in kilohertz;
- **totalConstMem** is the total amount of constant memory available on the device in bytes;
- **major, minor** are the major and minor revision numbers defining the device’s compute capability;
- **textureAlignment** is the alignment requirement; texture base addresses that are aligned to `textureAlignment` bytes do not need an offset applied to texture fetches;
- **texturePitchAlignment** is the pitch alignment requirement for 2D texture references that are bound to pitched memory;
- **deviceOverlap** is 1 if the device can concurrently copy memory between host and device while executing a kernel, or 0 if not. Deprecated, use instead `asyncEngineCount`.
- **multiProcessorCount** is the number of multiprocessors on the device;
- **kernelExecTimeoutEnabled** is 1 if there is a run time limit for kernels executed on the device, or 0 if not.

- **integrated** is 1 if the device is an integrated (motherboard) GPU and 0 if it is a discrete (card) component.

- **canMapHostMemory** is 1 if the device can map host memory into the CUDA address space for use with `cudaHostAlloc() / cudaHostGetDevicePointer()`, or 0 if not;

- **computeMode** is the compute mode that the device is currently in. Available modes are as follows:
  - `cudaComputeModeDefault`: Default mode - Device is not restricted and multiple threads can use `cudaSetDevice()` with this device.
  - `cudaComputeModeExclusive`: Compute-exclusive mode - Only one thread will be able to use `cudaSetDevice()` with this device.
  - `cudaComputeModeProhibited`: Compute-prohibited mode - No threads can use `cudaSetDevice()` with this device.
  - `cudaComputeModeExclusiveProcess`: Compute-exclusive-process mode - Many threads in one process will be able to use `cudaSetDevice()` with this device.

  If `cudaSetDevice()` is called on an already occupied device with `computeMode cudaComputeModeExclusive`, `cudaErrorDeviceAlreadyInUse` will be immediately returned indicating the device cannot be used. When an occupied exclusive mode device is chosen with `cudaSetDevice`, all subsequent non-device management runtime functions will return `cudaErrorDevicesUnavailable`.

- **maxTexture1D** is the maximum 1D texture size.

- **maxTexture1DMipmap** is the maximum 1D mipmapped texture texture size.

- **maxTexture1DLinear** is the maximum 1D texture size for textures bound to linear memory.

- **maxTexture2D[2]** contains the maximum 2D texture dimensions.

- **maxTexture2DMipmap[2]** contains the maximum 2D mipmapped texture dimensions.

- **maxTexture2DLinear[3]** contains the maximum 2D texture dimensions for 2D textures bound to pitch linear memory.

- **maxTexture2DGather[2]** contains the maximum 2D texture dimensions if texture gather operations have to be performed.

- **maxTexture3D[3]** contains the maximum 3D texture dimensions.

- **maxTexture3DAlt[3]** contains the maximum alternate 3D texture dimensions.

- **maxTextureCubemap** is the maximum cubemap texture width or height.

- **maxTexture1DLayered[2]** contains the maximum 1D layered texture dimensions.
- `maxSurface1D` is the maximum 1D surface size.
- `maxSurfaceCubemap` is the maximum cubemap surface width or height.
- `surfaceAlignment` specifies the alignment requirements for surfaces.
- `concurrentKernels` is 1 if the device supports executing multiple kernels within the same context simultaneously, or 0 if not. It is not guaranteed that multiple kernels will be resident on the device concurrently so this feature should not be relied upon for correctness;
- `ECCEnabled` is 1 if the device has ECC support turned on, or 0 if not.
- `pciBusID` is the PCI bus identifier of the device.
- `pciDeviceID` is the PCI device (sometimes called slot) identifier of the device.
- `pciDomainID` is the PCI domain identifier of the device.
- `tccDriver` is 1 if the device is using a TCC driver or 0 if not.
- `asyncEngineCount` is 1 when the device can concurrently copy memory between host and device while executing a kernel. It is 2 when the device can concurrently copy memory between host and device in both directions and execute a kernel at the same time. It is 0 if neither of these is supported.
- `unifiedAddressing` is 1 if the device shares a unified address space with the host and 0 otherwise.
- `memoryClockRate` is the peak memory clock frequency in kilohertz.
- `memoryBusWidth` is the memory bus width in bits.
- `l2CacheSize` is L2 cache size in bytes.
- `persistingL2CacheMaxSize` is L2 cache’s maximum persisting lines size in bytes.
- `maxThreadsPerMultiProcessor` is the number of maximum resident threads per multiprocessor.
- **streamPrioritiesSupported** is 1 if the device supports stream priorities, or 0 if it is not supported.
- **globalL1CacheSupported** is 1 if the device supports caching of globals in L1 cache, or 0 if it is not supported.
- **localL1CacheSupported** is 1 if the device supports caching of locals in L1 cache, or 0 if it is not supported.
- **sharedMemPerMultiprocessor** is the maximum amount of shared memory available to a multiprocessor in bytes; this amount is shared by all thread blocks simultaneously resident on a multiprocessor;
- **regsPerMultiprocessor** is the maximum number of 32-bit registers available to a multiprocessor; this number is shared by all thread blocks simultaneously resident on a multiprocessor;
- **managedMemory** is 1 if the device supports allocating managed memory on this system, or 0 if it is not supported.
- **isMultiGpuBoard** is 1 if the device is on a multi-GPU board (e.g. Gemini cards), and 0 if not;
- **multiGpuBoardGroupID** is a unique identifier for a group of devices associated with the same board. Devices on the same multi-GPU board will share the same identifier;
- **singleToDoublePrecisionPerfRatio** is the ratio of single precision performance (in floating-point operations per second) to double precision performance.
- **pageableMemoryAccess** is 1 if the device supports coherently accessing pageable memory without calling cudaHostRegister on it, and 0 otherwise.
- **concurrentManagedAccess** is 1 if the device can coherently access managed memory concurrently with the CPU, and 0 otherwise.
- **computePreemptionSupported** is 1 if the device supports Compute Preemption, and 0 otherwise.
- **canUseHostPointerForRegisteredMem** is 1 if the device can access host registered memory at the same virtual address as the CPU, and 0 otherwise.
- **cooperativeLaunch** is 1 if the device supports launching cooperative kernels via cudaLaunchCooperativeKernel, and 0 otherwise.
- **cooperativeMultiDeviceLaunch** is 1 if the device supports launching cooperative kernels via cudaLaunchCooperativeKernelMultiDevice, and 0 otherwise.
- **pageableMemoryAccessUsesHostPageTables** is 1 if the device accesses pageable memory via the host’s page tables, and 0 otherwise.
- **directManagedMemAccessFromHost** is 1 if the host can directly access managed memory on the device without migration, and 0 otherwise.
**maxBlocksPerMultiProcessor** is the maximum number of thread blocks that can reside on a multiprocessor.

**accessPolicyMaxWindowSize** is the maximum value of `cudaAccessPolicyWindow::num_bytes`.

---

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

---

**See also:**

`cudaGetDeviceCount`, `cudaGetDevice`, `cudaSetDevice`, `cudaChooseDevice`, `cudaDeviceGetAttribute`, `cuDeviceGetAttribute`, `cuDeviceGetName`

---

```c
__host__ cudaError_t cudaIpcCloseMemHandle (void *devPtr)
```

Attempts to close memory mapped with `cudaIpcOpenMemHandle`.

**Parameters**

- `devPtr` - Device pointer returned by `cudaIpcOpenMemHandle`

**Returns**

- `cudaSuccess`, `cudaErrorMapBufferObjectFailed`, `cudaErrorNotSupported`, `cudaErrorInvalidValue`

**Description**

Decrements the reference count of the memory returned by `cudaIpcOpenMemHandle` by 1. When the reference count reaches 0, this API unmaps the memory. The original allocation in the exporting process as well as imported mappings in other processes will be unaffected.

Any resources used to enable peer access will be freed if this is the last mapping using them.

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems. IPC functionality is not supported on Tegra platforms.
__host__ cudaError_t cudaIpcGetEventHandle(
cudaIpcEventHandle_t *handle, cudaEvent_t event)

Gets an interprocess handle for a previously allocated event.

**Parameters**

- **handle**
  - Pointer to a user allocated cudaIpcEventHandle in which to return the opaque event handle
- **event**
  - Event allocated with cudaEventInterprocess and cudaEventDisableTiming flags.

**Returns**

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorMemoryAllocation, cudaErrorMapBufferObjectFailed, cudaErrorNotSupported, cudaErrorInvalidValue

**Description**

Takes as input a previously allocated event. This event must have been created with the cudaEventInterprocess and cudaEventDisableTiming flags set. This opaque handle may be copied into other processes and opened with cudaIpcOpenEventHandle to allow efficient hardware synchronization between GPU work in different processes.

After the event has been opened in the importing process, cudaEventRecord, cudaEventSynchronize, cudaStreamWaitEvent and cudaEventQuery may be used in either process. Performing operations on the imported event after the exported event has been freed with cudaEventDestroy will result in undefined behavior.
IPC functionality is restricted to devices with support for unified addressing on Linux operating systems. IPC functionality is not supported on Tegra platforms.

**Note:**
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**
`cudaEventCreate`, `cudaEventDestroy`, `cudaEventSynchronize`, `cudaEventQuery`, `cudaStreamWaitEvent`, `cudaIpcOpenEventHandle`, `cudaIpcGetMemHandle`, `cudaIpcOpenMemHandle`, `cudaIpcCloseMemHandle`, `cuIpcGetEventHandle`

```c
__host__ cudaError_t cudaIpcGetMemHandle (cudaIpcMemHandle_t *handle, void *devPtr)
```

Gets an interprocess memory handle for an existing device memory allocation.

**Parameters**

- `handle` - Pointer to user allocated cudaIpcMemHandle to return the handle in.
- `devPtr` - Base pointer to previously allocated device memory

**Returns**

`cudaSuccess`, `cudaErrorMemoryAllocation`, `cudaErrorMapBufferObjectFailed`, `cudaErrorNotSupported`, `cudaErrorInvalidValue`

**Description**

Takes a pointer to the base of an existing device memory allocation created with `cudaMalloc` and exports it for use in another process. This is a lightweight operation and may be called multiple times on an allocation without adverse effects.

If a region of memory is freed with `cudaFree` and a subsequent call to `cudaMalloc` returns memory with the same device address, `cudaIpcGetMemHandle` will return a unique handle for the new memory.
IPC functionality is restricted to devices with support for unified addressing on Linux operating systems. IPC functionality is not supported on Tegra platforms.

**Note:**

- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

`cudaMalloc`, `cudaFree`, `cudaIpcGetEventHandle`, `cudaIpcOpenEventHandle`,
`cudaIpcOpenMemHandle`, `cudaIpcCloseMemHandle`, `culpcGetMemHandle`

```c
__host__ cudaError_t cudaIpcOpenEventHandle (cudaEvent_t *event, cudaIpcEventHandle_t handle)
```

Opens an interprocess event handle for use in the current process.

**Parameters**

- `event` - Returns the imported event
- `handle` - Interprocess handle to open

**Returns**

`cudaSuccess`, `cudaErrorMapBufferObjectFailed`, `cudaErrorNotSupported`,
`cudaErrorInvalidValue`, `cudaErrorDeviceUninitialized`

**Description**

Opens an interprocess event handle exported from another process with `cudaIpcGetEventHandle`. This function returns a `cudaEvent_t` that behaves like a locally created event with the `cudaEventDisableTiming` flag specified. This event must be freed with `cudaEventDestroy`.

Performing operations on the imported event after the exported event has been freed with `cudaEventDestroy` will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems. IPC functionality is not supported on Tegra platforms.
Note:

- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

`cudaEventCreate`, `cudaEventDestroy`, `cudaEventSynchronize`, `cudaEventQuery`, `cudaStreamWaitEvent`, `cudaIpcGetEventHandle`, `cudaIpcGetMemHandle`, `cudaIpcOpenMemHandle`, `cudaIpcCloseMemHandle`, `culpcOpenEventHandle`

```c
__host__ cudaError_t cudaIpcOpenMemHandle (void **devPtr, cudaIpcMemHandle_t handle, unsigned int flags)
```

Opens an interprocess memory handle exported from another process and returns a device pointer usable in the local process.

**Parameters**

- **devPtr**
  - Returned device pointer
- **handle**
  - `cudaIpcMemHandle` to open
- **flags**
  - Flags for this operation. Must be specified as `cudaIpcMemLazyEnablePeerAccess`

**Returns**

`cudaSuccess`, `cudaErrorMapBufferObjectFailed`, `cudaErrorInvalidResourceHandle`, `cudaErrorDeviceUninitialized`, `cudaErrorTooManyPeers`, `cudaErrorNotSupported`, `cudaErrorInvalidValue`

**Description**

Maps memory exported from another process with `cudaIpcGetMemHandle` into the current device address space. For contexts on different devices `cudaIpcOpenMemHandle` can attempt to enable peer access between the devices as if the user called `cudaDeviceEnablePeerAccess`. This behavior is controlled by the `cudaIpcMemLazyEnablePeerAccess` flag. `cudaDeviceCanAccessPeer` can determine if a mapping is possible.
**cudaIpcOpenMemHandle** can open handles to devices that may not be visible in the process calling the API.

Contexts that may open cudaIpcMemHandles are restricted in the following way. cudaIpcMemHandles from each device in a given process may only be opened by one context per device per other process.

If the memory handle has already been opened by the current context, the reference count on the handle is incremented by 1 and the existing device pointer is returned.

Memory returned from cudaIpcOpenMemHandle must be freed with cudaIpcCloseMemHandle.

Calling cudaFree on an exported memory region before calling cudaIpcCloseMemHandle in the importing context will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems. IPC functionality is not supported on Tegra platforms.

---

**Note:**

- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.

- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

- No guarantees are made about the address returned in *devPtr. In particular, multiple processes may not receive the same address for the same handle.

---

See also:
cudaMalloc, cudaMemcpy, cudaIpcGetMemHandle, cudaIpcOpenEventHandle, cudaIpcGetEventHandle, cudaIpcCloseMemHandle, cudaDeviceEnablePeerAccess, cudaDeviceCanAccessPeer, cuIpcOpenMemHandle

__host__cudaError_t cudaSetDevice (int device)

Set device to be used for GPU executions.

**Parameters**
device
- Device on which the active host thread should execute the device code.
Returns

cudaSuccess, cudaErrorInvalidDevice, cudaErrorDeviceAlreadyInUse

Description

Sets device as the current device for the calling host thread. Valid device id’s are 0 to (cudaGetDeviceCount() - 1).

Any device memory subsequently allocated from this host thread using cudaMalloc(), cudaMallocPitch() or cudaMallocArray() will be physically resident on device. Any host memory allocated from this host thread using cudaMallocHost() or cudaHostAlloc() or cudaHostRegister() will have its lifetime associated with device. Any streams or events created from this host thread will be associated with device. Any kernels launched from this host thread using the <<<>>> operator or cudaLaunchKernel() will be executed on device.

This call may be made from any host thread, to any device, and at any time. This function will do no synchronization with the previous or new device, and should be considered a very low overhead call.

Note:

‣ Note that this function may also return error codes from previous, asynchronous launches.
‣ Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
‣ Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaGetDeviceCount, cudaGetDevice, cudaGetDeviceProperties, cudaChooseDevice, cuCtxSetCurrent

__host__cudaError_t cudaSetDeviceFlags (unsigned int flags)

Sets flags to be used for device executions.

Parameters

flags

- Parameters for device operation
Returns

cudaSuccess, cudaErrorInvalidValue.

Description

Records flags as the flags for the current device. If the current device has been set and that device has already been initialized, the previous flags are overwritten. If the current device has not been initialized, it is initialized with the provided flags. If no device has been made current to the calling thread, a default device is selected and initialized with the provided flags.

The two LSBs of the flags parameter can be used to control how the CPU thread interacts with the OS scheduler when waiting for results from the device.

- **cudaDeviceScheduleAuto**: The default value if the flags parameter is zero, uses a heuristic based on the number of active CUDA contexts in the process C and the number of logical processors in the system P. If C > P, then CUDA will yield to other OS threads when waiting for the device, otherwise CUDA will not yield while waiting for results and actively spin on the processor. Additionally, on Tegra devices, cudaDeviceScheduleAuto uses a heuristic based on the power profile of the platform and may choose cudaDeviceScheduleBlockingSync for low-powered devices.

- **cudaDeviceScheduleSpin**: Instruct CUDA to actively spin when waiting for results from the device. This can decrease latency when waiting for the device, but may lower the performance of CPU threads if they are performing work in parallel with the CUDA thread.

- **cudaDeviceScheduleYield**: Instruct CUDA to yield its thread when waiting for results from the device. This can increase latency when waiting for the device, but can increase the performance of CPU threads performing work in parallel with the device.

- **cudaDeviceScheduleBlockingSync**: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the device to finish work.

- **cudaDeviceBlockingSync**: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the device to finish work.

  **Deprecated**: This flag was deprecated as of CUDA 4.0 and replaced with cudaDeviceScheduleBlockingSync.

- **cudaDeviceMapHost**: This flag enables allocating pinned host memory that is accessible to the device. It is implicit for the runtime but may be absent if a context is created using the driver API. If this flag is not set, cudaHostGetDevicePointer[] will always return a failure code.

- **cudaDeviceLmemResizeToMax**: Instruct CUDA to not reduce local memory after resizing local memory for a kernel. This can prevent thrashing by local memory allocations when launching many kernels with high local memory usage at the cost of potentially increased memory usage.
**Deprecated:** This flag is deprecated and the behavior enabled by this flag is now the default and cannot be disabled.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
- `cudaGetDeviceFlags`, `cudaGetDeviceCount`, `cudaGetDevice`, `cudaGetDeviceProperties`, `cudaSetDevice`, `cudaSetValidDevices`, `cudaChooseDevice`, `cuDevicePrimaryCtxSetFlags`

```c
__host__ cudaError_t cudaSetValidDevices (int *device_arr, int len)
```

Set a list of devices that can be used for CUDA.

**Parameters**

- **device_arr**
  - List of devices to try

- **len**
  - Number of devices in specified list

**Returns**

- `cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`

**Description**

Sets a list of devices for CUDA execution in priority order using `device_arr`. The parameter `len` specifies the number of elements in the list. CUDA will try devices from the list sequentially until it finds one that works. If this function is not called, or if it is called with a `len` of 0, then CUDA will go back to its default behavior of trying devices sequentially from a default list containing all of the available CUDA devices in the system. If a specified device ID in the list does not exist, this function will return `cudaErrorInvalidDevice`. If `len` is not 0 and `device_arr` is NULL or if `len` exceeds the number of devices in the system, then `cudaErrorInvalidValue` is returned.
6.2. Thread Management [DEPRECATED]

This section describes deprecated thread management functions of the CUDA runtime application programming interface.

`__host__ cudaError_t cudaThreadExit (void)`

Exit and clean up from CUDA launches.

**Returns**

`cudaSuccess`

**Description**

`Deprecated`

Note that this function is deprecated because its name does not reflect its behavior. Its functionality is identical to the non-deprecated function `cudaDeviceReset()`, which should be used instead.

Explicitly destroys all cleans up all resources associated with the current device in the current process. Any subsequent API call to this device will reinitialize the device.

Note that this function will reset the device immediately. It is the caller’s responsibility to ensure that the device is not being accessed by any other host threads from the process when this function is called.

See also:

`cudaGetDeviceCount`, `cudaSetDevice`, `cudaGetDeviceProperties`, `cudaSetDeviceFlags`, `cudaChooseDevice`
Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaDeviceReset

__host__ cudaError_t cudaThreadGetCacheConfig (cudaFuncCache *pCacheConfig)

Returns the preferred cache configuration for the current device.

Parameters

pCacheConfig
- Returned cache configuration

Returns

cudaSuccess

Description

Deprecated

Note that this function is deprecated because its name does not reflect its behavior. Its functionality is identical to the non-deprecated function `cudaDeviceGetCacheConfig()`, which should be used instead.

On devices where the L1 cache and shared memory use the same hardware resources, this returns through pCacheConfig the preferred cache configuration for the current device. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute functions.

This will return a pCacheConfig of `cudaFuncCachePreferNone` on devices where the size of the L1 cache and shared memory are fixed.

The supported cache configurations are:

- `cudaFuncCachePreferNone`: no preference for shared memory or L1 (default)
• `cudaFuncCachePreferShared`: prefer larger shared memory and smaller L1 cache
• `cudaFuncCachePreferL1`: prefer larger L1 cache and smaller shared memory

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**
`cudaDeviceGetCacheConfig`

```__host__cudaError_t cudaMemcpy (void *dest, const void *src, size_t size, cudaMemcpyType type)`

Returns resource limits.

**Parameters**
- `pValue` - Returned size in bytes of limit
- `limit` - Limit to query

**Returns**
`cudaSuccess`, `cudaErrorUnsupportedLimit`, `cudaErrorInvalidValue`

**Description**
`Deprecated`

Note that this function is deprecated because its name does not reflect its behavior. Its functionality is identical to the non-deprecated function `cudaDeviceGetLimit()`, which should be used instead.

Returns in `*pValue` the current size of `limit`. The supported `cudaLimit` values are:
- `cudaLimitStackSize`: stack size of each GPU thread;
- `cudaLimitPrintfFifoSize`: size of the shared FIFO used by the printf() device system call.
Modules

- **cudaLimitMallocHeapSize**: size of the heap used by the malloc() and free() device system calls;

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**
`cudaDeviceGetLimit`

```c
__host__ cudaError_t cudaThreadSetCacheConfig (cudaFuncCache cacheConfig)
```

Sets the preferred cache configuration for the current device.

**Parameters**

- **cacheConfig**
  - Requested cache configuration

**Returns**

- `cudaSuccess`

**Description**

**Deprecated**

Note that this function is deprecated because its name does not reflect its behavior. Its functionality is identical to the non-deprecated function `cudaDeviceSetCacheConfig[]`, which should be used instead.

On devices where the L1 cache and shared memory use the same hardware resources, this sets through `cacheConfig` the preferred cache configuration for the current device. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute the function. Any function preference set via `cudaFuncSetCacheConfig [ C API]` or `cudaFuncSetCacheConfig [ C++ API]` will be preferred over this device-wide setting. Setting the device-wide cache configuration to
CUDA Runtime API

**cudaFuncCachePreferNone** will cause subsequent kernel launches to prefer to not change the cache configuration unless required to launch the kernel.

This setting does nothing on devices where the size of the L1 cache and shared memory are fixed.

Launching a kernel with a different preference than the most recent preference setting may insert a device-side synchronization point.

The supported cache configurations are:

- **cudaFuncCachePreferNone**: no preference for shared memory or L1 (default)
- **cudaFuncCachePreferShared**: prefer larger shared memory and smaller L1 cache
- **cudaFuncCachePreferL1**: prefer larger L1 cache and smaller shared memory

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**

`cudaDeviceSetCacheConfig`

**__host__cudaError_t cudaThreadSetLimit (cudaLimit limit, size_t value)**

Set resource limits.

**Parameters**

- **limit**
  - Limit to set
- **value**
  - Size in bytes of limit

**Returns**

`cudaSuccess`, `cudaErrorUnsupportedLimit`, `cudaErrorInvalidValue`
Description

Deprecated

Note that this function is deprecated because its name does not reflect its behavior. Its functionality is identical to the non-deprecated function cudaDeviceSetLimit(), which should be used instead.

Setting limit to value is a request by the application to update the current limit maintained by the device. The driver is free to modify the requested value to meet h/w requirements [this could be clamping to minimum or maximum values, rounding up to nearest element size, etc]. The application can use cudaThreadGetLimit() to find out exactly what the limit has been set to.

Setting each cudaLimit has its own specific restrictions, so each is discussed here.

- **cudaLimitStackSize** controls the stack size of each GPU thread.
- **cudaLimitPrintfFifoSize** controls the size of the shared FIFO used by the printf() device system call. Setting cudaLimitPrintfFifoSize must be performed before launching any kernel that uses the printf() device system call, otherwise cudaErrorInvalidValue will be returned.
- **cudaLimitMallocHeapSize** controls the size of the heap used by the malloc() and free() device system calls. Setting cudaLimitMallocHeapSize must be performed before launching any kernel that uses the malloc() or free() device system calls, otherwise cudaErrorInvalidValue will be returned.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaDeviceSetLimit
__host__ cudaError_t cudaThreadSynchronize (void)

Wait for compute device to finish.

Returns
cudaSuccess

Description
Deprecated

Note that this function is deprecated because its name does not reflect its behavior. Its functionality is similar to the non-deprecated function cudaDeviceSynchronize(), which should be used instead.

Blocks until the device has completed all preceding requested tasks. cudaThreadSynchronize() returns an error if one of the preceding tasks has failed. If the cudaDeviceScheduleBlockingSync flag was set for this device, the host thread will block until the device has finished its work.

\[\text{Note:}\]
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaDeviceSynchronize

6.3. Error Handling

This section describes the error handling functions of the CUDA runtime application programming interface.
__host__ __device__ __const char *cudaGetErrorName (cudaError_t error)

Returns the string representation of an error code enum name.

Parameters

error
  - Error code to convert to string

Returns

cchar* pointer to a NULL-terminated string

Description

Returns a string containing the name of an error code in the enum. If the error code is not recognized, “unrecognized error code” is returned.

See also:

cudaGetErrorString, cudaGetLastError, cudaPeekAtLastError, cudaError, cuGetErrorName

__host__ __device__ __const char *cudaGetErrorString (cudaError_t error)

Returns the description string for an error code.

Parameters

error
  - Error code to convert to string

Returns

cchar* pointer to a NULL-terminated string

Description

Returns the description string for an error code. If the error code is not recognized, “unrecognized error code” is returned.

See also:

cudaGetErrorName, cudaGetLastError, cudaPeekAtLastError, cudaError, cuGetErrorString
__host__ __device__ cudaError_t cudaGetLastError (void)

Returns the last error from a runtime call.

Returns

cudaSuccess, cudaErrorMissingConfiguration, cudaErrorMemoryAllocation,
cudaErrorInitializationError, cudaErrorLaunchFailure, cudaErrorLaunchTimeout,
cudaErrorLaunchOutOfResources, cudaErrorInvalidDeviceFunction,
cudaErrorInvalidConfiguration, cudaErrorInvalidDevice, cudaErrorInvalidValue,
cudaErrorInvalidPitchValue, cudaErrorInvalidSymbol, cudaErrorUnmapBufferObjectFailed,
cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture, cudaErrorInvalidTextureBinding,
cudaErrorInvalidChannelDescriptor, cudaErrorInvalidMemcpyDirection,
cudaErrorInvalidFilterSetting, cudaErrorInvalidNormSetting, cudaErrorUnknown,
cudaErrorInvalidResourceHandle, cudaErrorInsufficientDriver, cudaErrorNoDevice,
cudaErrorSetOnActiveProcess, cudaErrorStartupFailure, cudaErrorInvalidPtx,
cudaErrorUnsupportedPtxVersion, cudaErrorNoKernelImageForDevice,
cudaErrorJitCompilerNotFound, cudaErrorJitCompilationDisabled

Description

Returns the last error that has been produced by any of the runtime calls in the same host thread and resets it to cudaSuccess.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaPeekAtLastError, cudaGetErrorName, cudaGetErrorString, cudaError
__host__ __device__ cudaError_t cudaPeekAtLastError (void)

Returns the last error from a runtime call.

**Returns**


**Description**

Returns the last error that has been produced by any of the runtime calls in the same host thread. Note that this call does not reset the error to cudaSuccess like cudaGetLastError.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**

cudaGetLastError, cudaGetErrorName, cudaGetErrorString, cudaError
6.4. Stream Management

This section describes the stream management functions of the CUDA runtime application programming interface.

typedef void (CUDART_CB *cudaStreamCallback_t)(cudaStream_t stream, cudaError_t status, void* userData)

Type of stream callback functions.

__host__cudaError_t cudaCtxResetPersistingL2Cache (void)

Resets all persisting lines in cache to normal status.

Returns

cudaSuccess.

Description

Resets all persisting lines in cache to normal status. Takes effect on function return.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaAccessPolicyWindow

__host__cudaError_t cudaStreamAddCallback (cudaStream_t stream, cudaStreamCallback_t callback, void *userData, unsigned int flags)

Add a callback to a compute stream.

Parameters

stream

- Stream to add callback to
**callback**
- The function to call once preceding stream operations are complete

**userData**
- User specified data to be passed to the callback function

**flags**
- Reserved for future use, must be 0

**Returns**
cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorInvalidValue, cudaErrorNotSupported

**Description**

**Note:**
This function is slated for eventual deprecation and removal. If you do not require the callback to execute in case of a device error, consider using cudaLaunchHostFunc. Additionally, this function is not supported with cudaStreamBeginCapture and cudaStreamEndCapture, unlike cudaLaunchHostFunc.

Adds a callback to be called on the host after all currently enqueued items in the stream have completed. For each cudaStreamAddCallback call, a callback will be executed exactly once. The callback will block later work in the stream until it is finished.

The callback may be passed cudaSuccess or an error code. In the event of a device error, all subsequently executed callbacks will receive an appropriate cudaError_t.

Callbacks must not make any CUDA API calls. Attempting to use CUDA APIs may result in cudaErrorNotPermitted. Callbacks must not perform any synchronization that may depend on outstanding device work or other callbacks that are not mandated to run earlier. Callbacks without a mandated order (in independent streams) execute in undefined order and may be serialized.

For the purposes of Unified Memory, callback execution makes a number of guarantees:

- The callback stream is considered idle for the duration of the callback. Thus, for example, a callback may always use memory attached to the callback stream.
- The start of execution of a callback has the same effect as synchronizing an event recorded in the same stream immediately prior to the callback. It thus synchronizes streams which have been “joined” prior to the callback.
- Adding device work to any stream does not have the effect of making the stream active until all preceding callbacks have executed. Thus, for example, a callback might use global attached memory even if work has been added to another stream, if it has been properly ordered with an event.
Completion of a callback does not cause a stream to become active except as described above. The callback stream will remain idle if no device work follows the callback, and will remain idle across consecutive callbacks without device work in between. Thus, for example, stream synchronization can be done by signaling from a callback at the end of the stream.

Note:
- This function uses standard default stream semantics.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaStreamCreate, cudaStreamCreateWithFlags, cudaStreamQuery, cudaStreamSynchronize, cudaStreamWaitEvent, cudaStreamDestroy, cudaMallocManaged, cudaStreamAttachMemAsync, cudaLaunchHostFunc, cuStreamAddCallback

__host__cudaError_t cudaStreamAttachMemAsync (cudaStream_t stream, void *devPtr, size_t length, unsigned int flags)
Attach memory to a stream asynchronously.

Parameters

stream
- Stream in which to enqueue the attach operation
devPtr
- Pointer to memory (must be a pointer to managed memory or to a valid host-accessible region of system-allocated memory)
length
- Length of memory (defaults to zero)
flags
- Must be one of cudaMemAttachGlobal, cudaMemAttachHost or cudaMemAttachSingle (defaults to cudaMemAttachSingle)
Returns

```
cudaSuccess, cudaErrorNotReady, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle
```

Description

Enqueues an operation in `stream` to specify stream association of `length` bytes of memory starting from `devPtr`. This function is a stream-ordered operation, meaning that it is dependent on, and will only take effect when, previous work in stream has completed. Any previous association is automatically replaced.

`devPtr` must point to one of the following types of memories:

- managed memory declared using the `__managed__` keyword or allocated with `cudaMallocManaged`
- a valid host-accessible region of system-allocated pageable memory. This type of memory may only be specified if the device associated with the stream reports a non-zero value for the device attribute `cudaDevAttrPageableMemoryAccess`.

For managed allocations, `length` must be either zero or the entire allocation’s size. Both indicate that the entire allocation’s stream association is being changed. Currently, it is not possible to change stream association for a portion of a managed allocation.

For pageable allocations, `length` must be non-zero.

The stream association is specified using `flags` which must be one of `cudaMemAttachGlobal`, `cudaMemAttachHost` or `cudaMemAttachSingle`. The default value for `flags` is `cudaMemAttachSingle`. If the `cudaMemAttachGlobal` flag is specified, the memory can be accessed by any stream on any device. If the `cudaMemAttachHost` flag is specified, the program makes a guarantee that it won’t access the memory on the device from any stream on a device that has a zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`. If the `cudaMemAttachSingle` flag is specified and `stream` is associated with a device that has a zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`, the program makes a guarantee that it will only access the memory on the device from `stream`. It is illegal to attach singly to the NULL stream, because the NULL stream is a virtual global stream and not a specific stream. An error will be returned in this case.

When memory is associated with a single stream, the Unified Memory system will allow 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.

Accessing memory on the device from streams that are not associated with it will produce undefined results. No error checking is performed by the Unified Memory system to ensure that kernels launched into other streams do not access this region.
It is a program’s responsibility to order calls to `cudaStreamAttachMemAsync` via events, synchronization or other means to ensure legal access to memory at all times. Data visibility and coherency will be changed appropriately for all kernels which follow a stream-association change.

If `stream` is destroyed while data is associated with it, the association is removed and the association reverts to the default visibility of the allocation as specified at `cudaMallocManaged`. For `__managed__` variables, the default association is always `cudaMemAttachGlobal`. Note that destroying a stream is an asynchronous operation, and as a result, the change to default association won’t happen until all work in the stream has completed.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**
- `cudaStreamCreate`, `cudaStreamCreateWithFlags`, `cudaStreamWaitEvent`, `cudaStreamSynchronize`, `cudaStreamAddCallback`, `cudaStreamDestroy`, `cudaMallocManaged`, `cuStreamAttachMemAsync`

**__host__cudaError_t cudaStreamBeginCapture**

(cudaStream_t stream, cudaStreamCaptureMode mode)

Begins graph capture on a stream.

**Parameters**

- **stream**
  - Stream in which to initiate capture

- **mode**
  - Controls the interaction of this capture sequence with other API calls that are potentially unsafe. For more details see `cudaThreadExchangeStreamCaptureMode`.  

See also:
- `cudaStreamCreate`, `cudaStreamCreateWithFlags`, `cudaStreamWaitEvent`, `cudaStreamSynchronize`, `cudaStreamAddCallback`, `cudaStreamDestroy`, `cudaMallocManaged`, `cuStreamAttachMemAsync`
Returns
cudaSuccess, cudaErrorInvalidValue

Description

Begin graph capture on stream. When a stream is in capture mode, all operations pushed into the stream will not be executed, but will instead be captured into a graph, which will be returned via cudaStreamEndCapture. Capture may not be initiated if stream is cudaStreamLegacy. Capture must be ended on the same stream in which it was initiated, and it may only be initiated if the stream is not already in capture mode. The capture mode may be queried via cudaStreamIsCapturing. A unique id representing the capture sequence may be queried via cudaStreamGetCaptureInfo.

If mode is not cudaStreamCaptureModeRelaxed, cudaStreamEndCapture must be called on this stream from the same thread.

Note:
Kernels captured using this API must not use texture and surface references. Reading or writing through any texture or surface reference is undefined behavior. This restriction does not apply to texture and surface objects.

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaStreamCreate, cudaStreamIsCapturing, cudaStreamEndCapture, cudaThreadExchangeStreamCaptureMode

__host__cudaError_t cudaStreamCopyAttributes (cudaStream_t dst, cudaStream_t src)
Copies attributes from source stream to destination stream.

Parameters
dst
  Destination stream
src
  Source stream For attributes see cudaStreamAttrID
Returns

cudaSuccess, cudaErrorNotSupported

Description

Copies attributes from source stream \texttt{src} to destination stream \texttt{dst}. Both streams must have the same context.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaAccessPolicyWindow

\_\_host\_\_cudaError\_t cudaStreamCreate (cudaStream\_t *pStream)

Create an asynchronous stream.

Parameters

\texttt{pStream}

- Pointer to new stream identifier

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Creates a new asynchronous stream.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
__host__ __device__ cudaError_t

cudaStreamCreateWithFlags (cudaStream_t *pStream, unsigned int flags)

Create an asynchronous stream.

Parameters

pStream
- Pointer to new stream identifier

flags
- Parameters for stream creation

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Creates a new asynchronous stream. The flags argument determines the behaviors of the stream. Valid values for flags are

- cudaStreamDefault: Default stream creation flag.
- cudaStreamNonBlocking: Specifies that work running in the created stream may run concurrently with work in stream 0 (the NULL stream), and that the created stream should perform no implicit synchronization with stream 0.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
__host__ cudaError_t cudaStreamCreateWithPriority (cudaStream_t *pStream, unsigned int flags, int priority)
Create an asynchronous stream with the specified priority.

Parameters
pStream
- Pointer to new stream identifier
flags
- Flags for stream creation. See cudaStreamCreateWithFlags for a list of valid flags that can be passed
priority
- Priority of the stream. Lower numbers represent higher priorities. See cudaDeviceGetStreamPriorityRange for more information about the meaningful stream priorities that can be passed.

Returns
cudaSuccess, cudaErrorInvalidValue

Description
Creates a stream with the specified priority and returns a handle in pStream. This API alters the scheduler priority of work in the stream. Work in a higher priority stream may preempt work already executing in a low priority stream.

priority follows a convention where lower numbers represent higher priorities. '0' represents default priority. The range of meaningful numerical priorities can be queried using cudaDeviceGetStreamPriorityRange. If the specified priority is outside the numerical range returned by cudaDeviceGetStreamPriorityRange, it will automatically be clamped to the lowest or the highest number in the range.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.

Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

Stream priorities are supported only on GPUs with compute capability 3.5 or higher.

In the current implementation, only compute kernels launched in priority streams are affected by the stream’s priority. Stream priorities have no effect on host-to-device and device-to-host memory operations.

See also:


___

```c
__host____device__cudaError_t cudaStreamDestroy (cudaStream_t stream)
```

Destroys and cleans up an asynchronous stream.

**Parameters**

* `stream` - Stream identifier

**Returns**

* `cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`

**Description**

Destroys and cleans up the asynchronous stream specified by `stream`.

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

**Note:**

* This function uses standard `default stream` semantics.
* Note that this function may also return error codes from previous, asynchronous launches.
__host__ cudaError_t cudaStreamEndCapture (cudaStream_t stream, cudaGraph_t *pGraph)

Ends capture on a stream, returning the captured graph.

Parameters

stream
  - Stream to query

pGraph
  - The captured graph

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorStreamCaptureWrongThread

Description

End capture on stream, returning the captured graph via pGraph. Capture must have been initiated on stream via a call to cudaStreamBeginCapture. If capture was invalidated, due to a violation of the rules of stream capture, then a NULL graph will be returned.

If the mode argument to cudaStreamBeginCapture was not cudaStreamCaptureModeRelaxed, this call must be from the same thread as cudaStreamBeginCapture.

See also:

cudaStreamCreate, cudaStreamBeginCapture, cudaStreamIsCapturing

Note:

Note that this function may also return error codes from previous, asynchronous launches.
__host__ cudaError_t cudaStreamGetAttribute (cudaStream_t hStream, cudaStreamAttrID attr, cudaStreamAttrValue *value_out)

Queries stream attribute.

Parameters

hStream
attr
value_out

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle

Description

Queries attribute attr from hStream and stores it in corresponding member of value_out.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaAccessPolicyWindow

__host__ cudaError_t cudaStreamGetCaptureInfo (cudaStream_t stream, cudaStreamCaptureStatus *pCaptureStatus, unsigned long long *pId)

Query capture status of a stream.

Parameters

stream
pCaptureStatus
pId

- Stream to query
- Returns the stream’s capture status
- Returns the unique id of the capture sequence
Returns
cudaSuccess, cudaErrorStreamCaptureImplicit

Description
Note there is a later version of this API, cudaStreamGetCaptureInfo_v2. It will supplant this version in 12.0, which is retained for minor version compatibility.

Query the capture status of a stream and get a unique id representing the capture sequence over the lifetime of the process.

If called on cudaStreamLegacy (the "null stream") while a stream not created with cudaStreamNonBlocking is capturing, returns cudaErrorStreamCaptureImplicit.

A valid id is returned only if both of the following are true:

- the call returns cudaSuccess
- captureStatus is set to cudaStreamCaptureStatusActive

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaStreamGetCaptureInfo_v2, cudaStreamBeginCapture, cudaStreamIsCapturing

__host__cudaError_t cudaStreamGetCaptureInfo_v2 (cudaStream_t stream, cudaStreamCaptureStatus *captureStatus_out, unsigned long long *id_out, cudaGraph_t *graph_out, const cudaGraphNode_t **dependencies_out, size_t *numDependencies_out)
Query a stream’s capture state (11.3+).

Parameters

stream
- The stream to query
captureStatus_out
- Location to return the capture status of the stream; required
id_out
- Optional location to return an id for the capture sequence, which is unique over the lifetime of the process

graph_out
- Optional location to return the graph being captured into. All operations other than destroy and node removal are permitted on the graph while the capture sequence is in progress. This API does not transfer ownership of the graph, which is transferred or destroyed at cudaStreamEndCapture. Note that the graph handle may be invalidated before end of capture for certain errors. Nodes that are or become unreachable from the original stream at cudaStreamEndCapture due to direct actions on the graph do not trigger cudaErrorStreamCaptureUnjoined.
dependencies_out
- Optional location to store a pointer to an array of nodes. The next node to be captured in the stream will depend on this set of nodes, absent operations such as event wait which modify this set. The array pointer is valid until the next API call which operates on the stream or until end of capture. The node handles may be copied out and are valid until they or the graph is destroyed. The driver-owned array may also be passed directly to APIs that operate on the graph (not the stream) without copying.
numDependencies_out
- Optional location to store the size of the array returned in dependencies_out.

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorStreamCaptureImplicit

Description
Query stream state related to stream capture.
If called on cudaStreamLegacy (the “null stream”) while a stream not created with cudaStreamNonBlocking is capturing, returns cudaErrorStreamCaptureImplicit. Valid data (other than capture status) is returned only if both of the following are true:

- the call returns cudaSuccess
- the returned capture status is cudaStreamCaptureStatusActive

This version of cudaStreamGetCaptureInfo is introduced in CUDA 11.3 and will supplant the previous version cudaStreamGetCaptureInfo in 12.0. Developers requiring compatibility across minor versions to CUDA 11.0 (driver version 445) can do one of the following:

- Use the older version of the API, cudaStreamGetCaptureInfo
- Pass null for all of graph_out, dependencies_out, and numDependencies_out.

Note:
Graph objects are not threadsafe. [More here.]

Note that this function may also return error codes from previous, asynchronous launches.

See also:

`cudaStreamGetCaptureInfo`, `cudaStreamBeginCapture`, `cudaStreamIsCapturing`, `cudaStreamUpdateCaptureDependencies`

```c
__host__ cudaError_t cudaStreamGetFlags(cudaStream_t hStream, unsigned int *flags)
```

Query the flags of a stream.

**Parameters**

- `hStream` - Handle to the stream to be queried
- `flags` - Pointer to an unsigned integer in which the stream’s flags are returned

**Returns**

- `cudaSuccess`
- `cudaErrorInvalidValue`
- `cudaErrorInvalidResourceHandle`

**Description**

Query the flags of a stream. The flags are returned in `flags`. See `cudaStreamCreateWithFlags` for a list of valid flags.

**Note:**

- This function uses standard `default stream` semantics.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

`cudaStreamCreateWithPriority`, `cudaStreamCreateWithFlags`, `cudaStreamGetPriority`, `cuStreamGetFlags`
__host__ cudaError_t cudaStreamGetPriority (cudaStream_t hStream, int *priority)
Query the priority of a stream.

Parameters

**hStream**
- Handle to the stream to be queried

**priority**
- Pointer to a signed integer in which the stream’s priority is returned

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle

Description
Query the priority of a stream. The priority is returned in priority. Note that if the stream was created with a priority outside the meaningful numerical range returned by cudaDeviceGetStreamPriorityRange, this function returns the clamped priority. See cudaStreamCreateWithPriority for details about priority clamping.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaStreamCreateWithPriority, cudaDeviceGetStreamPriorityRange, cudaStreamGetFlags, cuStreamGetPriority
__host__ cudaError_t cudaStreamIsCapturing (cudaStream_t stream, cudaStreamCaptureStatus *pCaptureStatus)

Returns a stream’s capture status.

Parameters

stream
- Stream to query

pCaptureStatus
- Returns the stream’s capture status

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorStreamCaptureImplicit

Description

Return the capture status of stream via pCaptureStatus. After a successful call, *pCaptureStatus will contain one of the following:

- cudaStreamCaptureStatusNone: The stream is not capturing.
- cudaStreamCaptureStatusActive: The stream is capturing.
- cudaStreamCaptureStatusInvalidated: The stream was capturing but an error has invalidated the capture sequence. The capture sequence must be terminated with cudaStreamEndCapture on the stream where it was initiated in order to continue using stream.

Note that, if this is called on cudaStreamLegacy (the “null stream”) while a blocking stream on the same device is capturing, it will return cudaErrorStreamCaptureImplicit and *pCaptureStatus is unspecified after the call. The blocking stream capture is not invalidated.

When a blocking stream is capturing, the legacy stream is in an unusable state until the blocking stream capture is terminated. The legacy stream is not supported for stream capture, but attempted use would have an implicit dependency on the capturing stream[s].

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaStreamCreate, cudaStreamBeginCapture, cudaStreamEndCapture
__host__ cudaError_t cudaStreamQuery (cudaStream_t stream)
Queries an asynchronous stream for completion status.

Parameters

stream
- Stream identifier

Returns
cudaSuccess, cudaErrorNotReady, cudaErrorInvalidResourceHandle

Description
Returns cudaSuccess if all operations in stream have completed, or cudaErrorNotReady if not.
For the purposes of Unified Memory, a return value of cudaSuccess is equivalent to having called cudaStreamSynchronize().

Note:
- This function uses standard default stream semantics.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaStreamCreate, cudaStreamCreateWithFlags, cudaStreamWaitEvent, cudaStreamSynchronize, cudaStreamAddCallback, cudaStreamDestroy, cuStreamQuery
__host__ cudaError_t cudaStreamSetAttribute (cudaStream_t hStream, cudaStreamAttrID attr, const cudaStreamAttrValue *value)
Sets stream attribute.

Parameters
hStream
attr
tvalue

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle

Description
Sets attribute attr on hStream from corresponding attribute of value. The updated attribute will be applied to subsequent work submitted to the stream. It will not affect previously submitted work.

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaAccessPolicyWindow

__host__ cudaError_t cudaStreamSynchronize (cudaStream_t stream)
Waits for stream tasks to complete.

Parameters
stream

- Stream identifier

Returns
cudaSuccess, cudaErrorInvalidResourceHandle
Description
Blocks until stream has completed all operations. If the cudaDeviceScheduleBlockingSync flag was set for this device, the host thread will block until the stream is finished with all of its tasks.

Note:
- This function uses standard default stream semantics.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaStreamCreate, cudaStreamCreateWithFlags, cudaStreamQuery, cudaStreamWaitEvent, cudaStreamAddCallback, cudaStreamDestroy, cuStreamSynchronize

__host__cudaError_t cudaStreamUpdateCaptureDependencies (cudaStream_t stream, cudaGraphNode_t *dependencies, size_t numDependencies, unsigned int flags)
Update the set of dependencies in a capturing stream (11.3+).

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorIllegalState

Description
Modifies the dependency set of a capturing stream. The dependency set is the set of nodes that the next captured node in the stream will depend on.

Valid flags are cudaStreamAddCaptureDependencies and cudaStreamSetCaptureDependencies. These control whether the set passed to
the API is added to the existing set or replaces it. A flags value of 0 defaults to `cudaStreamAddCaptureDependencies`.

Nodes that are removed from the dependency set via this API do not result in `cudaErrorStreamCaptureUnjoined` if they are unreachable from the stream at `cudaStreamEndCapture`.

Returns `cudaErrorIllegalState` if the stream is not capturing.

This API is new in CUDA 11.3. Developers requiring compatibility across minor versions of the CUDA driver to 11.0 should not use this API or provide a fallback.

### Note:

Note that this function may also return error codes from previous, asynchronous launches.

### See also:

`cudaStreamBeginCapture`, `cudaStreamGetCaptureInfo`, `cudaStreamGetCaptureInfo_v2`

```c
__host__ __device__ cudaError_t
cudaStreamWaitEvent (cudaStream_t stream,
cudaEvent_t event, unsigned int flags)
```

Make a compute stream wait on an event.

#### Parameters

- **stream**
  - Stream to wait
- **event**
  - Event to wait on
- **flags**
  - Parameters for the operation (See above)

#### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`

#### Description

Makes all future work submitted to `stream` wait for all work captured in `event`. See `cudaEventRecord()` for details on what is captured by an event. The synchronization will be performed efficiently on the device when applicable. `event` may be from a different device than `stream`.

flags include:
CUDA Runtime API

Modules

- **cudaEventWaitDefault**: Default event creation flag.
- **cudaEventWaitExternal**: Event is captured in the graph as an external event node when performing stream capture.

**Note:**
- This function uses standard **default stream** semantics.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return **cudaErrorInitializationError**, **cudaErrorInsufficientDriver** or **cudaErrorNoDevice** if this call tries to initialize internal CUDA RT state.
- Note that as specified by **cudaStreamAddCallback** no CUDA function may be called from callback. **cudaErrorNotPermitted** may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**
cudaStreamCreate, cudaStreamCreateWithFlags, cudaStreamQuery, cudaStreamSynchronize, cudaStreamAddCallback, cudaStreamDestroy, cuStreamWaitEvent

```c
__host__ cudaError_t cudaThreadExchangeStreamCaptureMode (cudaStreamCaptureMode *mode)
```

Swaps the stream capture interaction mode for a thread.

**Parameters**

`mode`
- Pointer to mode value to swap with the current mode

**Returns**
cudaSuccess, cudaErrorInvalidValue

**Description**

Sets the calling thread’s stream capture interaction mode to the value contained in `mode`, and overwrites `mode` with the previous mode for the thread. To facilitate deterministic behavior across function or module boundaries, callers are encouraged to use this API in a push-pop fashion:

```c
cudaStreamCaptureMode mode = desiredMode;
cudaThreadExchangeStreamCaptureMode(&mode);
```
During stream capture (see \texttt{cudaStreamBeginCapture}), some actions, such as a call to \texttt{cudaMalloc}, may be unsafe. In the case of \texttt{cudaMalloc}, the operation is not enqueued asynchronously to a stream, and is not observed by stream capture. Therefore, if the sequence of operations captured via \texttt{cudaStreamBeginCapture} depended on the allocation being replayed whenever the graph is launched, the captured graph would be invalid.

Therefore, stream capture places restrictions on API calls that can be made within or concurrently to a \texttt{cudaStreamBeginCapture-cudaStreamEndCapture} sequence. This behavior can be controlled via this API and flags to \texttt{cudaStreamBeginCapture}.

A thread’s mode is one of the following:

- \texttt{cudaStreamCaptureModeGlobal}: This is the default mode. If the local thread has an ongoing capture sequence that was not initiated with \texttt{cudaStreamCaptureModeRelaxed} at \texttt{cuStreamBeginCapture}, or if any other thread has a concurrent capture sequence initiated with \texttt{cudaStreamCaptureModeGlobal}, this thread is prohibited from potentially unsafe API calls.

- \texttt{cudaStreamCaptureModeThreadLocal}: If the local thread has an ongoing capture sequence not initiated with \texttt{cudaStreamCaptureModeRelaxed}, it is prohibited from potentially unsafe API calls. Concurrent capture sequences in other threads are ignored.

- \texttt{cudaStreamCaptureModeRelaxed}: The local thread is not prohibited from potentially unsafe API calls. Note that the thread is still prohibited from API calls which necessarily conflict with stream capture, for example, attempting \texttt{cudaEventQuery} on an event that was last recorded inside a capture sequence.

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

\texttt{cudaStreamBeginCapture}

## 6.5. Event Management

This section describes the event management functions of the CUDA runtime application programming interface.
__host__cudaError_t cudaEventCreate (cudaEvent_t *event)

Creates an event object.

Parameters

event
- Newly created event

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorLaunchFailure, cudaErrorMemoryAllocation

Description

Creates an event object for the current device using cudaEventDefault.

Note:

▶ Note that this function may also return error codes from previous, asynchronous launches.
▶ Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
▶ Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaEventCreate [ C++ API], cudaEventCreateWithFlags, cudaEventRecord, cudaEventQuery, cudaEventSynchronize, cudaEventDestroy, cudaEventElapsedTime, cudaStreamWaitEvent, cuEventCreate

__host____device__cudaError_t cudaEventCreateWithFlags (cudaEvent_t *event, unsigned int flags)

Creates an event object with the specified flags.

Parameters

event
- Newly created event

flags
- Flags for new event

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorLaunchFailure, cudaErrorMemoryAllocation

Description
Creates an event object for the current device with the specified flags. Valid flags include:

- **cudaEventDefault**: Default event creation flag.
- **cudaEventBlockingSync**: Specifies that event should use blocking synchronization. A host thread that uses `cudaEventSynchronize()` to wait on an event created with this flag will block until the event actually completes.
- **cudaEventDisableTiming**: Specifies that the created event does not need to record timing data. Events created with this flag specified and the `cudaEventBlockingSync` flag not specified will provide the best performance when used with `cudaStreamWaitEvent()` and `cudaEventQuery()`.
- **cudaEventInterprocess**: Specifies that the created event may be used as an interprocess event by `cudaIpcGetEventHandle()`. `cudaEventInterprocess` must be specified along with `cudaEventDisableTiming`.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaEventCreate [ C API], cudaEventSynchronize, cudaEventDestroy, cudaEventElapsedTime, cudaStreamWaitEvent, cuEventCreate
__host__ __device__ cudaError_t cudaEventDestroy (cudaEvent_t event)

Destroys an event object.

Parameters

**event**
- Event to destroy

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorLaunchFailure

Description

Destroys the event specified by `event`.

An event may be destroyed before it is complete (i.e., while `cudaEventQuery()` would return cudaErrorNotReady). In this case, the call does not block on completion of the event, and any associated resources will automatically be released asynchronously at completion.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
- Use of the handle after this call is undefined behavior.
- Returns cudaErrorInvalidResourceHandle in the event of being passed NULL as the input event.

See also:

cudaEventCreate [ C API], cudaEventCreateWithFlags, cudaEventQuery, cudaEventSynchronize, cudaEventRecord, cudaEventElapsedTime, cuEventDestroy
__host__ cudaError_t cudaEventElapsedTime (float *ms, cudaEvent_t start, cudaEvent_t end)

Computes the elapsed time between events.

Parameters

**ms**
- Time between *start* and *end* in ms

**start**
- Starting event

**end**
- Ending event

Returns

cudaSuccess, cudaErrorNotReady, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorLaunchFailure

Description

Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds).

If either event was last recorded in a non-NULL stream, the resulting time may be greater than expected (even if both used the same stream handle). This happens because the cudaEventRecord() operation takes place asynchronously and there is no guarantee that the measured latency is actually just between the two events. Any number of other different stream operations could execute in between the two measured events, thus altering the timing in a significant way.

If cudaEventRecord[] has not been called on either event, then cudaErrorInvalidResourceHandle is returned. If cudaEventRecord[] has been called on both events but one or both of them has not yet been completed [that is, cudaEventQuery[] would return cudaErrorNotReady on at least one of the events], cudaErrorNotReady is returned. If either event was created with the cudaEventDisableTiming flag, then this function will return cudaErrorInvalidResourceHandle.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback`, no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

Returns `cudaErrorInvalidResourceHandle` in the event of being passed NULL as the input event.

See also:

`cudaEventCreate (C API)`, `cudaEventCreateWithFlags`, `cudaEventQuery`, `cudaEventSynchronize`, `cudaEventDestroy`, `cudaEventRecord`, `cuEventElapsedTime`

```c
__host__cudaError_t cudaEventQuery (cudaEvent_t event)
```

Queries an event’s status.

**Parameters**

**event**

- Event to query

**Returns**

`cudaSuccess`, `cudaErrorNotReady`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`, `cudaErrorLaunchFailure`

**Description**

Queries the status of all work currently captured by `event`. See `cudaEventRecord()` for details on what is captured by an event.

Returns `cudaSuccess` if all captured work has been completed, or `cudaErrorNotReady` if any captured work is incomplete.

For the purposes of Unified Memory, a return value of `cudaSuccess` is equivalent to having called `cudaEventSynchronize()`.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

Returns `cudaErrorInvalidResourceHandle` in the event of being passed NULL as the input event.

See also:
`cudaEventCreate` [C API], `cudaEventCreateWithFlags`, `cudaEventRecord`, `cudaEventSynchronize`, `cudaEventDestroy`, `cudaEventElapsedTime`, `cuEventQuery`

```c
__host__ __device__ cudaError_t cudaEventRecord(cudaEvent_t event, cudaStream_t stream)
```
Records an event.

**Parameters**

- **event**
  - Event to record
- **stream**
  - Stream in which to record event

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`, `cudaErrorLaunchFailure`

**Description**

Captures in `event` the contents of `stream` at the time of this call. `event` and `stream` must be on the same device. Calls such as `cudaEventQuery()` or `cudaStreamWaitEvent()` will then examine or wait for completion of the work that was captured. Uses of `stream` after this call do not modify `event`. See note on default stream behavior for what is captured in the default case.

`cudaEventRecord()` can be called multiple times on the same event and will overwrite the previously captured state. Other APIs such as `cudaStreamWaitEvent()` use the most recently captured state at the time of the API call, and are not affected by later calls to `cudaEventRecord()`. Before the first call to `cudaEventRecord()`, an event represents an empty set of work, so for example `cudaEventQuery()` would return `cudaSuccess`.

**Note:**

- This function uses standard `default` stream semantics.
See also:
cudaEventCreate (C API), cudaEventCreateWithFlags, cudaEventQuery, cudaEventSynchronize, cudaEventDestroy, cudaEventElapsedTime, cudaStreamWaitEvent, cudaEventRecordWithFlags, cuEventRecord

__host__ cudaError_t cudaEventRecordWithFlags (cudaEvent_t event, cudaStream_t stream, unsigned int flags)

Records an event.

Parameters

event
- Event to record

stream
- Stream in which to record event

flags
- Parameters for the operation [See above]

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorLaunchFailure

Description

Captures in event the contents of stream at the time of this call. event and stream must be on the same device. Calls such as cudaEventQuery() or cudaStreamWaitEvent() will then examine or wait for completion of the work that was captured. Uses of stream after this call do not modify event. See note on default stream behavior for what is captured in the default case.
cudaEventRecordWithFlags() can be called multiple times on the same event and will overwrite the previously captured state. Other APIs such as cudaStreamWaitEvent() use the most recently captured state at the time of the API call, and are not affected by later calls to cudaEventRecordWithFlags(). Before the first call to cudaEventRecordWithFlags(), an event represents an empty set of work, so for example cudaEventQuery() would return cudaSuccess.

flags include:
- **cudaEventRecordDefault**: Default event creation flag.
- **cudaEventRecordExternal**: Event is captured in the graph as an external event node when performing stream capture.

**Note:**
- This function uses standard default stream semantics.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
- Returns cudaErrorInvalidResourceHandle in the event of being passed NULL as the input event.

See also:
cudaEventCreate (C API), cudaEventCreateWithFlags, cudaEventQuery, cudaEventSynchronize, cudaEventDestroy, cudaEventElapsedTime, cudaStreamWaitEvent, cudaEventRecord, cuEventRecord,

```c
__host__ cudaError_t cudaEventSynchronize (cudaEvent_t event)
```

Waits for an event to complete.

**Parameters**

**event**
- Event to wait for
Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorLaunchFailure

Description

Waits until the completion of all work currently captured in event. See cudaEventRecord() for details on what is captured by an event.

Waiting for an event that was created with the cudaEventBlockingSync flag will cause the calling CPU thread to block until the event has been completed by the device. If the cudaEventBlockingSync flag has not been set, then the CPU thread will busy-wait until the event has been completed by the device.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
- Returns cudaErrorInvalidResourceHandle in the event of being passed NULL as the input event.

See also:
cudaEventCreate (C API), cudaEventCreateWithFlags, cudaEventRecord, cudaEventQuery, cudaEventDestroy, cudaEventElapsedTime, cuEventSynchronize

6.6. External Resource Interoperability

This section describes the external resource interoperability functions of the CUDA runtime application programming interface.
__host__ cudaError_t cudaDestroyExternalMemory (cudaExternalMemory_t extMem)

Destroys an external memory object.

Parameters
extMem
   - External memory object to be destroyed

Returns
cudaSuccess, cudaErrorInvalidResourceHandle

Description
Destroys the specified external memory object. Any existing buffers and CUDA mipmapped arrays mapped onto this object must no longer be used and must be explicitly freed using cudaFree and cudaFreeMipmappedArray respectively.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
- Use of the handle after this call is undefined behavior.

See also:
cudaImportExternalMemory, cudaExternalMemoryGetMappedBuffer, cudaExternalMemoryGetMappedMipmappedArray
__host__ cudaError_t
cudaDestroyExternalSemaphore
cudaExternalSemaphore_t extSem)

Destroys an external semaphore.

Parameters

extSem
- External semaphore to be destroyed

Returns

cudaSuccess, cudaErrorInvalidResourceHandle

Description

Destroys an external semaphore object and releases any references to the underlying resource. Any outstanding signals or waits must have completed before the semaphore is destroyed.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
- Use of the handle after this call is undefined behavior.

See also:

cudainportExternalSemaphore, cudaSignalExternalSemaphoresAsync, cudaWaitExternalSemaphoresAsync
__host__ cudaError_t
cudaExternalMemoryGetMappedBuffer (void **devPtr, cudaExternalMemory_t extMem, const
cudaExternalMemoryBufferDesc *bufferDesc)

Maps a buffer onto an imported memory object.

Parameters

devPtr
- Returned device pointer to buffer

extMem
- Handle to external memory object

bufferDesc
- Buffer descriptor

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle

Description

Maps a buffer onto an imported memory object and returns a device pointer in devPtr.

The properties of the buffer being mapped must be described in bufferDesc. The
cudaExternalMemoryBufferDesc structure is defined as follows:

```
typedef struct cudaExternalMemoryBufferDesc_st {
    unsigned long long offset;
    unsigned long long size;
    unsigned int flags;
} cudaExternalMemoryBufferDesc;
```

where cudaExternalMemoryBufferDesc::offset is the offset in the memory object where
the buffer’s base address is. cudaExternalMemoryBufferDesc::size is the size of the buffer.
cudaExternalMemoryBufferDesc::flags must be zero.

The offset and size have to be suitably aligned to match the requirements of the external API.
Mapping two buffers whose ranges overlap may or may not result in the same virtual address
being returned for the overlapped portion. In such cases, the application must ensure that all
accesses to that region from the GPU are volatile. Otherwise writes made via one address are
not guaranteed to be visible via the other address, even if they’re issued by the same thread.
It is recommended that applications map the combined range instead of mapping separate
buffers and then apply the appropriate offsets to the returned pointer to derive the individual
buffers.

The returned pointer devPtr must be freed using cudaFree.
Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

- `cudaImportExternalMemory`
- `cudaDestroyExternalMemory`
- `cudaExternalMemoryGetMappedMipmappedArray`

```c
__host__ cudaError_t
cudaExternalMemoryGetMappedMipmappedArray
(cudaMipmappedArray_t *mipmap,
cudaExternalMemory_t extMem, const
cudaExternalMemoryMipmappedArrayDesc
*mipmapDesc)
```

Maps a CUDA mipmapped array onto an external memory object.

**Parameters**

- **mipmap**
  - Returned CUDA mipmapped array
- **extMem**
  - Handle to external memory object
- **mipmapDesc**
  - CUDA array descriptor

**Returns**

- `cudaSuccess`
- `cudaErrorInvalidValue`
- `cudaErrorInvalidResourceHandle`

**Description**

Maps a CUDA mipmapped array onto an external object and returns a handle to it in `mipmap`. 
The properties of the CUDA mipmapped array being mapped must be described in `mipmapDesc`. The structure `cudaExternalMemoryMipmappedArrayDesc` is defined as follows:

```c
typedef struct cudaExternalMemoryMipmappedArrayDesc_st {
    unsigned long long offset;
    cudaChannelFormatDesc formatDesc;
    cudaExtent extent;
    unsigned int flags;
    unsigned int numLevels;
} cudaExternalMemoryMipmappedArrayDesc;
```

where `cudaExternalMemoryMipmappedArrayDesc::offset` is the offset in the memory object where the base level of the mipmap chain is. `cudaExternalMemoryMipmappedArrayDesc::formatDesc` describes the format of the data. `cudaExternalMemoryMipmappedArrayDesc::extent` specifies the dimensions of the base level of the mipmap chain. `cudaExternalMemoryMipmappedArrayDesc::flags` are flags associated with CUDA mipmapped arrays. For further details, please refer to the documentation for `cudaMalloc3DArray`. Note that if the mipmapped array is bound as a color target in the graphics API, then the flag `cudaArrayColorAttachment` must be specified in `cudaExternalMemoryMipmappedArrayDesc::flags`. `cudaExternalMemoryMipmappedArrayDesc::numLevels` specifies the total number of levels in the mipmap chain.

The returned CUDA mipmapped array must be freed using `cudaFreeMipmappedArray`.

### Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

### See also:
- `cudaImportExternalMemory`, `cudaDestroyExternalMemory`,
- `cudaExternalMemoryGetMappedBuffer`

### Note:
If `cudaExternalMemoryHandleDesc::type` is `cudaExternalMemoryHandleTypeNvSciBuf`, then `cudaExternalMemoryMipmappedArrayDesc::numLevels` must not be greater than 1.
__host__ cudaError_t cudaImportExternalMemory
(cudaExternalMemory_t *extMem_out, const
cudaExternalMemoryHandleDesc *memHandleDesc)

Imports an external memory object.

Parameters

**extMem_out**
- Returned handle to an external memory object

**memHandleDesc**
- Memory import handle descriptor

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle

Description

Imports an externally allocated memory object and returns a handle to that in extMem_out.

The properties of the handle being imported must be described in memHandleDesc. The
**cudaExternalMemoryHandleDesc** structure is defined as follows:

```c
typedef struct cudaExternalMemoryHandleDesc_st {
    cudaExternalMemoryHandleType type;
    union {
        int fd;
        struct {
            void *handle;
            const void *name;
        } win32;
        const void *nvSciBufObject;
    } handle;
    unsigned long long size;
    unsigned int flags;
} cudaExternalMemoryHandleDesc;
```

where **cudaExternalMemoryHandleDesc::type** specifies the type of handle being imported.
**cudaExternalMemoryHandleType** is defined as:

```c
typedef enum cudaExternalMemoryHandleType_enum {
    cudaExternalMemoryHandleTypeOpaqueFd = 1,
    cudaExternalMemoryHandleTypeOpaqueWin32 = 2,
    cudaExternalMemoryHandleTypeOpaqueWin32Kmt = 3,
    cudaExternalMemoryHandleTypeD3D12Heap = 4,
    cudaExternalMemoryHandleTypeD3D12Resource = 5,
    cudaExternalMemoryHandleTypeD3D11Resource = 6,
    cudaExternalMemoryHandleTypeD3D11ResourceKmt = 7,
    cudaExternalMemoryHandleTypeNvSciBuf = 8
} cudaExternalMemoryHandleType;
```

If **cudaExternalMemoryHandleDesc::type** is **cudaExternalMemoryHandleTypeOpaqueFd**, then
cudaExternalMemoryHandleDesc::handle::fd must be a valid file descriptor referencing a
memory object. Ownership of the file descriptor is transferred to the CUDA driver when the handle is imported successfully. Performing any operations on the file descriptor after it is imported results in undefined behavior.

If `cudaExternalMemoryHandleDesc::type` is `cudaExternalMemoryHandleTypeOpaqueWin32`, then exactly one of `cudaExternalMemoryHandleDesc::handle::win32::handle` and `cudaExternalMemoryHandleDesc::handle::win32::name` must not be NULL. If `cudaExternalMemoryHandleDesc::handle::win32::handle` is not NULL, then it must represent a valid shared NT handle that references a memory object. Ownership of this handle is not transferred to CUDA after the import operation, so the application must release the handle using the appropriate system call. If `cudaExternalMemoryHandleDesc::handle::win32::name` is not NULL, then it must point to a NULL-terminated array of UTF-16 characters that refers to a memory object.

If `cudaExternalMemoryHandleDesc::type` is `cudaExternalMemoryHandleTypeOpaqueWin32Kmt`, then `cudaExternalMemoryHandleDesc::handle::win32::handle` must be non-NULL and `cudaExternalMemoryHandleDesc::handle::win32::name` must be NULL. The handle specified must be a globally shared KMT handle. This handle does not hold a reference to the underlying object, and thus will be invalid when all references to the memory object are destroyed.

If `cudaExternalMemoryHandleDesc::type` is `cudaExternalMemoryHandleTypeD3D12Heap`, then exactly one of `cudaExternalMemoryHandleDesc::handle::win32::handle` and `cudaExternalMemoryHandleDesc::handle::win32::name` must not be NULL. If `cudaExternalMemoryHandleDesc::handle::win32::handle` is not NULL, then it must represent a valid shared NT handle that is returned by `ID3D12Device::CreateSharedHandle` when referring to a `ID3D12Heap` object. This handle holds a reference to the underlying object. If `cudaExternalMemoryHandleDesc::handle::win32::name` is not NULL, then it must point to a NULL-terminated array of UTF-16 characters that refers to a `ID3D12Heap` object.

If `cudaExternalMemoryHandleDesc::type` is `cudaExternalMemoryHandleTypeD3D12Resource`, then exactly one of `cudaExternalMemoryHandleDesc::handle::win32::handle` and `cudaExternalMemoryHandleDesc::handle::win32::name` must not be NULL. If `cudaExternalMemoryHandleDesc::handle::win32::handle` is not NULL, then it must represent a valid shared NT handle that is returned by `ID3D12Device::CreateSharedHandle` when referring to a `ID3D12Resource` object. This handle holds a reference to the underlying object. If `cudaExternalMemoryHandleDesc::handle::win32::name` is not NULL, then it must point to a NULL-terminated array of UTF-16 characters that refers to a `ID3D12Resource` object.

If `cudaExternalMemoryHandleDesc::type` is `cudaExternalMemoryHandleTypeD3D11Resource`, then exactly one of `cudaExternalMemoryHandleDesc::handle::win32::handle` and `cudaExternalMemoryHandleDesc::handle::win32::name` must not be NULL. If `cudaExternalMemoryHandleDesc::handle::win32::handle` is not NULL, then it must represent a valid shared NT handle that is returned by
IDXGIResource1::CreateSharedHandle when referring to a ID3D11Resource object. If cudaExternalMemoryHandleDesc::handle::win32::name is not NULL, then it must point to a NULL-terminated array of UTF-16 characters that refers to a ID3D11Resource object.

If cudaExternalMemoryHandleDesc::type is cudaExternalMemoryHandleTypeD3D11ResourceKmt, then cudaExternalMemoryHandleDesc::handle::win32::handle must be non-NULL and cudaExternalMemoryHandleDesc::handle::win32::name must be NULL. The handle specified must be a valid shared KMT handle that is returned by IDXGIResource::GetSharedHandle when referring to a ID3D11Resource object.

If cudaExternalMemoryHandleDesc::type is cudaExternalMemoryHandleTypeNvSciBuf, then cudaExternalMemoryHandleDesc::handle::nvSciBufObject must be NON-NULL and reference a valid NvSciBuf object. If the NvSciBuf object imported into CUDA is also mapped by other drivers, then the application must use cudaWaitExternalSemaphoresAsync or cudaSignalExternalSemaphoresAsync as appropriate barriers to maintain coherence between CUDA and the other drivers. See cudaExternalSemaphoreWaitSkipNvSciBufMemSync and cudaExternalSemaphoreSignalSkipNvSciBufMemSync for memory synchronization.

The size of the memory object must be specified in cudaExternalMemoryHandleDesc::size.

Specifying the flag cudaExternalMemoryDedicated in cudaExternalMemoryHandleDesc::flags indicates that the resource is a dedicated resource. The definition of what a dedicated resource is outside the scope of this extension. This flag must be set if cudaExternalMemoryHandleDesc::type is one of the following: cudaExternalMemoryHandleTypeD3D12Resource cudaExternalMemoryHandleTypeD3D11Resource cudaExternalMemoryHandleTypeD3D11ResourceKmt

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
- If the Vulkan memory imported into CUDA is mapped on the CPU then the application must use vkInvalidateMappedMemoryRanges/vkFlushMappedMemoryRanges as well as appropriate Vulkan pipeline barriers to maintain coherence between CPU and GPU. For more information on these APIs, please refer to "Synchronization and Cache Control" chapter from Vulkan specification.
__host__ cudaError_t cudaImportExternalSemaphore (cudaExternalSemaphore_t *extSem_out, const cudaExternalSemaphoreHandleDesc *semHandleDesc)
Imports an external semaphore.

Parameters

extSem_out
- Returned handle to an external semaphore
semHandleDesc
- Semaphore import handle descriptor

Returns
cudaSuccess, cudaErrorInvalidResourceHandle

Description
Imports an externally allocated synchronization object and returns a handle to that in extSem_out.

The properties of the handle being imported must be described in semHandleDesc. The cudaExternalSemaphoreHandleDesc is defined as follows:

```c
typedef struct cudaExternalSemaphoreHandleDesc_st {
    cudaExternalSemaphoreHandleType type;
    union {
        int fd;
        struct {
            void *handle;
            const void *name;
        } win32;
        const void* NvSciSyncObj;
    } handle;
    unsigned int flags;
} cudaExternalSemaphoreHandleDesc;
```

where cudaExternalSemaphoreHandleDesc::type specifies the type of handle being imported. cudaExternalSemaphoreHandleType is defined as:

```c
typedef enum cudaExternalSemaphoreHandleType_enum {
    cudaExternalSemaphoreHandleTypeOpaqueFd = 1,
    cudaExternalSemaphoreHandleTypeOpaqueWin32 = 2,
    cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt = 3,
    cudaExternalSemaphoreHandleTypeD3D12Fence = 4,
    cudaExternalSemaphoreHandleTypeD3D11Fence = 5,
} cudaExternalSemaphoreHandleType;
```
If \texttt{cudaExternalSemaphoreHandleDesc::type} is \texttt{cudaExternalSemaphoreHandleTypeOpaqueFd}, then \texttt{cudaExternalSemaphoreHandleDesc::handle::fd} must be a valid file descriptor referencing a synchronization object. Ownership of the file descriptor is transferred to the CUDA driver when the handle is imported successfully. Performing any operations on the file descriptor after it is imported results in undefined behavior.

If \texttt{cudaExternalSemaphoreHandleDesc::type} is \texttt{cudaExternalSemaphoreHandleTypeOpaqueWin32}, then exactly one of \texttt{cudaExternalSemaphoreHandleDesc::handle::win32::handle} and \texttt{cudaExternalSemaphoreHandleDesc::handle::win32::name} must not be NULL. If \texttt{cudaExternalSemaphoreHandleDesc::handle::win32::handle} is not NULL, then it must represent a valid shared NT handle that references a synchronization object. Ownership of this handle is not transferred to CUDA after the import operation, so the application must release the handle using the appropriate system call. If \texttt{cudaExternalSemaphoreHandleDesc::handle::win32::name} is not NULL, then it must name a valid synchronization object.

If \texttt{cudaExternalSemaphoreHandleDesc::type} is \texttt{cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt}, then \texttt{cudaExternalSemaphoreHandleDesc::handle::win32::handle} must be non-NULL and \texttt{cudaExternalSemaphoreHandleDesc::handle::win32::name} must be NULL. The handle specified must be a globally shared KMT handle. This handle does not hold a reference to the underlying object, and thus will be invalid when all references to the synchronization object are destroyed.

If \texttt{cudaExternalSemaphoreHandleDesc::type} is \texttt{cudaExternalSemaphoreHandleTypeD3D12Fence}, then exactly one of \texttt{cudaExternalSemaphoreHandleDesc::handle::win32::handle} and \texttt{cudaExternalSemaphoreHandleDesc::handle::win32::name} must not be NULL. If \texttt{cudaExternalSemaphoreHandleDesc::handle::win32::handle} is not NULL, then it must represent a valid shared NT handle that is returned by \texttt{ID3D12Device::CreateSharedHandle} when referring to a ID3D12Fence object. This handle holds a reference to the underlying object. If \texttt{cudaExternalSemaphoreHandleDesc::handle::win32::name} is not NULL, then it must name a valid synchronization object that refers to a valid ID3D12Fence object.

If \texttt{cudaExternalSemaphoreHandleDesc::type} is \texttt{cudaExternalSemaphoreHandleTypeD3D11Fence}, then exactly one of \texttt{cudaExternalSemaphoreHandleDesc::handle::win32::handle} and \texttt{cudaExternalSemaphoreHandleDesc::handle::win32::name} must not be NULL. If \texttt{cudaExternalSemaphoreHandleDesc::handle::win32::handle} is not NULL, then it must represent a valid shared NT handle that is returned by \texttt{ID3D11Fence::CreateSharedHandle}. If
cudaExternalSemaphoreHandleDesc::handle::win32::name is not NULL, then it must name a valid synchronization object that refers to a valid ID3D11Fence object.

If `cudaExternalSemaphoreHandleDesc::type` is `cudaExternalSemaphoreHandleTypeNvSciSync`, then `cudaExternalSemaphoreHandleDesc::handle::nvSciSyncObj` represents a valid NvSciSyncObj.

`cudaExternalSemaphoreHandleTypeKeyedMutex`, then exactly one of `cudaExternalSemaphoreHandleDesc::handle::win32::handle` and `cudaExternalSemaphoreHandleDesc::handle::win32::name` must not be NULL. If `cudaExternalSemaphoreHandleDesc::handle::win32::handle` is not NULL, then it represent a valid shared NT handle that is returned by `IDXGIResource1::CreateSharedHandle` when referring to a IDXGIKeyedMutex object.

If `cudaExternalSemaphoreHandleDesc::type` is `cudaExternalSemaphoreHandleTypeKeyedMutexKmt`, then `cudaExternalSemaphoreHandleDesc::handle::win32::handle` must be non-NULL and `cudaExternalSemaphoreHandleDesc::handle::win32::name` must be NULL. The handle specified must represent a valid KMT handle that is returned by `IDXGIResource::GetSharedHandle` when referring to a IDXGIKeyedMutex object.

If `cudaExternalSemaphoreHandleDesc::type` is `cudaExternalSemaphoreHandleTypeTimelineSemaphoreFd`, then `cudaExternalSemaphoreHandleDesc::handle::fd` must be a valid file descriptor referencing a synchronization object. Ownership of the file descriptor is transferred to the CUDA driver when the handle is imported successfully. Performing any operations on the file descriptor after it is imported results in undefined behavior.

If `cudaExternalSemaphoreHandleDesc::type` is `cudaExternalSemaphoreHandleTypeTimelineSemaphoreWin32`, then exactly one of `cudaExternalSemaphoreHandleDesc::handle::win32::handle` and `cudaExternalSemaphoreHandleDesc::handle::win32::name` must not be NULL. If `cudaExternalSemaphoreHandleDesc::handle::win32::handle` is not NULL, then it must represent a valid shared NT handle that references a synchronization object. Ownership of this handle is not transferred to CUDA after the import operation, so the application must release the handle using the appropriate system call. If `cudaExternalSemaphoreHandleDesc::handle::win32::name` is not NULL, then it must name a valid synchronization object.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback`, no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**
- `cudaDestroyExternalSemaphore`
- `cudaSignalExternalSemaphoresAsync`
- `cudaWaitExternalSemaphoresAsync`

```c
__host__ cudaError_t cudaSignalExternalSemaphoresAsync (const cudaExternalSemaphore_t *extSemArray, const cudaExternalSemaphoreSignalParams *paramsArray, unsigned int numExtSems, cudaStream_t stream)
```

Signals a set of external semaphore objects.

**Parameters**

- **extSemArray**
  - Set of external semaphores to be signaled
- **paramsArray**
  - Array of semaphore parameters
- **numExtSems**
  - Number of semaphores to signal
- **stream**
  - Stream to enqueue the signal operations in

**Returns**

- `cudaSuccess`
- `cudaErrorInvalidResourceHandle`

**Description**

Enqueues a signal operation on a set of externally allocated semaphore object in the specified stream. The operations will be executed when all prior operations in the stream complete.

The exact semantics of signaling a semaphore depends on the type of the object.

If the semaphore object is any one of the following types:
- `cudaExternalSemaphoreHandleTypeOpaqueFd`
- `cudaExternalSemaphoreHandleTypeOpaqueWin32`
- `cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt`

then signaling the semaphore will set it to the signaled state.
If the semaphore object is any one of the following types: `cudaExternalSemaphoreHandleTypeD3D12Fence`, `cudaExternalSemaphoreHandleTypeD3D11Fence`, `cudaExternalSemaphoreHandleTypeTimelineSemaphoreFd`, `cudaExternalSemaphoreHandleTypeTimelineSemaphoreWin32` then the semaphore will be set to the value specified in `cudaExternalSemaphoreSignalParams::params::fence::value`.

If the semaphore object is of the type `cudaExternalSemaphoreHandleTypeNvSciSync` this API sets `cudaExternalSemaphoreSignalParams::params::nvSciSync::fence` to a value that can be used by subsequent waiters of the same NvSciSync object to order operations with those currently submitted in stream. Such an update will overwrite previous contents of `cudaExternalSemaphoreSignalParams::params::nvSciSync::fence`. By default, signaling such an external semaphore object causes appropriate memory synchronization operations to be performed over all the external memory objects that are imported as `cudaExternalMemoryHandleTypeNvSciBuf`. This ensures that any subsequent accesses made by other importers of the same set of NvSciBuf memory object(s) are coherent. These operations can be skipped by specifying the flag `cudaExternalSemaphoreSignalSkipNvSciBufMemSync`, which can be used as a performance optimization when data coherency is not required. But specifying this flag in scenarios where data coherency is required results in undefined behavior. Also, for semaphore object of the type `cudaExternalSemaphoreHandleTypeNvSciSync`, if the NvSciSyncAttrList used to create the NvSciSyncObj had not set the flags in `cudaDeviceGetNvSciSyncAttributes` to `cudaNvSciSyncAttrSignal`, this API will return `cudaErrorNotSupported`.

If the semaphore object is any one of the following types: `cudaExternalSemaphoreHandleTypeKeyedMutex`, `cudaExternalSemaphoreHandleTypeKeyedMutexKmt`, then the keyed mutex will be released with the key specified in `cudaExternalSemaphoreSignalParams::params::keyedmutex::key`.

---

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**

- `cudaImportExternalSemaphore`
- `cudaDestroyExternalSemaphore`
- `cudaWaitExternalSemaphoresAsync`
__host__ cudaError_t
__host__ cudaError_t

__host__ cudaError_t

cudaWaitExternalSemaphoresAsync (const
cudaExternalSemaphore_t *extSemArray, const
cudaExternalSemaphoreWaitParams *paramsArray,
unsigned int numExtSems, cudaStream_t stream)

Waits on a set of external semaphore objects.

Parameters

extSemArray
- External semaphores to be waited on

paramsArray
- Array of semaphore parameters

numExtSems
- Number of semaphores to wait on

stream
- Stream to enqueue the wait operations in

Returns

cudaSuccess, cudaErrorInvalidResourceHandle cudaErrorTimeout

Description

Enqueues a wait operation on a set of externally allocated semaphore object in the specified
stream. The operations will be executed when all prior operations in the stream complete.

The exact semantics of waiting on a semaphore depends on the type of the object.

If the semaphore object is any one of the following types:
cudaExternalSemaphoreHandleTypeOpaqueFd,
cudaExternalSemaphoreHandleTypeOpaqueWin32,
cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt then waiting on the semaphore will
wait until the semaphore reaches the signaled state. The semaphore will then be reset to the
unsignaled state. Therefore for every signal operation, there can only be one wait operation.

If the semaphore object is any one of the following types:
cudaExternalSemaphoreHandleTypeD3D12Fence,
cudaExternalSemaphoreHandleTypeD3D11Fence,
cudaExternalSemaphoreHandleTypeTimelineSemaphoreFd,
cudaExternalSemaphoreHandleTypeTimelineSemaphoreWin32 then waiting on the
semaphore will wait until the value of the semaphore is greater than or equal to
cudaExternalSemaphoreWaitParams::params::fence::value.
If the semaphore object is of the type `cudaExternalSemaphoreHandleTypeNvSciSync` then, waiting on the semaphore will wait until the `cudaExternalSemaphoreSignalParams::params::nvSciSync::fence` is signaled by the signaler of the NvSciSyncObj that was associated with this semaphore object. By default, waiting on such an external semaphore object causes appropriate memory synchronization operations to be performed over all external memory objects that are imported as `cudaExternalMemoryHandleTypeNvSciBuf`. This ensures that any subsequent accesses made by other importers of the same set of NvSciBuf memory object(s) are coherent. These operations can be skipped by specifying the flag `cudaExternalSemaphoreWaitSkipNvSciBufMemSync`, which can be used as a performance optimization when data coherency is not required. But specifying this flag in scenarios where data coherency is required results in undefined behavior. Also, for semaphore object of the type `cudaExternalSemaphoreHandleTypeNvSciSync`, if the NvSciSyncAttrList used to create the NvSciSyncObj had not set the flags in `cudaDeviceGetNvSciSyncAttributes` to `cudaNvSciSyncAttrWait`, this API will return `cudaErrorNotSupported`.

If the semaphore object is any one of the following types: `cudaExternalSemaphoreHandleTypeKeyedMutex`, `cudaExternalSemaphoreHandleTypeKeyedMutexKmt`, then the keyed mutex will be acquired when it is released with the key specified in `cudaExternalSemaphoreSignalParams::params::keyedmutex::key` or until the timeout specified by `cudaExternalSemaphoreSignalParams::params::keyedmutex::timeoutMs` has lapsed. The timeout interval can either be a finite value specified in milliseconds or an infinite value. In case an infinite value is specified the timeout never elapses. The windows INFINITE macro must be used to specify infinite timeout.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**

- `cudaImportExternalSemaphore`, `cudaDestroyExternalSemaphore`,
- `cudaSignalExternalSemaphoresAsync`
6.7. Execution Control

This section describes the execution control functions of the CUDA runtime application programming interface.

Some functions have overloaded C++ API template versions documented separately in the C++ API Routines module.

`__host___device__cudaError_t cudaFuncGetAttributes (cudaFuncAttributes *attr, const void *func)`

Find out attributes for a given function.

**Parameters**

- `attr` - Return pointer to function’s attributes
- `func` - Device function symbol

**Returns**

`cudaSuccess, cudaErrorInvalidDeviceFunction`

**Description**

This function obtains the attributes of a function specified via `func`. `func` is a device function symbol and must be declared as a `__global__` function. The fetched attributes are placed in `attr`. If the specified function does not exist, then `cudaErrorInvalidDeviceFunction` is returned. For templated functions, pass the function symbol as follows: `func_name<template_arg_0,...,template_arg_N>`

Note that some function attributes such as `maxThreadsPerBlock` may vary based on the device that is currently being used.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Use of a string naming a function as the `func` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.
Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.

Note that as specified by cudaMemcpyAsync no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaFuncSetCacheConfig (C API), cudaFuncGetAttributes (C++ API), cudaLaunchKernel (C API), cuFuncGetAttribute

__host__cudaError_t cudaFuncSetAttribute (const void *func, cudaFuncAttribute attr, int value)
Set attributes for a given function.

Parameters

func
- Function to get attributes of

attr
- Attribute to set

value
- Value to set

Returns
cudaSuccess, cudaErrorInvalidDeviceFunction, cudaErrorInvalidValue

Description

This function sets the attributes of a function specified via func. The parameter func must be a pointer to a function that executes on the device. The parameter specified by func must be declared as a __global__ function. The enumeration defined by attr is set to the value defined by value. If the specified function does not exist, then cudaErrorInvalidDeviceFunction is returned. If the specified attribute cannot be written, or if the value is incorrect, then cudaErrorInvalidValue is returned.

Valid values for attr are:

- cudaFuncAttributeMaxDynamicSharedMemorySize - The requested maximum size in bytes of dynamically-allocated shared memory. The sum of this value and the function attribute sharedSizeBytes cannot exceed the device attribute cudaDevAttrMaxSharedMemoryPerBlockOptin. The maximal size of requestable dynamic shared memory may differ by GPU architecture.
- **cudaFuncAttributePreferredSharedMemoryCarveout** - On devices where the L1 cache and shared memory use the same hardware resources, this sets the shared memory carveout preference, in percent of the total shared memory. See [cudaDevAttrMaxSharedMemoryPerMultiprocessor](https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE__ATTR.html). This is only a hint, and the driver can choose a different ratio if required to execute the function.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by [cudaStreamAddCallback](https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDA__STREAM.html) no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**cudaLaunchKernel (C++ API), cudaFuncSetCacheConfig (C++ API), cudaFuncGetAttributes (C API)**

```c
__host__cudaError_t cudaFuncSetCacheConfig (const void *func, cudaFuncCache cacheConfig)
```

Sets the preferred cache configuration for a device function.

**Parameters**
- **func**
  - Device function symbol
- **cacheConfig**
  - Requested cache configuration

**Returns**
- `cudaSuccess`, `cudaErrorInvalidDeviceFunction`

**Description**

On devices where the L1 cache and shared memory use the same hardware resources, this sets through `cacheConfig` the preferred cache configuration for the function specified via `func`. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute `func`.

`func` is a device function symbol and must be declared as a `__global__` function. If the specified function does not exist, then `cudaErrorInvalidDeviceFunction`
is returned. For templated functions, pass the function symbol as follows:
func_name<template_arg_0,...,template_arg_N>

This setting does nothing on devices where the size of the L1 cache and shared memory are fixed.

Launching a kernel with a different preference than the most recent preference setting may insert a device-side synchronization point.

The supported cache configurations are:

- \texttt{cudaFuncCachePreferNone}: no preference for shared memory or L1 (default)
- \texttt{cudaFuncCachePreferShared}: prefer larger shared memory and smaller L1 cache
- \texttt{cudaFuncCachePreferL1}: prefer larger L1 cache and smaller shared memory
- \texttt{cudaFuncCachePreferEqual}: prefer equal size L1 cache and shared memory

\textbf{Note:}

- Note that this function may also return error codes from previous, asynchronous launches.
- Use of a string naming a function as the \texttt{func} parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.
- Note that this function may also return \texttt{cudaErrorInitializationError}, \texttt{cudaErrorInsufficientDriver} or \texttt{cudaErrorNoDevice} if this call tries to initialize internal CUDA RT state.
- Note that as specified by \texttt{cudaStreamAddCallback} no CUDA function may be called from callback. \texttt{cudaErrorNotPermitted} may, but is not guaranteed to, be returned as a diagnostic in such case.

\textbf{See also:}
\texttt{cudaFuncSetCacheConfig [ C++ API]}, \texttt{cudaFuncGetAttributes [ C API]}, \texttt{cudaLaunchKernel [ C API]}, \texttt{cudaThreadGetCacheConfig}, \texttt{cudaThreadSetCacheConfig}, \texttt{cuFuncSetCacheConfig}

\texttt{\_host\_cudaError\_t cudaFuncSetSharedMemConfig (const void *func, cudaSharedMemConfig config)}

Sets the shared memory configuration for a device function.

\textbf{Parameters}

\textbf{func}

- Device function function symbol
config
- Requested shared memory configuration

Returns
cudaSuccess, cudaErrorInvalidDeviceFunction, cudaErrorInvalidValue.

Description
On devices with configurable shared memory banks, this function will force all subsequent launches of the specified device function to have the given shared memory bank size configuration. On any given launch of the function, the shared memory configuration of the device will be temporarily changed if needed to suit the function’s preferred configuration. Changes in shared memory configuration between subsequent launches of functions, may introduce a device side synchronization point.

Any per-function setting of shared memory bank size set via cudaFuncSetSharedMemConfig will override the device wide setting set by cudaDeviceSetSharedMemConfig.

Changing the shared memory bank size will not increase shared memory usage or affect occupancy of kernels, but may have major effects on performance. Larger bank sizes will allow for greater potential bandwidth to shared memory, but will change what kinds of accesses to shared memory will result in bank conflicts.

This function will do nothing on devices with fixed shared memory bank size.

For templated functions, pass the function symbol as follows: func_name<template_arg_0,...,template_arg_N>

The supported bank configurations are:

- cudaSharedMemBankSizeDefault: use the device’s shared memory configuration when launching this function.
- cudaSharedMemBankSizeFourByte: set shared memory bank width to be four bytes natively when launching this function.
- cudaSharedMemBankSizeEightByte: set shared memory bank width to be eight bytes natively when launching this function.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Use of a string naming a function as the func parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback`, no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
`cudaDeviceSetSharedMemConfig`, `cudaDeviceGetSharedMemConfig`, `cudaDeviceSetCacheConfig`, `cudaDeviceGetCacheConfig`, `cudaFuncSetCacheConfig`, `cuFuncSetSharedMemConfig`

```c
#define __device__

void *cudaGetParameterBuffer(size_t alignment, size_t size)
```

Obtains a parameter buffer.

**Parameters**

- **alignment**
  - Specifies alignment requirement of the parameter buffer

- **size**
  - Specifies size requirement in bytes

**Returns**

Returns pointer to the allocated parameterBuffer

**Description**

Obtains a parameter buffer which can be filled with parameters for a kernel launch. Parameters passed to `cudaLaunchDevice` must be allocated via this function.

This is a low level API and can only be accessed from Parallel Thread Execution (PTX). CUDA user code should use `<<< >>>` to launch kernels.

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

See also:
`cudaLaunchDevice`
__device__ void *cudaGetParameterBufferV2 (void *func, dim3 gridDimension, dim3 blockDimension, unsigned int sharedMemSize)

Launches a specified kernel.

Parameters

func
- Pointer to the kernel to be launched

gridDimension
- Specifies grid dimensions

blockDimension
- Specifies block dimensions

sharedMemSize
- Specifies size of shared memory

Returns

cudaSuccess, cudaErrorInvalidDevice, cudaErrorLaunchMaxDepthExceeded, cudaErrorInvalidConfiguration, cudaErrorStartupFailure, cudaErrorLaunchPendingCountExceeded, cudaErrorLaunchOutOfResources

Description

Launches a specified kernel with the specified parameter buffer. A parameter buffer can be obtained by calling cudaGetParameterBuffer[].

This is a low level API and can only be accessed from Parallel Thread Execution (PTX). CUDA user code should use <<< >>> to launch the kernels.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

Please refer to Execution Configuration and Parameter Buffer Layout from the CUDA Programming Guide for the detailed descriptions of launch configuration and parameter layout respectively.

See also:
cudaGetParameterBuffer
`__host__ cudaError_t cudaLaunchCooperativeKernel
(const void *func, dim3 gridDim, dim3 blockDim, void **args, size_t sharedMem, cudaStream_t stream)`

Launches a device function where thread blocks can cooperate and synchronize as they execute.

**Parameters**

- `func` - Device function symbol
- `gridDim` - Grid dimensions
- `blockDim` - Block dimensions
- `args` - Arguments
- `sharedMem` - Shared memory
- `stream` - Stream identifier

**Returns**

- `cudaSuccess`, `cudaErrorInvalidDeviceFunction`, `cudaErrorInvalidConfiguration`, `cudaErrorLaunchFailure`, `cudaErrorLaunchTimeout`, `cudaErrorLaunchOutOfResources`, `cudaErrorCooperativeLaunchTooLarge`, `cudaErrorSharedObjectInitFailed`

**Description**


The device on which this kernel is invoked must have a non-zero value for the device attribute `cudaDevAttrCooperativeLaunch`.

The total number of blocks launched cannot exceed the maximum number of blocks per multiprocessor as returned by `cudaOccupancyMaxActiveBlocksPerMultiprocessor` (or `cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags`) times the number of multiprocessors as specified by the device attribute `cudaDevAttrMultiProcessorCount`.

The kernel cannot make use of CUDA dynamic parallelism.

If the kernel has N parameters the `args` should point to array of N pointers. Each pointer, from `args[0]` to `args[N - 1]`, point to the region of memory from which the actual parameter will be copied.
For templated functions, pass the function symbol as follows:
func_name<template_arg_0,...,template_arg_N>

-sharedMem sets the amount of dynamic shared memory that will be available to each thread block.

-stream specifies a stream the invocation is associated to.

**Note:**
- This function uses standard default stream semantics.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**
cudaLaunchCooperativeKernel [C++ API], cudaLaunchCooperativeKernelMultiDevice, cuLaunchCooperativeKernel

__host__ cudaError_t
cudaLaunchCooperativeKernelMultiDevice
cudaLaunchParams *launchParamsList, unsigned int numDevices, unsigned int flags)

Launches device functions on multiple devices where thread blocks can cooperate and synchronize as they execute.

**Parameters**
- **launchParamsList**
  - List of launch parameters, one per device
- **numDevices**
  - Size of the launchParamsList array
- **flags**
  - Flags to control launch behavior
Returns

cudaSuccess, cudaErrorInvalidDeviceFunction, cudaErrorInvalidConfiguration,
cudaErrorLaunchFailure, cudaErrorLaunchTimeout, cudaErrorLaunchOutOfResources,
cudaErrorCooperativeLaunchTooLarge, cudaErrorSharedObjectInitFailed

Description

**Deprecated** This function is deprecated as of CUDA 11.3.

Invokes kernels as specified in the launchParamsList array where each element of the array specifies all the parameters required to perform a single kernel launch. These kernels can cooperate and synchronize as they execute. The size of the array is specified by numDevices.

No two kernels can be launched on the same device. All the devices targeted by this multi-device launch must be identical. All devices must have a non-zero value for the device attribute cudaDevAttrCooperativeMultiDeviceLaunch.

The same kernel must be launched on all devices. Note that any __device__ or __constant__ variables are independently instantiated on every device. It is the application’s responsibility to ensure these variables are initialized and used appropriately.

The size of the grids as specified in blocks, the size of the blocks themselves and the amount of shared memory used by each thread block must also match across all launched kernels.

The streams used to launch these kernels must have been created via either cudaStreamCreate or cudaStreamCreateWithPriority or cudaStreamCreateWithPriority. The NULL stream or cudaStreamLegacy or cudaStreamPerThread cannot be used.

The total number of blocks launched per kernel cannot exceed the maximum number of blocks per multiprocessor as returned by cudaOccupancyMaxActiveBlocksPerMultiprocessor (or cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags) times the number of multiprocessors as specified by the device attribute cudaDevAttrMultiProcessorCount. Since the total number of blocks launched per device has to match across all devices, the maximum number of blocks that can be launched per device will be limited by the device with the least number of multiprocessors.

The kernel cannot make use of CUDA dynamic parallelism.

The **cudaLaunchParams** structure is defined as:

```c
struct cudaLaunchParams
{
    void *func;
    dim3 gridDim;
    dim3 blockDim;
    void **args;
    size_t sharedMem;
    cudaStream_t stream;
};
```
where:

- **cudaLaunchParams::func** specifies the kernel to be launched. This same function must be launched on all devices. For templated functions, pass the function symbol as follows: `func_name<template_arg_0,...,template_arg_N>`

- **cudaLaunchParams::gridDim** specifies the width, height and depth of the grid in blocks. This must match across all kernels launched.

- **cudaLaunchParams::blockDim** is the width, height and depth of each thread block. This must match across all kernels launched.

- **cudaLaunchParams::args** specifies the arguments to the kernel. If the kernel has N parameters then `cudaLaunchParams::args` should point to array of N pointers. Each pointer, from `cudaLaunchParams::args[0]` to `cudaLaunchParams::args[N - 1]`, point to the region of memory from which the actual parameter will be copied.

- **cudaLaunchParams::sharedMem** is the dynamic shared-memory size per thread block in bytes. This must match across all kernels launched.

- **cudaLaunchParams::stream** is the handle to the stream to perform the launch in. This cannot be the NULL stream or `cudaStreamLegacy` or `cudaStreamPerThread`.

By default, the kernel won’t begin execution on any GPU until all prior work in all the specified streams has completed. This behavior can be overridden by specifying the flag `cudaCooperativeLaunchMultiDeviceNoPreSync`. When this flag is specified, each kernel will only wait for prior work in the stream corresponding to that GPU to complete before it begins execution.

Similarly, by default, any subsequent work pushed in any of the specified streams will not begin execution until the kernels on all GPUs have completed. This behavior can be overridden by specifying the flag `cudaCooperativeLaunchMultiDeviceNoPostSync`. When this flag is specified, any subsequent work pushed in any of the specified streams will only wait for the kernel launched on the GPU corresponding to that stream to complete before it begins execution.

---

**Note:**

- This function uses standard default stream semantics.

- Note that this function may also return error codes from previous, asynchronous launches.

- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
`cudaLaunchCooperativeKernel` (C++ API), `cudaLaunchCooperativeKernel`, `cuLaunchCooperativeKernelMultiDevice`

```c
__host__ cudaError_t cudaLaunchHostFunc (cudaStream_t stream, cudaHostFn_t fn, void *userData)
```

Enqueues a host function call in a stream.

**Parameters**

- `stream`
- `fn`
- `userData`

**Returns**

`cudaSuccess`, `cudaErrorInvalidResourceHandle`, `cudaErrorInvalidValue`, `cudaErrorNotSupported`

**Description**

Enqueues a host function to run in a stream. The function will be called after currently enqueued work and will block work added after it.

The host function must not make any CUDA API calls. Attempting to use a CUDA API may result in `cudaErrorNotPermitted`, but this is not required. The host function must not perform any synchronization that may depend on outstanding CUDA work not mandated to run earlier.

Host functions without a mandated order (such as in independent streams) execute in undefined order and may be serialized.

For the purposes of Unified Memory, execution makes a number of guarantees:

- The stream is considered idle for the duration of the function’s execution. Thus, for example, the function may always use memory attached to the stream it was enqueued in.
- The start of execution of the function has the same effect as synchronizing an event recorded in the same stream immediately prior to the function. It thus synchronizes streams which have been “joined” prior to the function.
Adding device work to any stream does not have the effect of making the stream active until all preceding host functions and stream callbacks have executed. Thus, for example, a function might use global attached memory even if work has been added to another stream, if the work has been ordered behind the function call with an event.

Completion of the function does not cause a stream to become active except as described above. The stream will remain idle if no device work follows the function, and will remain idle across consecutive host functions or stream callbacks without device work in between. Thus, for example, stream synchronization can be done by signaling from a host function at the end of the stream.

Note that, in contrast to cuStreamAddCallback, the function will not be called in the event of an error in the CUDA context.

**Note:**

- This function uses standard default stream semantics.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cuStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaStreamCreate, cudaStreamQuery, cudaStreamSynchronize, cudaStreamWaitEvent, cudaStreamDestroy, cudaMallocManaged, cudaStreamAttachMemAsync, cudaStreamAddCallback, cuLaunchHostFunc

```c
__host__ cudaError_t cudaLaunchKernel (const void *func, dim3 gridDim, dim3 blockDim, void **args, size_t sharedMem, cudaStream_t stream)
```

Launches a device function.

**Parameters**

- `func` - Device function symbol
- `gridDim` - Grid dimensions
**blockDim**
- Block dimensions

**args**
- Arguments

**sharedMem**
- Shared memory

**stream**
- Stream identifier

**Returns**
- cudaSuccess, cudaErrorInvalidDeviceFunction, cudaErrorInvalidConfiguration,
cudaErrorLaunchFailure, cudaErrorLaunchTimeout, cudaErrorLaunchOutOfResources,
cudaErrorSharedObjectInitFailed, cudaErrorInvalidPtx, cudaErrorUnsupportedPtxVersion,
cudaErrorNoKernelImageForDevice, cudaErrorJitCompilerNotFound,
cudaErrorJitCompilationDisabled

**Description**

The function invokes kernel func on gridDim [gridDim.x gridDim.y gridDim.z] grid of
blocks. Each block contains blockDim [blockDim.x blockDim.y blockDim.z] threads.

If the kernel has N parameters the args should point to array of N pointers. Each pointer,
from args[0] to args[N - 1], point to the region of memory from which the actual
parameter will be copied.

For templated functions, pass the function symbol as follows:
func_name<template_arg_0,..,template_arg_N>

sharedMem sets the amount of dynamic shared memory that will be available to each thread
block.

stream specifies a stream the invocation is associated to.

---

**Note:**

- This function uses standard default stream semantics.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError,
cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal
CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called
from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a
diagnostic in such case.
See also:
cudaLaunchKernel [ C++ API], cuLaunchKernel

__host__ cudaError_t cudaSetDoubleForDevice (double *d)
Converts a double argument to be executed on a device.

Parameters
d
- Double to convert

Returns
cudaSuccess

Description
Deprecated This function is deprecated as of CUDA 7.5
Converts the double value of d to an internal float representation if the device does not support
double arithmetic. If the device does natively support doubles, then this function does nothing.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal
CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called
from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a
diagnostic in such case.

See also:
cudaFuncSetCacheConfig [ C API], cudaFuncGetAttributes [ C API], cudaSetDoubleForHost
__host__cudaError_t cudaMemcpyFromSymbol (char *symbol_ptr, void *host_ptr)

Copies data from the host memory to a symbol.

Parameters
symbol_ptr
- Symbol name
host_ptr
- Host memory address

Returns
cudaSuccess

description

This function copies data from the host memory to a symbol.

See also:
cudaSymbolRegister, cudaMemcpySymbol

6.8. Occupancy

This section describes the occupancy calculation functions of the CUDA runtime application programming interface.

Besides the occupancy calculator functions (cudaOccupancyMaxActiveBlocksPerMultiprocessor and
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags, there are also C++ only occupancy-based launch configuration functions documented in C++ API Routines module.

See cudaOccupancyMaxPotentialBlockSize [C++ API], cudaOccupancyMaxPotentialBlockSize [C++ API], cudaOccupancyMaxPotentialBlockSizeVariableSMem [C++ API], cudaOccupancyMaxPotentialBlockSizeVariableSMem [C++ API]

cudaOccupancyAvailableDynamicSMemPerBlock [C++ API],

__host__cudaError_t
cudaOccupancyAvailableDynamicSMemPerBlock
(size_t *dynamicSmemSize, const void *func, int numBlocks, int blockSize)

Returns dynamic shared memory available per block when launching numBlocks blocks on SM.

Parameters

dynamicSmemSize
- Returned maximum dynamic shared memory
func
- Kernel function for which occupancy is calculated
numBlocks
- Number of blocks to fit on SM
blockSize
- Size of the block

Returns
cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidDeviceFunction, cudaErrorInvalidValue, cudaErrorUnknown.

Description

Returns in *dynamicSmemSize the maximum size of dynamic shared memory to allow numBlocks blocks per SM.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

- `cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags`
- `cudaOccupancyMaxPotentialBlockSize [C++ API]`
- `cudaOccupancyMaxPotentialBlockSizeWithFlags [C++ API]`
- `cudaOccupancyMaxPotentialBlockSizeVariableSMem [C++ API]`
- `cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags [C++ API]`
- `cudaOccupancyAvailableDynamicSMemPerBlock`

```c
__host__ __device__ cudaError_t
cudaOccupancyMaxActiveBlocksPerMultiprocessor
(int *numBlocks, const void *func, int blockSize, size_t dynamicSMemSize)
```

Returns occupancy for a device function.

**Parameters**

- `numBlocks`
  - Returned occupancy
- `func`
  - Kernel function for which occupancy is calculated
- `blockSize`
  - Block size the kernel is intended to be launched with
- `dynamicSMemSize`
  - Per-block dynamic shared memory usage intended, in bytes

**Returns**

- `cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorInvalidDeviceFunction`, `cudaErrorInvalidValue`, `cudaErrorUnknown`.

**Description**

Returns in `*numBlocks` the maximum number of active blocks per streaming multiprocessor for the device function.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.

Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
- `cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags`
- `cudaOccupancyMaxPotentialBlockSize [C++ API]`
- `cudaOccupancyMaxPotentialBlockSizeWithFlags [C++ API]`
- `cudaOccupancyMaxPotentialBlockSizeVariableSMem [C++ API]`
- `cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags [C++ API]`
- `cudaOccupancyAvailableDynamicSMemPerBlock [C++ API]`
- `cuOccupancyMaxActiveBlocksPerMultiprocessor`

```c
__host__ cudaError_t
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
(int *numBlocks, const void *func, int blockSize, size_t dynamicSMemSize, unsigned int flags)
```

Returns occupancy for a device function with the specified flags.

**Parameters**

- `numBlocks` - Returned occupancy
- `func` - Kernel function for which occupancy is calculated
- `blockSize` - Block size the kernel is intended to be launched with
- `dynamicSMemSize` - Per-block dynamic shared memory usage intended, in bytes
- `flags` - Requested behavior for the occupancy calculator

**Returns**

- `cudaSuccess`
- `cudaErrorInvalidDevice`
- `cudaErrorInvalidDeviceFunction`
- `cudaErrorInvalidValue`
- `cudaErrorUnknown`
Description

Returns in *numBlocks* the maximum number of active blocks per streaming multiprocessor for the device function.

The *flags* parameter controls how special cases are handled. Valid flags include:

- **cudaOccupancyDefault**: keeps the default behavior as `cudaOccupancyMaxActiveBlocksPerMultiprocessor`
- **cudaOccupancyDisableCachingOverride**: This flag suppresses the default behavior on platform where global caching affects occupancy. On such platforms, if caching is enabled, but per-block SM resource usage would result in zero occupancy, the occupancy calculator will calculate the occupancy as if caching is disabled. Setting this flag makes the occupancy calculator to return 0 in such cases. More information can be found about this feature in the “Unified L1/Texture Cache” section of the Maxwell tuning guide.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**

- `cudaOccupancyMaxActiveBlocksPerMultiprocessor`
- `cudaOccupancyMaxPotentialBlockSize [C++ API]`
- `cudaOccupancyMaxPotentialBlockSizeWithFlags [C++ API]`
- `cudaOccupancyMaxPotentialBlockSizeVariableSMem [C++ API]`
- `cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags [C++ API]`
- `cudaOccupancyAvailableDynamicSMemPerBlock [C++ API]`
- `cuOccupancyMaxActiveBlocksPerMultiprocessorWithFlags`

6.9. Memory Management

This section describes the memory management functions of the CUDA runtime application programming interface.

Some functions have overloaded C++ API template versions documented separately in the **C++ API Routines** module.
__host__ cudaError_t cudaArrayGetInfo
(cuArrayFormatDesc *desc, cudaExtent *extent,
unsigned int *flags, cudaArray_t array)

Get info about the specified cudaArray.

Parameters

- **desc**
  - Returned array type
- **extent**
  - Returned array shape. 2D arrays will have depth of zero
- **flags**
  - Returned array flags
- **array**
  - The cudaArray to get info for

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Returns in *desc, *extent and *flags respectively, the type, shape and flags of array.

Any of *desc, *extent and *flags may be specified as NULL.

---

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cuArrayGetDescriptor, cuArray3DGetDescriptor
__host__ cudaError_t cudaArrayGetPlane (cudaArray_t *pPlaneArray, cudaArray_t hArray, unsigned int planeIdx)

Gets a CUDA array plane from a CUDA array.

Parameters

pPlaneArray - Returned CUDA array referenced by the planeIdx
hArray - CUDA array
planeIdx - Plane index

Returns
cudaSuccess, cudaErrorInvalidValue cudaErrorInvalidResourceHandle

Description

Returns in pPlaneArray a CUDA array that represents a single format plane of the CUDA array hArray.

If planeIdx is greater than the maximum number of planes in this array or if the array does not have a multi-planar format e.g: cudaChannelFormatKindNV12, then cudaErrorInvalidValue is returned.

Note that if the hArray has format cudaChannelFormatKindNV12, then passing in 0 for planeIdx returns a CUDA array of the same size as hArray but with one 8-bit channel and cudaChannelFormatKindUnsigned as its format kind. If 1 is passed for planeIdx, then the returned CUDA array has half the height and width of hArray with two 8-bit channels and cudaChannelFormatKindUnsigned as its format kind.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuArrayGetPlane
__host__ cudaError_t cudaArrayGetSparseProperties (cudaArraySparseProperties *sparseProperties, cudaArray_t array)

Returns the layout properties of a sparse CUDA array.

Parameters

sparseProperties
- Pointer to return the cudaArraySparseProperties

array
- The CUDA array to get the sparse properties of

Returns
cudaSuccess cudaErrorInvalidValue

Description

Returns the layout properties of a sparse CUDA array in sparseProperties. If the CUDA array is not allocated with flag cudaArraySparse(cudaErrorInvalidValue) will be returned.

If the returned value in cudaArraySparseProperties::flags contains cudaArraySparsePropertiesSingleMipTail, then cudaArraySparseProperties::miptailSize represents the total size of the array. Otherwise, it will be zero. Also, the returned value in cudaArraySparseProperties::miptailFirstLevel is always zero. Note that the array must have been allocated using cudaMallocArray or cudaMalloc3DArray. For CUDA arrays obtained using cudaMipmappedArrayGetLevel, cudaErrorInvalidValue will be returned. Instead, cudaMipmappedArrayGetSparseProperties must be used to obtain the sparse properties of the entire CUDA mipmapped array to which array belongs to.

See also:
cudaMipmappedArrayGetSparseProperties, cuMemMapArrayAsync

__host____device__ cudaError_t cudaFree (void *devPtr)

Frees memory on the device.

Parameters
devPtr
- Device pointer to memory to free
Returns
cudaSuccess, cudaErrorInvalidValue

Description
Frees the memory space pointed to by devPtr, which must have been returned by a previous call to cudaMemcpy or cudaMemcpyPitch. Otherwise, or if cudaFree(devPtr) has already been called before, an error is returned. If devPtr is 0, no operation is performed. cudaFree returns cudaErrorValue in case of failure.

The device version of cudaFree cannot be used with a *devPtr allocated using the host API, and vice versa.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaMalloc, cudaMemcpyPitch, cudaMemcpyArray, cudaMemcpyArray, cudaMemcpyHost [ C API], cudaMemcpyHost, cudaMemcpy3D, cudaMemcpy3DArray, cudaMemcpyHost, cudaMemcpyHost, cuMemFree

__host__cudaError_t cudaFreeArray (cudaArray_t array)
Frees an array on the device.

Parameters
array
  - Pointer to array to free

Returns
cudaSuccess, cudaErrorInvalidValue
Description

Frees the CUDA array array, which must have been returned by a previous call to `cudaMallocArray()`. If `devPtr` is 0, no operation is performed.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

- `cudaMalloc`, `cudaMallocPitch`, `cudaFree`, `cudaMallocArray`, `cudaMallocHost (C API)`, `cudaFreeHost`, `cudaHostAlloc`, `cuArrayDestroy`

```c
__host__ cudaError_t cudaFreeHost (void *ptr)
```

Frees page-locked memory.

Parameters

- `ptr` - Pointer to memory to free

Returns

- `cudaSuccess`, `cudaErrorInvalidValue`

Description

Frees the memory space pointed to by `hostPtr`, which must have been returned by a previous call to `cudaMallocHost()` or `cudaHostAlloc()`.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.

Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

`cudaMalloc`, `cudaMallocPitch`, `cudaFree`, `cudaMallocArray`, `cudaFreeArray`, `cudaMallocHost` ([C API], `cudaMalloc3D`, `cudaMalloc3DArray`, `cudaHostAlloc`, `cuMemFreeHost`)

```__host__ cudaError_t cudaFreeMipmappedArray (cudaMipmappedArray_t mipmappedArray)```
Frees a mipmapped array on the device.

**Parameters**

`mipmappedArray`
- Pointer to mipmapped array to free

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`

**Description**

Frees the CUDA mipmapped array `mipmappedArray`, which must have been returned by a previous call to `cudaMallocMipmappedArray()`. If `devPtr` is 0, no operation is performed.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
__host__ cudaError_t cudaGetMipmappedArrayLevel(cudaArray_t *levelArray, cudaMipmappedArray_const_t mipmappedArray, unsigned int level)

Gets a mipmap level of a CUDA mipmapped array.

**Parameters**

- **levelArray** - Returned mipmap level CUDA array
- **mipmappedArray** - CUDA mipmapped array
- **level** - Mipmap level

**Returns**

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle

**Description**

Returns in *levelArray* a CUDA array that represents a single mipmap level of the CUDA mipmapped array `mipmappedArray`.

If `level` is greater than the maximum number of levels in this mipmapped array, `cudaErrorInvalidValue` is returned.

If `mipmappedArray` is NULL, `cudaErrorInvalidResourceHandle` is returned.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotAllowed` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaMalloc3D, cudaMalloc, cudaMallocPitch, cudaMemcpy, cudaMemcpyHost [ C API], cudaMemcpy, cudaHostAlloc, make_cudaExtent, cuMipmappedArrayGetLevel

__host__ cudaError_t cudaMemcpyAddress (void **devPtr, const void *symbol)
Finds the address associated with a CUDA symbol.

Parameters

devPtr
- Return device pointer associated with symbol

symbol
- Device symbol address

Returns

cudaSuccess, cudaErrorInvalidSymbol, cudaErrorNoKernelImageForDevice

Description

Returns in *devPtr the address of symbol symbol on the device. symbol is a variable that resides in global or constant memory space. If symbol cannot be found, or if symbol is not declared in the global or constant memory space, *devPtr is unchanged and the error cudaErrorInvalidSymbol is returned.

See also:
cudaGetSymbolAddress [ C++ API], cudaMemcpySymbolSize [ C API], cuModuleGetGlobal

---

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Use of a string naming a variable as the symbol parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpyAddress, no CUDA function may be called from callback. cudaMemcpyNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
__host__ cudaError_t cudaMemcpySize (size_t *size, const void *symbol)

Finds the size of the object associated with a CUDA symbol.

Parameters

size
- Size of object associated with symbol
symbol
- Device symbol address

Returns

cudaSuccess, cudaErrorInvalidSymbol, cudaErrorNoKernelImageForDevice

Description

Returns in *size the size of symbol symbol. symbol is a variable that resides in global or constant memory space. If symbol cannot be found, or if symbol is not declared in global or constant memory space, *size is unchanged and the error cudaErrorInvalidSymbol is returned.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Use of a string naming a variable as the symbol parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpyAddress no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGetSymbolAddress [ C API], cudaMemcpySize [ C++ API], cuModuleGetGlobal
__host__ cudaError_t cudaHostAlloc (void **pHost, size_t size, unsigned int flags)

 Allocates page-locked memory on the host.

**Parameters**

- **pHost**
  - Device pointer to allocated memory
- **size**
  - Requested allocation size in bytes
- **flags**
  - Requested properties of allocated memory

**Returns**

cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation

**Description**

Allocates `size` bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as cudaMemcpy[]. Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as malloc[]. Allocating excessive amounts of pinned memory may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to allocate staging areas for data exchange between host and device.

The `flags` parameter enables different options to be specified that affect the allocation, as follows.

- **cudaHostAllocDefault**: This flag's value is defined to be 0 and causes cudaMemcpy[] to emulate cudaMemcpy[].
- **cudaHostAllocPortable**: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
- **cudaHostAllocMapped**: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling cudaMemcpy[].
- **cudaHostAllocWriteCombined**: Allocates the memory as write-combined (WC). WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs. WC memory is a good option for buffers that will be written by the CPU and read by the device via mapped pinned memory or host->device transfers.
All of these flags are orthogonal to one another: a developer may allocate memory that is portable, mapped and/or write-combined with no restrictions.

In order for the `cudaHostAllocMapped` flag to have any effect, the CUDA context must support the `cudaDeviceMapHost` flag, which can be checked via `cudaGetDeviceFlags[]`. The `cudaDeviceMapHost` flag is implicitly set for contexts created via the runtime API.

The `cudaHostAllocMapped` flag may be specified on CUDA contexts for devices that do not support mapped pinned memory. The failure is deferred to `cudaHostGetDevicePointer[]` because the memory may be mapped into other CUDA contexts via the `cudaHostAllocPortable` flag.

Memory allocated by this function must be freed with `cudaFreeHost[]`.

---

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

---

**See also:**

`cudaSetDeviceFlags`, `cudaMallocHost [ C API]`, `cudaFreeHost`, `cudaGetDeviceFlags`, `cuMemHostAlloc`

---

```c
__host__ cudaError_t cudaMemcpyHostToDevice (void *dst, const void *src, size_t size, cudaMemcpyKind kind)
```
Returns

**cudaSuccess**, **cudaErrorInvalidValue**, **cudaErrorMemoryAllocation**

Description

Passes back the device pointer corresponding to the mapped, pinned host buffer allocated by **cudaHostAlloc()** or registered by **cudaHostRegister()**.

**cudaHostGetDevicePointer()** will fail if the **cudaDeviceMapHost** flag was not specified before deferred context creation occurred, or if called on a device that does not support mapped, pinned memory.

For devices that have a non-zero value for the device attribute **cudaDevAttrCanUseHostPointerForRegisteredMem**, the memory can also be accessed from the device using the host pointer `pHost`. The device pointer returned by **cudaHostGetDevicePointer()** may or may not match the original host pointer `pHost` and depends on the devices visible to the application. If all devices visible to the application have a non-zero value for the device attribute, the device pointer returned by **cudaHostGetDevicePointer()** will match the original pointer `pHost`. If any device visible to the application has a zero value for the device attribute, the device pointer returned by **cudaHostGetDevicePointer()** will not match the original host pointer `pHost`, but it will be suitable for use on all devices provided Unified Virtual Addressing is enabled. In such systems, it is valid to access the memory using either pointer on devices that have a non-zero value for the device attribute. Note however that such devices should access the memory using only of the two pointers and not both.

flags provides for future releases. For now, it must be set to 0.

### Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return **cudaErrorInitializationError**, **cudaErrorInsufficientDriver** or **cudaErrorNoDevice** if this call tries to initialize internal CUDA RT state.
- Note that as specified by **cudaStreamAddCallback** no CUDA function may be called from callback. **cudaErrorNotPermitted** may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

**cudaSetDeviceFlags**, **cudaHostAlloc**, **cuMemHostGetDevicePointer**
__host__cudaError_t cudaHostGetFlags (unsigned int *pFlags, void *pHost)

Passes back flags used to allocate pinned host memory allocated by cudaHostAlloc.

Parameters

pFlags
- Returned flags word

pHost
- Host pointer

Returns

cudaSuccess, cudaErrorInvalidValue

Description

cudaHostGetFlags() will fail if the input pointer does not reside in an address range allocated by cudaHostAlloc().

Note:

‣ Note that this function may also return error codes from previous, asynchronous launches.
‣ Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
‣ Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaHostAlloc, cuMemHostGetFlags

__host__cudaError_t cudaHostRegister (void *ptr, size_t size, unsigned int flags)

Registers an existing host memory range for use by CUDA.

Parameters

ptr
- Host pointer to memory to page-lock
size
- Size in bytes of the address range to page-lock in bytes

flags
- Flags for allocation request

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation, cudaErrorHostMemoryAlreadyRegistered, cudaErrorNotSupported

Description
Page-locks the memory range specified by `ptr` and `size` and maps it for the device(s) as specified by `flags`. This memory range also is added to the same tracking mechanism as `cudaHostAlloc()` to automatically accelerate calls to functions such as `cudaMemcpy()`. Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory that has not been registered. Page-locking excessive amounts of memory may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to register staging areas for data exchange between host and device.

`cudaHostRegister` is supported only on I/O coherent devices that have a non-zero value for the device attribute `cudaDevAttrHostRegisterSupported`.

The `flags` parameter enables different options to be specified that affect the allocation, as follows.

- `cudaHostRegisterDefault`: On a system with unified virtual addressing, the memory will be both mapped and portable. On a system with no unified virtual addressing, the memory will be neither mapped nor portable.

- `cudaHostRegisterPortable`: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.

- `cudaHostRegisterMapped`: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling `cudaHostGetDevicePointer()`.

- `cudaHostRegisterIoMemory`: The passed memory pointer is treated as pointing to some memory-mapped I/O space, e.g. belonging to a third-party PCIe device, and it will marked as non cache-coherent and contiguous.

- `cudaHostRegisterReadOnly`: The passed memory pointer is treated as pointing to memory that is considered read-only by the device. On platforms without `cudaDevAttrPageableMemoryAccessUsesHostPageTables`, this flag is required in order to register memory mapped to the CPU as read-only. Support for the use of this flag can be queried from the device attribute `cudaDeviceAttrReadOnlyHostRegisterSupported`. Using this flag with a current context associated with a device that does not have this attribute set will cause `cudaHostRegister` to error with `cudaErrorNotSupported`. 


All of these flags are orthogonal to one another: a developer may page-lock memory that is portable or mapped with no restrictions.

The CUDA context must have been created with the cudaMapHost flag in order for the cudaHostRegisterMapped flag to have any effect.

The cudaHostRegisterMapped flag may be specified on CUDA contexts for devices that do not support mapped pinned memory. The failure is deferred to cudaHostGetDevicePointer because the memory may be mapped into other CUDA contexts via the cudaHostRegisterPortable flag.

For devices that have a non-zero value for the device attribute cudaDevAttrCanUseHostPointerForRegisteredMem, the memory can also be accessed from the device using the host pointer ptr. The device pointer returned by cudaHostGetDevicePointer may or may not match the original host pointer ptr and depends on the devices visible to the application. If all devices visible to the application have a non-zero value for the device attribute, the device pointer returned by cudaHostGetDevicePointer will match the original pointer ptr. If any device visible to the application has a zero value for the device attribute, the device pointer returned by cudaHostGetDevicePointer will not match the original host pointer ptr, but it will be suitable for use on all devices provided Unified Virtual Addressing is enabled. In such systems, it is valid to access the memory using either pointer on devices that have a non-zero value for the device attribute. Note however that such devices should access the memory using only of the two pointers and not both.

The memory page-locked by this function must be unregistered with cudaHostUnregister.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaHostUnregister, cudaHostGetFlags, cudaHostGetDevicePointer, cuMemHostRegister
__host__cudaError_t cudaHostUnregister (void *ptr)
Unregisters a memory range that was registered with cudaHostRegister.

Parameters

ptr
- Host pointer to memory to unregister

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorHostMemoryNotRegistered

Description
Unmaps the memory range whose base address is specified by ptr, and makes it pageable again.
The base address must be the same one specified to cudaHostRegister[].

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpy, no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaHostUnregister, cuMemHostUnregister

__host___device__cudaError_t cudaMalloc (void **devPtr, size_t size)
Allocate memory on the device.

Parameters
devPtr
- Pointer to allocated device memory
size
- Requested allocation size in bytes
Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation

Description

Allocates size bytes of linear memory on the device and returns in *devPtr a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. cudaMalloc returns cudaErrorMemoryAllocation in case of failure.

The device version of cudaFree cannot be used with a *devPtr allocated using the host API, and vice versa.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaMallocPitch, cudaFree, cudaMallocArray, cudaFreeArray, cudaMalloc3D, cudaMalloc3DArray, cudaMallocHost (C API), cudaFreeHost, cudaHostAlloc, cuMemAlloc

__host__cudaError_t cudaMalloc3D (cudaPitchedPtr *pitchedDevPtr, cudaExtent extent)

Allocates logical 1D, 2D, or 3D memory objects on the device.

Parameters

pitchedDevPtr
  - Pointer to allocated pitched device memory

extent
  - Requested allocation size (width field in bytes)

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation
Description

Allocates at least $width \times height \times depth$ bytes of linear memory on the device and returns a `cudaPitchedPtr` in which `ptr` is a pointer to the allocated memory. The function may pad the allocation to ensure hardware alignment requirements are met. The pitch returned in the `pitch` field of `pitchedDevPtr` is the width in bytes of the allocation.

The returned `cudaPitchedPtr` contains additional fields `xsize` and `ysize`, the logical width and height of the allocation, which are equivalent to the `width` and `height` extent parameters provided by the programmer during allocation.

For allocations of 2D and 3D objects, it is highly recommended that programmers perform allocations using `cudaMalloc3D()` or `cudaMallocPitch()`. Due to alignment restrictions in the hardware, this is especially true if the application will be performing memory copies involving 2D or 3D objects (whether linear memory or CUDA arrays).

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

`cudaMallocPitch`, `cudaFree`, `cudaMemcpy3D`, `cudaMemset3D`, `cudaMalloc3DArray`, `cudaMallocArray`, `cudaFreeArray`, `cudaMallocHost [C API]`, `cudaFreeHost`, `cudaHostAlloc`, `make_cudaPitchedPtr`, `make_cudaExtent`, `cuMemAllocPitch`

```c
__host__ cudaError_t cudaMalloc3DArray
  (cudaArray_t *array, const cudaChannelFormatDesc *desc, cudaExtent extent, unsigned int flags)
```

Allocate an array on the device.

Parameters

- **array**
  - Pointer to allocated array in device memory
desc
- Requested channel format

extent
- Requested allocation size \((\text{width field in elements})\)

flags
- Flags for extensions

Returns
\texttt{cudaSuccess}, \texttt{cudaErrorInvalidValue}, \texttt{cudaErrorMemoryAllocation}

Description
Allocates a CUDA array according to the \texttt{cudaChannelFormatDesc} structure \texttt{desc} and returns a handle to the new CUDA array in \texttt{*array}.

The \texttt{cudaChannelFormatDesc} is defined as:

```c
struct cudaChannelFormatDesc {
    int x, y, z, w;
    enum cudaChannelFormatKind f;
};
```

where \texttt{cudaChannelFormatKind} is one of \texttt{cudaChannelFormatKindSigned}, \texttt{cudaChannelFormatKindUnsigned}, or \texttt{cudaChannelFormatKindFloat}.

\texttt{cudaMalloc3DArray} can allocate the following:

- A 1D array is allocated if the height and depth extents are both zero.
- A 2D array is allocated if only the depth extent is zero.
- A 3D array is allocated if all three extents are non-zero.
- A 1D layered CUDA array is allocated if only the height extent is zero and the cudaArrayLayered flag is set. Each layer is a 1D array. The number of layers is determined by the depth extent.
- A 2D layered CUDA array is allocated if all three extents are non-zero and the cudaArrayLayered flag is set. Each layer is a 2D array. The number of layers is determined by the depth extent.
- A cubemap CUDA array is allocated if all three extents are non-zero and the cudaArrayCubemap flag is set. Width must be equal to height, and depth must be six. A cubemap is a special type of 2D layered CUDA array, where the six layers represent the six faces of a cube. The order of the six layers in memory is the same as that listed in \texttt{cudaGraphicsCubeFace}.
- A cubemap layered CUDA array is allocated if all three extents are non-zero, and both, cudaArrayCubemap and cudaArrayLayered flags are set. Width must be equal to height, and depth must be a multiple of six. A cubemap layered CUDA array is a special type of 2D
layered CUDA array that consists of a collection of cubemaps. The first six layers represent the first cubemap, the next six layers form the second cubemap, and so on.

The flags parameter enables different options to be specified that affect the allocation, as follows.

- **cudaArrayDefault**: This flag’s value is defined to be 0 and provides default array allocation.
- **cudaArrayLayered**: Allocates a layered CUDA array, with the depth extent indicating the number of layers.
- **cudaArrayCubemap**: Allocates a cubemap CUDA array. Width must be equal to height, and depth must be six. If the cudaArrayLayered flag is also set, depth must be a multiple of six.
- **cudaArraySurfaceLoadStore**: Allocates a CUDA array that could be read from or written to using a surface reference.
- **cudaArrayTextureGather**: This flag indicates that texture gather operations will be performed on the CUDA array. Texture gather can only be performed on 2D CUDA arrays.
- **cudaArraySparse**: Allocates a CUDA array without physical backing memory. The subregions within this sparse array can later be mapped to physical memory by calling `cuMemMapArrayAsync`. This flag can only be used for creating 2D, 3D or 2D layered sparse CUDA arrays. The physical backing memory must be allocated via `cuMemCreate`.

The width, height and depth extents must meet certain size requirements as listed in the following table. All values are specified in elements.

Note that 2D CUDA arrays have different size requirements if the `cudaArrayTextureGather` flag is set. In that case, the valid range for (width, height, depth) is \([1, \text{maxTexture2DGather}[0]], [1, \text{maxTexture2DGather}[1]], 0\).

<table>
<thead>
<tr>
<th>CUDA array type</th>
<th>Valid extents that must always be met {{\text{width range in elements}}, {\text{height range}}, {\text{depth range}}}</th>
<th>Valid extents with cudaArraySurfaceLoadStore set {{\text{width range in elements}}, {\text{height range}}, {\text{depth range}}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D</td>
<td>{1, \text{maxTexture1D}, 0, 0 }</td>
<td>{1, \text{maxSurface1D}, 0, 0 }</td>
</tr>
<tr>
<td>2D</td>
<td>{1, \text{maxTexture2D}[0], \text{maxTexture2D}[1], 0 }</td>
<td>{1, \text{maxSurface2D}[0], \text{maxSurface2D}[1], 0 }</td>
</tr>
</tbody>
</table>
| 3D              | \{1, \text{maxTexture3D}[0], \text{maxTexture3D}[1], \text{maxTexture3D}[2] \}  
|                 | OR \{1, \text{maxTexture3DAlt}[0], \text{maxTexture3DAlt}[1], \text{maxTexture3DAlt}[2] \} | \{1, \text{maxSurface3D}[0], \text{maxSurface3D}[1], \text{maxSurface3D}[2] \} |
| 1D Layered      | \{1, \text{maxTexture1DLayered}[0], 0, \text{maxTexture1DLayered}[1]\} | \{1, \text{maxSurface1DLayered}[0], 0, \text{maxSurface1DLayered}[1]\} |
### Modules

#### CUDA Runtime API

<table>
<thead>
<tr>
<th>CUDA array type</th>
<th>Valid extents that must always be met { (width range in elements), (height range), (depth range) }</th>
<th>Valid extents with cudaArraySurfaceLoadStore set { (width range in elements), (height range), (depth range) }</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D Layered</td>
<td>{ (1,maxTexture2DLayered[0]), (1,maxTexture2DLayered[1]), (1,maxTexture2DLayered[2]) }</td>
<td>{ (1,maxSurface2DLayered[0]), (1,maxSurface2DLayered[1]), (1,maxSurface2DLayered[2]) }</td>
</tr>
<tr>
<td>Cubemap</td>
<td>{ (1,maxTextureCubemap), (1,maxTextureCubemap) }</td>
<td>{ (1,maxSurfaceCubemap), (1,maxSurfaceCubemap) }</td>
</tr>
<tr>
<td>Cubemap Layered</td>
<td>{ (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[1]) }</td>
<td>{ (1,maxSurfaceCubemapLayered[0]), (1,maxSurfaceCubemapLayered[0]), (1,maxSurfaceCubemapLayered[1]) }</td>
</tr>
</tbody>
</table>

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
- `cudaMalloc3D`, `cudaMalloc`, `cudaMallocPitch`, `cudaFree`, `cudaFreeArray`, `cudaMallocHost` [C API], `cudaFreeHost`, `cudaHostAlloc`, `make_cudaExtent`, `cuArray3DCreate`

```c
__host__cudaError_t cudaMallocArray (cudaArray_t *array, const cudaChannelFormatDesc *desc, size_t width, size_t height, unsigned int flags)
```

Allocate an array on the device.

**Parameters**

- **array**
  - Pointer to allocated array in device memory
- **desc**
  - Requested channel format
width
- Requested array allocation width

height
- Requested array allocation height

flags
- Requested properties of allocated array

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation

Description
Allocates a CUDA array according to the cudaChannelFormatDesc structure desc and returns a handle to the new CUDA array in *array.

The cudaChannelFormatDesc is defined as:

```
struct cudaChannelFormatDesc {
  int x, y, z, w;
  enum cudaChannelFormatKind f;
};
```

where cudaChannelFormatKind is one of cudaChannelFormatKindSigned, cudaChannelFormatKindUnsigned, or cudaChannelFormatKindFloat.

The flags parameter enables different options to be specified that affect the allocation, as follows.

- **cudaArrayDefault**: This flag’s value is defined to be 0 and provides default array allocation
- **cudaArraySurfaceLoadStore**: Allocates an array that can be read from or written to using a surface reference
- **cudaArrayTextureGather**: This flag indicates that texture gather operations will be performed on the array.
- **cudaArraySparse**: Allocates a CUDA array without physical backing memory. The subregions within this sparse array can later be mapped to physical memory by calling cuMemMapArrayAsync. The physical backing memory must be allocated via cuMemCreate.

width and height must meet certain size requirements. See cudaMalloc3DArray for more details.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.

Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
- `cudaMalloc`, `cudaMallocPitch`, `cudaFree`, `cudaFreeArray`, `cudaMallocHost` ([C API]),
- `cudaFreeHost`, `cudaMalloc3D`, `cudaMalloc3DArray`, `cudaHostAlloc`, `cuArrayCreate`

```c
__host__ cudaError_t cudaMallocHost (void **ptr, size_t size)
```

Allocates page-locked memory on the host.

**Parameters**

- **ptr**
  - Pointer to allocated host memory
- **size**
  - Requested allocation size in bytes

**Returns**
- `cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorMemoryAllocation`

**Description**

Allocates `size` bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as `cudaMemcpy`[]. Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as malloc[]. Allocating excessive amounts of memory with `cudaMallocHost[]` may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to allocate staging areas for data exchange between host and device.

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.
Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.

Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
`cudaMalloc`, `cudaMallocPitch`, `cudaMallocArray`, `cudaMalloc3D`, `cudaMalloc3DArray`, `cudaHostAlloc`, `cudaFree`, `cudaFreeArray`, `cudaMallocHost [C++ API]`, `cudaFreeHost`, `cudaHostAlloc`, `cuMemAllocHost`

```c
__host__ cudaError_t cudaMallocManaged(void **devPtr, size_t size, unsigned int flags)
```

Allocates memory that will be automatically managed by the Unified Memory system.

**Parameters**

- **devPtr**: Pointer to allocated device memory
- **size**: Requested allocation size in bytes
- **flags**: Must be either `cudaMemAttachGlobal` or `cudaMemAttachHost` (defaults to `cudaMemAttachGlobal`)

**Returns**

`cudaSuccess`, `cudaErrorMemoryAllocation`, `cudaErrorNotSupported`, `cudaErrorInvalidValue`

**Description**

Allocates `size` bytes of managed memory on the device and returns in `*devPtr` a pointer to the allocated memory. If the device doesn’t support allocating managed memory, `cudaErrorNotSupported` is returned. Support for managed memory can be queried using the device attribute `cudaDevAttrManagedMemory`. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. If `size` is 0, `cudaMallocManaged` returns `cudaErrorInvalidValue`. The pointer is valid on the CPU and on all GPUs in the system that support managed memory. All accesses to this pointer must obey the Unified Memory programming model.

`flags` specifies the default stream association for this allocation. `flags` must be one of `cudaMemAttachGlobal` or `cudaMemAttachHost`. The default value for `flags` is `cudaMemAttachGlobal`. If `cudaMemAttachGlobal` is specified, then this memory is accessible
from any stream on any device. If `cudaMemAttachHost` is specified, then the allocation should not be accessed from devices that have a zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`; an explicit call to `cudaStreamAttachMemAsync` will be required to enable access on such devices.

If the association is later changed via `cudaStreamAttachMemAsync` to a single stream, the default association, as specified during `cudaMallocManaged`, is restored when that stream is destroyed. For `__managed__` variables, the default association is always `cudaMemAttachGlobal`. Note that destroying a stream is an asynchronous operation, and as a result, the change to default association won’t happen until all work in the stream has completed.

Memory allocated with `cudaMallocManaged` should be released with `cudaFree`.

Device memory oversubscription is possible for GPUs that have a non-zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`. Managed memory on such GPUs may be evicted from device memory to host memory at any time by the Unified Memory driver in order to make room for other allocations.

In a multi-GPU system where all GPUs have a non-zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`, managed memory may not be populated when this API returns and instead may be populated on access. In such systems, managed memory can migrate to any processor’s memory at any time. The Unified Memory driver will employ heuristics to maintain data locality and prevent excessive page faults to the extent possible. The application can also guide the driver about memory usage patterns via `cudaMemAdvise`. The application can also explicitly migrate memory to a desired processor’s memory via `cudaMemPrefetchAsync`.

In a multi-GPU system where not all GPUs have peer-to-peer support with each other and where the value of the device attribute `cudaDevAttrConcurrentManagedAccess` is zero for at least one of those GPUs, the location chosen for physical storage of managed memory is system-dependent.

- On Linux, the location chosen will be device memory as long as the current set of active contexts are on devices that either have peer-to-peer support with each other or have a non-zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`. If there is an active context on a GPU that does not have a non-zero value for that device attribute and it does not have peer-to-peer support with the other devices that have active contexts on them, then the location for physical storage will be ‘zero-copy’ or host memory. Note that this means that managed memory that is located in device memory is migrated to host memory if a new context is created on a GPU that doesn’t have a non-zero value
for the device attribute and does not support peer-to-peer with at least one of the other
devices that has an active context. This in turn implies that context creation may fail if
there is insufficient host memory to migrate all managed allocations.

‣ On Windows, the physical storage is always created in ‘zero-copy’ or host memory.
All GPUs will reference the data at reduced bandwidth over the PCIe bus. In these
circumstances, use of the environment variable CUDA_VISIBLE_DEVICES is recommended
to restrict CUDA to only use those GPUs that have peer-to-peer support. Alternatively,
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 the CUDA programming
guide under the “CUDA environment variables” section.

Note:
‣ Note that this function may also return error codes from previous, asynchronous launches.
‣ Note that this function may also return cudaErrorInitializationError,
cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal
CUDA RT state.
‣ Note that as specified by cudaStreamAddCallback no CUDA function may be called
from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a
diagnostic in such case.

See also:
cudaMallocPitch, cudaFree, cudaMallocArray, cudaFreeArray, cudaMalloc3D,
cudaMalloc3DArray, cudaMallocHost [C API], cudaFreeHost, cudaHostAlloc,
cudaDeviceGetAttribute, cudaStreamAttachMemAsync, cuMemAllocManaged
__host__ cudaError_t cudaMallocMipmappedArray

(cudaMipmappedArray_t *mipmappedArray, const
cudaChannelFormatDesc *desc, cudaExtent extent,
unsigned int numLevels, unsigned int flags)

Allocate a mipmapped array on the device.

Parameters

mipmappedArray
  - Pointer to allocated mipmapped array in device memory
desc
  - Requested channel format
extent
  - Requested allocation size (width field in elements)
numLevels
  - Number of mipmap levels to allocate
flags
  - Flags for extensions

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation

Description

Allocates a CUDA mipmapped array according to the cudaChannelFormatDesc structure desc
and returns a handle to the new CUDA mipmapped array in *mipmappedArray. numLevels
specifies the number of mipmap levels to be allocated. This value is clamped to the range [1, 1
+ floor(log2(max(width, height, depth)))].

The cudaChannelFormatDesc is defined as:

```c
struct cudaChannelFormatDesc {
  int x, y, z, w;
  enum cudaChannelFormatKind f;
};
```

where cudaChannelFormatKind is one of cudaChannelFormatKindSigned,
cudaChannelFormatKindUnsigned, or cudaChannelFormatKindFloat.

cudaMallocMipmappedArray() can allocate the following:

- A 1D mipmapped array is allocated if the height and depth extents are both zero.
- A 2D mipmapped array is allocated if only the depth extent is zero.
- A 3D mipmapped array is allocated if all three extents are non-zero.
A 1D layered CUDA mipmapped array is allocated if only the height extent is zero and the cudaArrayLayered flag is set. Each layer is a 1D mipmapped array. The number of layers is determined by the depth extent.

A 2D layered CUDA mipmapped array is allocated if all three extents are non-zero and the cudaArrayLayered flag is set. Each layer is a 2D mipmapped array. The number of layers is determined by the depth extent.

A cubemap CUDA mipmapped array is allocated if all three extents are non-zero and the cudaArrayCubemap flag is set. Width must be equal to height, and depth must be six. The order of the six layers in memory is the same as that listed in cudaGraphicsCubeFace.

A cubemap layered CUDA mipmapped array is allocated if all three extents are non-zero, and both, cudaArrayCubemap and cudaArrayLayered flags are set. Width must be equal to height, and depth must be a multiple of six. A cubemap layered CUDA mipmapped array is a special type of 2D layered CUDA mipmapped array that consists of a collection of cubemap mipmapped arrays. The first six layers represent the first cubemap mipmapped array, the next six layers form the second cubemap mipmapped array, and so on.

The flags parameter enables different options to be specified that affect the allocation, as follows.

- **cudaArrayDefault**: This flag’s value is defined to be 0 and provides default mipmapped array allocation
- **cudaArrayLayered**: Allocates a layered CUDA mipmapped array, with the depth extent indicating the number of layers
- **cudaArrayCubemap**: Allocates a cubemap CUDA mipmapped array. Width must be equal to height, and depth must be six. If the cudaArrayLayered flag is also set, depth must be a multiple of six.
- **cudaArraySurfaceLoadStore**: This flag indicates that individual mipmap levels of the CUDA mipmapped array will be read from or written to using a surface reference.
- **cudaArrayTextureGather**: This flag indicates that texture gather operations will be performed on the CUDA array. Texture gather can only be performed on 2D CUDA mipmapped arrays, and the gather operations are performed only on the most detailed mipmap level.
- **cudaArraySparse**: Allocates a CUDA array without physical backing memory. The subregions within this sparse array can later be mapped to physical memory by calling cuMemMapArrayAsync. This flag can only be used for creating 2D, 3D or 2D layered sparse CUDA mipmapped arrays. The physical backing memory must be allocated via cuMemCreate.

The width, height and depth extents must meet certain size requirements as listed in the following table. All values are specified in elements.
<p>| CUDA array type | Valid extents that must always be met { | Valid extents with cudaArraySurfaceLoadStore set { |</p>
<table>
<thead>
<tr>
<th></th>
<th>(width range in elements), (height range), (depth range)}</th>
<th>(width range in elements), (height range), (depth range)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D</td>
<td>{(1, maxTexture1DMipmap[0]), 0, 0}</td>
<td>{(1, maxSurface1D), 0, 0}</td>
</tr>
<tr>
<td>2D</td>
<td>{(1, maxTexture2DMipmap[0]), (1, maxTexture2DMipmap[1]), 0}</td>
<td>{(1, maxSurface2D[0]), (1, maxSurface2D[1]), 0}</td>
</tr>
<tr>
<td>3D</td>
<td>{(1, maxTexture3D[0]), (1, maxTexture3D[1]), (1, maxTexture3D[2])} OR {(1, maxTexture3DAlt[0]), (1, maxTexture3DAlt[1]), (1, maxTexture3DAlt[2])}</td>
<td>{(1, maxSurface3D[0]), (1, maxSurface3D[1]), (1, maxSurface3D[2])}</td>
</tr>
<tr>
<td>1D Layered</td>
<td>{(1, maxTexture1DLayered[0]), 0, (1, maxTexture1DLayered[1])}</td>
<td>{(1, maxSurface1DLayered[0]), 0, (1, maxSurface1DLayered[1])}</td>
</tr>
<tr>
<td>2D Layered</td>
<td>{(1, maxTexture2DLayered[0]), (1, maxTexture2DLayered[1]), (1, maxTexture2DLayered[2])}</td>
<td>{(1, maxSurface2DLayered[0]), (1, maxSurface2DLayered[1]), (1, maxSurface2DLayered[2])}</td>
</tr>
<tr>
<td>Cubemap</td>
<td>{(1, maxTextureCubemap), (1, maxSurfaceCubemap), 6}</td>
<td>{(1, maxSurfaceCubemap), (1, maxSurfaceCubemap), 6}</td>
</tr>
<tr>
<td>Cubemap Layered</td>
<td>{(1, maxTextureCubemapLayered[0]), (1, maxTextureCubemapLayered[1]), (1, maxTextureCubemapLayered[2])}</td>
<td>{(1, maxSurfaceCubemapLayered[0]), (1, maxSurfaceCubemapLayered[1]), (1, maxSurfaceCubemapLayered[2])}</td>
</tr>
</tbody>
</table>

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**

`cudaMalloc3D`, `cudaMalloc`, `cudaMallocPitch`, `cudaFree`, `cudaFreeArray`, `cudaMallocHost` [C API], `cudaFreeHost`, `cudaHostAlloc`, `make_cudeExtent`, `cuMipmappedArrayCreate`
__host__ cudaError_t cudaMemcpy (void **devPtr, const void *hostPtr, size_t size, cudaMemcpyKind kind)

Allocates and copies data to/from the device.

Parameters

devPtr - Pointer to allocated pitched device memory
hostPtr - Pointer to host memory
size - Size of the copy
kind - Copy direction (cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyHostToHost, cudaMemcpyDeviceToDevice)

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation, cudaErrorInitializationError, cudaErrorInsufficientDriver, cudaErrorNoDevice

Description

Allocates at least size bytes of linear memory on the device and returns a pointer to the allocated memory. The function may pad the allocation to ensure that corresponding pointers in any given row will continue to meet the alignment requirements for coalescing as the address is updated from row to row. The pitch returned in *pitch by cudaMemcpy() is the width in bytes of the allocation. The intended usage of *pitch is as a separate parameter of the allocation, used to compute addresses within the 2D array. Given the row and column of an array element of type T, the address is computed as:

T* pElement = (T*)((char*)BaseAddress + Row * pitch) + Column;

For allocations of 2D arrays, it is recommended that programmers consider performing pitch allocations using cudaMemcpy(). Due to pitch alignment restrictions in the hardware, this is especially true if the application will be performing 2D memory copies between different regions of device memory (whether linear memory or CUDA arrays).

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaMemcpyInitializationError, cudaMemcpyInsufficientDriver or cudaMemcpyNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**__host__cudaError_t cudaMemAdvise (const void *devPtr, size_t count, cudaMemoryAdvise advice, int device)**

Advise about the usage of a given memory range.

**Parameters**

- **devPtr**
  - Pointer to memory to set the advice for
- **count**
  - Size in bytes of the memory range
- **advice**
  - Advice to be applied for the specified memory range
- **device**
  - Device to apply the advice for

**Returns**

- `cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`

**Description**

Advise the Unified Memory subsystem about the usage pattern for the memory range starting at `devPtr` with a size of `count` bytes. The start address and end address of the memory range will be rounded down and rounded up respectively to be aligned to CPU page size before the advice is applied. The memory range must refer to managed memory allocated via `cudaMallocManaged` or declared via `__managed__` variables. The memory range could also refer to system-allocated pageable memory provided it represents a valid, host-accessible region of memory and all additional constraints imposed by `advice` as outlined below are also satisfied. Specifying an invalid system-allocated pageable memory range results in an error being returned.

The `advice` parameter can take the following values:

- `cudaMemAdviseSetReadMostly`: This implies that the data is mostly going to be read from and only occasionally written to. Any read accesses from any processor to this
region will create a read-only copy of at least the accessed pages in that processor’s
memory. Additionally, if `cudaMemPrefetchAsync` is called on this region, it will create a
read-only copy of the data on the destination processor. If any processor writes to this
region, all copies of the corresponding page will be invalidated except for the one where
the write occurred. The `device` argument is ignored for this advice. Note that for a page
to be read-duplicated, the accessing processor must either be the CPU or a GPU that
has a non-zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`.
Also, if a context is created on a device that does not have the device attribute
`cudaDevAttrConcurrentManagedAccess` set, then read-duplication will not occur until
all such contexts are destroyed. If the memory region refers to valid system-allocated
pageable memory, then the accessing device must have a non-zero value for the device
attribute `cudaDevAttrPageableMemoryAccess` for a read-only copy to be created on that
device. Note however that if the accessing device also has a non-zero value for the device
attribute `cudaDevAttrPageableMemoryAccessUsesHostPageTables`, then setting this
advice will not create a read-only copy when that device accesses this memory region.

- **cudaMemAdviceUnsetReadMostly**: Undoes the effect of `cudaMemAdviceReadMostly` and
  also prevents the Unified Memory driver from attempting heuristic read-duplication on
  the memory range. Any read-duplicated copies of the data will be collapsed into a single
  copy. The location for the collapsed copy will be the preferred location if the page has a
  preferred location and one of the read-duplicated copies was resident at that location.
  Otherwise, the location chosen is arbitrary.

- **cudaMemAdviseSetPreferredLocation**: This advice sets the preferred location for the data
to be the memory belonging to `device`. Passing in `cudaCpuDeviceId` for `device` sets the
preferred location as host memory. If `device` is a GPU, then it must have a non-zero value
for the device attribute `cudaDevAttrConcurrentManagedAccess`. Setting the preferred
location does not cause data to migrate to that location immediately. Instead, it guides the
migration policy when a fault occurs on that memory region. If the data is already in its
preferred location and the faulting processor can establish a mapping without requiring
the data to be migrated, then data migration will be avoided. On the other hand, if the data
is not in its preferred location or if a direct mapping cannot be established, then it will be
migrated to the processor accessing it. It is important to note that setting the preferred
location does not prevent data prefetching done using `cudaMemPrefetchAsync`. Having
a preferred location can override the page thrash detection and resolution logic in the
Unified Memory driver. Normally, if a page is detected to be constantly thrashing between
for example host and device memory, the page may eventually be pinned to host memory
by the Unified Memory driver. But if the preferred location is set as device memory, then
the page will continue to thrash indefinitely. If `cudaMemAdviseSetReadMostly` is also set
on this memory region or any subset of it, then the policies associated with that advice
will override the policies of this advice, unless read accesses from `device` will not result
in a read-only copy being created on that device as outlined in description for the advice
`cudaMemAdviseSetReadMostly`. If the memory region refers to valid system-allocated
pageable memory, then `device` must have a non-zero value for the device attribute
**cudaDevAttrPageableMemoryAccess**: Additionally, if `device` has a non-zero value for the device attribute `cudaDevAttrPageableMemoryAccessUsesHostPageTables`, then this call has no effect. Note however that this behavior may change in the future.

- **cudaMemAdviseUnsetPreferredLocation**: Undoes the effect of `cudaMemAdviseSetPreferredLocation` and changes the preferred location to none.

- **cudaMemAdviseSetAccessedBy**: This advice implies that the data will be accessed by `device`. Passing in `cudaCpuDeviceId` for `device` will set the advice for the CPU. If `device` is a GPU, then the device attribute `cudaDevAttrConcurrentManagedAccess` must be non-zero. This advice does not cause data migration and has no impact on the location of the data per se. Instead, it causes the data to always be mapped in the specified processor’s page tables, as long as the location of the data permits a mapping to be established. If the data gets migrated for any reason, the mappings are updated accordingly. This advice is recommended in scenarios where data locality is not important, but avoiding faults is. Consider for example a system containing multiple GPUs with peer-to-peer access enabled, where the data located on one GPU is occasionally accessed by peer GPUs. In such scenarios, migrating data over to the other GPUs is not as important because the accesses are infrequent and the overhead of migration may be too high. But preventing faults can still help improve performance, and so having a mapping set up in advance is useful. Note that on CPU access of this data, the data may be migrated to host memory because the CPU typically cannot access device memory directly. Any GPU that had the `cudaMemAdviceSetAccessedBy` flag set for this data will now have its mapping updated to point to the page in host memory. If `cudaMemAdviseSetReadMostly` is also set on this memory region or any subset of it, then the policies associated with that advice will override the policies of this advice. Additionally, if the preferred location of this memory region or any subset of it is also `device`, then the policies associated with `cudaMemAdviseSetPreferredLocation` will override the policies of this advice. If the memory region refers to valid system-allocated pageable memory, then `device` must have a non-zero value for the device attribute `cudaDevAttrPageableMemoryAccess`. Additionally, if `device` has a non-zero value for the device attribute `cudaDevAttrPageableMemoryAccessUsesHostPageTables`, then this call has no effect.

- **cudaMemAdviseUnsetAccessedBy**: Undoes the effect of `cudaMemAdviseSetAccessedBy`. Any mappings to the data from `device` may be removed at any time causing accesses to result in non-fatal page faults. If the memory region refers to valid system-allocated pageable memory, then `device` must have a non-zero value for the device attribute `cudaDevAttrPageableMemoryAccess`. Additionally, if `device` has a non-zero value for the device attribute `cudaDevAttrPageableMemoryAccessUsesHostPageTables`, then this call has no effect.

---

**Note:**
Note that this function may also return error codes from previous, asynchronous launches.

This function exhibits asynchronous behavior for most use cases.

This function uses standard default stream semantics.

Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.

Note that as specified by cudaMemcpyHostToDevice, no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaMemcpy, cudaMemcpyPeer, cudaMemcpyAsync, cudaMemcpy3DPeerAsync, cudaMemcpyPrefetchAsync, cuMemAdvise

__host__ cudaError_t cudaMemcpy (void *dst, const void *src, size_t count, cudaMemcpyKind kind)

Copies data between host and device.

Parameters

dst
  - Destination memory address
src
  - Source memory address
count
  - Size in bytes to copy
kind
  - Type of transfer

Returns
cudaSuccess, cudaErrorInvalidValue, cudaMemcpyInvalidMemcpyDirection

Description

Copies count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing.
Calling `cudaMemcpy()` with dst and src pointers that do not match the direction of the copy results in an undefined behavior.

### Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.
- This function exhibits **synchronous** behavior for most use cases.
- Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

See also:
- `cudaMemcpy2D`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cuMemcpyDtoH`, `cuMemcpyHtoD`, `cuMemcpyDtoD`, `cuMemcpy`

```c
__host__cudaError_t cudaMemcpy2D (void *dst, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, cudaMemcpyKind kind)
```

Copies data between host and device.

**Parameters**

- **dst**
  - Destination memory address

- **dpitch**
  - Pitch of destination memory

- **src**
  - Source memory address
spitch
- Pitch of source memory

width
- Width of matrix transfer [columns in bytes]

height
- Height of matrix transfer [rows]

kind
- Type of transfer

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidPitchValue, cudaErrorInvalidMemcpyDirection

Description
Copies a matrix (height rows of width bytes each) from the memory area pointed to by src to the memory area pointed to by dst, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing. dpitch and spitch are the widths in memory in bytes of the 2D arrays pointed to by dst and src, including any padding added to the end of each row. The memory areas may not overlap. width must not exceed either dpitch or spitch. Calling cudaMemcpy2D with dst and src pointers that do not match the direction of the copy results in an undefined behavior. cudaMemcpy2D returns an error if dpitch or spitch exceeds the maximum allowed.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpy2D, no CUDA function may be called from callback. cudaMemcpyNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
- Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.
See also:
cudaMemcpy, cudaMemcpy2DToArray, cudaMemcpy2DFromArray,
cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol,
cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DToArrayAsync,
cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync,
cudaMemcpyFromSymbolAsync, cuMemcpy2D, cuMemcpy2DUnaligned

__host__cudaError_t cudaMemcpy2DArrayToArray(cudArray_t dst, size_t wOffsetDst, size_t hOffsetDst, 
cudArray_const_t src, size_t wOffsetSrc, size_t hOffsetSrc, size_t width, size_t height, 
cudaMemcpyKind kind)
Copies data between host and device.

**Parameters**

dst
- Destination memory address

wOffsetDst
- Destination starting X offset (columns in bytes)

hOffsetDst
- Destination starting Y offset (rows)

src
- Source memory address

wOffsetSrc
- Source starting X offset (columns in bytes)

hOffsetSrc
- Source starting Y offset (rows)

width
- Width of matrix transfer (columns in bytes)

height
- Height of matrix transfer (rows)

kind
- Type of transfer

**Returns**
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemcpyDirection
Description

Copies a matrix (height rows of width bytes each) from the CUDA array src starting at hOffsetSrc rows and wOffsetSrc bytes from the upper left corner to the CUDA array dst starting at hOffsetDst rows and wOffsetDst bytes from the upper left corner, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing. wOffsetDst + width must not exceed the width of the CUDA array dst. wOffsetSrc + width must not exceed the width of the CUDA array src.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.
- Note that this function may also return cudaMemcpyInitializationError, cudaMemcpyInsufficientDriver or cudaMemcpyNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpyAddCallback no CUDA function may be called from callback. cudaMemcpyNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cupyMemcpy, cudaMemcpy2D, cudaMemcpy2DToArray, cudaMemcpy2DFromArray, cudaMemcpyToSymbol, cudaMemcpyFromArraySymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DToArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromArraySymbolAsync, cudaMemcpyAsyncFromSymbolAsync, cudaMemcpy2D, cudaMemcpy2DUnaligned
__host__ __device__ cudaError_t cudaMemcpy2DAsync (void *dst, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, cudaMemcpyKind kind, cudaStream_t stream)

Copies data between host and device.

Parameters

dst
- Destination memory address
dpitch
- Pitch of destination memory
csrc
- Source memory address
spitch
- Pitch of source memory
width
- Width of matrix transfer (columns in bytes)
height
- Height of matrix transfer (rows)
kind
- Type of transfer
stream
- Stream identifier

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidPitchValue, cudaErrorInvalidMemcpyDirection

Description

Copies a matrix (height rows of width bytes each) from the memory area pointed to by src to the memory area pointed to by dst, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing. dpitch and spitch are the widths in memory in bytes of the 2D arrays pointed to by dst and src, including any padding added to the end of each row. The memory areas may not overlap. width must not exceed either dpitch or spitch.
Calling `cudaMemcpy2DAsync()` with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior. `cudaMemcpy2DAsync()` returns an error if `dpitch` or `spitch` is greater than the maximum allowed.

`cudaMemcpy2DAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero stream argument. If `kind` is `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost` and stream is non-zero, the copy may overlap with operations in other streams.

The device version of this function only handles device to device copies and cannot be given local or shared pointers.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.
- Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return `CUDA_ERROR_INVALID_VALUE`.

See also:
- `cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`, `cuMemcpy2DAsync`
__host__ cudaError_t cudaMemcpy2DFromArray (void *dst, size_t dpitch, cudaArray_const_t src, size_t wOffset, size_t hOffset, size_t width, size_t height, cudaMemcpyKind kind)

Copies data between host and device.

Parameters

dst
- Destination memory address
dpitch
- Pitch of destination memory
csrc
- Source memory address
wOffset
- Source starting X offset (columns in bytes)
hOffset
- Source starting Y offset (rows)
width
- Width of matrix transfer (columns in bytes)
height
- Height of matrix transfer (rows)
kind
- Type of transfer

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidPitchValue, cudaErrorInvalidMemcpyDirection

Description

Copies a matrix (height rows of width bytes each) from the CUDA array src starting at hOffset rows and wOffset bytes from the upper left corner to the memory area pointed to by dst, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing. dpitch is the width in memory in bytes of the 2D array pointed to by dst, including any padding added to the end of each row. wOffset + width must not exceed the width of the CUDA array src, width must not exceed dpitch. cudaMemcpy2DFromArray returns an error if dpitch exceeds the maximum allowed.
Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
- Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

See also:
cudaMemcpy, cudaMemcpy2D, cudaMemcpy2DToArray, cudaMemcpy2DToArrayArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpy2DFromArrayToArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, cuMemcpy2D, cuMemcpy2DUnaligned

__host__cudaError_t

__host__cudaError_t
cudaMemcpy2DFromArrayAsync (void *dst, size_t dpitch, cudaArray_const_t src, size_t wOffset, size_t hOffset, size_t width, size_t height, cudaMemcpyKind kind, cudaStream_t stream)

Copies data between host and device.

Parameters

**dst**
- Destination memory address

**dpitch**
- Pitch of destination memory

**src**
- Source memory address

**wOffset**
- Source starting X offset (columns in bytes)
Modules

CUDA Runtime API

hOffset
- Source starting Y offset (rows)

width
- Width of matrix transfer (columns in bytes)

height
- Height of matrix transfer (rows)

kind
- Type of transfer

stream
- Stream identifier

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidPitchValue, cudaErrorInvalidMemcpyDirection

Description

Copies a matrix (height rows of width bytes each) from the CUDA array src starting at hOffset rows and wOffset bytes from the upper left corner to the memory area pointed to by dst, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing.

dpitch is the width in memory in bytes of the 2D array pointed to by dst, including any padding added to the end of each row. wOffset + width must not exceed the width of the CUDA array src. width must not exceed dpitch. cudaMemcpy2DFromArrayAsync() returns an error if dpitch exceeds the maximum allowed.

cudaMemcpy2DFromArrayAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and stream is non-zero, the copy may overlap with operations in other streams.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.

Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return `CUDA_ERROR_INVALID_VALUE`.

See also:

- `cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`,
- `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`,
- `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpy2DToArrayAsync`,
- `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`, `cuMemcpy2DAsync`

```c
__host__ cudaError_t cudaMemcpy2DToArray
(cuArray_t dst, size_t wOffset, size_t hOffset, const void *src, size_t spitch, size_t width, size_t height, cudaMemcpyKind kind)
```

Copies data between host and device.

**Parameters**

- **dst**
  - Destination memory address
- **wOffset**
  - Destination starting X offset (columns in bytes)
- **hOffset**
  - Destination starting Y offset (rows)
- **src**
  - Source memory address
- **spitch**
  - Pitch of source memory
- **width**
  - Width of matrix transfer (columns in bytes)
- **height**
  - Height of matrix transfer (rows)
kind
- Type of transfer

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidPitchValue, cudaErrorInvalidMemcpyDirection

Description
Copies a matrix [height rows of width bytes each] from the memory area pointed to by src to the CUDA array dst starting at hOffset rows and wOffset bytes from the upper left corner, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing. spitch is the width in memory in bytes of the 2D array pointed to by src, including any padding added to the end of each row. wOffset + width must not exceed the width of the CUDA array dst. width must not exceed spitch. cudaMemcpy2DToArray() returns an error if spitch exceeds the maximum allowed.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.
- Note that this function may also return cudaMemcpyInitializationError, cudaMemcpyInsufficientDriver or cudaMemcpyNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpyStreamAddCallback no CUDA function may be called from callback. cudaMemcpyNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
- Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

See also:
ccudaMemcpy, cudaMemcpy2D, cudaMemcpy2DFromArray, cudaMemcpy2DToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DToArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, cudaMemcpy2D, cudaMemcpy2DUnaligned
__host__ cudaError_t cudaMemcpy2DToArrayAsync (cudaArray_t dst, size_t wOffset, size_t hOffset, const void *src, size_t spitch, size_t width, size_t height, cudaMemcpyKind kind, cudaStream_t stream)

Copies data between host and device.

**Parameters**

- **dst**
  - Destination memory address
- **wOffset**
  - Destination starting X offset (columns in bytes)
- **hOffset**
  - Destination starting Y offset (rows)
- **src**
  - Source memory address
- **spitch**
  - Pitch of source memory
- **width**
  - Width of matrix transfer (columns in bytes)
- **height**
  - Height of matrix transfer (rows)
- **kind**
  - Type of transfer
- **stream**
  - Stream identifier

**Returns**

- cudaSuccess
- cudaErrorInvalidValue
- cudaErrorInvalidPitchValue
- cudaErrorInvalidMemcpyDirection

**Description**

Copies a matrix (height rows of width bytes each) from the memory area pointed to by src to the CUDA array dst starting at hOffset rows and wOffset bytes from the upper left corner, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing. spitch is the width in memory in bytes of the 2D array pointed to by src, including any padding added to the end of each row. wOffset + width must not exceed the width of the
CUDA array dst. width must not exceed spitch. `cudaMemcpy2DToArrayAsync()` returns an error if spitch exceeds the maximum allowed.

`cudaMemcpy2DToArrayAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost` and stream is non-zero, the copy may overlap with operations in other streams.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.
- Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return `CUDA_ERROR_INVALID_VALUE`.

See also:
- `cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`,
- `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`,
- `cudaMemcpyAsync`, `cudaMemcpy2DAsync`,
- `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`,
- `cudaMemcpyFromSymbolAsync`, `cuMemcpy2DAsync`.

`__host__cudaError_t cudaMemcpy3D (const cudaMemcpy3DParms *p)`
Copies data between 3D objects.

**Parameters**
- `p` - 3D memory copy parameters
Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidPitchValue, cudaErrorInvalidMemcpyDirection

Description

```c
struct cudaExtent {
    size_t width;
    size_t height;
    size_t depth;
};
struct cudaExtent
    make_cudaExtent(size_t w, size_t h, size_t d);

struct cudaPos {
    size_t x;
    size_t y;
    size_t z;
};
struct cudaPos
    make_cudaPos(size_t x, size_t y, size_t z);

struct cudaMemcpy3DParms {
    cudaArray_t srcArray;
    struct cudaPos srcPos;
    struct cudaMemcpy3DParms srcPtr;
    cudaArray_t dstArray;
    struct cudaPos dstPos;
    struct cudaMemcpy3DParms dstPtr;
    struct cudaExtent extent;
    enum cudaMemcpyKind kind;
};
```

cudaMemcpy3D() copies data between two 3D objects. The source and destination objects may be in either host memory, device memory, or a CUDA array. The source, destination, extent, and kind of copy performed is specified by the cudaMemcpy3DParms struct which should be initialized to zero before use:

```c
cudaMemcpy3DParms myParms = {0};
```

The struct passed to cudaMemcpy3D() must specify one of srcArray or srcPtr and one of dstArray or dstPtr. Passing more than one non-zero source or destination will cause cudaMemcpy3D() to return an error.

The srcPos and dstPos fields are optional offsets into the source and destination objects and are defined in units of each object’s elements. The element for a host or device pointer is assumed to be unsigned char.
The `extent` field defines the dimensions of the transferred area in elements. If a CUDA array is participating in the copy, the extent is defined in terms of that array’s elements. If no CUDA array is participating in the copy then the extents are defined in elements of `unsigned char`.

The `kind` field defines the direction of the copy. It must be one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing. For `cudaMemcpyHostToHost` or `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost` passed as kind and cudaArray type passed as source or destination, if the kind implies cudaArray type to be present on the host, `cudaMemcpy3D()` will disregard that implication and silently correct the kind based on the fact that cudaArray type can only be present on the device.

If the source and destination are both arrays, `cudaMemcpy3D()` will return an error if they do not have the same element size.

The source and destination object may not overlap. If overlapping source and destination objects are specified, undefined behavior will result.

The source object must entirely contain the region defined by `srcPos` and `extent`. The destination object must entirely contain the region defined by `dstPos` and `extent`.

`cudaMemcpy3D()` returns an error if the pitch of `srcPtr` or `dstPtr` exceeds the maximum allowed. The pitch of a `cudaPitchedPtr` allocated with `cudaMalloc3D()` will always be valid.

---

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits **synchronous** behavior for most use cases.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

---

See also:

- `cudaMalloc3D`, `cudaMalloc3DArray`, `cudaMemset3D`, `cudaMemcpy3DAsync`, `cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`, `make_cudaExtent`, `make_cudaPos`, `cuMemcpy3D`
**__host__ __device__ cudaError_t**
cudaMemcpy3DAsync (const cudaMemcpy3DParms *p, cudaStream_t stream)

Copies data between 3D objects.

**Parameters**

- **p**
  - 3D memory copy parameters

- **stream**
  - Stream identifier

**Returns**

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidPitchValue, cudaErrorInvalidMemcpyDirection

**Description**

```c
struct cudaExtent {
   size_t width;
   size_t height;
   size_t depth;
};

struct cudaMemcpy3DParms {
   cudaArray_t srcArray;
   struct cudaPos srcPos;
   struct cudaPitchedPtr srcPtr;
   cudaArray_t dstArray;
   struct cudaPos dstPos;
   struct cudaPitchedPtr dstPtr;
   struct cudaExtent extent;
   enum cudaMemcpyKind kind;
};
```

```c
struct cudaPos {
   size_t x;
   size_t y;
   size_t z;
};

struct make_cudaPos(size_t x, size_t y, size_t z);
```

```c
struct cudaMemcpy3DParms {
   cudaArray_t srcArray;
   struct cudaPos srcPos;
   struct cudaPitchedPtr srcPtr;
   cudaArray_t dstArray;
   struct cudaPos dstPos;
   struct cudaPitchedPtr dstPtr;
   struct cudaExtent extent;
   enum cudaMemcpyKind kind;
};
```
`cudaMemcpy3DAsync()` copies data between two 3D objects. The source and destination objects may be in either host memory, device memory, or a CUDA array. The source, destination, extent, and kind of copy performed is specified by the `cudaMemcpy3DParms` struct which should be initialized to zero before use:

```c
cudaMemcpy3DParms myParms = {0};
```

The struct passed to `cudaMemcpy3DAsync()` must specify one of `srcArray` or `srcPtr` and one of `dstArray` or `dstPtr`. Passing more than one non-zero source or destination will cause `cudaMemcpy3DAsync()` to return an error.

The `srcPos` and `dstPos` fields are optional offsets into the source and destination objects and are defined in units of each object’s elements. The element for a host or device pointer is assumed to be `unsigned char`. For CUDA arrays, positions must be in the range [0, 2048] for any dimension.

The `extent` field defines the dimensions of the transferred area in elements. If a CUDA array is participating in the copy, the extent is defined in terms of that array’s elements. If no CUDA array is participating in the copy then the extents are defined in elements of `unsigned char`.

The `kind` field defines the direction of the copy. It must be one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing. For `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToDevice` passed as kind and cudaArray type passed as source or destination, if the kind implies cudaArray type to be present on the host, `cudaMemcpy3DAsync()` will disregard that implication and silently correct the kind based on the fact that cudaArray type can only be present on the device.

If the source and destination are both arrays, `cudaMemcpy3DAsync()` will return an error if they do not have the same element size.

The source and destination object may not overlap. If overlapping source and destination objects are specified, undefined behavior will result.

The source object must lie entirely within the region defined by `srcPos` and `extent`. The destination object must lie entirely within the region defined by `dstPos` and `extent`.

`cudaMemcpy3DAsync()` returns an error if the pitch of `srcPtr` or `dstPtr` exceeds the maximum allowed. The pitch of a `cudaPitchedPtr` allocated with `cudaMalloc3D()` will always be valid.

`cudaMemcpy3DAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost` and stream is non-zero, the copy may overlap with operations in other streams.

The device version of this function only handles device to device copies and cannot be given local or shared pointers.
__host__ cudaError_t cudaMemcpy3DPeer (const cudaMemcpy3DPeerParms *p)

Copies memory between devices.

Parameters

p
- Parameters for the memory copy

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevice

Description

Perform a 3D memory copy according to the parameters specified in p. See the definition of the cudaMemcpy3DPeerParms structure for documentation of its parameters.

Note that this function is synchronous with respect to the host only if the source or destination of the transfer is host memory. Note also that this copy is serialized with respect to all pending...
and future asynchronous work in to the current device, the copy’s source device, and the copy’s destination device (use cudaMemcpy3DPeerAsync to avoid this synchronization).

Note:

‣ Note that this function may also return error codes from previous, asynchronous launches.
‣ This function exhibits synchronous behavior for most use cases.
‣ Note that this function may also return cudaMemcpy3DPeerParms *p, cudaStream_t stream

See also:
cudaMemcpy, cudaMemcpyPeer, cudaMemcpyAsync, cudaMemcpyPeerAsync, cudaMemcpy3DPeerAsync, cuMemcpy3DPeer

__host__cudaError_t cudaMemcpy3DPeerAsync (const cudaMemcpy3DPeerParms *p, cudaStream_t stream)
Copies memory between devices asynchronously.

Parameters
p
  - Parameters for the memory copy
stream
  - Stream identifier

Returns
cudaSuccess, cudaMemcpy3DPeerParms *p, cudaStream_t stream

description
Perform a 3D memory copy according to the parameters specified in p. See the definition of the cudaMemcpy3DPeerParms structure for documentation of its parameters.
Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotAllowed may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaMemcpy, cudaMemcpyPeer, cudaMemcpyAsync, cudaMemcpyPeerAsync, cudaMemcpy3DPeerAsync, cuMemcpy3DPeerAsync

__host__ __device__ cudaMemcpyAsync

( void *dst, const void *src, size_t count, cudaMemcpyKind kind, cudaStream_t stream )

Copies data between host and device.

Parameters

dst
  - Destination memory address
src
  - Source memory address
count
  - Size in bytes to copy
kind
  - Type of transfer
stream
  - Stream identifier

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemcpyDirection
Description

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing.

The memory areas may not overlap. Calling `cudaMemcpyAsync()` with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior.

`cudaMemcpyAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero stream argument. If `kind` is `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost` and the stream is non-zero, the copy may overlap with operations in other streams.

The device version of this function only handles device to device copies and cannot be given local or shared pointers.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.
- Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpy2DAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpy2DFromArrayAsync`.
__host__ cudaError_t cudaMemcpyFromSymbol (void *dst, const void *symbol, size_t count, size_t offset, cudaMemcpyKind kind)

Copies data from the given symbol on the device.

Parameters

dst
- Destination memory address

symbol
- Device symbol address

count
- Size in bytes to copy

offset
- Offset from start of symbol in bytes

kind
- Type of transfer

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSymbol, cudaMemcpyDirection, cudaMemcpyDefault, cudaMemcpyNoKernelImageForDevice

Description

Copies count bytes from the memory area pointed to by offset bytes from the start of symbol symbol to the memory area pointed to by dst. The memory areas may not overlap. symbol is a variable that resides in global or constant memory space. kind can be either cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.
- Use of a string naming a variable as the symbol parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.
Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.

Note that as specified by cudaMemcpyAddCallback no CUDA function may be called from callback. cudaMemcpyNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaMemcpy, cudaMemcpy2D, cudaMemcpy2DtoArray, cudaMemcpy2DFromArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DtoArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, cuMemcpy, cuMemcpyDtoH, cuMemcpyDtoD

__host__ cudaError_t cudaMemcpyFromSymbolAsync (void *dst, const void *symbol, size_t count, size_t offset, cudaMemcpyKind kind, cudaStream_t stream)

Copies data from the given symbol on the device.

Parameters

dst
  - Destination memory address
symbol
  - Device symbol address
count
  - Size in bytes to copy
offset
  - Offset from start of symbol in bytes
kind
  - Type of transfer
stream
  - Stream identifier

Returns
cudaSuccess, cudaMemcpyInvalidValue, cudaMemcpyInvalidSymbol, cudaMemcpyInvalidMemcpyDirection, cudaMemcpyNoKernelImageForDevice

Description

Copies count bytes from the memory area pointed to by offset bytes from the start of symbol symbol to the memory area pointed to by dst. The memory areas may not overlap.
symbol is a variable that resides in global or constant memory space. kind can be either cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing.

cudaMemcpyFromSymbolAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is cudaMemcpyDeviceToHost and stream is non-zero, the copy may overlap with operations in other streams.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
- Use of a string naming a variable as the symbol parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpyPeer no CUDA function may be called from callback. cudaMemcpyNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaMemcpy, cudaMemcpy2D, cudaMemcpy2DToArray, cudaMemcpy2DFromArray, cudaMemcpy2DArrayToArray, cudaMemcpy2DFromArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpyDtoHAsync, cudaMemcpyDtoDAsync

__host__cudaError_t cudaMemcpyPeer (void *dst, int dstDevice, const void *src, int srcDevice, size_t count)
Copies memory between two devices.

Parameters

dst
- Destination device pointer
dstDevice
  - Destination device
dst
- Source device pointer
dstDevice
  - Source device
count
  - Size of memory copy in bytes

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevice

Description
Copies memory from one device to memory on another device. dst is the base device pointer of the destination memory and dstDevice is the destination device. src is the base device pointer of the source memory and srcDevice is the source device. count specifies the number of bytes to copy.

Note that this function is asynchronous with respect to the host, but serialized with respect all pending and future asynchronous work in to the current device, srcDevice, and dstDevice (use cudaMemcpyPeerAsync to avoid this synchronization).

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpyPeerAsync no CUDA function may be called from callback. cudaMemcpyPeerAsync may be returned as a diagnostic in such case.

See also:
cudaMemcpy, cudaMemcpyAsync, cudaMemcpyPeerAsync, cudaMemcpy3DPeerAsync, cuMemcpyPeer
__host__ cudaError_t cudaMemcpyPeerAsync (void *dst, int dstDevice, const void *src, int srcDevice, size_t count, cudaStream_t stream)

Copies memory between two devices asynchronously.

**Parameters**

dst
- Destination device pointer
dstDevice
- Destination device
src
- Source device pointer
srcDevice
- Source device
count
- Size of memory copy in bytes
stream
- Stream identifier

**Returns**
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevice

**Description**

Copies memory from one device to memory on another device. dst is the base device pointer of the destination memory and dstDevice is the destination device. src is the base device pointer of the source memory and srcDevice is the source device. count specifies the number of bytes to copy.

Note that this function is asynchronous with respect to the host and all work on other devices.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cud memcpy, cudaMemcpyPeer, cudaMemcpyAsync, cudaMemcpy3DPeerAsync, cuMemcpyPeerAsync

__host__cudaError_t cudaMemcpyToSymbol (const void *symbol, const void *src, size_t count, size_t offset, cudaMemcpyKind kind)

Copies data to the given symbol on the device.

Parameters
symbol
- Device symbol address
src
- Source memory address
count
- Size in bytes to copy
offset
- Offset from start of symbol in bytes
kind
- Type of transfer

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSymbol, cudaErrorInvalidMemcpyDirection, cudaErrorNoKernelImageForDevice

Description
Copies count bytes from the memory area pointed to by src to the memory area pointed to by offset bytes from the start of symbol symbol. The memory areas may not overlap. symbol is a variable that resides in global or constant memory space. kind can be either cudaMemcpyHostToDevice, cudaMemcpyDeviceTo Device, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing.
Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.
- Use of a string naming a variable as the symbol parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpy2DAsync no CUDA function may be called from callback. cudaMemcpyNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaMemcpy, cudaMemcpy2D, cudaMemcpy2DToArray, cudaMemcpy2DFromArray, cudaMemcpy2DToArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpy2DAsync, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DArrayToArray, cudaMemcpyFromSymbolAsync, cudaMemcpyToSymbolAsync, cudaMemcpy2DAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyFromSymbolAsync, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DFromArrayAsync.

cudaError_t cudaMemcpyToSymbolAsync(const void *symbol, const void *src, size_t count, size_t offset, cudaMemcpyKind kind, cudaStream_t stream)

Copies data to the given symbol on the device.

Parameters
symbol
- Device symbol address
src
- Source memory address
count
- Size in bytes to copy
offset
- Offset from start of symbol in bytes
kind
- Type of transfer
stream
- Stream identifier

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSymbol,
cudaErrorInvalidMemcpyDirection, cudaErrorNoKernelImageForDevice

Description
Copies count bytes from the memory area pointed to by src to the memory area pointed to by offset bytes from the start of symbol. The memory areas may not overlap. symbol is a variable that resides in global or constant memory space. kind can be either cudaMemcpyHostToDevice, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing.

cudaMemcpyToSymbolAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is cudaMemcpyHostToDevice and stream is non-zero, the copy may overlap with operations in other streams.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
- Use of a string naming a variable as the symbol parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpyAsync, cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaMemcpy, cudaMemcpy2D, cudaMemcpy2DToArray, cudaMemcpy2DFromArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DToArrayAsync.
__host__cudaError_t cudaMemGetInfo (size_t *free, size_t *total)

Gets free and total device memory.

Parameters

**free**
- Returned free memory in bytes

**total**
- Returned total memory in bytes

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorLaunchFailure

Description

Returns in **total** the total amount of memory available to the current context. Returns in **free** the amount of memory on the device that is free according to the OS. CUDA is not guaranteed to be able to allocate all of the memory that the OS reports as free. In a multi-tenant situation, free estimate returned is prone to race condition where a new allocation/free done by a different process or a different thread in the same process between the time when free memory was estimated and reported, will result in deviation in free value reported and actual free memory.

The integrated GPU on Tegra shares memory with CPU and other component of the SoC. The free and total values returned by the API excludes the SWAP memory space maintained by the OS on some platforms. The OS may move some of the memory pages into swap area as the GPU or CPU allocate or access memory. See Tegra app note on how to calculate total and free memory on Tegra.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
See also:

cuMemGetInfo

__host__ cudaError_t cudaMemPrefetchAsync
(const void *devPtr, size_t count, int dstDevice,
cudaStream_t stream)
Prefetches memory to the specified destination device.

Parameters

devPtr
  - Pointer to be prefetched
count
  - Size in bytes
dstDevice
  - Destination device to prefetch to
stream
  - Stream to enqueue prefetch operation

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevice

Description

Prefetches memory to the specified destination device. devPtr is the base device pointer of the memory to be prefetched and dstDevice is the destination device. count specifies the number of bytes to copy. stream is the stream in which the operation is enqueued. The memory range must refer to managed memory allocated via cudaMemcpyManaged or declared via __managed__ variables.

Passing in cudaCpuDeviceId for dstDevice will prefetch the data to host memory. If dstDevice is a GPU, then the device attribute cudaDevAttrConcurrentManagedAccess must be non-zero. Additionally, stream must be associated with a device that has a non-zero value for the device attribute cudaDevAttrConcurrentManagedAccess.

The start address and end address of the memory range will be rounded down and rounded up respectively to be aligned to CPU page size before the prefetch operation is enqueued in the stream.

If no physical memory has been allocated for this region, then this memory region will be populated and mapped on the destination device. If there’s insufficient memory to prefetch the desired region, the Unified Memory driver may evict pages from other cudaMemcpyManaged allocations to host memory in order to make room. Device memory allocated using cudaMemcpy or cudaMemcpyArray will not be evicted.
By default, any mappings to the previous location of the migrated pages are removed and mappings for the new location are only setup on \texttt{dstDevice}. The exact behavior however also depends on the settings applied to this memory range via \texttt{cudaMemAdvise} as described below:

If \texttt{cudaMemAdviseSetReadMostly} was set on any subset of this memory range, then that subset will create a read-only copy of the pages on \texttt{dstDevice}.

If \texttt{cudaMemAdviseSetPreferredLocation} was called on any subset of this memory range, then the pages will be migrated to \texttt{dstDevice} even if \texttt{dstDevice} is not the preferred location of any pages in the memory range.

If \texttt{cudaMemAdviseSetAccessedBy} was called on any subset of this memory range, then mappings to those pages from all the appropriate processors are updated to refer to the new location if establishing such a mapping is possible. Otherwise, those mappings are cleared.

Note that this API is not required for functionality and only serves to improve performance by allowing the application to migrate data to a suitable location before it is accessed. Memory accesses to this range are always coherent and are allowed even when the data is actively being migrated.

Note that this function is asynchronous with respect to the host and all work on other devices.

\begin{itemize}
\item Note that this function may also return error codes from previous, asynchronous launches.
\item This function exhibits \texttt{asynchronous} behavior for most use cases.
\item This function uses standard \texttt{default stream} semantics.
\item Note that this function may also return \texttt{cudaErrorInitializationError}, \texttt{cudaErrorInsufficientDriver} or \texttt{cudaErrorNoDevice} if this call tries to initialize internal CUDA RT state.
\item Note that as specified by \texttt{cudaStreamAddCallback} no CUDA function may be called from callback. \texttt{cudaErrorNotPermitted} may, but is not guaranteed to, be returned as a diagnostic in such case.
\end{itemize}

See also:

\texttt{cudaMemcpy}, \texttt{cudaMemcpyPeer}, \texttt{cudaMemcpyAsync}, \texttt{cudaMemcpy3DPeerAsync}, \texttt{cudaMemAdvise}, \texttt{cuMemPrefetchAsync}
`__host__ cudaError_t cudaMemRangeGetAttribute` (void *data, size_t dataSize, cudaMemRangeAttribute attribute, const void *devPtr, size_t count)

Query an attribute of a given memory range.

Parameters

- **data**
  - A pointer to a memory location where the result of each attribute query will be written to.
- **dataSize**
  - Array containing the size of data
- **attribute**
  - The attribute to query
- **devPtr**
  - Start of the range to query
- **count**
  - Size of the range to query

Returns

- **cudaSuccess**, **cudaErrorInvalidValue**

Description

Query an attribute about 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.

The `attribute` parameter can take the following values:

- **cudaMemRangeAttributeReadMostly**: If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. The result returned will be 1 if all pages in the given memory range have read-duplication enabled, or 0 otherwise.

- **cudaMemRangeAttributePreferredLocation**: If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. The result returned will be a GPU device id if all pages in the memory range have that GPU as their preferred location, or it will be `cudaCpuDeviceId` if all pages in the memory range have the CPU as their preferred location, or it will be `cudaInvalidDeviceId` 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. Note that the actual location of the pages in the memory range at the time of the query may be different from the preferred location.

- **cudaMemRangeAttributeAccessedBy**: If this attribute is specified, `data` will be interpreted as an array of 32-bit integers, and `dataSize` must be a non-zero multiple of 4. The result
returned will be a list of device ids that had cudaMemAdviceSetAccessedBy set for that entire memory range. If any device does not have that advice set for the entire memory range, that device will not be included. If data is larger than the number of devices that have that advice set for that memory range, cudaInvaldidDeviceId will be returned in all the extra space provided. For ex., if dataSize is 12 (i.e. data has 3 elements) and only device 0 has the advice set, then the result returned will be { 0, cudaInvaldidDeviceId, cudaInvaldidDeviceId }. If data is smaller than the number of devices that have that advice set, then only as many devices will be returned as can fit in the array. There is no guarantee on which specific devices will be returned, however.

- cudaMemRangeAttributeLastPrefetchLocation: If this attribute is specified, data will be interpreted as a 32-bit integer, and dataSize must be 4. The result returned will be the last location to which all pages in the memory range were prefetched explicitly via cudaMemPrefetchAsync. This will either be a GPU id or cudaCpuDeviceId depending on whether the last location for prefetch was a GPU or the CPU respectively. If any page in the memory range was never explicitly prefetched or if all pages were not prefetched to the same location, cudaInvaldidDeviceId will be returned. 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.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotAllowed may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaMemRangeGetAttributes, cudaMemPrefetchAsync, cudaMemAdvise, cuMemRangeGetAttribute
__host__ cudaError_t cudaMemRangeGetAttributes
(void **data, size_t *dataSizes,
cudaMemRangeAttribute *attributes, size_t
numAttributes, const void *devPtr, size_t count)

Query attributes of a given memory range.

Parameters

data
- A two-dimensional array containing pointers to memory locations where the result of
each attribute query will be written to.
dataSizes
- Array containing the sizes of each result
attributes
- An array of attributes to query [numAttributes and the number of attributes in this array
should match]
numAttributes
- Number of attributes to query
devPtr
- Start of the range to query
count
- Size of the range to query

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Query attributes 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. The attributes array will be interpreted to have numAttributes entries. The dataSizes array will also be interpreted to have numAttributes entries. The results of the query will be stored in data.

The list of supported attributes are given below. Please refer to cudaMemRangeGetAttribute for attribute descriptions and restrictions.

- cudaMemRangeAttributeReadMostly
- cudaMemRangeAttributePreferredLocation
- cudaMemRangeAttributeAccessedBy
- cudaMemRangeAttributeLastPrefetchLocation
__host__ cudaError_t cudaMemcpy (void *devPtr, int value, size_t count)

Initializes or sets device memory to a value.

Parameters

devPtr
- Pointer to device memory

value
- Value to set for each byte of specified memory

count
- Size in bytes to set

Returns
cudaSuccess, cudaErrorInvalidValue.

Description

Fills the first count bytes of the memory area pointed to by devPtr with the constant byte value value.

Note that this function is asynchronous with respect to the host unless devPtr refers to pinned host memory.
See also memset synchronization details.

Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.

Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cuMemsetD8, cuMemsetD16, cuMemsetD32

__host__ cudaError_t cudaMemcpy2D (void *devPtr,
size_t pitch, int value, size_t width, size_t height)
Initializes or sets device memory to a value.

Parameters

devPtr
  - Pointer to 2D device memory
pitch
  - Pitch in bytes of 2D device memory (Unused if height is 1)
value
  - Value to set for each byte of specified memory
width
  - Width of matrix set (columns in bytes)
height
  - Height of matrix set (rows)

Returns
cudaSuccess, cudaErrorInvalidValue,

Description

Sets to the specified value value a matrix (height rows of width bytes each) pointed to by devPtr. pitch is the width in bytes of the 2D array pointed to by dstPtr, including any padding added to the end of each row. This function performs fastest when the pitch is one that has been passed back by cudaMemcpy2D().

Note that this function is asynchronous with respect to the host unless devPtr refers to pinned host memory.
Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- See also [memset synchronization details](#).
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

- `cudaMemset`, `cudaMemset3D`, `cudaMemsetAsync`, `cudaMemset2DAsync`, `cudaMemset3DAsync`, `cuMemsetD2D8`, `cuMemsetD2D16`, `cuMemsetD2D32`

```c
__host__ _device__ __cudaError_t
cudaMemset2DAsync (void *devPtr, size_t pitch, int value, size_t width, size_t height, cudaStream_t stream)
```

Initializes or sets device memory to a value.

**Parameters**

- **devPtr**
  - Pointer to 2D device memory
- **pitch**
  - Pitch in bytes of 2D device memory (Unused if `height` is 1)
- **value**
  - Value to set for each byte of specified memory
- **width**
  - Width of matrix set (columns in bytes)
- **height**
  - Height of matrix set (rows)
- **stream**
  - Stream identifier

**Returns**

- `cudaSuccess`, `cudaErrorInvalidValue`
Description

Sets to the specified value value a matrix [height rows of width bytes each] pointed by dstPtr. pitch is the width in bytes of the 2D array pointed to by dstPtr, including any padding added to the end of each row. This function performs fastest when the pitch is one that has been passed back by cudaMemcpyPitch().

cudaMemset2DAsync() is asynchronous with respect to the host, so the call may return before the memset is complete. The operation can optionally be associated to a stream by passing a non-zero stream argument. If stream is non-zero, the operation may overlap with operations in other streams.

The device version of this function only handles device to device copies and cannot be given local or shared pointers.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- See also memset synchronization details.
- This function uses standard default stream semantics.
- Note that this function may also return cudaMemcpyInitializationError, cudaMemcpyInsufficientDriver or cudaMemcpyNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpyAddCallback no CUDA function may be called from callback. cudaMemcpyNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaMemset, cudaMemcpy2D, cudaMemcpy3D, cudaMemcpyAsync, cudaMemcpy3DAsync, cuMemcpyD2D8Async, cuMemcpyD2D16Async, cuMemcpyD2D32Async

__host__cudaError_t cudaMemcpy3D (cudaPitchedPtr pitchedDevPtr, int value, cudaExtent extent)

Initializes or sets device memory to a value.

Parameters

pitchedDevPtr

- Pointer to pitched device memory

value

- Value to set for each byte of specified memory
**extent**
- Size parameters for where to set device memory (width field in bytes)

**Returns**
cudaSuccess, cudaErrorInvalidValue.

**Description**
Initializes each element of a 3D array to the specified value `value`. The object to initialize is defined by `pitchedDevPtr`. The pitch field of `pitchedDevPtr` is the width in memory in bytes of the 3D array pointed to by `pitchedDevPtr`, including any padding added to the end of each row. The `xsize` field specifies the logical width of each row in bytes, while the `ysize` field specifies the height of each 2D slice in rows. The pitch field of `pitchedDevPtr` is ignored when `height` and `depth` are both equal to 1.

The extents of the initialized region are specified as a width in bytes, a height in rows, and a depth in slices.

Extents with width greater than or equal to the `xsize` of `pitchedDevPtr` may perform significantly faster than extents narrower than the `xsize`. Secondarily, extents with height equal to the `ysize` of `pitchedDevPtr` will perform faster than when the height is shorter than the `ysize`.

This function performs fastest when the `pitchedDevPtr` has been allocated by `cudaMalloc3D()`. Note that this function is asynchronous with respect to the host unless `pitchedDevPtr` refers to pinned host memory.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- See also memset synchronization details.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpyAsync no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**
cudaMemset, cudaMemset2D, cudaMemsetAsync, cudaMemset2DAsync, cudaMemset3DAsync, cudaMalloc3D, make_cudPitchedPtr, make_cudExtent
__host__ __device__ cudaError_t cudaMemset3DAsync (cudaPitchedPtr pitchedDevPtr, int value, cudaExtent extent, cudaStream_t stream)

Initializes or sets device memory to a value.

Parameters
pitchedDevPtr
- Pointer to pitched device memory
value
- Value to set for each byte of specified memory
extent
- Size parameters for where to set device memory [width field in bytes]
stream
- Stream identifier

Returns
cudaSuccess, cudaErrorInvalidValue.

Description
Initializes each element of a 3D array to the specified value value. The object to initialize is defined by pitchedDevPtr. The pitch field of pitchedDevPtr is the width in memory in bytes of the 3D array pointed to by pitchedDevPtr, including any padding added to the end of each row. The xsize field specifies the logical width of each row in bytes, while the ysize field specifies the height of each 2D slice in rows. The pitch field of pitchedDevPtr is ignored when height and depth are both equal to 1.

The extents of the initialized region are specified as a width in bytes, a height in rows, and a depth in slices.

Extents with width greater than or equal to the xsize of pitchedDevPtr may perform significantly faster than extents narrower than the xsize. Secondarily, extents with height equal to the ysize of pitchedDevPtr will perform faster than when the height is shorter than the ysize.

This function performs fastest when the pitchedDevPtr has been allocated by cudaMalloc3D[].

cudaMemset3DAsync() is asynchronous with respect to the host, so the call may return before the memset is complete. The operation can optionally be associated to a stream by passing a non-zero stream argument. If stream is non-zero, the operation may overlap with operations in other streams.
The device version of this function only handles device to device copies and cannot be given local or shared pointers.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- See also **memset synchronization details**.
- This function uses standard **default stream** semantics.
- Note that this function may also return **cudaErrorInitializationError**, **cudaErrorInsufficientDriver** or **cudaErrorNoDevice** if this call tries to initialize internal CUDA RT state.
- Note that as specified by **cudaStreamAddCallback** no CUDA function may be called from callback. **cudaErrorNotPermitted** may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
- cudaMemset, cudaMemset2D, cudaMemset3D, cudaMemsetAsync, cudaMemset2DAync, cudaMemcpy3D, make_cudaPitchedPtr, make_cudaExtent

```c
__host__ __device__ cudaError_t cudaMemsetAsync(void *devPtr, int value, size_t count, cudaStream_t stream)
```

Initializes or sets device memory to a value.

**Parameters**
- **devPtr** - Pointer to device memory
- **value** - Value to set for each byte of specified memory
- **count** - Size in bytes to set
- **stream** - Stream identifier

**Returns**
- **cudaSuccess**, **cudaErrorInvalidValue**.
Description

Fills the first `count` bytes of the memory area pointed to by `devPtr` with the constant byte value `value`.

`cudaMemsetAsync()` is asynchronous with respect to the host, so the call may return before the memset is complete. The operation can optionally be associated to a stream by passing a non-zero `stream` argument. If `stream` is non-zero, the operation may overlap with operations in other streams.

The device version of this function only handles device to device copies and cannot be given local or shared pointers.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- See also [memset synchronization details](#).
- This function uses standard [default stream](#) semantics.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

`cudaMemset`, `cudaMemset2D`, `cudaMemset3D`, `cudaMemset2DAsync`, `cudaMemset3DAsync`, `cuMemsetD8Async`, `cuMemsetD16Async`, `cuMemsetD32Async`

```c
__host__cudaError_t
cudaMipmappedArrayGetSparseProperties(
cudaArraySparseProperties *sparseProperties,
cudaMipmappedArray_t mipmap)
```

Returns the layout properties of a sparse CUDA mipmapped array.

Parameters

`sparseProperties`

- Pointer to return `cudaArraySparseProperties`
mipmap
- The CUDA mipmapped array to get the sparse properties of

Returns
cudaSuccess cudaErrorInvalidValue

Description
Returns the sparse array layout properties in sparseProperties. If the CUDA mipmapped array is not allocated with flag cudaArraySparse cudaErrorInvalidValue will be returned.

For non-layered CUDA mipmapped arrays, cudaArraySparseProperties::miptailSize returns the size of the mip tail region. The mip tail region includes all mip levels whose width, height or depth is less than that of the tile. For layered CUDA mipmapped arrays, if cudaArraySparseProperties::flags contains cudaArraySparsePropertiesSingleMipTail, then cudaArraySparseProperties::miptailSize specifies the size of the mip tail of all layers combined. Otherwise, cudaArraySparseProperties::miptailSize specifies mip tail size per layer. The returned value of cudaArraySparseProperties::miptailFirstLevel is valid only if cudaArraySparseProperties::miptailSize is non-zero.

See also:
cuArrayGetSparseProperties, cuMemMapArrayAsync

__host__make_cudaExtent (size_t w, size_t h, size_t d)
Returns a cudaExtent based on input parameters.

Parameters
w
- Width in elements when referring to array memory, in bytes when referring to linear memory
h
- Height in elements
d
- Depth in elements

Returns
cudaExtent specified by w, h, and d

Description
Returns a cudaExtent based on the specified input parameters w, h, and d.
See also:
make_cudaPitchedPtr, make_cudaPos

__host__make_cudaPitchedPtr (void *d, size_t p, size_t xsz, size_t ysz)
Returns a cudaPitchedPtr based on input parameters.

Parameters

*d*  
- Pointer to allocated memory

*p*  
- Pitch of allocated memory in bytes

*xsz*  
- Logical width of allocation in elements

*ysz*  
- Logical height of allocation in elements

Returns
cudaPitchedPtr specified by *d*, *p*, *xsz*, and *ysz*

Description
Returns a cudaPitchedPtr based on the specified input parameters *d*, *p*, *xsz*, and *ysz*.

See also:
make_cudaExtent, make_cudaPos

__host__make_cudaPos (size_t x, size_t y, size_t z)
Returns a cudaPos based on input parameters.

Parameters

*x*  
- X position

*y*  
- Y position

*z*  
- Z position

Returns
cudaPos specified by *x*, *y*, and *z*
Description

Returns a `cudaPos` based on the specified input parameters x, y, and z.

See also:

`make_cudaExtent`, `make_cudaPitchedPtr`

6.10. Memory Management

[DEPRECATED]

This section describes deprecated memory management functions of the CUDA runtime application programming interface.

Some functions have overloaded C++ API template versions documented separately in the C++ API Routines module.

```c
__host__cudaError_t cudaMemcpyArrayToArray (cudaArray_t dst, size_t wOffsetDst, size_t hOffsetDst, cudaArray_const_t src, size_t wOffsetSrc, size_t hOffsetSrc, size_t count, cudaMemcpyKind kind)
```

Copies data between host and device.

Parameters

dst
  - Destination memory address
wOffsetDst
  - Destination starting X offset (columns in bytes)
hOffsetDst
  - Destination starting Y offset (rows)
src
  - Source memory address
wOffsetSrc
  - Source starting X offset (columns in bytes)
hOffsetSrc
  - Source starting Y offset (rows)
count
  - Size in bytes to copy
kind
- Type of transfer

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemcpyDirection

Description
Deprecated
Copies count bytes from the CUDA array src starting at hOffsetSrc rows and wOffsetSrc bytes from the upper left corner to the CUDA array dst starting at hOffsetDst rows and wOffsetDst bytes from the upper left corner, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpyHostToDevice no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToarray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromArraySymbol, cudaMemcpyasync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToarrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromarrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromArrayAsync, cudaMemcpyFromSymbolAsync, cudaMemcpyAtoA
__host__ cudaError_t cudaMemcpyFromArray (void *dst, cudaArray_const_t src, size_t wOffset, size_t hOffset, size_t count, cudaMemcpyKind kind)

Copies data between host and device.

Parameters

dst
- Destination memory address

src
- Source memory address

wOffset
- Source starting X offset (columns in bytes)

hOffset
- Source starting Y offset (rows)

count
- Size in bytes to copy

kind
- Type of transfer

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemcpyDirection

Description

Deprecated

Copies count bytes from the CUDA array src starting at hOffset rows and wOffset bytes from the upper left corner to the memory area pointed to by dst, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.
- Note that this function may also return cudaMemcpyInitializationError, cudaMemcpyInsufficientDriver or cudaMemcpyNoDevice if this call tries to initialize internal CUDA RT state.
See also: cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArrays, cudaMemcpy2DArrays, cudaMemcpy2DFromArrays, cudaMemcpyFromArrays, cudaMemcpy2DAsync, cudaMemcpyFromAsync, cudaMemcpyToSymbolAsync, cudaMemcpy2DArrayToArrays, cudaMemcpy2DFromArrayToArrays, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpy2DArrayToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, cuMemcpyAtoH, cuMemcpyAtoD

__host__ cudaError_t cudaMemcpyFromArrayAsync
(void *dst, cudaArray_const_t src, size_t wOffset,
size_t hOffset, size_t count, cudaMemcpyKind kind, 
cudaStream_t stream)
Copies data between host and device.

Parameters

dst
- Destination memory address

src
- Source memory address

wOffset
- Source starting X offset (columns in bytes)

hOffset
- Source starting Y offset (rows)

count
- Size in bytes to copy

kind
- Type of transfer

stream
- Stream identifier

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemcpyDirection

Description
Deprecated
Copies `count` bytes from the CUDA array `src` starting at `hOffset` rows and `wOffset` bytes from the upper left corner to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing.

`cudaMemcpyFromArrayAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost` and `stream` is non-zero, the copy may overlap with operations in other streams.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`, `cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyToArrayArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromSymbolAsync`, `cuMemcpyAtoHAsync`, `cuMemcpy2DAsync`
__host__ cudaError_t cudaMemcpyToArray (cudaArray_t dst, size_t wOffset, size_t hOffset, const void *src, size_t count, cudaMemcpyKind kind)

Copies data between host and device.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dst</td>
<td>Destination memory address</td>
</tr>
<tr>
<td>wOffset</td>
<td>Destination starting X offset (columns in bytes)</td>
</tr>
<tr>
<td>hOffset</td>
<td>Destination starting Y offset (rows)</td>
</tr>
<tr>
<td>src</td>
<td>Source memory address</td>
</tr>
<tr>
<td>count</td>
<td>Size in bytes to copy</td>
</tr>
<tr>
<td>kind</td>
<td>Type of transfer</td>
</tr>
</tbody>
</table>

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemcpyDirection

Description

Deprecated

Copies count bytes from the memory area pointed to by src to the CUDA array dst starting at hOffset rows and wOffset bytes from the upper left corner, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudAmemcpy, cudAmemcpy2D, cudAmemcpy2DToArray, cudAmemcpyFromArray, cudAmemcpy2DFromArray, cudAmemcpyArrayToArray, cudAmemcpy2DArrayToArray, cudAmemcpyToSymbol, cudAmemcpyFromArraySymbol, cudAmemcpyAsync, cudAmemcpy2DAsync, cudAmemcpyToArrayAsync, cudAmemcpy2DToArrayAsync, cudAmemcpyFromArrayAsync, cudAmemcpy2DFromArrayAsync, cudAmemcpyToSymbolAsync, cudAmemcpyFromArraySymbolAsync, cudAmemcpyHtoA, cudAmemcpyDtoA

__host__cudaError_t cudaMemcpyToArrayAsync (cudArray_t dst, size_t wOffset, size_t hOffset, const void *src, size_t count, cudAmemcpyKind kind, cudStream_t stream)

Copies data between host and device.

Parameters

dst
   - Destination memory address
wOffset
   - Destination starting X offset (columns in bytes)
hOffset
   - Destination starting Y offset (rows)
src
   - Source memory address
count
   - Size in bytes to copy
kind
   - Type of transfer
stream
   - Stream identifier

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemcpyDirection

Description

Deprecated
Copies count bytes from the memory area pointed to by src to the CUDA array dst starting at hOffset rows and wOffset bytes from the upper left corner, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToDevice, cudaMemcpyDeviceToDevice, cudaMemcpyHostToHost, cudaMemcpyDeviceToHost, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing.

cudaMemcpyToArrayAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and stream is non-zero, the copy may overlap with operations in other streams.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
- Note that this function may also return cudaMemcpyInitializationError, cudaMemcpyInsufficientDriver or cudaMemcpyNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpyAddCallback no CUDA function may be called from callback. cudaMemcpyNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudamcpypy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpySymbol, cudaMemcpyFromArraySymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, cudaMemcpyFromSymbolAsync, cudaMemcpyHtoAAsync, cudaMemcpy2DAsync
allocation and the free. If the memory is accessed outside of the promised stream order, a use before allocation / use after free error will cause undefined behavior.

The allocator is free to reallocate the memory as long as it can guarantee that compliant memory accesses will not overlap temporally. The allocator may refer to internal stream ordering as well as inter-stream dependencies (such as CUDA events and null stream dependencies) when establishing the temporal guarantee. The allocator may also insert inter-stream dependencies to establish the temporal guarantee.

**Supported Platforms**

Whether or not a device supports the integrated stream ordered memory allocator may be queried by calling `cudaDeviceGetAttribute()` with the device attribute `cudaDevAttrMemoryPoolsSupported`.

```c
__host__cudaError_t cudaFreeAsync (void *devPtr, cudaStream_t hStream)
```

Frees memory with stream ordered semantics.

**Parameters**

- `devPtr`
- `hStream`

  - The stream establishing the stream ordering promise

**Returns**

- `cudaSuccess`
- `cudaErrorInvalidValue`
- `cudaErrorNotSupported`

**Description**

Inserts a free operation into `hStream`. The allocation must not be accessed after stream execution reaches the free. After this API returns, accessing the memory from any subsequent work launched on the GPU or querying its pointer attributes results in undefined behavior.

**Note:**

During stream capture, this function results in the creation of a free node and must therefore be passed the address of a graph allocation.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- This function uses standard `default stream` semantics.
Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.

Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
`cuMemFreeAsync`, `cudaMallocAsync`

```c
__host__ cudaError_t cudaMallocAsync (void **devPtr, size_t size, cudaStream_t hStream)
```

Allocates memory with stream ordered semantics.

**Parameters**

- **devPtr**
  - Returned device pointer
- **size**
  - Number of bytes to allocate
- **hStream**
  - The stream establishing the stream ordering contract and the memory pool to allocate from

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorNotSupported`, `cudaErrorOutOfMemory`

**Description**

Inserts an allocation operation into `hStream`. A pointer to the allocated memory is returned immediately in `*dptr`. The allocation must not be accessed until the the allocation operation completes. The allocation comes from the memory pool associated with the stream’s device.

**Note:**

- The default memory pool of a device contains device memory from that device.
- Basic stream ordering allows future work submitted into the same stream to use the allocation. Stream query, stream synchronize, and CUDA events can be used to guarantee that the allocation operation completes before work submitted in a separate stream runs.
During stream capture, this function results in the creation of an allocation node. In this case, the allocation is owned by the graph instead of the memory pool. The memory pool’s properties are used to set the node’s creation parameters.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- This function uses standard default stream semantics.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

```c
__host__ cudaError_t cudaMallocFromPoolAsync(void **ptr, size_t size, cudaMemPool_t memPool, cudaStream_t stream)
```

Allocates memory from a specified pool with stream ordered semantics.

Parameters
- `ptr` - Returned device pointer
- `size` - The pool to allocate from
- `memPool` - The stream establishing the stream ordering semantic

Returns
- `cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorNotSupported`, `cudaErrorOutOfMemory`
Description

Inserts an allocation operation into hStream. A pointer to the allocated memory is returned immediately in *dptr. The allocation must not be accessed until the the allocation operation completes. The allocation comes from the specified memory pool.

Note:
- The specified memory pool may be from a device different than that of the specified hStream.
- Basic stream ordering allows future work submitted into the same stream to use the allocation. Stream query, stream synchronize, and CUDA events can be used to guarantee that the allocation operation completes before work submitted in a separate stream runs.

Note:
During stream capture, this function results in the creation of an allocation node. In this case, the allocation is owned by the graph instead of the memory pool. The memory pool’s properties are used to set the node’s creation parameters.

See also:
cuMemAllocFromPoolAsync, cudaMallocAsync [ C++ API], cudaFreeAsync, cudaDeviceGetDefaultMemPool, cudaMemPoolCreate, cudaMemPoolSetAccess, cudaMemPoolSetAttribute

__host__ cudaError_t cudaMemPoolCreate (cudaMemPool_t *memPool, const cudaMemPoolProps *poolProps)

Creates a memory pool.

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorNotSupported

Description

Creates a CUDA memory pool and returns the handle in pool. The poolProps determines the properties of the pool such as the backing device and IPC capabilities.
By default, the pool’s memory will be accessible from the device it is allocated on.

**Note:**
Specifying `cudaMemHandleTypeNone` creates a memory pool that will not support IPC.

**See also:**
- `cuMemPoolCreate`, `cudaDeviceSetMemPool`, `cudaMallocFromPoolAsync`,
- `cudaMemPoolExportToShareableHandle`, `cudaDeviceGetDefaultMemPool`,
- `cudaDeviceGetMemPool`

```__host__cudaError_t cudaMemPoolDestroy (cudaMemPool_t memPool)`
```
Destroys the specified memory pool.

**Returns**
- `cudaSuccess`, `cudaErrorInvalidValue`

**Description**
If any pointers obtained from this pool haven’t been freed or the pool has free operations that haven’t completed when `cudaMemPoolDestroy` is invoked, the function will return immediately and the resources associated with the pool will be released automatically once there are no more outstanding allocations.

Destroying the current mempool of a device sets the default mempool of that device as the current mempool for that device.

**Note:**
A device’s default memory pool cannot be destroyed.

**See also:**
- `cuMemPoolDestroy`, `cudaFreeAsync`, `cudaDeviceSetMemPool`,
- `cudaDeviceGetDefaultMemPool`, `cudaDeviceGetMemPool`, `cudaMemPoolCreate`
__host__ cudaError_t cudaMemPoolExportPointer (cudaMemPoolPtrExportData *exportData, void *ptr)

Export data to share a memory pool allocation between processes.

Parameters
exportData
ptr - pointer to memory being exported

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorOutOfMemory

Description
Constructs shareData_out for sharing a specific allocation from an already shared memory pool. The recipient process can import the allocation with the cudaMemPoolImportPointer api. The data is not a handle and may be shared through any IPC mechanism.

See also:
cuMemPoolExportPointer, cudaMemPoolExportToShareableHandle, cudaMemPoolImportFromShareableHandle, cudaMemPoolImportPointer

__host__ cudaError_t cudaMemPoolExportToShareableHandle (void *shareableHandle, cudaMemPool_t memPool, cudaMemAllocationHandleType handleType, unsigned int flags)

Exports a memory pool to the requested handle type.

Parameters
shareableHandle
memPool
handleType - the type of handle to create
flags - must be 0
Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorOutOfMemory

Description

Given an IPC capable mempool, create an OS handle to share the pool with another process. A recipient process can convert the shareable handle into a mempool with cudaMemPoolImportFromShareableHandle. Individual pointers can then be shared with the cudaMemPoolExportPointer and cudaMemPoolImportPointer APIs. The implementation of what the shareable handle is and how it can be transferred is defined by the requested handle type.

Note:

: To create an IPC capable mempool, create a mempool with a CUmemAllocationHandleType other than cudaMemHandleTypeNone.

See also:

cuMemPoolExportToShareableHandle, cudaMemPoolImportFromShareableHandle, cudaMemPoolExportPointer, cudaMemPoolImportPointer

__host__cudaError_t cudaMemPoolGetAccess (cudaMemAccessFlags *flags, cudaMemPool_t memPool, cudaMemLocation_t location)

Returns the accessibility of a pool from a device.

Parameters

flags
  - the accessibility of the pool from the specified location
memPool
  - the pool being queried
location
  - the location accessing the pool

Description

Returns the accessibility of the pool’s memory from the specified location.

See also:

cuMemPoolGetAccess, cudaMemPoolSetAccess
__host__ cudaError_t cudaMemPoolGetAttribute(cudaMemPool_t memPool, cudaMemPoolAttr attr, void *value)

Gets attributes of a memory pool.

Parameters

cudaMemPool:
- The memory pool to get attributes from

cudaMemPoolAttr:
- The attribute to get

cudaMemPoolAttrGet:
- Retrieved value

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Supported attributes are:

- **cudaMemPoolAttrReleaseThreshold**: (value type = cuuint64_t) Amount of reserved memory in bytes to 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 context synchronize. (default 0)

- **cudaMemPoolReuseFollowEventDependencies**: (value type = int) Allow cudaMallocAsync to use memory asynchronously freed in another stream as long as a stream ordering dependency of the allocating stream on the free action exists. Cuda events and null stream interactions can create the required stream ordered dependencies. (default enabled)

- **cudaMemPoolReuseAllowOpportunistic**: (value type = int) Allow reuse of already completed frees when there is no dependency between the free and allocation. (default enabled)

- **cudaMemPoolReuseAllowInternalDependencies**: (value type = int) Allow cudaMallocAsync to insert new stream dependencies in order to establish the stream ordering required to reuse a piece of memory released by cudaFreeAsync (default enabled).

Note:
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

```c
__host__ cudaError_t
cudaMemPoolImportFromShareableHandle(
    cudaMemPool_t *memPool, void *shareableHandle, 
    cudaMemAllocationHandleType handleType, unsigned int flags)
```

imports a memory pool from a shared handle.

**Parameters**
- `memPool`:
- `shareableHandle`:
- `handleType`:
  - The type of handle being imported
- `flags`:
  - must be 0

**Returns**
- `cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorOutOfMemory`

**Description**
Specific allocations can be imported from the imported pool with `cudaMemPoolImportPointer`.

**Note:**
Imported memory pools do not support creating new allocations. As such imported memory pools may not be used in `cudaDeviceSetMemPool` or `cudaMallocFromPoolAsync` calls.

See also:
`cuMemPoolImportFromShareableHandle`, `cudaMemPoolExportToShareableHandle`, `cudaMemPoolExportPointer`, `cudaMemPoolImportPointer`
__host__ cudaError_t cudaMemPoolImportPointer(void **ptr, cudaMemPool_t memPool, cudaMemPoolPtrExportData *exportData)
Import a memory pool allocation from another process.

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_OUT_OF_MEMORY

Description
Returns in ptr_out a pointer to the imported memory. The imported memory must not be accessed before the allocation operation completes in the exporting process. The imported memory must be freed from all importing processes before being freed in the exporting process. The pointer may be freed with cudaFree or cudaFreeAsync. If cudaFreeAsync is used, the free must be completed on the importing process before the free operation on the exporting process.

Note:
The cudaFreeAsync api may be used in the exporting process before the cudaFreeAsync operation completes in its stream as long as the cudaFreeAsync in the exporting process specifies a stream with a stream dependency on the importing process's cudaFreeAsync.

See also:
cuMemPoolImportPointer, cudaMemPoolExportToShareableHandle, cudaMemPoolImportFromShareableHandle, cudaMemPoolExportPointer

__host__ cudaError_t cudaMemPoolSetAccess(cudaMemPool_t memPool, const cudaMemAccessDesc *descList, size_t count)
Controls visibility of pools between devices.

Parameters
memPool
descList
count
- Number of descriptors in the map array.
Returns
cudaSuccess, cudaErrorInvalidValue

Description

See also:
cuMemPoolSetAccess, cudaMemPoolGetAccess, cudaMallocAsync, cudaFreeAsync

__host__cudaError_t cudaMemPoolSetAttribute
(cuMemPool_t memPool, cudaMemPoolAttr attr, void *value)
Sets attributes of a memory pool.

Parameters

memPool
attr
- The attribute to modify
value
- Pointer to the value to assign

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Supported attributes are:

- **cudaMemPoolAttrReleaseThreshold**: [value type = cuuint64_t] Amount of reserved memory in bytes to 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 context synchronize. (default 0)

- **cudaMemPoolReuseFollowEventDependencies**: [value type = int] Allow cudaMallocAsync to use memory asynchronously freed in another stream as long as a stream ordering dependency of the allocating stream on the free action exists. Cuda events and null stream interactions can create the required stream ordered dependencies. (default enabled)

- **cudaMemPoolReuseAllowOpportunistic**: [value type = int] Allow reuse of already completed frees when there is no dependency between the free and allocation. (default enabled)
cudaMemPoolReuseAllowInternalDependencies: [value type = int] Allow cudaMallocAsync to insert new stream dependencies in order to establish the stream ordering required to reuse a piece of memory released by cudaFreeAsync [default enabled].

Note:
Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cuMemPoolSetAttribute, cudaMallocAsync, cudaFreeAsync, cudaDeviceGetDefaultMemPool, cudaDeviceGetMemPool, cudaMemPoolCreate

__host__cudaError_t cudaMemPoolTrimTo(cudaMemPool_t memPool, size_t minBytesToKeep)
Tries to release memory back to the OS.

Parameters
memPool
minBytesToKeep
- If the pool has less than minBytesToKeep reserved, the TrimTo operation is a no-op.
Otherwise the pool will be guaranteed to have at least minBytesToKeep bytes reserved after the operation.

Returns
cudaSuccess, cudaErrorInvalidValue

Description
Releases memory back to the OS until the pool contains fewer than minBytesToKeep reserved bytes, or there is no more memory that the allocator can safely release. The allocator cannot release OS allocations that back outstanding asynchronous allocations. The OS allocations may happen at different granularity from the user allocations.

Note:
- Allocations that have not been freed count as outstanding.
- Allocations that have been asynchronously freed but whose completion has not been observed on the host [eg. by a synchronize] can count as outstanding.
6.12. Unified Addressing

This section describes the unified addressing functions of the CUDA runtime application programming interface.

Overview

CUDA devices can share a unified address space with the host. For these devices there is no distinction between a device pointer and a host pointer -- the same pointer value may be used to access memory from the host program and from a kernel running on the device (with exceptions enumerated below).

Supported Platforms

Whether or not a device supports unified addressing may be queried by calling cudaGetDeviceProperties() with the device property cudaDeviceProp::unifiedAddressing.

Unified addressing is automatically enabled in 64-bit processes.

Looking Up Information from Pointer Values

It is possible to look up information about the memory which backs a pointer value. For instance, one may want to know if a pointer points to host or device memory. As another example, in the case of device memory, one may want to know on which CUDA device the memory resides. These properties may be queried using the function cudaPointerGetAttributes().

Since pointers are unique, it is not necessary to specify information about the pointers specified to cudaMemcpy() and other copy functions. The copy direction cudaMemcpyDefault may be used to specify that the CUDA runtime should infer the location of the pointer from its value.

Automatic Mapping of Host Allocated Host Memory

All host memory allocated through all devices using cudaMemcpyHost() and cudaMemcpyHost() is always directly accessible from all devices that support unified addressing. This is the case
regardless of whether or not the flags `cudaHostAllocPortable` and `cudaHostAllocMapped` are specified.

The pointer value through which allocated host memory may be accessed in kernels on all devices that support unified addressing is the same as the pointer value through which that memory is accessed on the host. It is not necessary to call `cudaHostGetDevicePointer()` to get the device pointer for these allocations.

Note that this is not the case for memory allocated using the flag `cudaHostAllocWriteCombined`, as discussed below.

**Direct Access of Peer Memory**

Upon enabling direct access from a device that supports unified addressing to another peer device that supports unified addressing using `cudaDeviceEnablePeerAccess()` all memory allocated in the peer device using `cudaMalloc()` and `cudaMallocPitch()` will immediately be accessible by the current device. The device pointer value through which any peer’s memory may be accessed in the current device is the same pointer value through which that memory may be accessed from the peer device.

**Exceptions, Disjoint Addressing**

Not all memory may be accessed on devices through the same pointer value through which they are accessed on the host. These exceptions are host memory registered using `cudaHostRegister()` and host memory allocated using the flag `cudaHostAllocWriteCombined`. For these exceptions, there exists a distinct host and device address for the memory. The device address is guaranteed to not overlap any valid host pointer range and is guaranteed to have the same value across all devices that support unified addressing.

This device address may be queried using `cudaHostGetDevicePointer()` when a device using unified addressing is current. Either the host or the unified device pointer value may be used to refer to this memory in `cudaMemcpy()` and similar functions using the `cudaMemcpyDefault` memory direction.

```c
__host__ cudaError_t cudaMemcpy
    (void *hostMem, const void *devMem, size_t size, cudaMemcpyFlags flags);```

Returns attributes about a specified pointer.

**Parameters**

- `attributes` - Attributes for the specified pointer
- `ptr` - Pointer to get attributes for

**Returns**

`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorInvalidValue`
**Description**

Returns in `*attributes` the attributes of the pointer `ptr`. If pointer was not allocated in, mapped by or registered with context supporting unified addressing `cudaErrorInvalidValue` is returned.

**Note:**

In CUDA 11.0 forward passing host pointer will return `cudaMemoryTypeUnregistered` in `cudaPointerAttributes::type` and call will return `cudaSuccess`.

The `cudaPointerAttributes` structure is defined as:

```c
struct cudaPointerAttributes {
    enum cudaMemoryType type;
    int device;
    void *devicePointer;
    void *hostPointer;
};
```

In this structure, the individual fields mean

- `cudaPointerAttributes::type` identifies type of memory. It can be `cudaMemoryTypeUnregistered` for unregistered host memory, `cudaMemoryTypeHost` for registered host memory, `cudaMemoryTypeDevice` for device memory or `cudaMemoryTypeManaged` for managed memory.

- `device` is the device against which `ptr` was allocated. If `ptr` has memory type `cudaMemoryTypeDevice` then this identifies the device on which the memory referred to by `ptr` physically resides. If `ptr` has memory type `cudaMemoryTypeHost` then this identifies the device which was current when the allocation was made (and if that device is deinitialized then this allocation will vanish with that device’s state).

- `devicePointer` is the device pointer alias through which the memory referred to by `ptr` may be accessed on the current device. If the memory referred to by `ptr` cannot be accessed directly by the current device then this is NULL.

- `hostPointer` is the host pointer alias through which the memory referred to by `ptr` may be accessed on the host. If the memory referred to by `ptr` cannot be accessed directly by the host then this is NULL.

**Note:**

- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
`cudaGetDeviceCount`, `cudaGetDevice`, `cudaSetDevice`, `cudaChooseDevice`, `cuPointerGetAttributes`

6.13. Peer Device Memory Access

This section describes the peer device memory access functions of the CUDA runtime application programming interface.

```c
__host__ cudaError_t cudaDeviceCanAccessPeer (int *canAccessPeer, int device, int peerDevice)
```

Queries if a device may directly access a peer device’s memory.

Parameters

- `canAccessPeer`: Returned access capability
- `device`: Device from which allocations on `peerDevice` are to be directly accessed.
- `peerDevice`: Device on which the allocations to be directly accessed by `device` reside.

Returns

- `cudaSuccess`, `cudaErrorInvalidDevice`

Description

Returns in `*canAccessPeer` a value of 1 if `device` `device` is capable of directly accessing memory from `peerDevice` and 0 otherwise. If direct access of `peerDevice` from `device` is possible, then access may be enabled by calling `cudaDeviceEnablePeerAccess()`.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

- `cudaDeviceEnablePeerAccess`
- `cudaDeviceDisablePeerAccess`
- `cuDeviceCanAccessPeer`

```c
__host__ cudaError_t cudaDeviceDisablePeerAccess(int peerDevice)
```

Disables direct access to memory allocations on a peer device.

**Parameters**

**peerDevice**
- Peer device to disable direct access to

**Returns**

- `cudaSuccess`
- `cudaErrorPeerAccessNotEnabled`
- `cudaErrorInvalidDevice`

**Description**

Returns `cudaErrorPeerAccessNotEnabled` if direct access to memory on `peerDevice` has not yet been enabled from the current device.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

- `cudaDeviceCanAccessPeer`
- `cudaDeviceEnablePeerAccess`
- `cuCtxDisablePeerAccess`
__host__ cudaError_t cudaDeviceEnablePeerAccess
(int peerDevice, unsigned int flags)

Enables direct access to memory allocations on a peer device.

Parameters

peerDevice
- Peer device to enable direct access to from the current device

flags
- Reserved for future use and must be set to 0

Returns
cudaSuccess, cudaErrorInvalidDevice, cudaErrorPeerAccessAlreadyEnabled,
cudaErrorInvalidValue

Description

On success, all allocations from peerDevice will immediately be accessible by the current device. They will remain accessible until access is explicitly disabled using cudaDeviceDisablePeerAccess() or either device is reset using cudaDeviceReset().

Note that access granted by this call is unidirectional and that in order to access memory on the current device from peerDevice, a separate symmetric call to cudaDeviceEnablePeerAccess() is required.

Note that there are both device-wide and system-wide limitations per system configuration, as noted in the CUDA Programming Guide under the section “Peer-to-Peer Memory Access”.

Returns cudaErrorInvalidDevice if cudaDeviceCanAccessPeer() indicates that the current device cannot directly access memory from peerDevice.

Returns cudaErrorPeerAccessAlreadyEnabled if direct access of peerDevice from the current device has already been enabled.

Returns cudaErrorInvalidValue if flags is not 0.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.

This section describes the OpenGL interoperability functions of the CUDA runtime application programming interface. Note that mapping of OpenGL resources is performed with the graphics API agnostic, resource mapping interface described in Graphics Interopability.

```c
enum cudaGLDeviceList
```

CUDA devices corresponding to the current OpenGL context

**Values**

- `cudaGLDeviceListAll = 1`
  - The CUDA devices for all GPUs used by the current OpenGL context
- `cudaGLDeviceListCurrentFrame = 2`
  - The CUDA devices for the GPUs used by the current OpenGL context in its currently rendering frame
- `cudaGLDeviceListNextFrame = 3`
  - The CUDA devices for the GPUs to be used by the current OpenGL context in the next frame

```c
__host__cudaError_t cudaGLGetDevices (unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, cudaGLDeviceList deviceList)
```

Gets the CUDA devices associated with the current OpenGL context.

**Parameters**

- `pCudaDeviceCount`
  - Returned number of CUDA devices corresponding to the current OpenGL context
- `pCudaDevices`
  - Returned CUDA devices corresponding to the current OpenGL context
- `cudaDeviceCount`
  - The size of the output device array `pCudaDevices`
**deviceList**
- The set of devices to return. This set may be cudaGLDeviceListAll for all devices, cudaGLDeviceListCurrentFrame for the devices used to render the current frame (in SLI), or cudaGLDeviceListNextFrame for the devices used to render the next frame (in SLI).

**Returns**
cudaSuccess, cudaErrorNoDevice, cudaErrorInvalidGraphicsContext, cudaErrorUnknown

**Description**
Returns in *pCudaDeviceCount the number of CUDA-compatible devices corresponding to the current OpenGL context. Also returns in *pCudaDevices at most cudaDeviceCount of the CUDA-compatible devices corresponding to the current OpenGL context. If any of the GPUs being used by the current OpenGL context are not CUDA capable then the call will return cudaErrorNoDevice.

**Note:**
- This function is not supported on Mac OS X.
- Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

```c
__host__ cudaError_t cudaGraphicsGLRegisterBuffer (cudaGraphicsResource **resource, GLuint buffer, unsigned int flags)
```

Registers an OpenGL buffer object.

**Parameters**
- `resource`
  - Pointer to the returned object handle
- `buffer`
  - name of buffer object to be registered
- `flags`
  - Register flags
Returns

cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description

Registers the buffer object specified by buffer for access by CUDA. A handle to the registered object is returned as resource. The register flags flags specify the intended usage, as follows:

- **cudaGraphicsRegisterFlagsNone**: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- **cudaGraphicsRegisterFlagsReadOnly**: Specifies that CUDA will not write to this resource.
- **cudaGraphicsRegisterFlagsWriteDiscard**: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

**Note:** Note that this function may also return error codes from previous, asynchronous launches.

See also:


__host__cudaError_t cudaGraphicsGLRegisterImage (cudaGraphicsResource **resource, GLuint image, GLenum target, unsigned int flags)

Register an OpenGL texture or renderbuffer object.

Parameters

**resource**
- Pointer to the returned object handle

**image**
- name of texture or renderbuffer object to be registered

**target**
- Identifies the type of object specified by image
flags
- Register flags

Returns
cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description
Registers the texture or renderbuffer object specified by image for access by CUDA. A handle to the registered object is returned as resource.

target must match the type of the object, and must be one of GL_TEXTURE_2D, GL_TEXTURE_RECTANGLE, GL_TEXTURE_CUBE_MAP, GL_TEXTURE_3D, GL_TEXTURE_2D_ARRAY, or GL_RENDERBUFFER.

The register flags flags specify the intended usage, as follows:

- cudaGraphicsRegisterFlagsNone: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- cudaGraphicsRegisterFlagsReadOnly: Specifies that CUDA will not write to this resource.
- cudaGraphicsRegisterFlagsWriteDiscard: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.
- cudaGraphicsRegisterFlagsSurfaceLoadStore: Specifies that CUDA will bind this resource to a surface reference.
- cudaGraphicsRegisterFlagsTextureGather: Specifies that CUDA will perform texture gather operations on this resource.

The following image formats are supported. For brevity’s sake, the list is abbreviated. For ex., \{GL_R, GL_RG\} X \{8, 16\} would expand to the following 4 formats \{GL_R8, GL_R16, GL_RG8, GL_RG16\}:

- GL_RED, GL_RG, GL_RGBA, GL_LUMINANCE, GL_ALPHA, GL_LUMINANCE_ALPHA, GL_INTENSITY
- \{GL_R, GL_RG, GL_RGBA\} X \{8, 16, 16F, 32F, 8UI, 16UI, 32UI, 8I, 16I, 32I\}
- \{GL_LUMINANCE, GL_ALPHA, GL_LUMINANCE_ALPHA, GL_INTENSITY\} X \{8, 16, 16F_ARB, 32F_ARB, 8UI_EXT, 16UI_EXT, 32UI_EXT, 8I_EXT, 16I_EXT, 32I_EXT\}

The following image classes are currently disallowed:
- Textures with borders
- Multisampled renderbuffers
Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:

__host__cudaError_t cudaWGLGetDevice (int *device, HGPUNV hGpu)
Gets the CUDA device associated with hGpu.

Parameters

device
- Returns the device associated with hGpu, or -1 if hGpu is not a compute device.

hGpu
- Handle to a GPU, as queried via WGL_NV_gpu_affinity

Returns
cudaSuccess

Description
Returns the CUDA device associated with a hGpu, if applicable.

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
WGL_NV_gpu_affinity, cuWGLGetDevice

6.15. OpenGL Interoperability
[DEPRECATED]

This section describes deprecated OpenGL interoperability functionality.
enum cudaGLMapFlags

CUDA GL Map Flags

Values

cudaGLMapFlagsNone = 0
  Default; Assume resource can be read/written

cudaGLMapFlagsReadOnly = 1
  CUDA kernels will not write to this resource

cudaGLMapFlagsWriteDiscard = 2
  CUDA kernels will only write to and will not read from this resource

__host__ cudaError_t cudaGLMapBufferObject (void **devPtr, GLuint bufObj)

Maps a buffer object for access by CUDA.

Parameters

devPtr
  - Returned device pointer to CUDA object

bufObj
  - Buffer object ID to map

Returns

cudaSuccess, cudaErrorMapBufferObjectFailed

Description

Deprecated This function is deprecated as of CUDA 3.0.

Maps the buffer object of ID bufObj into the address space of CUDA and returns in *devPtr the base pointer of the resulting mapping. The buffer must have previously been registered by calling cudaGLRegisterBufferObject[]. While a buffer is mapped by CUDA, any OpenGL operation which references the buffer will result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

All streams in the current thread are synchronized with the current GL context.

Note:

Note that this function may also return error codes from previous, asynchronous launches.
___host__cudaError_t cudaGLMapBufferObjectAsync (void **devPtr, GLuint bufObj, cudaStream_t stream)
Maps a buffer object for access by CUDA.

Parameters

- **devPtr**: Returned device pointer to CUDA object
- **bufObj**: Buffer object ID to map
- **stream**: Stream to synchronize

Returns

cudaSuccess, cudaErrorMapBufferObjectFailed

Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Maps the buffer object of ID bufObj into the address space of CUDA and returns in *devPtr* the base pointer of the resulting mapping. The buffer must have previously been registered by calling cudaGLRegisterBufferObject[]. While a buffer is mapped by CUDA, any OpenGL operation which references the buffer will result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

Stream /p stream is synchronized with the current GL context.

- **Note:**
  Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaGraphicsMapResources
__host__cudaError_t cudaGLRegisterBufferObject (GLuint bufObj)
Registers a buffer object for access by CUDA.

Parameters
bufObj
- Buffer object ID to register

Returns
cudaSuccess, cudaErrorInitializationError

Description
Deprecated
This function is deprecated as of CUDA 3.0.
Registers the buffer object of ID bufObj for access by CUDA. This function must be called before CUDA can map the buffer object. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaGraphicsGLRegisterBuffer

__host__cudaError_t
cudaGLSetBufferObjectMapFlags (GLuint bufObj, unsigned int flags)
Set usage flags for mapping an OpenGL buffer.

Parameters
bufObj
- Registered buffer object to set flags for
flags
- Parameters for buffer mapping
Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description
Deprecated
This function is deprecated as of CUDA 3.0.

Set flags for mapping the OpenGL buffer bufObj

Changes to flags will take effect the next time bufObj is mapped. The flags argument may be any of the following:

- cudaGLMapFlagsNone: Specifies no hints about how this buffer will be used. It is therefore assumed that this buffer will be read from and written to by CUDA kernels. This is the default value.
- cudaGLMapFlagsReadOnly: Specifies that CUDA kernels which access this buffer will not write to the buffer.
- cudaGLMapFlagsWriteDiscard: Specifies that CUDA kernels which access this buffer will not read from the buffer and will write over the entire contents of the buffer, so none of the data previously stored in the buffer will be preserved.

If bufObj has not been registered for use with CUDA, then cudaErrorInvalidResourceHandle is returned. If bufObj is presently mapped for access by CUDA, then cudaErrorUnknown is returned.

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaGraphicsResourceSetMapFlags

__host__cudaError_t cudaGLSetGLDevice (int device)
Sets a CUDA device to use OpenGL interoperability.

Parameters
device
- Device to use for OpenGL interoperability

Returns
cudaSuccess, cudaErrorInvalidDevice, cudaErrorSetOnActiveProcess
Description

Deprecated This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with an OpenGL context in order to achieve maximum interoperability performance.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaGraphicsGLRegisterBuffer, cudaGraphicsGLRegisterImage

__host__cudaError_t cudaGLUnmapBufferObject(GLuint bufObj)

Unmaps a buffer object for access by CUDA.

Parameters

bufObj
  - Buffer object to unmap

Returns
cudaSuccess, cudaErrorUnmapBufferObjectFailed

Description

Deprecated This function is deprecated as of CUDA 3.0.

Unmaps the buffer object of ID bufObj for access by CUDA. When a buffer is unmapped, the base address returned by cudaGLMapBufferObject() is invalid and subsequent references to the address result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

All streams in the current thread are synchronized with the current GL context.

Note:

Note that this function may also return error codes from previous, asynchronous launches.
See also:
 cudaGraphicsUnmapResources

__host__cudaError_t
cudaGLUnmapBufferObjectAsync (GLuint bufObj, cudaStream_t stream)
Unmaps a buffer object for access by CUDA.

Parameters
bufObj
- Buffer object to unmap
stream
- Stream to synchronize

Returns
cudaSuccess, cudaErrorUnmapBufferObjectFailed

Description
Deprecated This function is deprecated as of CUDA 3.0.
Unmaps the buffer object of ID bufObj for access by CUDA. When a buffer is unmapped, the base address returned by cudaGLMapBufferObject() is invalid and subsequent references to the address result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.
Stream /p stream is synchronized with the current GL context.

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
 cudaGraphicsUnmapResources
__host__ cudaError_t cudaGLUnregisterBufferObject(GLuint bufObj)
Unregisters a buffer object for access by CUDA.

Parameters
bufObj
- Buffer object to unregister

Returns
cudaSuccess

Description
Deprecated This function is deprecated as of CUDA 3.0.
Unregisters the buffer object of ID bufObj for access by CUDA and releases any CUDA resources associated with the buffer. Once a buffer is unregistered, it may no longer be mapped by CUDA. The GL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaGraphicsUnregisterResource

6.16. Direct3D 9 Interoperability
This section describes the Direct3D 9 interoperability functions of the CUDA runtime application programming interface. Note that mapping of Direct3D 9 resources is performed with the graphics API agnostic, resource mapping interface described in Graphics Interopability.

enum cudaD3D9DeviceList
CUDA devices corresponding to a D3D9 device

Values
cudaD3D9DeviceListAll = 1
The CUDA devices for all GPUs used by a D3D9 device

\texttt{cudaD3D9DeviceListCurrentFrame} = 2

The CUDA devices for the GPUs used by a D3D9 device in its currently rendering frame

\texttt{cudaD3D9DeviceListNextFrame} = 3

The CUDA devices for the GPUs to be used by a D3D9 device in the next frame

\begin{verbatim}
__host__cudaError_t cudaD3D9GetDevice (int *device, const char *pszAdapterName)

Gets the device number for an adapter.

Parameters

\begin{itemize}
  \item \texttt{device} - Returns the device corresponding to \texttt{pszAdapterName}
  \item \texttt{pszAdapterName} - D3D9 adapter to get device for
\end{itemize}

Returns

\texttt{cudaSuccess}, \texttt{cudaErrorInvalidValue}, \texttt{cudaErrorUnknown}

Description

Returns in \texttt{*device} the CUDA-compatible device corresponding to the adapter name \texttt{pszAdapterName} obtained from \texttt{EnumDisplayDevices} or \texttt{IDirect3D9::GetAdapterIdentifier[]}. If no device on the adapter with name \texttt{pszAdapterName} is CUDA-compatible then the call will fail.

\begin{itemize}
  \item \textbf{Note:}
  \begin{quote}
    Note that this function may also return error codes from previous, asynchronous launches.
  \end{quote}
\end{itemize}

See also:

\texttt{cudaD3D9SetDirect3DDevice}, \texttt{cudaGraphicsD3D9RegisterResource}, \texttt{cuD3D9GetDevice}
__host__ cudaError_t cudaD3D9GetDevices (unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, IDirect3DDevice9 *pD3D9Device, cudaD3D9DeviceList deviceList)

Gets the CUDA devices corresponding to a Direct3D 9 device.

Parameters

pCudaDeviceCount
- Returned number of CUDA devices corresponding to pD3D9Device

pCudaDevices
- Returned CUDA devices corresponding to pD3D9Device

cudaDeviceCount
- The size of the output device array pCudaDevices

pD3D9Device
- Direct3D 9 device to query for CUDA devices

deviceList
- The set of devices to return. This set may be cudaD3D9DeviceListAll for all devices, cudaD3D9DeviceListCurrentFrame for the devices used to render the current frame (in SLI), or cudaD3D9DeviceListNextFrame for the devices used to render the next frame (in SLI).

Returns

cudaSuccess, cudaErrorNoDevice, cudaErrorUnknown

Description

Returns in *pCudaDeviceCount the number of CUDA-compatible devices corresponding to the Direct3D 9 device pD3D9Device. Also returns in *pCudaDevices at most cudaDeviceCount of the the CUDA-compatible devices corresponding to the Direct3D 9 device pD3D9Device.

If any of the GPUs being used to render pDevice are not CUDA capable then the call will return cudaErrorNoDevice.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaGraphicsUnregisterResource, cudaGraphicsMapResources,
cudaGraphicsSubResourceGetMappedArray, cudaGraphicsResourceGetMappedPointer,
cuD3D9GetDevices

__host__cudaError_t cudaD3D9GetDirect3DDevice
(IDirect3DDevice9 **ppD3D9Device)

Gets the Direct3D device against which the current CUDA context was created.

Parameters

ppD3D9Device
- Returns the Direct3D device for this thread

Returns
cudaSuccess, cudaErrorInvalidGraphicsContext, cudaErrorUnknown

Description

Returns in *ppD3D9Device the Direct3D device against which this CUDA context was created in cudaD3D9SetDirect3DDevice.

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaD3D9SetDirect3DDevice, cuD3D9GetDirect3DDevice

__host__cudaError_t cudaD3D9SetDirect3DDevice
(IDirect3DDevice9 *pD3D9Device, int device)

Sets the Direct3D 9 device to use for interoperability with a CUDA device.

Parameters

pD3D9Device
- Direct3D device to use for this thread
device
- The CUDA device to use. This device must be among the devices returned when querying cudaD3D9DeviceListAll from cudaD3D9GetDevices, may be set to -1 to automatically select an appropriate CUDA device.
Returns

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorSetOnActiveProcess

Description

Records pD3D9Device as the Direct3D 9 device to use for Direct3D 9 interoperability with the CUDA device device and sets device as the current device for the calling host thread.

If device has already been initialized then this call will fail with the error cudaErrorSetOnActiveProcess. In this case it is necessary to reset device using cudaDeviceReset() before Direct3D 9 interoperability on device may be enabled.

Successfully initializing CUDA interoperability with pD3D9Device will increase the internal reference count on pD3D9Device. This reference count will be decremented when device is reset using cudaDeviceReset().

Note that this function is never required for correct functionality. Use of this function will result in accelerated interoperability only when the operating system is Windows Vista or Windows 7, and the device pD3DDDevice is not an IDirect3DDevice9Ex. In all other circumstances, this function is not necessary.

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaD3D9GetDevice, cudaGraphicsD3D9RegisterResource, cudaDeviceReset

__host__cudaError_t
cudaGraphicsD3D9RegisterResource
(cudaGraphicsResource **resource, IDirect3DResource9 *pD3DResource, unsigned int flags)

Register a Direct3D 9 resource for access by CUDA.

Parameters

resource
- Pointer to returned resource handle
**pD3DResource**
- Direct3D resource to register

**flags**
- Parameters for resource registration

**Returns**
cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

**Description**
Registers the Direct3D 9 resource pD3DResource for access by CUDA.

If this call is successful then the application will be able to map and unmap this resource until it is unregistered through cudaGraphicsUnregisterResource(). Also on success, this call will increase the internal reference count on pD3DResource. This reference count will be decremented when this resource is unregistered through cudaGraphicsUnregisterResource().

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of pD3DResource must be one of the following.

- IDirect3DVertexBuffer9: may be accessed through a device pointer
- IDirect3DIndexBuffer9: may be accessed through a device pointer
- IDirect3DSurface9: may be accessed through an array. Only stand-alone objects of type IDirect3DSurface9 may be explicitly shared. In particular, individual mipmap levels and faces of cube maps may not be registered directly. To access individual surfaces associated with a texture, one must register the base texture object.
- IDirect3DBaseTexture9: individual surfaces on this texture may be accessed through an array.

The flags argument may be used to specify additional parameters at register time. The valid values for this parameter are

- cudaGraphicsRegisterFlagsNone: Specifies no hints about how this resource will be used.
- cudaGraphicsRegisterFlagsSurfaceLoadStore: Specifies that CUDA will bind this resource to a surface reference.
- cudaGraphicsRegisterFlagsTextureGather: Specifies that CUDA will perform texture gather operations on this resource.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
• Resources allocated as shared may not be registered with CUDA.
• Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
• Surfaces of depth or stencil formats cannot be shared.

A complete list of supported formats is as follows:

• D3DFMT_L8
• D3DFMT_L16
• D3DFMT_A8R8G8B8
• D3DFMT_X8R8G8B8
• D3DFMT_G16R16
• D3DFMT_A8B8G8R8
• D3DFMT_A8
• D3DFMT_A8L8
• D3DFMT_Q8W8V8U8
• D3DFMT_V16U16
• D3DFMT_A16B16G16R16F
• D3DFMT_A16B16G16R16
• D3DFMT_R32F
• D3DFMT_G16R16F
• D3DFMT_A32B32G32R32F
• D3DFMT_G32R32F
• D3DFMT_R16F

If pD3DResource is of incorrect type or is already registered, then cudaInvalidDeviceCall is returned. If pD3DResource cannot be registered, then cudaErrorUnknown is returned.

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:

6.17. Direct3D 9 Interoperability
[DEPRECATED]

This section describes deprecated Direct3D 9 interoperability functions.

enum cudaD3D9MapFlags

CUDA D3D9 Map Flags

Values

cudaD3D9MapFlagsNone = 0
Default; Assume resource can be read/written

cudaD3D9MapFlagsReadOnly = 1
CUDA kernels will not write to this resource

cudaD3D9MapFlagsWriteDiscard = 2
CUDA kernels will only write to and will not read from this resource

enum cudaD3D9RegisterFlags

CUDA D3D9 Register Flags

Values

cudaD3D9RegisterFlagsNone = 0
Default; Resource can be accessed through void*

cudaD3D9RegisterFlagsArray = 1
Resource can be accessed through a CUarray*

__host__ cudaError_t cudaD3D9MapResources (int count, IDirect3DResource9 **ppResources)
Map Direct3D resources for access by CUDA.

Parameters

count
- Number of resources to map for CUDA

ppResources
- Resources to map for CUDA
Returns

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description

Deprecated This function is deprecated as of CUDA 3.0.

Maps the count Direct3D resources in ppResources for access by CUDA.

The resources in ppResources may be accessed in CUDA kernels until they are unmapped. Direct3D should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any Direct3D calls issued before cudaD3D9MapResources[] will complete before any CUDA kernels issued after cudaD3D9MapResources[] begin.

If any of ppResources have not been registered for use with CUDA or if ppResources contains any duplicate entries then cudaErrorInvalidResourceHandle is returned. If any of ppResources are presently mapped for access by CUDA then cudaErrorUnknown is returned.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsMapResources

__host__cudaError_t cudaD3D9RegisterResource
(IDirect3DResource9 *pResource, unsigned int flags)

Registers a Direct3D resource for access by CUDA.

Parameters

pResource
- Resource to register

flags
- Parameters for resource registration

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown
Description

Deprecated  This function is deprecated as of CUDA 3.0.

Registers the Direct3D resource pResource for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through cudaD3D9UnregisterResource(). Also on success, this call will increase the internal reference count on pResource. This reference count will be decremented when this resource is unregistered through cudaD3D9UnregisterResource().

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of pResource must be one of the following.

- IDirect3DVertexBuffer9: No notes.
- IDirect3DIndexBuffer9: No notes.
- IDirect3DSurface9: Only stand-alone objects of type IDirect3DSurface9 may be explicitly shared. In particular, individual mipmap levels and faces of cube maps may not be registered directly. To access individual surfaces associated with a texture, one must register the base texture object.

- IDirect3DBaseTexture9: When a texture is registered, all surfaces associated with all mipmap levels of all faces of the texture will be accessible to CUDA.

The flags argument specifies the mechanism through which CUDA will access the Direct3D resource. The following value is allowed:

- cudaD3D9RegisterFlagsNone: Specifies that CUDA will access this resource through a void*. The pointer, size, and pitch for each subresource of this resource may be queried through cudaD3D9ResourceGetMappedPointer(), cudaD3D9ResourceGetMappedSize(), and cudaD3D9ResourceGetMappedPitch() respectively. This option is valid for all resource types.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations:

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Any resources allocated in D3DPOOL_SYSTEMMEM or D3DPOOL_MANAGED may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.
If Direct3D interoperability is not initialized on this context, then `cudaErrorInvalidDevice` is returned. If `pResource` is of incorrect type (e.g., a non-stand-alone IDirect3DSurface9) or is already registered, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` cannot be registered then `cudaErrorUnknown` is returned.

**Note:**
Note that this function may also return error codes from previous, asynchronous launches.

See also:
`cudaGraphicsD3D9RegisterResource`

```c
__host__cudaError_t
cudaD3D9ResourceGetMappedArray (cudaArray **ppArray, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)
```

Get an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

**Parameters**
- **ppArray**
  - Returned array corresponding to subresource
- **pResource**
  - Mapped resource to access
- **face**
  - Face of resource to access
- **level**
  - Level of resource to access

**Returns**
- `cudaSuccess`
- `cudaErrorInvalidResourceHandle`
- `cudaErrorUnknown`

**Description**
`Deprecated` This function is deprecated as of CUDA 3.0.

Returns in `ppArray` an array through which the subresource of the mapped Direct3D resource `pResource`, which corresponds to `face` and `level` may be accessed. The value set in `pArray` may change every time that `pResource` is mapped.

If `pResource` is not registered then `cudaErrorInvalidResourceHandle` is returned. If `pResource` was not registered with usage flags `cudaD3D9RegisterFlagsArray`, then
cudaErrorInvalidResourceHandle is returned. If pResource is not mapped, then cudaErrorUnknown is returned.

For usage requirements of face and level parameters, see cudaD3D9ResourceGetMappedPointer().

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaGraphicsSubResourceGetMappedArray

__host__ cudaError_t
cudaD3D9ResourceGetMappedPitch (size_t *pPitch, size_t *pPitchSlice, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)
Get the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.

Parameters
pPitch
- Returned pitch of subresource
pPitchSlice
- Returned Z-slice pitch of subresource
pResource
- Mapped resource to access
face
- Face of resource to access
level
- Level of resource to access

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description
Deprecated This function is deprecated as of CUDA 3.0.

Returns in *pPitch and *pPitchSlice the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource pResource, which corresponds to face and level. The values set in pPitch and pPitchSlice may change every time that pResource is mapped.
The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position $x, y$ from the base pointer of the surface is:

$$y \times \text{pitch} + (\text{bytes per pixel}) \times x$$

For a 3D surface, the byte offset of the sample at position $x, y, z$ from the base pointer of the surface is:

$$z \times \text{slicePitch} + y \times \text{pitch} + (\text{bytes per pixel}) \times x$$

Both parameters $pPitch$ and $pPitchSlice$ are optional and may be set to NULL.

If $pResource$ is not of type IDirect3DBaseTexture9 or one of its sub-types or if $pResource$ has not been registered for use with CUDA, then cudaErrorInvalidResourceHandle is returned. If $pResource$ was not registered with usage flags cudaD3D9RegisterFlagsNone, then cudaErrorInvalidResourceHandle is returned. If $pResource$ is not mapped for access by CUDA then cudaErrorUnknown is returned.

For usage requirements of $\text{face}$ and $\text{level}$ parameters, see cudaD3D9ResourceGetMappedPointer[].

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaGraphicsResourceGetMappedPointer

---

__host__ cudaError_t
cudaD3D9ResourceGetMappedPointer (void **pPointer, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)

Get a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

Parameters

pPointer
  - Returned pointer corresponding to subresource

pResource
  - Mapped resource to access
**face**
- Face of resource to access

**level**
- Level of resource to access

**Returns**
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

**Description**
**Deprecated** This function is deprecated as of CUDA 3.0.

Returns in *pPointer the base pointer of the subresource of the mapped Direct3D resource pResource, which corresponds to face and level. The value set in pPointer may change every time that pResource is mapped.

If pResource is not registered, then cudaErrorInvalidResourceHandle is returned.
If pResource was not registered with usage flags cudaD3D9RegisterFlagsNone, then cudaErrorInvalidResourceHandle is returned. If pResource is not mapped, then cudaErrorUnknown is returned.

If pResource is of type IDirect3DCubeTexture9, then face must one of the values enumerated by type D3DCUBEMAP_FACES. For all other types, face must be 0. If face is invalid, then cudaErrorInvalidValue is returned.

If pResource is of type IDirect3DBaseTexture9, then level must correspond to a valid mipmap level. Only mipmap level 0 is supported for now. For all other types level must be 0. If level is invalid, then cudaErrorInvalidValue is returned.

**Note:**
Note that this function may also return error codes from previous, asynchronous launches.

**See also:**
cudaGraphicsResourceGetMappedPointer
__host__ cudaError_t
cudaD3D9ResourceGetMappedSize (size_t *pSize,
IDirect3DResource9 *pResource, unsigned int face,
unsigned int level)

Get the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.

Parameters

pSize
- Returned size of subresource

pResource
- Mapped resource to access

face
- Face of resource to access

level
- Level of resource to access

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description

Deprecated
This function is deprecated as of CUDA 3.0.

Returns in *pSize the size of the subresource of the mapped Direct3D resource pResource, which corresponds to face and level. The value set in pSize may change every time that pResource is mapped.

If pResource has not been registered for use with CUDA then cudaErrorInvalidResourceHandle is returned. If pResource was not registered with usage flags cudaD3D9RegisterFlagsNone, then cudaErrorInvalidResourceHandle is returned. If pResource is not mapped for access by CUDA then cudaErrorUnknown is returned.

For usage requirements of face and level parameters, see cudaD3D9ResourceGetMappedPointer().

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaGraphicsResourceGetMappedPointer
__host__ cudaError_t
cudaD3D9ResourceGetSurfaceDimensions
(size_t *pWidth, size_t *pHeight, size_t *pDepth,
IDirect3DResource9 *pResource, unsigned int face,
unsigned int level)

Get the dimensions of a registered Direct3D surface.

Parameters

pWidth
- Returned width of surface

pHeight
- Returned height of surface

pDepth
- Returned depth of surface

pResource
- Registered resource to access

face
- Face of resource to access

level
- Level of resource to access

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle.

Description

Deprecated This function is deprecated as of CUDA 3.0.

Returns in *pWidth, *pHeight, and *pDepth the dimensions of the subresource of the
mapped Direct3D resource pResource which corresponds to face and level.

Since anti-aliased surfaces may have multiple samples per pixel, it is possible that the
dimensions of a resource will be an integer factor larger than the dimensions reported by the
Direct3D runtime.

The parameters pWidth, pHeight, and pDepth are optional. For 2D surfaces, the value
returned in *pDepth will be 0.

If pResource is not of type IDirect3DBaseTexture9 or IDirect3DSurface9 or if pResource has
not been registered for use with CUDA, then cudaErrorInvalidResourceHandle is returned.

For usage requirements of face and level parameters, see
Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaGraphicsSubResourceGetMappedArray

__host__cudaError_t
cudaD3D9ResourceSetMapFlags (IDirect3DResource9 *pResource, unsigned int flags)
Set usage flags for mapping a Direct3D resource.

Parameters

pResource
- Registered resource to set flags for
flags
- Parameters for resource mapping

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description

Deprecated This function is deprecated as of CUDA 3.0.
Set flags for mapping the Direct3D resource pResource.
Changes to flags will take effect the next time pResource is mapped. The flags argument may be any of the following:

- cudaD3D9MapFlagsNone: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- cudaD3D9MapFlagsReadOnly: Specifies that CUDA kernels which access this resource will not write to this resource.
- cudaD3D9MapFlagsWriteDiscard: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If pResource has not been registered for use with CUDA, then cudaErrorInvalidResourceHandle is returned. If pResource is presently mapped for access by CUDA, then cudaErrorUnknown is returned.
__host__ cudaError_t cudaD3D9UnmapResources (int count, IDirect3DResource9 **ppResources)

Unmap Direct3D resources for access by CUDA.

Parameters

- **count**
  - Number of resources to unmap for CUDA

- **ppResources**
  - Resources to unmap for CUDA

Returns

- cudaSuccess
- cudaErrorInvalidResourceHandle
- cudaErrorUnknown

Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Unmaps the *count* Direct3D resources in *ppResources*.

This function provides the synchronization guarantee that any CUDA kernels issued before `cudaD3D9UnmapResources` will complete before any Direct3D calls issued after `cudaD3D9UnmapResources` begin.

If any of *ppResources* have not been registered for use with CUDA or if *ppResources* contains any duplicate entries, then `cudaErrorInvalidResourceHandle` is returned. If any of *ppResources* are not presently mapped for access by CUDA then `cudaErrorUnknown` is returned.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnmapResources
__host__cudaError_t cudaD3D9UnregisterResource (IDirect3DResource9 *pResource)

Unregisters a Direct3D resource for access by CUDA.

Parameters

pResource
- Resource to unregister

Returns

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description

Deprecated

This function is deprecated as of CUDA 3.0.

Unregisters the Direct3D resource pResource so it is not accessible by CUDA unless registered again.

If pResource is not registered, then cudaErrorInvalidResourceHandle is returned.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnregisterResource

6.18. Direct3D 10 Interoperability

This section describes the Direct3D 10 interoperability functions of the CUDA runtime application programming interface. Note that mapping of Direct3D 10 resources is performed with the graphics API agnostic, resource mapping interface described in Graphics Interopability.

enum cudaD3D10DeviceList

CUDA devices corresponding to a D3D10 device

Values

cudaD3D10DeviceListAll = 1
The CUDA devices for all GPUs used by a D3D10 device
\texttt{cudaD3D10DeviceListCurrentFrame} = 2
The CUDA devices for the GPUs used by a D3D10 device in its currently rendering frame
\texttt{cudaD3D10DeviceListNextFrame} = 3
The CUDA devices for the GPUs to be used by a D3D10 device in the next frame

\begin{verbatim}
\_host\_\_cudaError\_t cudaD3D10GetDevice (int *device, IDXGIAdapter *pAdapter)
\end{verbatim}
Gets the device number for an adapter.

\textbf{Parameters}

\textbf{device}
- Returns the device corresponding to pAdapter

\textbf{pAdapter}
- D3D10 adapter to get device for

\textbf{Returns}
\texttt{cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown}

\textbf{Description}

Returns in *device the CUDA-compatible device corresponding to the adapter \texttt{pAdapter} obtained from IDXGIFactory::EnumAdapters. This call will succeed only if a device on adapter \texttt{pAdapter} is CUDA-compatible.

\textbf{Note:}
Note that this function may also return error codes from previous, asynchronous launches.

\textbf{See also:}
\texttt{cudaGraphicsD3D10RegisterResource, cuD3D10GetDevice}
__host__ cudaError_t cudaD3D10GetDevices(unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, ID3D10Device *pD3D10Device, cudaD3D10DeviceList deviceList)

Gets the CUDA devices corresponding to a Direct3D 10 device.

**Parameters**

- **pCudaDeviceCount**
  - Returned number of CUDA devices corresponding to pD3D10Device
- **pCudaDevices**
  - Returned CUDA devices corresponding to pD3D10Device
- **cudaDeviceCount**
  - The size of the output device array pCudaDevices
- **pD3D10Device**
  - Direct3D 10 device to query for CUDA devices
- **deviceList**
  - The set of devices to return. This set may be `cudaD3D10DeviceListAll` for all devices, `cudaD3D10DeviceListCurrentFrame` for the devices used to render the current frame (in SLI), or `cudaD3D10DeviceListNextFrame` for the devices used to render the next frame (in SLI).

**Returns**

- `cudaSuccess`
- `cudaErrorNoDevice`
- `cudaErrorUnknown`

**Description**

Returns in *pCudaDeviceCount the number of CUDA-compatible devices corresponding to the Direct3D 10 device pD3D10Device. Also returns in *pCudaDevices at most cudaDeviceCount of the the CUDA-compatible devices corresponding to the Direct3D 10 device pD3D10Device.

If any of the GPUs being used to render pDevice are not CUDA capable then the call will return `cudaErrorNoDevice`.

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**
__host__ cudaError_t
cudaGraphicsD3D10RegisterResource
(cudaGraphicsResource **resource, ID3D10Resource *pD3DResource, unsigned int flags)
Registers a Direct3D 10 resource for access by CUDA.

Parameters
resource
- Pointer to returned resource handle
pD3DResource
- Direct3D resource to register
flags
- Parameters for resource registration

Returns
cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue,
cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description
Registers the Direct3D 10 resource pD3DResource for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through cudaGraphicsUnregisterResource(). Also on success, this call will increase the internal reference count on pD3DResource. This reference count will be decremented when this resource is unregistered through cudaGraphicsUnregisterResource().

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of pD3DResource must be one of the following.

- ID3D10Buffer: may be accessed via a device pointer
- ID3D10Texture1D: individual subresources of the texture may be accessed via arrays
- ID3D10Texture2D: individual subresources of the texture may be accessed via arrays
- ID3D10Texture3D: individual subresources of the texture may be accessed via arrays

The flags argument may be used to specify additional parameters at register time. The valid values for this parameter are

cudaGraphicsUnregisterResource, cudaGraphicsMapResources,
cudaGraphicsSubResourceGetMappedArray, cudaGraphicsResourceGetMappedPointer,
cuD3D10GetDevices
- **cudaGraphicsRegisterFlagsNone**: Specifies no hints about how this resource will be used.
- **cudaGraphicsRegisterFlagsSurfaceLoadStore**: Specifies that CUDA will bind this resource to a surface reference.
- **cudaGraphicsRegisterFlagsTextureGather**: Specifies that CUDA will perform texture gather operations on this resource.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

A complete list of supported DXGI formats is as follows. For compactness the notation $A_{B,C,D}$ represents $A_B$, $A_C$, and $A_D$.

- DXGI_FORMAT_A8_UNORM
- DXGI_FORMAT_B8G8R8A8_UNORM
- DXGI_FORMAT_B8G8R8X8_UNORM
- DXGI_FORMAT_R16_FLOAT
- DXGI_FORMAT_R16G16B16A16_{FLOAT,SINT,SNORM,UINT,UNORM}
- DXGI_FORMAT_R16G16_{FLOAT,SINT,SNORM,UINT,UNORM}
- DXGI_FORMAT_R16_{SINT,SNORM,UINT,UNORM}
- DXGI_FORMAT_R32_FLOAT
- DXGI_FORMAT_R32G32B32A32_{FLOAT,SINT,UINT}
- DXGI_FORMAT_R32G32_{FLOAT,SINT,UINT}
- DXGI_FORMAT_R32_{SINT,UINT}
- DXGI_FORMAT_R8G8B8A8_{SINT,SNORM,UINT,UNORM,UNORM_SRGB}
- DXGI_FORMAT_R8G8_{SINT,SNORM,UINT,UNORM}
- DXGI_FORMAT_R8_{SINT,SNORM,UINT,UNORM}

If `pD3DResource` is of incorrect type or is already registered, then `cudaErrorInvalidResourceHandle` is returned. If `pD3DResource` cannot be registered, then `cudaErrorUnknown` is returned.
6.19. Direct3D 10 Interoperability
[DEPRECATED]

This section describes deprecated Direct3D 10 interoperability functions.

enum cudaD3D10MapFlags

CUDA D3D10 Map Flags

Values

cudaD3D10MapFlagsNone = 0
  Default; Assume resource can be read/written

cudaD3D10MapFlagsReadOnly = 1
  CUDA kernels will not write to this resource

cudaD3D10MapFlagsWriteDiscard = 2
  CUDA kernels will only write to and will not read from this resource

enum cudaD3D10RegisterFlags

CUDA D3D10 Register Flags

Values

cudaD3D10RegisterFlagsNone = 0
  Default; Resource can be accessed through a void*

cudaD3D10RegisterFlagsArray = 1
  Resource can be accessed through a CUarray*
__host__ cudaError_t cudaD3D10GetDirect3DDevice (ID3D10Device **ppD3D10Device)

Gets the Direct3D device against which the current CUDA context was created.

Parameters

**ppD3D10Device**
- Returns the Direct3D device for this thread

Returns

cuErrorSuccess, cuErrorUnknown

Description

Deprecated
This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with a D3D10 device in order to achieve maximum interoperability performance.

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaD3D10SetDirect3DDevice

__host__ cudaError_t cudaD3D10MapResources (int count, ID3D10Resource **ppResources)

Maps Direct3D Resources for access by CUDA.

Parameters

**count**
- Number of resources to map for CUDA

**ppResources**
- Resources to map for CUDA

Returns

cuErrorSuccess, cuErrorInvalidResourceHandle, cuErrorUnknown

Description

Deprecated This function is deprecated as of CUDA 3.0.

Maps the count Direct3D resources in ppResources for access by CUDA.

The resources in ppResources may be accessed in CUDA kernels until they are unmapped. Direct3D should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any Direct3D calls issued before cudaD3D10MapResources() will complete before any CUDA kernels issued after cudaD3D10MapResources() begin.

If any of ppResources have not been registered for use with CUDA or if ppResources contains any duplicate entries then cudaErrorInvalidResourceHandle is returned. If any of ppResources are presently mapped for access by CUDA then cudaErrorUnknown is returned.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsMapResources

__host__cudaError_t cudaD3D10RegisterResource(ID3D10Resource *pResource, unsigned int flags)

Registers a Direct3D 10 resource for access by CUDA.

Parameters

pResource
  - Resource to register
flags
  - Parameters for resource registration

Returns

cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description

Deprecated This function is deprecated as of CUDA 3.0.
Registers the Direct3D resource pResource for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through cudaD3D10UnregisterResource(). Also on success, this call will increase the internal reference count on pResource. This reference count will be decremented when this resource is unregistered through cudaD3D10UnregisterResource().

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of pResource must be one of the following:

- ID3D10Buffer: Cannot be used with flags set to cudaD3D10RegisterFlagsArray.
- ID3D10Texture1D: No restrictions.
- ID3D10Texture2D: No restrictions.
- ID3D10Texture3D: No restrictions.

The flags argument specifies the mechanism through which CUDA will access the Direct3D resource. The following values are allowed.

- cudaD3D10RegisterFlagsNone: Specifies that CUDA will access this resource through a void*. The pointer, size, and pitch for each subresource of this resource may be queried through cudaD3D10ResourceGetMappedPointer(), cudaD3D10ResourceGetMappedSize(), and cudaD3D10ResourceGetMappedPitch() respectively. This option is valid for all resource types.
- cudaD3D10RegisterFlagsArray: Specifies that CUDA will access this resource through a CUarray queried on a sub-resource basis through cudaD3D10ResourceGetMappedArray(). This option is only valid for resources of type ID3D10Texture1D, ID3D10Texture2D, and ID3D10Texture3D.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized on this context then cudaErrorInvalidDevice is returned. If pResource is of incorrect type or is already registered then cudaErrorInvalidResourceHandle is returned. If pResource cannot be registered then cudaErrorUnknown is returned.
__host__ cudaError_t

cudaD3D10ResourceGetMappedArray (cudaArray **ppArray, ID3D10Resource *pResource, unsigned int subResource)

Gets an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

Parameters

ppArray
- Returned array corresponding to subresource

pResource
- Mapped resource to access

subResource
- Subresource of pResource to access

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description

Deprecated This function is deprecated as of CUDA 3.0.

Returns in *ppArray an array through which the subresource of the mapped Direct3D resource pResource which corresponds to subResource may be accessed. The value set in ppArray may change every time that pResource is mapped.

If pResource is not registered, then cudaErrorInvalidResourceHandle is returned. If pResource was not registered with usage flags cudaD3D10RegisterFlagsArray, then cudaErrorInvalidResourceHandle is returned. If pResource is not mapped then cudaErrorUnknown is returned.

For usage requirements of the subResource parameter, see cudaD3D10ResourceGetMappedPointer().
Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaGraphicsSubResourceGetMappedArray

g__host__cudaError_t
cudaD3D10ResourceGetMappedPitch (size_t *pPitch, size_t *pPitchSlice, ID3D10Resource *pResource, unsigned int subResource)

Gets the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.

Parameters

pPitch
- Returned pitch of subresource

pPitchSlice
- Returned Z-slice pitch of subresource

pResource
- Mapped resource to access

subResource
- Subresource of pResource to access

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description

Deprecated This function is deprecated as of CUDA 3.0.

Returns in *pPitch and *pPitchSlice the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource pResource, which corresponds to subResource. The values set in pPitch and pPitchSlice may change every time that pResource is mapped.

The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position x, y from the base pointer of the surface is:

\[ y \times \text{pitch} + \lfloor \text{bytes per pixel} \rfloor \times x \]
For a 3D surface, the byte offset of the sample at position \( x, y, z \) from the base pointer of the surface is:

\[
z \cdot \text{slicePitch} + y \cdot \text{pitch} + (\text{bytes per pixel}) \cdot x
\]

Both parameters \( \text{pPitch} \) and \( \text{pPitchSlice} \) are optional and may be set to NULL.

If \( \text{pResource} \) is not of type ID3D10Texture1D, ID3D10Texture2D, or ID3D10Texture3D, or if \( \text{pResource} \) has not been registered for use with CUDA, then \text{cudaErrorInvalidResourceHandle} \) is returned. If \( \text{pResource} \) was not registered with usage flags \text{cudaD3D10RegisterFlagsNone}, then \text{cudaErrorInvalidResourceHandle} \) is returned. If \( \text{pResource} \) is not mapped for access by CUDA then \text{cudaErrorUnknown} \) is returned.

For usage requirements of the \( \text{subResource} \) parameter see \text{cudaD3D10ResourceGetMappedPointer}().

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
\text{cudaGraphicsSubResourceGetMappedArray}

__host__cudaError_t
\text{cudaD3D10ResourceGetMappedPointer} (void **\text{pPointer}, ID3D10Resource *\text{pResource}, unsigned \text{int} \text{subResource})

Gets a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

Parameters
\text{pPointer} 
- Returned pointer corresponding to subresource
\text{pResource} 
- Mapped resource to access
\text{subResource} 
- Subresource of \text{pResource} to access

Returns
\text{cudaSuccess}, \text{cudaErrorInvalidValue}, \text{cudaErrorInvalidResourceHandle}, \text{cudaErrorUnknown}
Description

Deprecated This function is deprecated as of CUDA 3.0.

Returns in *pPointer the base pointer of the subresource of the mapped Direct3D resource pResource which corresponds to subResource. The value set in pPointer may change every time that pResource is mapped.

If pResource is not registered, then cudaErrorInvalidResourceHandle is returned. If pResource was not registered with usage flags cudaD3D9RegisterFlagsNone, then cudaErrorInvalidResourceHandle is returned. If pResource is not mapped then cudaErrorUnknown is returned.

If pResource is of type ID3D10Buffer then subResource must be 0. If pResource is of any other type, then the value of subResource must come from the subresource calculation in D3D10CalcSubResource().

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsResourceGetMappedPointer

__host__cudaError_t
cudaD3D10ResourceGetMappedSize (size_t *pSize, ID3D10Resource *pResource, unsigned int subResource)

Gets the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.

Parameters

pSize
  - Returned size of subresource
pResource
  - Mapped resource to access
subResource
  - Subresource of pResource to access

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown
Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Returns in \*pSize the size of the subresource of the mapped Direct3D resource pResource which corresponds to subResource. The value set in pSize may change every time that pResource is mapped.

If pResource has not been registered for use with CUDA then cudaErrorInvalidHandle is returned. If pResource was not registered with usage flags cudaD3D10RegisterFlagsNone, then cudaErrorInvalidResourceHandle is returned. If pResource is not mapped for access by CUDA then cudaErrorUnknown is returned.

For usage requirements of the subResource parameter see cudaD3D10ResourceGetMappedPointer().

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsResourceGetMappedPointer

```c
__host__cudaError_t
cudaD3D10ResourceGetSurfaceDimensions(
    size_t *pWidth, size_t *pHeight, size_t *pDepth,
    ID3D10Resource *pResource, unsigned int
    subResource)
```

Gets the dimensions of a registered Direct3D surface.

**Parameters**

- **pWidth**
  - Returned width of surface
- **pHeight**
  - Returned height of surface
- **pDepth**
  - Returned depth of surface
- **pResource**
  - Registered resource to access
- **subResource**
  - Subresource of pResource to access
Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle.

Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Returns in *pWidth, *pHeight, and *pDepth the dimensions of the subresource of the mapped Direct3D resource pResource which corresponds to subResource.

Since anti-aliased surfaces may have multiple samples per pixel, it is possible that the dimensions of a resource will be an integer factor larger than the dimensions reported by the Direct3D runtime.

The parameters pWidth, pHeight, and pDepth are optional. For 2D surfaces, the value returned in *pDepth will be 0.

If pResource is not of type ID3D10Texture1D, ID3D10Texture2D, or ID3D10Texture3D, or if pResource has not been registered for use with CUDA, then cudaErrorInvalidHandle is returned.

For usage requirements of subResource parameters see cudaD3D10ResourceGetMappedPointer().

---

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

cudaGraphicsSubResourceGetMappedArray

---

**__host__cudaError_t**

cudaD3D10ResourceSetMapFlags (ID3D10Resource *pResource, unsigned int flags)

Set usage flags for mapping a Direct3D resource.

**Parameters**

- **pResource**
  - Registered resource to set flags for

- **flags**
  - Parameters for resource mapping
Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown.

Description

Deprecated This function is deprecated as of CUDA 3.0.

Set usage flags for mapping the Direct3D resource pResource.

Changes to flags will take effect the next time pResource is mapped. The flags argument may be any of the following:

- **cudaD3D10MapFlagsNone**: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.

- **cudaD3D10MapFlagsReadOnly**: Specifies that CUDA kernels which access this resource will not write to this resource.

- **cudaD3D10MapFlagsWriteDiscard**: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If pResource has not been registered for use with CUDA then cudaErrorInvalidHandle is returned. If pResource is presently mapped for access by CUDA then cudaErrorUnknown is returned.

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaGraphicsResourceSetMapFlags

```c
__host__ cudaError_t cudaD3D10SetDirect3DDevice(ID3D10Device *pD3D10Device, int device)
```

Sets the Direct3D 10 device to use for interoperability with a CUDA device.

**Parameters**

- **pD3D10Device**: Direct3D device to use for interoperability
device
- The CUDA device to use. This device must be among the devices returned when querying cudaD3D10DeviceListAll from cudaD3D10GetDevices, may be set to -1 to automatically select an appropriate CUDA device.

Returns
cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorSetOnActiveProcess

Description
Deprecated
This function is deprecated as of CUDA 5.0.
This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with a D3D10 device in order to achieve maximum interoperability performance.

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaD3D10GetDevice, cudaGraphicsD3D10RegisterResource, cudaDeviceReset

__host__cudaError_t cudaD3D10UnmapResources(int count, ID3D10Resource **ppResources)
Unmaps Direct3D resources.

Parameters
count
- Number of resources to unmap for CUDA
ppResources
- Resources to unmap for CUDA

Returns
cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description
Deprecated
This function is deprecated as of CUDA 3.0.
Unmaps the count Direct3D resource in ppResources.
This function provides the synchronization guarantee that any CUDA kernels issued before `cudaD3D10UnmapResources()` will complete before any Direct3D calls issued after `cudaD3D10UnmapResources()` begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries, then `cudaErrorInvalidResourceHandle` is returned. If any of `ppResources` are not presently mapped for access by CUDA then `cudaErrorUnknown` is returned.

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

See also:

`cudaGraphicsUnmapResources`

```__host__cudaError_t cudaD3D10UnregisterResource(ID3D10Resource *pResource)`

Unregisters a Direct3D resource.

**Parameters**

- **pResource**
  - Resource to unregister

**Returns**

`cudaSuccess`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

**Description**

**Deprecated** This function is deprecated as of CUDA 3.0.

Unregisters the Direct3D resource `resource` so it is not accessible by CUDA unless registered again.

If `pResource` is not registered, then `cudaErrorInvalidResourceHandle` is returned.

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

See also:

`cudaGraphicsUnregisterResource`
6.20. Direct3D 11 Interoperability

This section describes the Direct3D 11 interoperability functions of the CUDA runtime application programming interface. Note that mapping of Direct3D 11 resources is performed with the graphics API agnostic, resource mapping interface described in [Graphics Interopability](#).

```c
enum cudaD3D11DeviceList

CUDA devices corresponding to a D3D11 device

Values

cudaD3D11DeviceListAll = 1
    The CUDA devices for all GPUs used by a D3D11 device

cudaD3D11DeviceListCurrentFrame = 2
    The CUDA devices for the GPUs used by a D3D11 device in its currently rendering frame

cudaD3D11DeviceListNextFrame = 3
    The CUDA devices for the GPUs to be used by a D3D11 device in the next frame

__host__ cudaError_t cudaD3D11GetDevice (int *device, IDXGIAdapter *pAdapter)

Gets the device number for an adapter.

Parameters

device
    - Returns the device corresponding to pAdapter

pAdapter
    - D3D11 adapter to get device for

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown

Description

Returns in *device the CUDA-compatible device corresponding to the adapter pAdapter obtained from IDXGIFactory::EnumAdapters. This call will succeed only if a device on adapter pAdapter is CUDA-compatible.

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:


__host__ cudaError_t cudaD3D11GetDevices
(unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, ID3D11Device *pD3D11Device, cudaD3D11DeviceList deviceList)

Gets the CUDA devices corresponding to a Direct3D 11 device.

Parameters

pCudaDeviceCount
- Returned number of CUDA devices corresponding to pD3D11Device

pCudaDevices
- Returned CUDA devices corresponding to pD3D11Device

cudaDeviceCount
- The size of the output device array pCudaDevices

pD3D11Device
- Direct3D 11 device to query for CUDA devices

deviceList
- The set of devices to return. This set may be cudaD3D11DeviceListAll for all devices, cudaD3D11DeviceListCurrentFrame for the devices used to render the current frame (in SLI), or cudaD3D11DeviceListNextFrame for the devices used to render the next frame (in SLI).

Returns
cudaSuccess, cudaErrorNoDevice, cudaErrorUnknown

Description

Returns in *pCudaDeviceCount the number of CUDA-compatible devices corresponding to the Direct3D 11 device pD3D11Device. Also returns in *pCudaDevices at most cudaDeviceCount of the the CUDA-compatible devices corresponding to the Direct3D 11 device pD3D11Device.

If any of the GPUs being used to render pDevice are not CUDA capable then the call will return cudaErrorNoDevice.
Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:

__host__cudaError_t
cudaGraphicsD3D11RegisterResource
(cudaGraphicsResource **resource, ID3D11Resource *pD3DResource, unsigned int flags)
Register a Direct3D 11 resource for access by CUDA.

Parameters

resource
- Pointer to returned resource handle
pD3DResource
- Direct3D resource to register
flags
- Parameters for resource registration

Returns
cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description

Registers the Direct3D 11 resource pD3DResource for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through cudaGraphicsUnregisterResource[]. Also on success, this call will increase the internal reference count on pD3DResource. This reference count will be decremented when this resource is unregistered through cudaGraphicsUnregisterResource[].

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of pD3DResource must be one of the following.

- ID3D11Buffer: may be accessed via a device pointer
ID3D11Texture1D: individual subresources of the texture may be accessed via arrays
ID3D11Texture2D: individual subresources of the texture may be accessed via arrays
ID3D11Texture3D: individual subresources of the texture may be accessed via arrays

The `flags` argument may be used to specify additional parameters at register time. The valid values for this parameter are

- `cudaGraphicsRegisterFlagsNone`: Specifies no hints about how this resource will be used.
- `cudaGraphicsRegisterFlagsSurfaceLoadStore`: Specifies that CUDA will bind this resource to a surface reference.
- `cudaGraphicsRegisterFlagsTextureGather`: Specifies that CUDA will perform texture gather operations on this resource.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

A complete list of supported DXGI formats is as follows. For compactness the notation $A_{(B,C,D)}$ represents $A_B$, $A_C$, and $A_D$.

- `DXGI_FORMAT_A8_UNORM`
- `DXGI_FORMAT_B8G8R8A8_UNORM`
- `DXGI_FORMAT_B8G8R8X8_UNORM`
- `DXGI_FORMAT_R16_FLOAT`
- `DXGI_FORMAT_R16G16B16A16_{FLOAT,SINT,SNORM,UINT,UNORM}`
- `DXGI_FORMAT_R16G16_{FLOAT,SINT,SNORM,UINT,UNORM}`
- `DXGI_FORMAT_R16_{SINT,SNORM,UINT,UNORM}`
- `DXGI_FORMAT_R32_FLOAT`
- `DXGI_FORMAT_R32G32B32A32_{FLOAT,SINT,UINT}`
- `DXGI_FORMAT_R32G32_{FLOAT,SINT,UINT}`
- `DXGI_FORMAT_R32_{SINT,UINT}`
- `DXGI_FORMAT_R8G8B8A8_{SINT,SNORM,UINT,UNORM,UNORM_SRGB}`
- `DXGI_FORMAT_R8G8_{SINT,SNORM,UINT,UNORM}`
- `DXGI_FORMAT_R8_{SINT,SNORM,UINT,UNORM}`
If `pD3DResource` is of incorrect type or is already registered, then `cudaErrorInvalidResourceHandle` is returned. If `pD3DResource` cannot be registered, then `cudaErrorUnknown` is returned.

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
- `cudaGraphicsUnregisterResource`
- `cudaGraphicsMapResources`
- `cudaGraphicsSubResourceGetMappedArray`
- `cudaGraphicsResourceGetMappedPointer`
- `cuGraphicsD3D11RegisterResource`

6.21. Direct3D 11 Interoperability
[DEPRECATED]

This section describes deprecated Direct3D 11 interoperability functions.

```c
__host__cudaError_t cudaD3D11GetDirect3DDevice(ID3D11Device **ppD3D11Device)
```

Gets the Direct3D device against which the current CUDA context was created.

Parameters

`ppD3D11Device`
- Returns the Direct3D device for this thread

Returns

`cudaSuccess`, `cudaErrorUnknown`

Description

**Deprecated** This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with a D3D11 device in order to achieve maximum interoperability performance.

Note:
__host__cudaError_t cudaD3D11SetDirect3DDevice
(ID3D11Device *pD3D11Device, int device)
Sets the Direct3D 11 device to use for interoperability with a CUDA device.

Parameters

pD3D11Device
- Direct3D device to use for interoperability
device
- The CUDA device to use. This device must be among the devices returned when querying cudaD3D11DeviceListAll from cudaD3D11GetDevices, may be set to -1 to automatically select an appropriate CUDA device.

Returns

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorSetOnActiveProcess

Description

Deprecated This function is deprecated as of CUDA 5.0.
This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with a D3D11 device in order to achieve maximum interoperability performance.

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:

6.22. VDPAU Interoperability

This section describes the VDPAU interoperability functions of the CUDA runtime application programming interface.
__host__ cudaError_t
cudaGraphicsVDPAURegisterOutputSurface
(cudaGraphicsResource **resource,
VdpOutputSurface vdpSurface, unsigned int flags)

Register a VdpOutputSurface object.

Parameters

resource
- Pointer to the returned object handle

vdpSurface
- VDPAU object to be registered

flags
- Map flags

Returns

cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue,
cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description

Registers the VdpOutputSurface specified by vdpSurface for access by CUDA. A handle to
the registered object is returned as resource. The surface’s intended usage is specified
using flags, as follows:

- **cudaGraphicsMapFlagsNone**: Specifies no hints about how this resource will be used. It is
  therefore assumed that this resource will be read from and written to by CUDA. This is the
  default value.

- **cudaGraphicsMapFlagsReadOnly**: Specifies that CUDA will not write to this resource.

- **cudaGraphicsMapFlagsWriteDiscard**: Specifies that CUDA will not read from this resource
  and will write over the entire contents of the resource, so none of the data previously
  stored in the resource will be preserved.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaVDPAUSetVDPAUDevice, cudaGraphicsUnregisterResource,
cudaGraphicsSubResourceGetMappedArray, cuGraphicsVDPAURegisterOutputSurface
__host__ cudaError_t
cudaGraphicsVDPAURegisterVideoSurface
(cudaGraphicsResource **resource, VdpVideoSurface vdpSurface, unsigned int flags)

Register a VdpVideoSurface object.

Parameters

resource
  - Pointer to the returned object handle
vdpSurface
  - VDPAU object to be registered
flags
  - Map flags

Returns
cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue,
cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description

Registers the VdpVideoSurface specified by vdpSurface for access by CUDA. A handle to the registered object is returned as resource. The surface’s intended usage is specified using flags, as follows:

- **cudaGraphicsMapFlagsNone**: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- **cudaGraphicsMapFlagsReadOnly**: Specifies that CUDA will not write to this resource.
- **cudaGraphicsMapFlagsWriteDiscard**: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaVDPAUSetVDPAUDevice, cudaGraphicsUnregisterResource,
cudaGraphicsSubResourceGetMappedArray, cuGraphicsVDPAURegisterVideoSurface
__host__cudaError_t cudaVDPAUGetDevice (int *device, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)

Gets the CUDA device associated with a VdpDevice.

Parameters

device
- Returns the device associated with vdpDevice, or -1 if the device associated with vdpDevice is not a compute device.

vdpDevice
- A VdpDevice handle

vdpGetProcAddress
- VDPAU’s VdpGetProcAddress function pointer

Returns
cudaSuccess

Description

Returns the CUDA device associated with a VdpDevice, if applicable.

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaVDPAUSetVDPAUDevice, cuVDPAUGetDevice

__host__cudaError_t cudaVDPAUSetVDPAUDevice (int device, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)

Sets a CUDA device to use VDPAU interoperability.

Parameters

device
- Device to use for VDPAU interoperability

vdpDevice
- The VdpDevice to interoperate with
**vdpGetProcAddress**
- VDPAU's VdpGetProcAddress function pointer

**Returns**
cudaSuccess, cudaErrorInvalidDevice, cudaErrorSetOnActiveProcess

**Description**
Records vdpDevice as the VdpDevice for VDPAU interoperability with the CUDA device device and sets device as the current device for the calling host thread.

If device has already been initialized then this call will fail with the error cudaErrorSetOnActiveProcess. In this case it is necessary to reset device using cudaDeviceReset() before VDPAU interoperability on device may be enabled.

**Note:**
Note that this function may also return error codes from previous, asynchronous launches.

**See also:**
cudaGraphicsVDPAURegisterVideoSurface, cudaGraphicsVDPAURegisterOutputSurface, cudaDeviceReset

### 6.23. EGL Interoperability

This section describes the EGL interoperability functions of the CUDA runtime application programming interface.

```c
__host__ cudaError_t cudaEGLStreamConsumerAcquireFrame (cudaEglStreamConnection *conn,
cudaGraphicsResource_t *pCudaResource,
cudaStream_t *pStream, unsigned int timeout)
```

Acquire an image frame from the EGLStream with CUDA as a consumer.

**Parameters**

- **conn**
  - Connection on which to acquire
**pCudaResource**
- CUDA resource on which the EGLStream frame will be mapped for use.

**pStream**
- CUDA stream for synchronization and any data migrations implied by `cudaEglResourceLocationFlags`.

**timeout**
- Desired timeout in usec.

**Returns**
cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown, cudaErrorLaunchTimeout

**Description**
Acquire an image frame from EGLStreamKHR. `cudaGraphicsResourceGetMappedEglFrame` can be called on `pCudaResource` to get `cudaEglFrame`.

**See also:**
cudaEGLStreamConsumerConnect, cudaEGLStreamConsumerDisconnect, cudaEGLStreamConsumerReleaseFrame, cuEGLStreamConsumerAcquireFrame

```c
__host__ cudaError_t cudaEGLStreamConsumerConnect(cudaEglStreamConnection *conn, EGLStreamKHR eglStream)
```

Connect CUDA to EGLStream as a consumer.

**Parameters**

**conn**
- Pointer to the returned connection handle

**eglStream**
- EGLStreamKHR handle

**Returns**
cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown

**Description**
Connect CUDA as a consumer to EGLStreamKHR specified by `eglStream`. The EGLStreamKHR is an EGL object that transfers a sequence of image frames from one API to another.
__host__ cudaError_t
cudaEGLStreamConsumerConnectWithFlags
cudaEglStreamConnection *conn, EGLStreamKHR eglStream, unsigned int flags)

Connect CUDA to EGLStream as a consumer with given flags.

Parameters

conn
- Pointer to the returned connection handle
eglStream
- EGLStreamKHR handle
flags
- Flags denote intended location - system or video.

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown

Description

Connect CUDA as a consumer to EGLStreamKHR specified by stream with specified flags defined by cudaEglResourceLocationFlags.

The flags specify whether the consumer wants to access frames from system memory or video memory. Default is cudaEglResourceLocationVidmem.

See also:
cudaEGLStreamConsumerDisconnect, cudaEGLStreamConsumerAcquireFrame,
cudaEGLStreamConsumerReleaseFrame, cuEGLStreamConsumerConnectWithFlags
__host__ cudaError_t

cudaEGLStreamConsumerDisconnect
(cudaEglStreamConnection *conn)

Disconnect CUDA as a consumer to EGLStream.

Parameters

conn
- Connection to disconnect.

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown

Description

Disconnect CUDA as a consumer to EGLStreamKHR.

See also:
cudaEGLStreamConsumerConnect, cudaEGLStreamConsumerAcquireFrame,
cudaEGLStreamConsumerReleaseFrame, cuEGLStreamConsumerDisconnect

__host__ cudaError_t

cudaEGLStreamConsumerReleaseFrame
(cudaEglStreamConnection *conn,
cudaGraphicsResource_t pCudaResource,
cudaStream_t *pStream)

Releases the last frame acquired from the EGLStream.

Parameters

conn
- Connection on which to release

pCudaResource
- CUDA resource whose corresponding frame is to be released

pStream
- CUDA stream on which release will be done.

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown
Description

Release the acquired image frame specified by pCudaResource to EGLStreamKHR.

See also:

cudaEGLStreamConsumerConnect, cudaEGLStreamConsumerDisconnect,
cudaEGLStreamConsumerAcquireFrame, cuEGLStreamConsumerReleaseFrame

__host__cudaError_t

cudaEGLStreamProducerConnect

cudaEglStreamConnection *conn, EGLStreamKHR eglStream, EGLint width, EGLint height)

Connect CUDA to EGLStream as a producer.

Parameters

c conn
  - Pointer to the returned connection handle
c eglStream
  - EGLStreamKHR handle
c width
  - width of the image to be submitted to the stream
c height
  - height of the image to be submitted to the stream

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown

Description

Connect CUDA as a producer to EGLStreamKHR specified by stream.
The EGLStreamKHR is an EGL object that transfers a sequence of image frames from one API to another.

See also:

cudaEGLStreamProducerDisconnect, cudaEGLStreamProducerPresentFrame,
cudaEGLStreamProducerReturnFrame, cuEGLStreamProducerConnect
__host__ cudaError_t
cudaEGLStreamProducerDisconnect
cudaEglStreamConnection *conn)

Disconnect CUDA as a producer to EGLStream.

Parameters

conn
- Connection to disconnect.

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown

Description

Disconnect CUDA as a producer to EGLStreamKHR.

See also:
cudaEGLStreamProducerConnect, cudaEGLStreamProducerPresentFrame,
cudaEGLStreamProducerReturnFrame, cuEGLStreamProducerDisconnect

__host__ cudaError_t
cudaEGLStreamProducerPresentFrame
cudaEglStreamConnection *conn, cudaEglFrame
eglframe, cudaStream_t *pStream)

Present a CUDA eglFrame to the EGLStream with CUDA as a producer.

Parameters

conn
- Connection on which to present the CUDA array
eglframe
- CUDA Eglstream Producer Frame handle to be sent to the consumer over EglStream.
pStream
- CUDA stream on which to present the frame.

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown
Description

The `cudaEglFrame` is defined as:

```c
typedef struct cudaEglFrame_st {
    union {
        cudaArray_t pArray[CUDA_EGL_MAX_PLANES];
        struct cudaPitchedPtr pPitch[CUDA_EGL_MAX_PLANES];
    } frame;
    cudaEglPlaneDesc planeDesc[CUDA_EGL_MAX_PLANES];
    unsigned int planeCount;
    cudaEglFrameType frameType;
    cudaEglColorFormat eglColorFormat;
} cudaEglFrame;
```

For `cudaEglFrame` of type `cudaEglFrameTypePitch`, the application may present sub-region of a memory allocation. In that case, `cudaPitchedPtr::ptr` will specify the start address of the sub-region in the allocation and `cudaEglPlaneDesc` will specify the dimensions of the sub-region.

See also:

`cudaEGLStreamProducerConnect`, `cudaEGLStreamProducerDisconnect`, `cudaEGLStreamProducerReturnFrame`, `cuEGLStreamProducerPresentFrame`

```c
__host__ cudaError_t cudaEGLStreamProducerReturnFrame(cudaEglStreamConnection *conn, cudaEglFrame *eglframe, cudaStream_t *pStream)
```

Return the CUDA eglFrame to the EGLStream last released by the consumer.

Parameters

- **conn**
  - Connection on which to present the CUDA array

- **eglframe**
  - CUDA EglStream Proucer Frame handle returned from the consumer over EglStream.

- **pStream**
  - CUDA stream on which to return the frame.

Returns

- `cudaSuccess`, `cudaErrorLaunchTimeout`, `cudaErrorInvalidValue`, `cudaErrorUnknown`
Description

This API can potentially return cudaErrorLaunchTimeout if the consumer has not returned a frame to EGL stream. If timeout is returned the application can retry.

See also:

cudaEGLStreamProducerConnect, cudaEGLStreamProducerDisconnect, cudaEGLStreamProducerPresentFrame, cuEGLStreamProducerReturnFrame

__host__cudaError_t cudaEventCreateFromEGLSync (cudaEvent_t *phEvent, EGLSyncKHR eglSync, unsigned int flags)

Creates an event from EGLSync object.

Parameters

**phEvent**
- Returns newly created event

**eglSync**
- Opaque handle to EGLSync object

**flags**
- Event creation flags

Returns

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorLaunchFailure, cudaErrorMemoryAllocation

Description

Creates an event *phEvent from an EGLSyncKHR eglSync with the flags specified via flags. Valid flags include:

- **cudaEventDefault**: Default event creation flag.
- **cudaEventBlockingSync**: Specifies that the created event should use blocking synchronization. A CPU thread that uses cudaEventSynchronize() to wait on an event created with this flag will block until the event has actually been completed.

cudaEventRecord and TimingData are not supported for events created from EGLSync.

The EGLSyncKHR is an opaque handle to an EGL sync object. typedef void* EGLSyncKHR

See also:
cudaEventQuery, cudaEventSynchronize, cudaEventDestroy

__host__ cudaError_t
cudaGraphicsEGLRegisterImage
(cudaGraphicsResource **pCudaResource,
EGLImageKHR image, unsigned int flags)

Registers an EGL image.

Parameters

pCudaResource
- Pointer to the returned object handle

image
- An EGLImageKHR image which can be used to create target resource.

flags
- Map flags

Returns
cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorInvalidValue, cudaErrorUnknown

Description

Registers the EGLImageKHR specified by image for access by CUDA. A handle to the
registered object is returned as pCudaResource. Additional Mapping/Unmapping is not
required for the registered resource and cudaGraphicsResourceGetMappedEglFrame can be
directly called on the pCudaResource.

The application will be responsible for synchronizing access to shared objects. The application
must ensure that any pending operation which access the objects have completed before
passing control to CUDA. This may be accomplished by issuing and waiting for glFinish
command on all GLcontexts (for OpenGL and likewise for other APIs). The application will be
also responsible for ensuring that any pending operation on the registered CUDA resource has
completed prior to executing subsequent commands in other APIs accessing the same memory
objects. This can be accomplished by calling cuCtxSynchronize or cuEventSynchronize
(preferably).

The surface’s intended usage is specified using flags, as follows:

- **cudaGraphicsRegisterFlagsNone**: Specifies no hints about how this resource will be used.
  It is therefore assumed that this resource will be read from and written to by CUDA. This is
  the default value.

- **cudaGraphicsRegisterFlagsReadOnly**: Specifies that CUDA will not write to this resource.
• `cudaGraphicsRegisterFlagsWriteDiscard`: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

The EGLImageKHR is an object which can be used to create EGLImage target resource. It is defined as a void pointer. typedef void* EGLImageKHR

See also:

```c
__host__ cudaError_t
cudaGraphicsResourceGetMappedEglFrame(
cudaEglFrame *eglFrame, cudaGraphicsResource_t resource,
unsigned int index, unsigned int mipLevel)
```

Get an eglFrame through which to access a registered EGL graphics resource.

**Parameters**

- `eglFrame` - Returned eglFrame.
- `resource` - Registered resource to access.
- `index` - Index for cubemap surfaces.
- `mipLevel` - Mipmap level for the subresource to access.

**Returns**

- `cudaSuccess`
- `cudaErrorInvalidValue`
- `cudaErrorUnknown`

**Description**

Returns in `eglFrame` an eglFrame pointer through which the registered graphics resource may be accessed. This API can only be called for EGL graphics resources.

The `cudaEglFrame` is defined as

```c
typedef struct cudaEglFrame_st {
    union {
        cudaArray_t pArray[CUDA_EGL_MAX_PLANES];
        struct cudaPitchedPtr pPitch[CUDA_EGL_MAX_PLANES];
    } frame;
    cudaEglPlaneDesc planeDesc[CUDA_EGL_MAX_PLANES];
    unsigned int planeCount;
    cudaEglFrameType frameType;
} cudaEglFrame_t;
```
6.24. Graphics Interoperability

This section describes the graphics interoperability functions of the CUDA runtime application programming interface.

__host__ cudaError_t cudaGraphicsMapResources
(int count, cudaGraphicsResource_t *resources, cudaStream_t stream)

Map graphics resources for access by CUDA.

Parameters

count
  - Number of resources to map

resources
  - Resources to map for CUDA

stream
  - Stream for synchronization

Returns

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description

Maps the count graphics resources in resources for access by CUDA.

The resources in resources may be accessed by CUDA until they are unmapped. The graphics API from which resources were registered should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.
This function provides the synchronization guarantee that any graphics calls issued before `cudaGraphicsMapResources()` will complete before any subsequent CUDA work issued in stream begins.

If `resources` contains any duplicate entries then `cudaErrorInvalidResourceHandle` is returned. If any of `resources` are presently mapped for access by CUDA then `cudaErrorUnknown` is returned.

**Note:**

- This function uses standard `default stream` semantics.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**


```c
__host__ cudaError_t cudaGraphicsResourceGetMappedMipmappedArray (cudaMipmappedArray_t *mipmappedArray, cudaGraphicsResource_t resource)
```

Get a mipmapped array through which to access a mapped graphics resource.

**Parameters**

- **mipmappedArray**
  - Returned mipmapped array through which `resource` may be accessed
- **resource**
  - Mapped resource to access

**Returns**

`cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown`
Description

Returns in `mipmappedArray` a mipmapped array through which the mapped graphics resource may be accessed. The value set in `mipmappedArray` may change every time that resource is mapped.

If `resource` is not a texture then it cannot be accessed via an array and `cudaErrorUnknown` is returned. If `resource` is not mapped then `cudaErrorUnknown` is returned.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**


```c
__host__ cudaError_t
cudaGraphicsResourceGetMappedPointer (void **devPtr, size_t *size, cudaGraphicsResource_t resource)
```

Get an device pointer through which to access a mapped graphics resource.

**Parameters**

- **devPtr**
  - Returned pointer through which `resource` may be accessed
- **size**
  - Returned size of the buffer accessible starting at `*devPtr`
- **resource**
  - Mapped resource to access

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`
Description

Returns in *devPtr a pointer through which the mapped graphics resource resource may be accessed. Returns in *size the size of the memory in bytes which may be accessed from that pointer. The value set in devPtr may change every time that resource is mapped.

If resource is not a buffer then it cannot be accessed via a pointer and cudaErrorUnknown is returned. If resource is not mapped then cudaErrorUnknown is returned. *

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

__host__cudaError_t
cudaGraphicsResourceSetMapFlags
(cudaGraphicsResource_t resource, unsigned int flags)

Set usage flags for mapping a graphics resource.

Parameters

- resource
  - Registered resource to set flags for

- flags
  - Parameters for resource mapping

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown,
Description

Set flags for mapping the graphics resource resource.

Changes to flags will take effect the next time resource is mapped. The flags argument may be any of the following:

- **cudaGraphicsMapFlagsNone**: Specifies no hints about how resource will be used. It is therefore assumed that CUDA may read from or write to resource.
- **cudaGraphicsMapFlagsReadOnly**: Specifies that CUDA will not write to resource.
- **cudaGraphicsMapFlagsWriteDiscard**: Specifies CUDA will not read from resource and will write over the entire contents of resource, so none of the data previously stored in resource will be preserved.

If resource is presently mapped for access by CUDA then **cudaErrorUnknown** is returned. If flags is not one of the above values then **cudaErrorInvalidValue** is returned.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return **cudaErrorInitializationError**, **cudaErrorInsufficientDriver** or **cudaErrorNoDevice** if this call tries to initialize internal CUDA RT state.
- Note that as specified by **cudaStreamAddCallback** no CUDA function may be called from callback. **cudaErrorNotPermitted** may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

**cudaGraphicsMapResources, cuGraphicsResourceSetMapFlags**
__host__ cudaError_t
cudaGraphicsSubResourceGetMappedArray
(cudaArray_t *array, cudaGraphicsResource_t
resource, unsigned int arrayIndex, unsigned int
mipLevel)

Get an array through which to access a subresource of a mapped graphics resource.

Parameters
array
- Returned array through which a subresource of resource may be accessed

resource
- Mapped resource to access

arrayIndex
- Array index for array textures or cubemap face index as defined by
cudaGraphicsCubeFace for cubemap textures for the subresource to access

mipLevel
- Mipmap level for the subresource to access

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description
Returns in *array an array through which the subresource of the mapped graphics resource
resource which corresponds to array index arrayIndex and mipmap level mipLevel may
be accessed. The value set in array may change every time that resource is mapped.

If resource is not a texture then it cannot be accessed via an array and cudaErrorUnknown
is returned. If arrayIndex is not a valid array index for resource then
cudaErrorInvalidValue is returned. If mipLevel is not a valid mipmap level for resource
then cudaErrorInvalidValue is returned. If resource is not mapped then cudaErrorUnknown
is returned.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError,
cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal
  CUDA RT state.
Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

__host__ cudaError_t cudaGraphicsUnmapResources (int count, cudaGraphicsResource_t *resources, cudaStream_t stream)
Unmap graphics resources.

Parameters

count
- Number of resources to unmapped

resources
- Resources to unmapped

stream
- Stream for synchronization

Returns
cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description
Unmaps the count graphics resources in resources.
Once unmapped, the resources in resources may not be accessed by CUDA until they are mapped again.
This function provides the synchronization guarantee that any CUDA work issued in stream before cudaGraphicsUnmapResources() will complete before any subsequently issued graphics work begins.
If resources contains any duplicate entries then cudaErrorInvalidResourceHandle is returned. If any of resources are not presently mapped for access by CUDA then cudaErrorUnknown is returned.

Note:

- This function uses standard default stream semantics.
- Note that this function may also return error codes from previous, asynchronous launches.
__host__ cudaError_t cudaGraphicsUnregisterResource (cudaGraphicsResource_t resource)

Unregisters a graphics resource for access by CUDA.

Parameters

resource
  - Resource to unregister

Returns

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Description

Unregisters the graphics resource resource so it is not accessible by CUDA unless registered again.

If resource is invalid then cudaErrorInvalidResourceHandle is returned.

Note:

- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
- Use of the handle after this call is undefined behavior.

See also:
cudaGraphicsMapResources, cuGraphicsUnmapResources
6.25. Texture Reference Management [DEPRECATED]

This section describes the low level texture reference management functions of the CUDA runtime application programming interface.

Some functions have overloaded C++ API template versions documented separately in the C++ API Routines module.

__host__cudaError_t cudaBindTexture (size_t *offset, const textureReference *texref, const void *devPtr, const cudaChannelFormatDesc *desc, size_t size)

Binds a memory area to a texture.

Parameters

offset
  - Offset in bytes
texref
  - Texture to bind
devPtr
  - Memory area on device
desc
  - Channel format
size
  - Size of the memory area pointed to by devPtr

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidTexture

Description

Depreciated

Binds size bytes of the memory area pointed to by devPtr to the texture reference texref. desc describes how the memory is interpreted when fetching values from the texture. Any memory previously bound to texref is unbound.
Since the hardware enforces an alignment requirement on texture base addresses, `cudaBindTexture()` returns in *offset* a byte offset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the `tex1Dfetch()` function. If the device memory pointer was returned from `cudaMalloc()`, the offset is guaranteed to be 0 and NULL may be passed as the *offset* parameter.

The total number of elements (or texels) in the linear address range cannot exceed `cudaDeviceProp::maxTexture1DLinear[0]`. The number of elements is computed as *(size / elementSize)*, where elementSize is determined from *desc*.

---

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**

- `cudaCreateChannelDesc [C API]`, `cudaGetChannelDesc`, `cudaGetTextureReference`,
- `cudaBindTexture [C++ API]`, `cudaBindTexture2D [C API]`, `cudaBindTextureToArray [C API]`,
- `cudaUnbindTexture [C API]`, `cudaGetTextureAlignmentOffset [C API]`, `cuTexRefSetAddress`,
- `cuTexRefSetAddressMode`, `cuTexRefSetFormat`, `cuTexRefSetFlags`, `cuTexRefSetBorderColor`

```c
__host__cudaError_t cudaBindTexture2D (size_t *offset, const textureReference *texref, const void *devPtr, const cudaChannelFormatDesc *desc, size_t width, size_t height, size_t pitch)
```

Binds a 2D memory area to a texture.

**Parameters**

- **offset**
  - Offset in bytes
- **texref**
  - Texture reference to bind
devPtr
- 2D memory area on device
desc
- Channel format
width
- Width in texel units
height
- Height in texel units
pitch
- Pitch in bytes

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidTexture

Description
Deprecated

Binds the 2D memory area pointed to by devPtr to the texture reference texref. The size of the area is constrained by width in texel units, height in texel units, and pitch in byte units. desc describes how the memory is interpreted when fetching values from the texture. Any memory previously bound to texref is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, cudaBindTexture2D() returns in *offset a byte offset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the tex2D() function. If the device memory pointer was returned from cudaMemcpy(), the offset is guaranteed to be 0 and NULL may be passed as the offset parameter.

width and height, which are specified in elements (or texels), cannot exceed cudaDeviceProp::maxTexture2DLinear[0] and cudaDeviceProp::maxTexture2DLinear[1] respectively. pitch, which is specified in bytes, cannot exceed cudaDeviceProp::maxTexture2DLinear[2].

The driver returns cudaErrorInvalidValue if pitch is not a multiple of cudaDeviceProp::texturePitchAlignment.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaCreateChannelDesc [ C API], cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture [ C API], cudaBindTexture2D [ C++ API], cudaBindTexture2D [ C++ API, inherited channel descriptor], cudaBindTextureToArray [ C API], cudaBindTextureToArray [ C API], cudaGetTextureAlignmentOffset [ C API], cuTexRefSetAddress2D, cuTexRefSetFormat, cuTexRefSetFlags, cuTexRefSetAddressMode, cuTexRefSetBorderColor

__host__cudaError_t cudaBindTextureToArray (const textureReference *texref, cudaArray_const_t array, const cudaChannelFormatDesc *desc)

Binds an array to a texture.

Parameters

texref
   - Texture to bind
array
   - Memory array on device
desc
   - Channel format

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidTexture

Description

Deprecated

Binds the CUDA array array to the texture reference texref.desc describes how the memory is interpreted when fetching values from the texture. Any CUDA array previously bound to texref is unbound.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaCreateChannelDesc [C API], cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture [C API], cudaBindTexture2D [C API], cudaBindTextureToArray [C++ API], cudaUnbindTexture [C API], cudaGetTextureAlignmentOffset [C API], cuTexRefSetArray, cuTexRefSetFormat, cuTexRefSetFlags, cuTexRefSetAddressMode, cuTexRefSetFilterMode, cuTexRefSetBorderColor, cuTexRefSetMaxAnisotropy

__host__ cudaError_t
cudaBindTextureToMipmappedArray
(const textureReference *texref,
cudaMipmappedArray_const_t mipmappedArray,
const cudaChannelFormatDesc *desc)
Binds a mipmapped array to a texture.

Parameters

texref
  - Texture to bind
mipmappedArray
  - Memory mipmapped array on device
desc
  - Channel format

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidTexture

Description

Deprecated

Binds the CUDA mipmapped array mipmappedArray to the texture reference texref. desc describes how the memory is interpreted when fetching values from the texture. Any CUDA mipmapped array previously bound to texref is unbound.

Note:
  - Note that this function may also return error codes from previous, asynchronous launches.
Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.

Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

- `cudaCreateChannelDesc (C API)`, `cudaGetChannelDesc`, `cudaGetTextureReference`, `cudaBindTexture (C API)`, `cudaBindTexture2D (C API)`, `cudaBindTextureToArray (C++ API)`, `cudaUnbindTexture (C API)`, `cudaGetTextureAlignmentOffset (C API)`, `cuTexRefSetMipmappedArray`, `cuTexRefSetMipmapFilterMode`, `cuTexRefSetMipmapLevelClamp`, `cuTexRefSetMipmapLevelBias`, `cuTexRefSetFormat`, `cuTexRefSetFlags`, `cuTexRefSetAddressMode`, `cuTexRefSetBorderColor`, `cuTexRefSetMaxAnisotropy`

```c
__host__ cudaError_t cudaGetTextureAlignmentOffset (size_t *offset, const textureReference *texref)
```

Get the alignment offset of a texture.

**Parameters**

- `offset`
  - Offset of texture reference in bytes

- `texref`
  - Texture to get offset of

**Returns**

- `cudaSuccess`
- `cudaErrorInvalidTexture`
- `cudaErrorInvalidTextureBinding`

**Description**

**Deprecated**

Returns in `*offset` the offset that was returned when texture reference `texref` was bound.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
- `cudaCreateChannelDesc [ C API]`
- `cudaGetChannelDesc`
- `cudaGetTextureReference`
- `cudaBindTexture [ C API]`
- `cudaBindTexture2D [ C API]`
- `cudaBindTextureToArray [ C API]`
- `cudaUnbindTexture [ C API]`
- `cudaGetTextureAlignmentOffset [ C++ API]`

```c
__host__cudaError_t cudaGetTextureReference
(const textureReference **texref, const void *symbol)
```

Get the texture reference associated with a symbol.

**Parameters**
- `texref`
  - Texture reference associated with symbol
- `symbol`
  - Texture to get reference for

**Returns**
- `cudaSuccess`
- `cudaErrorInvalidTexture`

**Description**
- **Deprecated**

Returns in `*texref` the structure associated to the texture reference defined by symbol `symbol`.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- Use of a string naming a variable as the `symbol` parameter was removed in CUDA 5.0.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.
See also:
cudaCreateChannelDesc [ C API], cudaGetChannelDesc, cudaGetTextureAlignmentOffset [ C API], cudaBindTexture [ C API], cudaBindTexture2D [ C API], cudaBindTextureToArray [ C API], cudaUnbindTexture [ C API], cuModuleGetTexRef


__host__cudaError_t cudaUnbindTexture (const textureReference *texref)

Unbinds a texture.

Parameters
texref
  - Texture to unbind

Returns
cudaSuccess, cudaErrorInvalidTexture

Description
Deprecated
Unbinds the texture bound to texref. If texref is not currently bound, no operation is performed.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaCreateChannelDesc [ C API], cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture [ C API], cudaBindTexture2D [ C API], cudaBindTextureToArray [ C API], cudaUnbindTexture [ C++ API], cudaGetTextureAlignmentOffset [ C API]
[DEPRECATED]

This section describes the low level surface reference management functions of the CUDA runtime application programming interface.

Some functions have overloaded C++ API template versions documented separately in the C++ API Routines module.

```c
__host__cudaError_t cudaBindSurfaceToArray (const surfaceReference *surfref, cudaArray_const_t array, const cudaChannelFormatDesc *desc)
```

Binds an array to a surface.

**Parameters**

- **surfref**
  - Surface to bind
- **array**
  - Memory array on device
- **desc**
  - Channel format

**Returns**

- `cudaSuccess`
- `cudaErrorInvalidValue`
- `cudaErrorInvalidSurface`

**Description**

**Deprecated**

Binds the CUDA array `array` to the surface reference `surfref`. `desc` describes how the memory is interpreted when fetching values from the surface. Any CUDA array previously bound to `surfref` is unbound.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
`cudaBindSurfaceToArray (C++ API)`, `cudaBindSurfaceToArray (C++ API, inherited channel descriptor)`, `cudaGetSurfaceReference`, `cuSurfRefSetArray`

__host__cudaError_t cudaGetSurfaceReference (const surfaceReference **surfref, const void *symbol)

Get the surface reference associated with a symbol.

Parameters
- `surfref` - Surface reference associated with symbol
- `symbol` - Surface to get reference for

Returns
- `cudaSuccess`, `cudaErrorInvalidSurface`

Description
- Deprecated

Returns in `*surfref` the structure associated to the surface reference defined by symbol `symbol`.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Use of a string naming a variable as the `symbol` parameter was removed in CUDA 5.0.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.
6.27. Texture Object Management

This section describes the low level texture object management functions of the CUDA runtime application programming interface. The texture object API is only supported on devices of compute capability 3.0 or higher.

`__host__cudaCreateChannelDesc (int x, int y, int z, int w, cudaChannelFormatKind f)`

Returns a channel descriptor using the specified format.

**Parameters**

- `x` - X component
- `y` - Y component
- `z` - Z component
- `w` - W component
- `f` - Channel format

**Returns**

Channel descriptor with format `f`

**Description**

Returns a channel descriptor with format `f` and number of bits of each component `x`, `y`, `z`, and `w`. The `cudaChannelFormatDesc` is defined as:

```c
struct cudaChannelFormatDesc {
    int x, y, z, w;
    enum cudaChannelFormatKind f;
};
```

where `cudaChannelFormatKind` is one of `cudaChannelFormatKindSigned`, `cudaChannelFormatKindUnsigned`, or `cudaChannelFormatKindFloat`.

**See also:**

`cudaBindSurfaceToArray [C API]`, `cuModuleGetSurfRef`
cudaCreateTextureObject (C++ API), cudaGetChannelDesc, cudaCreateTextureObject, cudaCreateSurfaceObject

__host__ cudaError_t cudaCreateTextureObject (cudaTextureObject_t *pTexObject, const cudaResourceDesc *pResDesc, const cudaTextureDesc *pTexDesc, const cudaResourceViewDesc *pResViewDesc)

Creates a texture object.

Parameters

pTexObject  
- Texture object to create

pResDesc  
- Resource descriptor

pTexDesc  
- Texture descriptor

pResViewDesc  
- Resource view descriptor

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Creates a texture object and returns it in pTexObject. pResDesc describes the data to texture from. pTexDesc describes how the data should be sampled. pResViewDesc is an optional argument that specifies an alternate format for the data described by pResDesc, and also describes the subresource region to restrict access to when texturing. pResViewDesc can only be specified if the type of resource is a CUDA array or a CUDA mipmapped array.

Texture objects are only supported on devices of compute capability 3.0 or higher. Additionally, a texture object is an opaque value, and, as such, should only be accessed through CUDA API calls.

The __host__ cudaError_t cudaCreateTextureObject (cudaTextureObject_t *pTexObject, const cudaResourceDesc *pResDesc, const cudaTextureDesc *pTexDesc, const cudaResourceViewDesc *pResViewDesc)

Creates a texture object.

Parameters

pTexObject  
- Texture object to create

pResDesc  
- Resource descriptor

pTexDesc  
- Texture descriptor

pResViewDesc  
- Resource view descriptor

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Creates a texture object and returns it in pTexObject. pResDesc describes the data to texture from. pTexDesc describes how the data should be sampled. pResViewDesc is an optional argument that specifies an alternate format for the data described by pResDesc, and also describes the subresource region to restrict access to when texturing. pResViewDesc can only be specified if the type of resource is a CUDA array or a CUDA mipmapped array.

Texture objects are only supported on devices of compute capability 3.0 or higher. Additionally, a texture object is an opaque value, and, as such, should only be accessed through CUDA API calls.

The __host__ cudaError_t cudaCreateTextureObject (cudaTextureObject_t *pTexObject, const cudaResourceDesc *pResDesc, const cudaTextureDesc *pTexDesc, const cudaResourceViewDesc *pResViewDesc)

Creates a texture object.

Parameters

pTexObject  
- Texture object to create

pResDesc  
- Resource descriptor

pTexDesc  
- Texture descriptor

pResViewDesc  
- Resource view descriptor

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Creates a texture object and returns it in pTexObject. pResDesc describes the data to texture from. pTexDesc describes how the data should be sampled. pResViewDesc is an optional argument that specifies an alternate format for the data described by pResDesc, and also describes the subresource region to restrict access to when texturing. pResViewDesc can only be specified if the type of resource is a CUDA array or a CUDA mipmapped array.

Texture objects are only supported on devices of compute capability 3.0 or higher. Additionally, a texture object is an opaque value, and, as such, should only be accessed through CUDA API calls.
where:

- `cudaResourceDesc::resType` specifies the type of resource to texture from.

```
CUresourceType is defined as:

```
        enum cudaResourceType {
            cudaResourceTypeArray = 0x00,
            cudaResourceTypeMipmappedArray = 0x01,
            cudaResourceTypeLinear = 0x02,
            cudaResourceTypePitch2D = 0x03
        };
```

If `cudaResourceDesc::resType` is set to `cudaResourceTypeArray`, `cudaResourceDesc::res::array::array` must be set to a valid CUDA array handle.

If `cudaResourceDesc::resType` is set to `cudaResourceTypeMipmappedArray`, `cudaResourceDesc::res::mipmap::mipmap` must be set to a valid CUDA mipmapped array handle and `cudaTextureDesc::normalizedCoords` must be set to true.

If `cudaResourceDesc::resType` is set to `cudaResourceTypeLinear`, `cudaResourceDesc::res::linear::devPtr` must be set to a valid device pointer, that is aligned to `cudaDeviceProp::textureAlignment`. `cudaResourceDesc::res::linear::desc` describes the format and the number of components per array element. `cudaResourceDesc::res::linear::sizeInBytes` specifies the size of the array in bytes. The total number of elements in the linear address range cannot exceed `cudaDeviceProp::maxTexture1DLinear`. The number of elements is computed as `(sizeInBytes / sizeof(desc))`.

If `cudaResourceDesc::resType` is set to `cudaResourceTypePitch2D`, `cudaResourceDesc::res::pitch2D::devPtr` must be set to a valid device pointer, that is aligned to `cudaDeviceProp::textureAlignment`. `cudaResourceDesc::res::pitch2D::desc` describes the format and the number of components per array element. `cudaResourceDesc::res::pitch2D::width` and `cudaResourceDesc::res::pitch2D::height`
specify the width and height of the array in elements, and cannot exceed `cudaDeviceProp::maxTexture2DLinear[0]` and `cudaDeviceProp::maxTexture2DLinear[1]` respectively. `cudaResourceDesc::res::pitch2D::pitchInBytes` specifies the pitch between two rows in bytes and has to be aligned to `cudaDeviceProp::texturePitchAlignment`. Pitch cannot exceed `cudaDeviceProp::maxTexture2DLinear[2]`.

The `cudaTextureDesc` struct is defined as

```c
struct cudaTextureDesc {
    enum cudaTextureAddressMode addressMode[3];
    enum cudaTextureFilterMode filterMode;
    enum cudaTextureReadMode readMode;
    int sRGB;
    float borderColor[4];
    int normalizedCoords;
    unsigned int maxAnisotropy;
    enum cudaTextureFilterMode mipmapFilterMode;
    float mipmapLevelBias;
    float minMipmapLevelClamp;
    float maxMipmapLevelClamp;
    int disableTrilinearOptimization;
};
```

where

- `cudaTextureDesc::addressMode` specifies the addressing mode for each dimension of the texture data. `cudaTextureAddressMode` is defined as:

```c
enum cudaTextureAddressMode {
    cudaAddressModeWrap = 0,
    cudaAddressModeClamp = 1,
    cudaAddressModeMirror = 2,
    cudaAddressModeBorder = 3
};
```

This is ignored if `cudaResourceDesc::resType` is `cudaResourceTypeLinear`. Also, if `cudaTextureDesc::normalizedCoords` is set to zero, `cudaAddressModeWrap` and `cudaAddressModeMirror` won’t be supported and will be switched to `cudaAddressModeClamp`.

- `cudaTextureDesc::filterMode` specifies the filtering mode to be used when fetching from the texture. `cudaTextureFilterMode` is defined as:

```c
enum cudaTextureFilterMode {
    cudaFilterModePoint = 0,
    cudaFilterModeLinear = 1
};
```

This is ignored if `cudaResourceDesc::resType` is `cudaResourceTypeLinear`.

- `cudaTextureDesc::readMode` specifies whether integer data should be converted to floating point or not. `cudaTextureReadMode` is defined as:

```c
enum cudaTextureReadMode {
    // Definition here
};
```
Note that this applies only to 8-bit and 16-bit integer formats. 32-bit integer format would not be promoted, regardless of whether or not this `cudaTextureDesc::readMode` is set `cudaReadModeNormalizedFloat` is specified.

- `cudaTextureDesc::sRGB` specifies whether sRGB to linear conversion should be performed during texture fetch.
- `cudaTextureDesc::borderColor` specifies the float values of color. where: `cudaTextureDesc::borderColor[0]` contains value of ‘R’, `cudaTextureDesc::borderColor[1]` contains value of ‘G’, `cudaTextureDesc::borderColor[2]` contains value of ‘B’, `cudaTextureDesc::borderColor[3]` contains value of ‘A’. Note that application using integer border color values will need to `reinterpret_cast` these values to float. The values are set only when the addressing mode specified by `cudaTextureDesc::addressMode` is `cudaAddressModeBorder`.
- `cudaTextureDesc::normalizedCoords` specifies whether the texture coordinates will be normalized or not.
- `cudaTextureDesc::maxAnisotropy` specifies the maximum anistropy ratio to be used when doing anisotropic filtering. This value will be clamped to the range [1,16].
- `cudaTextureDesc::mipmapFilterMode` specifies the filter mode when the calculated mipmap level lies between two defined mipmap levels.
- `cudaTextureDesc::mipmapLevelBias` specifies the offset to be applied to the calculated mipmap level.
- `cudaTextureDesc::minMipmapLevelClamp` specifies the lower end of the mipmap level range to clamp access to.
- `cudaTextureDesc::maxMipmapLevelClamp` specifies the upper end of the mipmap level range to clamp access to.
- `cudaTextureDesc::disableTrilinearOptimization` specifies whether the trilinear filtering optimizations will be disabled.

The `cudaResourceViewDesc` struct is defined as:

```c
struct cudaResourceViewDesc {
    enum cudaResourceViewFormat
        format;
    size_t width;
    size_t height;
    size_t depth;
    unsigned int firstMipmapLevel;
    unsigned int lastMipmapLevel;
    unsigned int firstLayer;
    unsigned int lastLayer;
};
```
where:

- **cudaResourceViewDesc::format** specifies how the data contained in the CUDA array or CUDA mipmapped array should be interpreted. Note that this can incur a change in size of the texture data. If the resource view format is a block compressed format, then the underlying CUDA array or CUDA mipmapped array has to have a 32-bit unsigned integer format with 2 or 4 channels, depending on the block compressed format. For ex., BC1 and BC4 require the underlying CUDA array to have a 32-bit unsigned int with 2 channels. The other BC formats require the underlying resource to have the same 32-bit unsigned int format but with 4 channels.

- **cudaResourceViewDesc::width** specifies the new width of the texture data. If the resource view format is a block compressed format, this value has to be 4 times the original width of the resource. For non block compressed formats, this value has to be equal to that of the original resource.

- **cudaResourceViewDesc::height** specifies the new height of the texture data. If the resource view format is a block compressed format, this value has to be 4 times the original height of the resource. For non block compressed formats, this value has to be equal to that of the original resource.

- **cudaResourceViewDesc::depth** specifies the new depth of the texture data. This value has to be equal to that of the original resource.

- **cudaResourceViewDesc::firstMipmapLevel** specifies the most detailed mipmap level. This will be the new mipmap level zero. For non-mipmapped resources, this value has to be zero. **cudaTextureDesc::minMipmapLevelClamp** and **cudaTextureDesc::maxMipmapLevelClamp** will be relative to this value. For ex., if the firstMipmapLevel is set to 2, and a minMipmapLevelClamp of 1.2 is specified, then the actual minimum mipmap level clamp will be 3.2.

- **cudaResourceViewDesc::lastMipmapLevel** specifies the least detailed mipmap level. For non-mipmapped resources, this value has to be zero.

- **cudaResourceViewDesc::firstLayer** specifies the first layer index for layered textures. This will be the new layer zero. For non-layered resources, this value has to be zero.

- **cudaResourceViewDesc::lastLayer** specifies the last layer index for layered textures. For non-layered resources, this value has to be zero.

**Note:**

- Note that this function may also return **cudaErrorInitializationError**, **cudaErrorInsufficientDriver** or **cudaErrorNoDevice** if this call tries to initialize internal CUDA RT state.
Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaDestroyTextureObject, cuTexObjectCreate

__host__ cudaError_t cudaDestroyTextureObject (cudaTextureObject_t texObject)
Destroys a texture object.

Parameters
	
texObject
- Texture object to destroy

Returns
cudaSuccess, cudaErrorInvalidValue

Description
Destroys the texture object specified by texObject.

Note:
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
- Use of the handle after this call is undefined behavior.

See also:
cudaCreateTextureObject, cuTexObjectDestroy
__host__ cudaError_t cudaGetChannelDesc(cudaChannelFormatDesc *desc, cudaArray_const_t array)

Get the channel descriptor of an array.

Parameters

- **desc**: Channel format
- **array**: Memory array on device

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Returns in *desc the channel descriptor of the CUDA array array.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaCreateChannelDesc [C API], cudaCreateTextureObject, cudaCreateSurfaceObject
__host__ cudaError_t
cudaGetTextureObjectResourceDesc
(cudaResourceDesc *pResDesc, cudaTextureObject_t texObject)

Returns a texture object’s resource descriptor.

Parameters
pResDesc
- Resource descriptor
texObject
- Texture object

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Returns the resource descriptor for the texture object specified by texObject.

Note:

- Note that this function may also return cudaErrorInitializationError,
cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal
CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called
from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a
diagnostic in such case.

See also:
cudaCreateTextureObject, cuTexObjectGetResourceDesc
__host__ cudaError_t
cudaGetTextureObjectResourceViewDesc
(cudaResourceViewDesc *pResViewDesc, cudaTextureObject_t texObject)

Returns a texture object’s resource view descriptor.

Parameters

pResViewDesc
- Resource view descriptor

texObject
- Texture object

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Returns the resource view descriptor for the texture object specified by texObject. If no resource view was specified, cudaErrorInvalidValue is returned.

Note:

- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaCreateTextureObject, cuTexObjectGetResourceViewDesc
__host__ cudaError_t
cudaGetTextureObjectTextureDesc (cudaTextureDesc
*pTexDesc, cudaTextureObject_t texObject)

Returns a texture object’s texture descriptor.

Parameters

pTexDesc
- Texture descriptor
texObject
- Texture object

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Returns the texture descriptor for the texture object specified by texObject.

Note:

- Note that this function may also return cudaErrorInitializationError,
cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaCreateTextureObject, cuTexObjectGetTextureDesc

6.28. Surface Object Management

This section describes the low level texture object management functions of the CUDA runtime application programming interface. The surface object API is only supported on devices of compute capability 3.0 or higher.
__host__ cudaError_t cudaCreateSurfaceObject (cudaSurfaceObject_t *pSurfObject, const cudaResourceDesc *pResDesc)

Creates a surface object.

Parameters

pSurfObject
  - Surface object to create

pResDesc
  - Resource descriptor

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidChannelDescriptor, cudaErrorInvalidResourceHandle

Description

Creates a surface object and returns it in pSurfObject. pResDesc describes the data to perform surface load/stores on. cudaResourceDesc::resType must be cudaResourceTypeArray and cudaResourceDesc::res::array::array must be set to a valid CUDA array handle.

Surface objects are only supported on devices of compute capability 3.0 or higher. Additionally, a surface object is an opaque value, and, as such, should only be accessed through CUDA API calls.

Note:

- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.

- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaDestroySurfaceObject, cuSurfObjectCreate
__host__ cudaError_t cudaDestroySurfaceObject (cudaSurfaceObject_t surfObject)

Destroys a surface object.

Parameters

surfObject
- Surface object to destroy

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Destroys the surface object specified by surfObject.

Note:

- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
- Use of the handle after this call is undefined behavior.

See also:

cudaCreateSurfaceObject, cuSurfObjectDestroy
__host__cudaError_t
cudaGetSurfaceObjectResourceDesc
cudaResourceDesc *pResDesc, cudaSurfaceObject_t surfObject)

Returns a surface object’s resource descriptor

Returns the resource descriptor for the
surface object specified by surfObject.

Parameters

pResDesc
- Resource descriptor

surfObject
- Surface object

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Note:

‣ Note that this function may also return cudaErrorInitializationError,
cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal
CUDA RT state.

‣ Note that as specified by cudaStreamAddCallback no CUDA function may be called
from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a
diagnostic in such case.

See also:
cudaCreateSurfaceObject, cuSurfObjectGetResourceDesc

6.29. Version Management
__host__cudaError_t cudaDriverGetVersion (int *driverVersion)

Returns the latest version of CUDA supported by the driver.

Parameters

driverVersion
  - Returns the CUDA driver version.

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Returns in *driverVersion the latest version of CUDA supported by the driver. The version is returned as [1000 major + 10 minor]. For example, CUDA 9.2 would be represented by 9020. If no driver is installed, then 0 is returned as the driver version.

This function automatically returns cudaErrorInvalidValue if driverVersion is NULL.

Note:

▶ Note that this function may also return error codes from previous, asynchronous launches.
▶ Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
▶ Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaRuntimeGetVersion, cuDriverGetVersion

__host__device__cudaError_t
cudaRuntimeGetVersion (int *runtimeVersion)

Returns the CUDA Runtime version.

Parameters

runtimeVersion
  - Returns the CUDA Runtime version.
Returns
cudaSuccess, cudaErrorInvalidValue

Description
Returns in `*runtimeVersion` the version number of the current CUDA Runtime instance. The version is returned as (1000 major + 10 minor). For example, CUDA 9.2 would be represented by 9020.

This function automatically returns `cudaErrorInvalidValue` if the `runtimeVersion` argument is NULL.

Note:
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaDriverGetVersion, cuDriverGetVersion

6.30. Graph Management

This section describes the graph management functions of CUDA runtime application programming interface.

```c
__host__ cudaError_t
cudaDeviceGetGraphMemAttribute (int device, cudaGraphMemAttributeType attr, void *value)
```

Query asynchronous allocation attributes related to graphs.

Parameters
device
  - Specifies the scope of the query
attr
  - attribute to get
value
- retrieved value

Returns
cudaSuccess, cudaErrorInvalidDevice

Description
Valid attributes are:

- `cudaGraphMemAttrUsedMemCurrent`: Amount of memory, in bytes, currently associated with graphs
- `cudaGraphMemAttrUsedMemHigh`: High watermark of memory, in bytes, associated with graphs since the last time it was reset. High watermark can only be reset to zero.
- `cudaGraphMemAttrReservedMemCurrent`: Amount of memory, in bytes, currently allocated for use by the CUDA graphs asynchronous allocator.
- `cudaGraphMemAttrReservedMemHigh`: High watermark of memory, in bytes, currently allocated for use by the CUDA graphs asynchronous allocator.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphAddMemAllocNode, cudaGraphAddMemFreeNode, cudaDeviceGraphMemTrim, cudaDeviceSetGraphMemAttribute, cudaMallocAsync, cudaFreeAsync,
__host__ cudaError_t cudaDeviceGraphMemTrim (int device)
Free unused memory that was cached on the specified device for use with graphs back to the OS.

Parameters
device
- The device for which cached memory should be freed.

Returns
cudaSuccess, cudaErrorInvalidValue

Description
Blocks which are not in use by a graph that is either currently executing or scheduled to execute are freed back to the operating system.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphAddMemAllocNode, cudaGraphAddMemFreeNode,
cudaDeviceGetGraphMemAttribute, cudaDeviceSetGraphMemAttribute, cudaMallocAsync,
cudaFreeAsync.
__host__cudaError_t

cudaDeviceSetGraphMemAttribute (int device,
cudaGraphMemAttributeType attr, void *value)

Set asynchronous allocation attributes related to graphs.

Parameters

device
  - Specifies the scope of the query
attr
  - attribute to get
value
  - pointer to value to set

Returns
cudaSuccess, cudaErrorInvalidDevice

Description

Valid attributes are:

- **cudaGraphMemAttrUsedMemHigh**: High watermark of memory, in bytes, associated with graphs since the last time it was reset. High watermark can only be reset to zero.

- **cudaGraphMemAttrReservedMemHigh**: High watermark of memory, in bytes, currently allocated for use by the CUDA graphs asynchronous allocator.

Note:

- Graph objects are not threadsafe. [More here](#).
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphAddMemAllocNode, cudaGraphAddMemFreeNode, cudaDeviceGraphMemTrim, cudaDeviceGetGraphMemAttribute, cudaMallocAsync, cudaFreeAsync.
__host__ cudaError_t cudaGraphAddChildGraphNode (cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, cudaGraph_t childGraph)

Creates a child graph node and adds it to a graph.

**Parameters**

- **pGraphNode**
  - Returns newly created node
- **graph**
  - Graph to which to add the node
- **pDependencies**
  - Dependencies of the node
- **numDependencies**
  - Number of dependencies
- **childGraph**
  - The graph to clone into this node

**Returns**

cudaSuccess, cudaErrorInvalidValue

**Description**

Creates a new node which executes an embedded graph, and adds it to graph with numDependencies dependencies specified via pDependencies. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. pDependencies may not have any duplicate entries. A handle to the new node will be returned in pGraphNode.

If hGraph contains allocation or free nodes, this call will return an error.

The node executes an embedded child graph. The child graph is cloned in this call.

**Note:**

- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphChildGraphNodeGetGraph, cudaGraphCreate, cudaGraphDestroyNode, cudaGraphAddEmptyNode, cudaGraphAddKernelNode, cudaGraphAddHostNode, cudaGraphAddMemcpyNode, cudaGraphAddMemsetNode, cudaGraphClone

__host__cudaError_t cudaGraphAddDependencies(cudaGraph_t graph, const cudaGraphNode_t *from, const cudaGraphNode_t *to, size_t numDependencies)

Adds dependency edges to a graph.

Parameters

graph
  - Graph to which dependencies are added
from
  - Array of nodes that provide the dependencies
to
  - Array of dependent nodes
numDependencies
  - Number of dependencies to be added

Returns
cudaSuccess, cudaErrorInvalidValue

Description

The number of dependencies to be added is defined by numDependencies. Elements in pFrom and pTo at corresponding indices define a dependency. Each node in pFrom and pTo must belong to graph.

If numDependencies is 0, elements in pFrom and pTo will be ignored. Specifying an existing dependency will return an error.

Note:

Graph objects are not threadsafe. More here.
Note that this function may also return error codes from previous, asynchronous launches.

Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.

Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

- `cudaGraphRemoveDependencies`
- `cudaGraphGetEdges`
- `cudaGraphNodeGetDependencies`
- `cudaGraphNodeGetDependentNodes`

```c
__host__ cudaError_t cudaGraphAddEmptyNode(
  cudaGraphNode_t *pGraphNode, cudaGraph_t graph,
  const cudaGraphNode_t *pDependencies, size_t numDependencies)
```

Creates an empty node and adds it to a graph.

**Parameters**

- `pGraphNode`
  - Returns newly created node
- `graph`
  - Graph to which to add the node
- `pDependencies`
  - Dependencies of the node
- `numDependencies`
  - Number of dependencies

**Returns**

- `cudaSuccess`
- `cudaErrorInvalidValue`

**Description**

Creates a new node which performs no operation, and adds it to `graph` with `numDependencies` dependencies specified via `pDependencies`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`. 
An empty node performs no operation during execution, but can be used for transitive ordering. For example, a phased execution graph with 2 groups of n nodes with a barrier between them can be represented using an empty node and 2*n dependency edges, rather than no empty node and n^2 dependency edges.

Note:

- Graph objects are not threadsafe. [More here.]
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaGraphCreate, cudaGraphDestroyNode, cudaGraphAddChildGraphNode, cudaGraphAddKernelNode, cudaGraphAddHostNode, cudaGraphAddMemcpyNode, cudaGraphAddMemsetNode

__host__ cudaError_t

`cudaGraphAddEventRecordNode (cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, cudaEvent_t event)`

Creates an event record node and adds it to a graph.

Parameters

- `pGraphNode` graph
- `pDependencies` numDependencies
- `event` Event for the node

Returns

cudaSuccess, cudaErrorInvalidValue
Description

Creates a new event record node and adds it to hGraph with numDependencies dependencies specified via dependencies and event specified in event. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. dependencies may not have any duplicate entries. A handle to the new node will be returned in phGraphNode.

Each launch of the graph will record event to capture execution of the node’s dependencies. These nodes may not be used in loops or conditionals.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaMemcpyInitializationError, cudaMemcpyInsufficientDriver or cudaMemcpyNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpyCallback no CUDA function may be called from callback. cudaMemcpyNotAllowed may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphAddEventWaitNode, cudaEventRecordWithFlags, cudaStreamWaitEvent, cudaGraphCreate, cudaGraphDestroyNode, cudaGraphAddChildGraphNode, cudaGraphAddEmptyNode, cudaGraphAddKernelNode, cudaGraphAddMemcpyNode, cudaGraphAddMemsetNode.
__host__ cudaError_t cudaGraphAddEventWaitNode (cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, cudaEvent_t event)

Creates an event wait node and adds it to a graph.

Parameters
- **pGraphNode**: graph node
- **graph**: graph
- **pDependencies**: dependencies
- **numDependencies**: Number of dependencies
- **event**: Event for the node

Returns
- **cudaSuccess**, **cudaErrorInvalidValue**

Description
Creates a new event wait node and adds it to hGraph with numDependencies dependencies specified via dependencies and event specified in event. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. dependencies may not have any duplicate entries. A handle to the new node will be returned in pGraphNode.

The graph node will wait for all work captured in event. See **cuEventRecord()** for details on what is captured by an event. The synchronization will be performed efficiently on the device when applicable. event may be from a different context or device than the launch stream.

These nodes may not be used in loops or conditionals.

Note:
- Graph objects are not threadsafe. [More here](#).
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
- `cudaGraphAddEventRecordNode`
- `cudaEventRecordWithFlags`
- `cudaStreamWaitEvent`
- `cudaGraphCreate`
- `cudaGraphDestroyNode`
- `cudaGraphAddChildGraphNode`
- `cudaGraphAddEmptyNode`
- `cudaGraphAddKernelNode`
- `cudaGraphAddMemcpyNode`
- `cudaGraphAddMemsetNode`
- `__host__cudaError_t cudaGraphAddExternalSemaphoresSignalNode (cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, const cudaExternalSemaphoreSignalNodeParams *nodeParams)`

Creates an external semaphore signal node and adds it to a graph.

**Parameters**

- `pGraphNode`
  - Returns newly created node
- `graph`
  - Graph to which to add the node
- `pDependencies`
  - Dependencies of the node
- `numDependencies`
  - Number of dependencies
- `nodeParams`
  - Parameters for the node

**Returns**

- `cudaSuccess`, `cudaErrorInvalidValue`

**Description**

Creates a new external semaphore signal node and adds it to `graph` with `numDependencies` dependencies specified via `dependencies` and arguments specified in `nodeParams`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of
the graph. Dependencies may not have any duplicate entries. A handle to the new node will be returned in pGraphNode.

Performs a signal operation on a set of externally allocated semaphore objects when the node is launched. The operation(s) will occur after all of the node’s dependencies have completed.

**Note:**

- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**

- `cudaGraphExternalSemaphoresSignalNodeGetParams`
- `cudaGraphExternalSemaphoresSignalNodeSetParams`
- `cudaGraphExecExternalSemaphoresSignalNodeSetParams`
- `cudaGraphAddExternalSemaphoresWaitNode`, `cudaImportExternalSemaphore`
- `cudaSignalExternalSemaphoresAsync`, `cudaWaitExternalSemaphoresAsync`
- `cudaGraphCreate`, `cudaGraphDestroyNode`, `cudaGraphAddEventRecordNode`
- `cudaGraphAddEventWaitNode`, `cudaGraphAddChildGraphNode`, `cudaGraphAddEmptyNode`
- `cudaGraphAddKernelNode`, `cudaGraphAddMemcpyNode`, `cudaGraphAddMemsetNode`
__host__ cudaError_t
cudaGraphAddExternalSemaphoresWaitNode
cudaGraphNode_t *pGraphNode,
cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies,
const cudaExternalSemaphoreWaitNodeParams *nodeParams)

Creates an external semaphore wait node and adds it to a graph.

Parameters

- **pGraphNode**
  - Returns newly created node

- **graph**
  - Graph to which to add the node

- **pDependencies**
  - Dependencies of the node

- **numDependencies**
  - Number of dependencies

- **nodeParams**
  - Parameters for the node

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Creates a new external semaphore wait node and adds it to graph with numDependencies dependencies specified via dependencies and arguments specified in nodeParams. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. dependencies may not have any duplicate entries. A handle to the new node will be returned in pGraphNode.

Performs a wait operation on a set of externally allocated semaphore objects when the node is launched. The node’s dependencies will not be launched until the wait operation has completed.

Note:

- Graph objects are not threadsafe. More here.
Note that this function may also return error codes from previous, asynchronous launches.

Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.

Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
- `cudaGraphExternalSemaphoresWaitNodeGetParams`,
- `cudaGraphExternalSemaphoresWaitNodeSetParams`,
- `cudaGraphExecExternalSemaphoresWaitNodeSetParams`,
- `cudaGraphAddExternalSemaphoresSignalNode`,
- `cudaImportExternalSemaphore`,
- `cudaSignalExternalSemaphoresAsync`,
- `cudaWaitExternalSemaphoresAsync`,
- `cudaGraphCreate`,
- `cudaGraphDestroyNode`,
- `cudaGraphAddEventRecordNode`,
- `cudaGraphAddEventWaitNode`,
- `cudaGraphAddChildGraphNode`,
- `cudaGraphAddEmptyNode`,
- `cudaGraphAddKernelNode`,
- `cudaGraphAddMemcpyNode`,
- `cudaGraphAddMemsetNode`.

```c
__host__ cudaError_t cudaGraphAddHostNode(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, const cudaHostNodeParams *pNodeParams)
```

Creates a host execution node and adds it to a graph.

**Parameters**

- **pGraphNode**
  - Returns newly created node
- **graph**
  - Graph to which to add the node
- **pDependencies**
  - Dependencies of the node
- **numDependencies**
  - Number of dependencies
- **pNodeParams**
  - Parameters for the host node
Returns

cudaSuccess, cudaErrorNotSupported, cudaErrorInvalidValue

Description

Creates a new CPU execution node and adds it to graph with numDependencies dependencies specified via pDependencies and arguments specified in pNodeParams. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. pDependencies may not have any duplicate entries. A handle to the new node will be returned in pGraphNode.

When the graph is launched, the node will invoke the specified CPU function. Host nodes are not supported under MPS with pre-Volta GPUs.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaLaunchHostFunc, cudaGraphHostNodeGetParams, cudaGraphHostNodeSetParams, cudaGraphCreate, cudaGraphDestroyNode, cudaGraphAddChildGraphNode, cudaGraphAddEmptyNode, cudaGraphAddKernelNode, cudaGraphAddMemcpyNode, cudaGraphAddMemsetNode
__host__ cudaError_t cudaGraphAddKernelNode (cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, const cudaKernelNodeParams *pNodeParams)

Creates a kernel execution node and adds it to a graph.

Parameters

pGraphNode
- Returns newly created node

graph
- Graph to which to add the node

pDependencies
- Dependencies of the node

numDependencies
- Number of dependencies

pNodeParams
- Parameters for the GPU execution node

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDeviceFunction

Description

Creates a new kernel execution node and adds it to graph with numDependencies dependencies specified via pDependencies and arguments specified in pNodeParams. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. pDependencies may not have any duplicate entries. A handle to the new node will be returned in pGraphNode.

The cudaKernelNodeParams structure is defined as:

```c
struct cudaKernelNodeParams
{
    void* func;
    dim3 gridDim;
    dim3 blockDim;
    unsigned int sharedMemBytes;
    void **kernelParams;
    void **extra;
};
```
When the graph is launched, the node will invoke kernel `func` on a `(gridDim.x \times gridDim.y \times gridDim.z)` grid of blocks. Each block contains `(blockDim.x \times blockDim.y \times blockDim.z)` threads.

`sharedMem` sets the amount of dynamic shared memory that will be available to each thread block.

Kernel parameters to `func` can be specified in one of two ways:

1) Kernel parameters can be specified via `kernelParams`. If the kernel has `N` parameters, then `kernelParams` needs to be an array of `N` pointers. Each pointer, from `kernelParams[0]` to `kernelParams[N-1]`, points to the region of memory from which the actual parameter will be copied. The number of kernel parameters and their offsets and sizes do not need to be specified as that information is retrieved directly from the kernel’s image.

2) Kernel parameters can also be packaged by the application into a single buffer that is passed in via `extra`. This places the burden on the application of knowing each kernel parameter’s size and alignment/padding within the buffer. The `extra` parameter exists to allow this function to take additional less commonly used arguments. `extra` specifies a list of names of extra settings and their corresponding values. Each extra setting name is immediately followed by the corresponding value. The list must be terminated with either `NULL` or `CU_LAUNCH_PARAM_END`.

- **CU_LAUNCH_PARAM_END**, which indicates the end of the `extra` array;
- **CU_LAUNCH_PARAM_BUFFER_POINTER**, which specifies that the next value in `extra` will be a pointer to a buffer containing all the kernel parameters for launching kernel `func`;
- **CU_LAUNCH_PARAM_BUFFER_SIZE**, which specifies that the next value in `extra` will be a pointer to a `size_t` containing the size of the buffer specified with **CU_LAUNCH_PARAM_BUFFER_POINTER**;

The error `cudaErrorInvalidValue` will be returned if kernel parameters are specified with both `kernelParams` and `extra` (i.e. both `kernelParams` and `extra` are non-NULL).

The `kernelParams` or `extra` array, as well as the argument values it points to, are copied during this call.

**Note:**

Kernels launched using graphs must not use texture and surface references. Reading or writing through any texture or surface reference is undefined behavior. This restriction does not apply to texture and surface objects.

**Note:**

- Graph objects are not threadsafe. [More here.](#)
Note that this function may also return error codes from previous, asynchronous launches.

Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.

Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaLaunchKernel, cudaGraphKernelNodeGetParams, cudaGraphKernelNodeSetParams, cudaGraphCreate, cudaGraphDestroyNode, cudaGraphAddChildGraphNode, cudaGraphAddEmptyNode, cudaGraphAddHostNode, cudaGraphAddMemcpyNode, cudaGraphAddMemsetNode

__host__ cudaError_t cudaGraphAddMemAllocNode(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, cudaMemAllocNodeParams *nodeParams)

Creates an allocation node and adds it to a graph.

Parameters

pGraphNode
- Returns newly created node

graph
- Graph to which to add the node

pDependencies
- Dependencies of the node

numDependencies
- Number of dependencies

nodeParams
- Parameters for the node

Returns
cudaSuccess, cudaErrorCudartUnloading, cudaErrorInitializationError, cudaErrorNotSupported, cudaErrorInvalidValue, cudaErrorOutOfMemory
Description

Creates a new allocation node and adds it to graph with numDependencies dependencies specified via pDependencies and arguments specified in nodeParams. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. pDependencies may not have any duplicate entries. A handle to the new node will be returned in pGraphNode.

When `cudaGraphAddMemAllocNode` creates an allocation node, it returns the address of the allocation in If the allocation is freed in the same graph, by creating a free node using `cudaGraphAddMemFreeNode`, the allocation can be accessed by nodes ordered after the allocation node but before the free node. These allocations cannot be freed outside the owning graph, and they can only be freed once in the owning graph.

If the allocation is not freed in the same graph, then it can be accessed not only by nodes in the graph which are ordered after the allocation node, but also by stream operations ordered after the graph’s execution but before the allocation is freed.

Allocations which are not freed in the same graph can be freed by:

- passing the allocation to `cudaMemFreeAsync` or `cudaMemFree`;
- launching a graph with a free node for that allocation; or
- specifying `cudaGraphInstantiateFlagAutoFreeOnLaunch` during instantiation, which makes each launch behave as though it called `cudaMemFreeAsync` for every unfreed allocation.

It is not possible to free an allocation in both the owning graph and another graph. If the allocation is freed in the same graph, a free node cannot be added to another graph. If the allocation is freed in another graph, a free node can no longer be added to the owning graph.

The following restrictions apply to graphs which contain allocation and/or memory free nodes:

- Nodes and edges of the graph cannot be deleted.
- The graph cannot be used in a child node.
- Only one instantiation of the graph may exist at any point in time.
- The graph cannot be cloned.

Note:

- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaGraphAddMemcpyNode

cudaGraphNode_t* pGraphNode, cudaGraph_t graph,
const cudaGraphNode_t* pDependencies, size_t numDependencies,
const cudaMemcpy3DParms* pCopyParams)

Creates a memcpy node and adds it to a graph.

Parameters

pGraphNode
- Returns newly created node

graph
- Graph to which to add the node

pDependencies
- Dependencies of the node

numDependencies
- Number of dependencies

pCopyParams
- Parameters for the memory copy

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Creates a new memcpy node and adds it to graph with numDependencies dependencies specified via pDependencies. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. pDependencies may not have any duplicate entries. A handle to the new node will be returned in pGraphNode.

When the graph is launched, the node will perform the memcpy described by pCopyParams. See cudaMemcpy3D() for a description of the structure and its restrictions.
Memcpy nodes have some additional restrictions with regards to managed memory, if the system contains at least one device which has a zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`.

Note:
- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
- `cudaMemcpy3D`, `cudaGraphAddMemcpyNodeToSymbol`, `cudaGraphAddMemcpyNodeFromSymbol`, `cudaGraphAddMemcpyNode1D`, `cudaGraphMemcpyNodeGetParams`, `cudaGraphMemcpyNodeSetParams`, `cudaGraphCreate`, `cudaGraphDestroyNode`, `cudaGraphAddChildGraphNode`, `cudaGraphAddEmptyNode`, `cudaGraphAddKernelNode`, `cudaGraphAddHostNode`, `cudaGraphAddMemsetNode`

```c
__host__cudaError_t cudaGraphAddMemcpyNode1D(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, void *dst, const void *src, size_t count, cudaMemcpyKind kind)
```

Creates a 1D memcpy node and adds it to a graph.

**Parameters**
- **pGraphNode**: Returns newly created node
- **graph**: Graph to which to add the node
- **pDependencies**: Dependencies of the node
- **numDependencies**: Number of dependencies
dst
- Destination memory address

src
- Source memory address

count
- Size in bytes to copy

kind
- Type of transfer

Returns
cudaSuccess, cudaErrorInvalidValue

Description
Creates a new 1D memcpy node and adds it to graph with numDependencies dependencies specified via pDependencies. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. pDependencies may not have any duplicate entries. A handle to the new node will be returned in pGraphNode.

When the graph is launched, the node will copy count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing. Launching a memcpy node with dst and src pointers that do not match the direction of the copy results in an undefined behavior.

Memcpy nodes have some additional restrictions with regards to managed memory, if the system contains at least one device which has a zero value for the device attribute cudaDevAttrConcurrentManagedAccess.

Note:
▶ Graph objects are not threadsafe. More here.
▶ Note that this function may also return error codes from previous, asynchronous launches.
▶ Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
▶ Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
See also:
cudamemcpy, cudagraphadmmemcpynode, cudagraphmemcpynodegetparams,
cudagraphmemcpynodesetparams, cudagraphmemcpynodesetparams1d,  
cudagraphcreate, cudagraphdestroynode, cudagraphaddchildgraphnode,  
cudagraphaddemptynode, cudagraphaddkernelnode, cudagraphaddhostnode,  
cudagraphaddmemsetnode

__host__cudaError_t

cudagraphaddmemcpynodefromsymbol
(cudagraphnode_t *pgraphnode, cudagraph_t graph,  
const cudagraphnode_t *pdependencies, size_t numdependencies, void *dst, const void *symbol,  
size_t count, size_t offset, cudamemcpykind kind)

Creates a memcpy node to copy from a symbol on the device and adds it to a graph.

Parameters

pgraphnode
- Returns newly created node

graph
- Graph to which to add the node

pdependencies
- Dependencies of the node

numdependencies
- Number of dependencies

dst
- Destination memory address

symbol
- Device symbol address

count
- Size in bytes to copy

offset
- Offset from start of symbol in bytes

kind
- Type of transfer

Returns

cudasuccess, cudaerrorinvalidvalue
Description

Creates a new memcpy node to copy from symbol and adds it to graph with numDependencies dependencies specified via pDependencies. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. pDependencies may not have any duplicate entries. A handle to the new node will be returned in pGraphNode.

When the graph is launched, the node will copy count bytes from the memory area pointed to by offset bytes from the start of symbol to the memory area pointed to by dst. The memory areas may not overlap. symbol is a variable that resides in global or constant memory space. kind can be either cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing.

Memcpy nodes have some additional restrictions with regards to managed memory, if the system contains at least one device which has a zero value for the device attribute cudaDevAttrConcurrentManagedAccess.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaMemcpyInitializationError, cudaMemcpyInsufficientDriver or cudaMemcpyNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpyFromSymbol if no CUDA function may be called from callback. cudaMemcpyNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaMemcpyFromSymbol, cudaGraphAddMemcpyNode,
cudaGraphAddMemcpyNodeToSymbol, cudaGraphMemcpyNodeGetParams,
cudaGraphMemcpyNodeSetParams, cudaGraphMemcpyNodeSetParamsFromSymbol,
cudaGraphMemcpyNodeSetParamsToSymbol, cudaGraphCreate, cudaGraphDestroyNode,
cudaGraphAddChildGraphNode, cudaGraphAddEmptyNode, cudaGraphAddKernelNode,
cudaGraphAddHostNode, cudaGraphAddMemsetNode
__host__ cudaError_t
cudaGraphAddMemcpyNodeToSymbol
cudaGraphNode_t *pGraphNode, cudaGraph_t graph,
const cudaGraphNode_t *pDependencies, size_t
numDependencies, const void *symbol, const void
*src, size_t count, size_t offset, cudaMemcpyKind
kind)
Creates a memcpy node to copy to a symbol on the device and adds it to a graph.

Parameters

pGraphNode  - Returns newly created node
graph  - Graph to which to add the node
pDependencies  - Dependencies of the node
numDependencies  - Number of dependencies
symbol  - Device symbol address
src  - Source memory address
count  - Size in bytes to copy
offset  - Offset from start of symbol in bytes
kind  - Type of transfer

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Creates a new memcpy node to copy to symbol and adds it to graph with
numDependencies dependencies specified via pDependencies. It is possible for
numDependencies to be 0, in which case the node will be placed at the root of the graph.
pDependencies may not have any duplicate entries. A handle to the new node will be
returned in pGraphNode.
When the graph is launched, the node will copy `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing.

Memcpy nodes have some additional restrictions with regards to managed memory, if the system contains at least one device which has a zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`.

**Note:**
- Graph objects are not threadsafe. [More here](#).
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**
- `cudaMemcpyToSymbol`, `cudaGraphAddMemcpyNode`, `cudaGraphAddMemcpyNodeFromSymbol`, `cudaGraphMemcpyNodeGetParams`, `cudaGraphMemcpyNodeSetParams`, `cudaGraphMemcpyNodeSetParamsToSymbol`, `cudaGraphMemcpyNodeSetParamsFromSymbol`, `cudaGraphCreate`, `cudaGraphDestroyNode`, `cudaGraphAddChildGraphNode`, `cudaGraphAddEmptyNode`, `cudaGraphAddKernelNode`, `cudaGraphAddHostNode`, `cudaGraphAddMemsetNode`
__host__ cudaError_t cudaGraphAddMemFreeNode (cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, void *dptr)

Creates a memory free node and adds it to a graph.

Parameters

**pGraphNode**
- Returns newly created node

**graph**
- Graph to which to add the node

**pDependencies**
- Dependencies of the node

**numDependencies**
- Number of dependencies

**dptr**
- Address of memory to free

Returns

cudaSuccess, cudaErrorCudartUnloading, cudaErrorInitializationError, cudaErrorNotSupported, cudaErrorInvalidValue, cudaErrorOutOfMemory

Description

Creates a new memory free node and adds it to graph with numDependencies dependencies specified via pDependencies and address specified in dptr. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. pDependencies may not have any duplicate entries. A handle to the new node will be returned in pGraphNode.

cudaGraphAddMemFreeNode will return cudaErrorInvalidValue if the user attempts to free:

- an allocation twice in the same graph.
- an address that was not returned by an allocation node.
- an invalid address.

The following restrictions apply to graphs which contain allocation and/or memory free nodes:

- Nodes and edges of the graph cannot be deleted.
- The graph cannot be used in a child node.
- Only one instantiation of the graph may exist at any point in time.
The graph cannot be cloned.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphAddMemAllocNode, cudaGraphMemFreeNodeGetParams, cudaDeviceGraphMemTrim, cudaDeviceGetGraphMemAttribute, cudaDeviceSetGraphMemAttribute, cudaMallocAsync, cudaFreeAsync, cudaGraphCreate, cudaGraphDestroyNode, cudaGraphAddChildGraphNode, cudaGraphAddEmptyNode, cudaGraphAddEventRecordNode, cudaGraphAddEventWaitNode, cudaGraphAddExternalSemaphoresSignalNode, cudaGraphAddExternalSemaphoresWaitNode, cudaGraphAddKernelNode, cudaGraphAddMemcpyNode, cudaGraphAddMemsetNode

__host__ cudaError_t cudaGraphAddMemsetNode (cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, const cudaMemsetParams *pMemsetParams)

Creates a memset node and adds it to a graph.

Parameters

- **pGraphNode**
  - Returns newly created node
- **graph**
  - Graph to which to add the node
- **pDependencies**
  - Dependencies of the node
- **numDependencies**
  - Number of dependencies
- **pMemsetParams**
  - Parameters for the memory set

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevice
Description

Creates a new memset node and adds it to graph with numDependencies dependencies specified via pDependencies. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. pDependencies may not have any duplicate entries. A handle to the new node will be returned in pGraphNode.

The element size must be 1, 2, or 4 bytes. When the graph is launched, the node will perform the memset described by pMemsetParams.

Note:

➤ Graph objects are not threadsafe. [More here.]
➤ Note that this function may also return error codes from previous, asynchronous launches.
➤ Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
➤ Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaMemset2D, cudaGraphMemsetNodeGetParams, cudaGraphMemsetNodeSetParams, cudaGraphCreate, cudaGraphDestroyNode, cudaGraphAddChildGraphNode, cudaGraphAddEmptyNode, cudaGraphAddKernelNode, cudaGraphAddHostNode, cudaGraphAddMemcpyNode

__host__ cudaError_t
cudaGraphChildGraphNodeGetGraph
(cudaGraphNode_t node, cudaGraph_t *pGraph)

Gets a handle to the embedded graph of a child graph node.

Parameters

node
  - Node to get the embedded graph for
pGraph
  - Location to store a handle to the graph
Returns
cudaSuccess, cudaErrorInvalidValue

Description
Gets a handle to the embedded graph in a child graph node. This call does not clone the graph. Changes to the graph will be reflected in the node, and the node retains ownership of the graph.

Allocation and free nodes cannot be added to the returned graph. Attempting to do so will return an error.

Note:
¬ Graph objects are not threadsafe. More here.
¬ Note that this function may also return error codes from previous, asynchronous launches.
¬ Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
¬ Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphAddChildGraphNode, cudaGraphNodeFindInClone

__host__cudaError_t cudaGraphClone (cudaGraph_t *pGraphClone, cudaGraph_t originalGraph)
Clones a graph.

Parameters
pGraphClone
- Returns newly created cloned graph
originalGraph
- Graph to clone

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation
Description

This function creates a copy of `originalGraph` and returns it in `pGraphClone`. All parameters are copied into the cloned graph. The original graph may be modified after this call without affecting the clone.

Child graph nodes in the original graph are recursively copied into the clone.

Note:

- Graph objects are not threadsafe. [More here.]
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaGraphCreate, cudaGraphNodeFindInClone

```cpp
__host__ cudaError_t cudaGraphCreate (cudaGraph_t *pGraph, unsigned int flags)
```

Creates a graph.

Parameters

- **pGraph**
  - Returns newly created graph
- **flags**
  - Graph creation flags, must be 0

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation

Description

Creates an empty graph, which is returned via `pGraph`. 
Note:

- Graph objects are not thread safe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphAddChildGraphNode, cudaGraphAddEmptyNode, cudaGraphAddKernelNode, cudaGraphAddHostNode, cudaGraphAddMemcpyNode, cudaGraphAddMemsetNode, cudaGraphInstantiate, cudaGraphDestroy, cudaGraphGetNodes, cudaGraphGetRootNodes, cudaGraphGetEdges, cudaGraphClone

#include <cudaError_t.h>

__host__ cudaError_t cudaGraphDebugDotPrint (cudaGraph_t graph, const char *path, unsigned int flags)

Write a DOT file describing graph structure.

Parameters

graph
  - The graph to create a DOT file from
path
  - The path to write the DOT file to
flags
  - Flags from cudaGraphDebugDotFlags for specifying which additional node information to write

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorOperatingSystem

Description

Using the provided graph, write to path a DOT formatted description of the graph. By default this includes the graph topology, node types, node id, kernel names and memcpy direction.
Flags can be specified to write more detailed information about each node type such as parameter values, kernel attributes, node and function handles.

```c
__host__ cudaError_t cudaGraphDestroy (cudaGraph_t graph)
```

Destroys a graph.

**Parameters**

- **graph**
  - Graph to destroy

**Returns**

- **cudaSuccess**
- **cudaErrorInvalidValue**

**Description**

Destroys the graph specified by `graph`, as well as all of its nodes.

**Note:**

- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.
- Use of the handle after this call is undefined behavior.

**See also:**

- [cudaGraphCreate](#)
__host__ cudaError_t cudaGraphDestroyNode (cudaGraphNode_t node)
Remove a node from the graph.

Parameters
node
- Node to remove

Returns
cudaSuccess, cudaErrorInvalidValue

Description
Removes node from its graph. This operation also severs any dependencies of other nodes on node and vice versa.

Dependencies cannot be removed from graphs which contain allocation or free nodes. Any attempt to do so will return an error.

Note:
- Graph objects are not thread safe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
- Use of the handle after this call is undefined behavior.

See also:
cudaGraphAddChildGraphNode, cudaGraphAddEmptyNode, cudaGraphAddKernelNode, cudaGraphAddHostNode, cudaGraphAddMemcpyNode, cudaGraphAddMemsetNode
__host__ cudaError_t
.cudaGraphEventRecordNodeGetEvent
(cudaGraphNode_t node, cudaEvent_t *event_out)

Returns the event associated with an event record node.

Parameters

node

- Pointer to return the event

event_out

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Returns the event of event record node hNode in event_out.

Note:
- Graph objects are not threadsafe. [More here.]
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphAddEventRecordNode, cudaGraphEventRecordNodeSetEvent, cudaGraphEventWaitNodeGetEvent, cudaEventRecordWithFlags, cudaStreamWaitEvent
__host__ cudaError_t
cudaGraphEventRecordNodeSetEvent
cudaGraphNode_t node, cudaEvent_t event)
Sets an event record node’s event.

Parameters
node
event
- Event to use

Returns
cudaSuccess, cudaErrorInvalidValue

Description
Sets the event of event record node hNode to event.

Note:
- Graph objects are not threadsafe. [More here](#).
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphAddEventRecordNode, cudaGraphEventRecordNodeGetEvent,
cudaGraphEventWaitNodeSetEvent, cudaEventRecordWithFlags, cudaStreamWaitEvent
__host__ cudaError_t
cudaGraphEventWaitNodeGetEvent (
cudaGraphNode_t node, cudaEvent_t *event_out)

Returns the event associated with an event wait node.

Parameters
  node
  event_out
    - Pointer to return the event

Returns
cudaSuccess, cudaErrorInvalidValue

Description
Returns the event of event wait node hNode in event_out.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphAddEventWaitNode, cudaGraphEventWaitNodeSetEvent,
cudaGraphNodeGetEvent, cudaEventRecordWithFlags, cudaStreamWaitEvent
__host__ cudaError_t
cudaGraphEventWaitNodeSetEvent
(cudaGraphNode_t node, cudaEvent_t event)
Sets an event wait node’s event.

Parameters
node
event
- Event to use

Returns
cudaSuccess, cudaErrorInvalidValue

Description
Sets the event of event wait node hNode to event.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphAddEventWaitNode, cudaGraphEventWaitNodeGetEvent, cudaGraphEventRecordNodeSetEvent, cudaEventRecordWithFlags, cudaStreamWaitEvent
__host__ cudaError_t
cudaGraphExecChildGraphNodeSetParams
  (cudaGraphExec_t hGraphExec, cudaGraphNode_t node, cudaGraph_t childGraph)

Updates node parameters in the child graph node in the given graphExec.

Parameters

  hGraphExec
    - The executable graph in which to set the specified node
  node
    - Host node from the graph which was used to instantiate graphExec
  childGraph
    - The graph supplying the updated parameters

Returns
cudaSuccess, cudaErrorInvalidValue,

Description

Updates the work represented by node in hGraphExec as though the nodes contained in
node's graph had the parameters contained in childGraph's nodes at instantiation. node
must remain in the graph which was used to instantiate hGraphExec. Changed edges to and
from node are ignored.

The modifications only affect future launches of hGraphExec. Already enqueued or running
launches of hGraphExec are not affected by this call. node is also not modified by this call.

The topology of childGraph, as well as the node insertion order, must match that of the
graph contained in node. See cudaGraphExecUpdate() for a list of restrictions on what can
be updated in an instantiated graph. The update is recursive, so child graph nodes contained
within the top level child graph will also be updated.

Note:

  ▶ Graph objects are not threadsafe. More here.
  ▶ Note that this function may also return error codes from previous, asynchronous launches.
  ▶ Note that this function may also return cudaErrorInitializationError,
cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal
CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
`cudaGraphAddChildGraphNode`, `cudaGraphChildGraphNodeGetGraph`, `cudaGraphExecKernelNodeSetParams`, `cudaGraphExecMemcpyNodeSetParams`, `cudaGraphExecMemsetNodeSetParams`, `cudaGraphExecHostNodeSetParams`, `cudaGraphExecEventRecordNodeSetEvent`, `cudaGraphExecEventWaitNodeSetEvent`, `cudaGraphExecExternalSemaphoresSignalNodeSetParams`, `cudaGraphExecExternalSemaphoresWaitNodeSetParams`, `cudaGraphExecUpdate`, `cudaGraphInstantiate`

```c
__host__ cudaError_t cudaGraphExecDestroy (cudaGraphExec_t graphExec)
```

Destroys an executable graph.

**Parameters**

**graphExec**
- Executable graph to destroy

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`

**Description**

Destroys the executable graph specified by `graphExec`.

---

**Note:**

- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.
- Use of the handle after this call is undefined behavior.
__host__ cuError_t

cudaGraphExecEventRecordNodeSetEvent

(cudaGraphExec_t hGraphExec, cudaGraphNode_t hNode, cudaEvent_t event)

Sets the event for an event record node in the given graphExec.

Parameters

hGraphExec
- The executable graph in which to set the specified node

hNode
- Event record node from the graph from which graphExec was instantiated

event
- Updated event to use

Returns
cuSuccess, cuErrorInvalidValue,

Description

Sets the event of an event record node in an executable graph hGraphExec. The node is identified by the corresponding node hNode in the non-executable graph, from which the executable graph was instantiated.

The modifications only affect future launches of hGraphExec. Already enqueued or running launches of hGraphExec are not affected by this call. hNode is also not modified by this call.

Note:

- Graph objects are not thread safe. More here.
- Note that this function may also return cuError codes from previous, asynchronous launches.
- Note that this function may also return cuErrorInitializationError, cuErrorInsufficientDriver or cuErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cuErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
__host__ cudaError_t

cudaGraphExecEventWaitNodeSetEvent
cudaGraphExec_t hGraphExec, cudaGraphNode_t hNode, cudaEvent_t event)

Sets the event for an event wait node in the given graphExec.

Parameters

hGraphExec
- The executable graph in which to set the specified node

hNode
- Event wait node from the graph from which graphExec was instantiated

event
- Updated event to use

Returns
cudaSuccess, cudaErrorInvalidValue.

Description

Sets the event of an event wait node in an executable graph hGraphExec. The node is identified by the corresponding node hNode in the non-executable graph, from which the executable graph was instantiated.

The modifications only affect future launches of hGraphExec. Already enqueued or running launches of hGraphExec are not affected by this call. hNode is also not modified by this call.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.

Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

`cudaGraphAddEventWaitNode`, `cudaGraphEventWaitNodeGetEvent`,
`cudaGraphEventRecordNodeSetEvent`, `cudaEventRecordWithFlags`, `cudaStreamWaitEvent`,
`cudaGraphExecKernelNodeSetParams`, `cudaGraphExecMemcpyNodeSetParams`,
`cudaGraphExecMemsetNodeSetParams`, `cudaGraphExecHostNodeSetParams`,
`cudaGraphExecChildGraphNodeSetParams`, `cudaGraphExecEventRecordNodeSetEvent`,
`cudaGraphExecExternalSemaphoresSignalNodeSetParams`,
`cudaGraphExecExternalSemaphoresWaitNodeSetParams`, `cudaGraphExecUpdate`,
`cudaGraphInstantiate`

```c
__host__ cudaError_t
cudaGraphExecExternalSemaphoresSignalNodeSetParams(
cudaGraphExec_t hGraphExec,
cudaGraphNode_t hNode, const
cudaExternalSemaphoreSignalNodeParams *nodeParams)
```

Sets the parameters for an external semaphore signal node in the given graphExec.

**Parameters**

- **hGraphExec**
  - The executable graph in which to set the specified node

- **hNode**
  - semaphore signal node from the graph from which graphExec was instantiated

- **nodeParams**
  - Updated Parameters to set

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`,

```c
__host__ cudaError_t
cudaGraphExecExternalSemaphoresSignalNodeSetParams(
cudaGraphExec_t hGraphExec,
cudaGraphNode_t hNode, const
cudaExternalSemaphoreSignalNodeParams *nodeParams)
```
Description

Sets the parameters of an external semaphore signal node in an executable graph \(\text{hGraphExec}\). The node is identified by the corresponding node \(\text{hNode}\) in the non-executable graph, from which the executable graph was instantiated.

\(\text{hNode}\) must not have been removed from the original graph.

The modifications only affect future launches of \(\text{hGraphExec}\). Already enqueued or running launches of \(\text{hGraphExec}\) are not affected by this call. \(\text{hNode}\) is also not modified by this call.

Changing \(\text{nodeParams->numExtSems}\) is not supported.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return \texttt{cudaErrorInitializationError}, \texttt{cudaErrorInsufficientDriver} or \texttt{cudaErrorNoDevice} if this call tries to initialize internal CUDA RT state.
- Note that as specified by \texttt{cudaStreamAddCallback} no CUDA function may be called from callback. \texttt{cudaErrorNotPermitted} may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

\texttt{cudaGraphAddExternalSemaphoresSignalNode}, \texttt{cudaImportExternalSemaphore}, \texttt{cudaSignalExternalSemaphoresAsync}, \texttt{cudaWaitExternalSemaphoresAsync}, \texttt{cudaGraphExecKernelNodeSetParams}, \texttt{cudaGraphExecMemcpyNodeSetParams}, \texttt{cudaGraphExecMemsetNodeSetParams}, \texttt{cudaGraphExecHostNodeSetParams}, \texttt{cudaGraphExecChildGraphNodeSetParams}, \texttt{cudaGraphExecEventRecordNodeSetEvent}, \texttt{cudaGraphExecEventWaitNodeSetEvent}, \texttt{cudaGraphExecExternalSemaphoresWaitNodeSetParams}, \texttt{cudaGraphExecUpdate}, \texttt{cudaGraphInstantiate}
__host__ cudaError_t
cudaGraphExecExternalSemaphoresWaitNodeSetParams
(cudaGraphExec_t hGraphExec, cudaGraphNode_t hNode, const
cudaExternalSemaphoreWaitNodeParams *nodeParams)

Sets the parameters for an external semaphore wait node in the given graphExec.

Parameters

hGraphExec
   - The executable graph in which to set the specified node
hNode
   - semaphore wait node from the graph from which graphExec was instantiated
nodeParams
   - Updated Parameters to set

Returns

cudaSuccess, cudaErrorInvalidValue,

Description

Sets the parameters of an external semaphore wait node in an executable graph 
hGraphExec. The node is identified by the corresponding node hNode in the non-executable graph, from which the executable graph was instantiated.

hNode must not have been removed from the original graph.

The modifications only affect future launches of hGraphExec. Already enqueued or running launches of hGraphExec are not affected by this call. hNode is also not modified by this call.

Changing nodeParams->numExtSems is not supported.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

- `cudaGraphAddExternalSemaphoresWaitNode`
- `cudaImportExternalSemaphore`
- `cudaSignalExternalSemaphoresAsync`
- `cudaWaitExternalSemaphoresAsync`
- `cudaGraphExecKernelNodeSetParams`
- `cudaGraphExecMemcpyNodeSetParams`
- `cudaGraphExecMemsetNodeSetParams`
- `cudaGraphExecHostNodeSetParams`
- `cudaGraphExecChildGraphNodeSetParams`
- `cudaGraphExecMemcpyNodeSetParams`
- `cudaGraphExecMemsetNodeSetParams`
- `cudaGraphExecHostNodeSetParams`
- `cudaGraphExecChildGraphNodeSetParams`
- `cudaGraphExecEventRecordNodeSetEvent`
- `cudaGraphExecEventWaitNodeSetEvent`
- `cudaGraphExecExternalSemaphoresSignalNodeSetParams`
- `cudaGraphExecUpdate`
- `cudaGraphInstantiate`

```c
__host__ cudaError_t
cudaGraphExecHostNodeSetParams
cudaGraphExec_t hGraphExec, cudaGraphNode_t node, const cudaHostNodeParams *pNodeParams)
```

Sets the parameters for a host node in the given graphExec.

**Parameters**

- **hGraphExec**
  - The executable graph in which to set the specified node
- **node**
  - Host node from the graph which was used to instantiate graphExec
- **pNodeParams**
  - Updated Parameters to set

**Returns**

- `cudaSuccess`
- `cudaErrorInvalidValue`

**Description**

Updates the work represented by `node` in `hGraphExec` as though `node` had contained `pNodeParams` at instantiation. `node` must remain in the graph which was used to instantiate `hGraphExec`. Changed edges to and from `node` are ignored.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `node` is also not modified by this call.
Note:

- Graph objects are not thread-safe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaGraphAddHostNode, cudaGraphHostNodeSetParams,
cudaGraphExecKernelNodeSetParams, cudaGraphExecMemcpyNodeSetParams,
cudaGraphExecMemsetNodeSetParams, cudaGraphExecChildGraphNodeSetParams,
cudaGraphExecEventRecordNodeSetEvent, cudaGraphExecEventWaitNodeSetEvent,
cudaGraphExecExternalSemaphoresSignalNodeSetParams,
cudaGraphExecExternalSemaphoresWaitNodeSetParams, cudaGraphExecUpdate,
cudaGraphInstantiate

__host__cudaError_t
cudaGraphExecKernelNodeSetParams
(cudaGraphExec_t hGraphExec, cudaGraphNode_t node, const cudaKernelNodeParams *pNodeParams)
Sets the parameters for a kernel node in the given graphExec.

Parameters

hGraphExec
- The executable graph in which to set the specified node
node
- kernel node from the graph from which graphExec was instantiated
pNodeParams
- Updated Parameters to set

Returns

cudaSuccess, cudaErrorInvalidValue,
Description

Sets the parameters of a kernel node in an executable graph `hGraphExec`. The node is identified by the corresponding `node` in the non-executable graph, from which the executable graph was instantiated.

`node` must not have been removed from the original graph. The `func` field of `nodeParams` cannot be modified and must match the original value. All other values can be modified.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `node` is also not modified by this call.

Note:

- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

- [cudaGraphAddKernelNode](#), [cudaGraphKernelNodeSetParams](#), [cudaGraphExecMemcpyNodeSetParams](#), [cudaGraphExecMemsetNodeSetParams](#), [cudaGraphExecHostNodeSetParams](#), [cudaGraphExecChildGraphNodeSetParams](#), [cudaGraphExecEventRecordNodeSetEvent](#), [cudaGraphExecEventWaitNodeSetEvent](#), [cudaGraphExecExternalSemaphoresSignalNodeSetParams](#), [cudaGraphExecExternalSemaphoresWaitNodeSetParams](#), [cudaGraphExecUpdate](#), [cudaGraphInstantiate](#)
__host__ cudaError_t
cudaGraphExecMemcpyNodeSetParams
(cudaGraphExec_t hGraphExec, cudaGraphNode_t
node, const cudaMemcpy3DParms *pNodeParams)

Sets the parameters for a memcpy node in the given graphExec.

Parameters

hGraphExec
  - The executable graph in which to set the specified node
node
  - Memcpy node from the graph which was used to instantiate graphExec
pNodeParams
  - Updated Parameters to set

Returns

cudaSuccess, cudaErrorInvalidValue.

Description

Updates the work represented by node in hGraphExec as though node had contained
pNodeParams at instantiation. node must remain in the graph which was used to instantiate
hGraphExec. Changed edges to and from node are ignored.

The source and destination memory in pNodeParams must be allocated from the same
contexts as the original source and destination memory. Both the instantiation-time memory
operands and the memory operands in pNodeParams must be 1-dimensional. Zero-length
operations are not supported.

The modifications only affect future launches of hGraphExec. Already enqueued or running
launches of hGraphExec are not affected by this call. node is also not modified by this call.

Returns cudaErrorInvalidValue if the memory operands’ mappings changed or either the
original or new memory operands are multidimensional.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
- `cudaGraphAddMemcpyNode`
- `cudaGraphMemcpyNodeSetParams`
- `cudaGraphExecMemcpyNodeSetParamsToSymbol`
- `cudaGraphExecMemcpyNodeSetParamsFromSymbol`
- `cudaGraphExecMemcpyNodeSetParams1D`
- `cudaGraphExecKernelNodeSetParams`
- `cudaGraphExecMemsetNodeSetParams`
- `cudaGraphExecHostNodeSetParams`
- `cudaGraphExecChildGraphNodeSetParams`
- `cudaGraphExecEventRecordNodeSetEvent`
- `cudaGraphExecEventWaitNodeSetEvent`
- `cudaGraphExecExternalSemaphoresSignalNodeSetParams`
- `cudaGraphExecExternalSemaphoresWaitNodeSetParams`
- `cudaGraphExecUpdate`
- `cudaGraphInstantiate`

```c
__host__cudaError_t
cudaGraphExecMemcpyNodeSetParams1D
cudaGraphExec_t hGraphExec, cudaGraphNode_t	node, void *dst, const void *src, size_t count,
cudaMemcpyKind kind)
```

Sets the parameters for a memcpy node in the given graphExec to perform a 1-dimensional copy.

**Parameters**

- `hGraphExec`
  - The executable graph in which to set the specified node
- `node`
  - Memcpy node from the graph which was used to instantiate graphExec
- `dst`
  - Destination memory address
- `src`
  - Source memory address
- `count`
  - Size in bytes to copy
- `kind`
  - Type of transfer

**Returns**

- `cudaSuccess`
- `cudaErrorInvalidValue`
Description

Updates the work represented by `node` in `hGraphExec` as though `node` had contained the given params at instantiation. `node` must remain in the graph which was used to instantiate `hGraphExec`. Changed edges to and from `node` are ignored.

`src` and `dst` must be allocated from the same contexts as the original source and destination memory. The instantiation-time memory operands must be 1-dimensional. Zero-length operations are not supported.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `node` is also not modified by this call.

Returns `cudaErrorInvalidValue` if the memory operands’ mappings changed or the original memory operands are multidimensional.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

- `cudaGraphAddMemcpyNode`, `cudaGraphAddMemcpyNode1D`,
- `cudaGraphNodeMemcpyNodeSetParams`, `cudaGraphNodeMemcpyNodeSetParams1D`,
- `cudaGraphExecMemcpyNodeSetParams`, `cudaGraphExecKernelNodeSetParams`,
- `cudaGraphExecMemsetNodeSetParams`, `cudaGraphExecHostNodeSetParams`,
- `cudaGraphExecChildGraphNodeSetParams`, `cudaGraphExecEventRecordNodeSetEvent`,
- `cudaGraphExecEventWaitNodeSetEvent`,
- `cudaGraphExecExternalSemaphoresSignalNodeSetParams`,
- `cudaGraphExecExternalSemaphoresWaitNodeSetParams`,
- `cudaGraphExecUpdate`,
- `cudaGraphInstantiate`
__host__ cudaError_t
cudaGraphExecMemcpyNodeSetParamsFromSymbol(
cudaGraphExec_t hGraphExec, cudaGraphNode_t node, void *dst, const void *symbol, size_t count,
size_t offset, cudaMemcpyKind kind)

Sets the parameters for a memcpy node in the given graphExec to copy from a symbol on the device.

Parameters

hGraphExec
- The executable graph in which to set the specified node

node
- Memcpy node from the graph which was used to instantiate graphExec

dst
- Destination memory address

symbol
- Device symbol address

count
- Size in bytes to copy

offset
- Offset from start of symbol in bytes

kind
- Type of transfer

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Updates the work represented by node in hGraphExec as though node had contained the given params at instantiation. node must remain in the graph which was used to instantiate hGraphExec. Changed edges to and from node are ignored.

symbol and dst must be allocated from the same contexts as the original source and destination memory. The instantiation-time memory operands must be 1-dimensional. Zero-length operations are not supported.

The modifications only affect future launches of hGraphExec. Already enqueued or running launches of hGraphExec are not affected by this call. node is also not modified by this call.

Returns cudaErrorInvalidValue if the memory operands’ mappings changed or the original memory operands are multidimensional.
Note:

- Graph objects are not threadsafe. [More here](#).
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

- `cudaGraphAddMemcpyNode`, `cudaGraphAddMemcpyNodeFromSymbol`, `cudaGraphMemcpyNodeSetParams`, `cudaGraphMemcpyNodeSetParamsFromSymbol`, `cudaGraphExecMemcpyNodeSetParams`, `cudaGraphExecMemcpyNodeSetParamsToSymbol`, `cudaGraphExecKernelNodeSetParams`, `cudaGraphExecMemsetNodeSetParams`, `cudaGraphExecHostNodeSetParams`, `cudaGraphExecChildGraphNodeSetParams`, `cudaGraphExecEventRecordNodeSetEvent`, `cudaGraphExecEventWaitNodeSetEvent`, `cudaGraphExecExternalSemaphoresSignalNodeSetParams`, `cudaGraphExecExternalSemaphoresWaitNodeSetParams`, `cudaGraphExecUpdate`, `cudaGraphInstantiate`

```c
__host__ cudaError_t
cudaGraphExecMemcpyNodeSetParamsToSymbol(
    cudaGraphExec_t hGraphExec, cudaGraphNode_t node,
    const void *symbol, const void *src, size_t count,
    size_t offset, cudaMemcpyKind kind)
```

Sets the parameters for a memcpy node in the given graphExec to copy to a symbol on the device.

**Parameters**

- **hGraphExec**
  - The executable graph in which to set the specified node
- **node**
  - Memcpy node from the graph which was used to instantiate graphExec
- **symbol**
  - Device symbol address
src
- Source memory address

count
- Size in bytes to copy

offset
- Offset from start of symbol in bytes

kind
- Type of transfer

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Updates the work represented by node in hGraphExec as though node had contained the given params at instantiation. node must remain in the graph which was used to instantiate hGraphExec. Changed edges to and from node are ignored.

src and symbol must be allocated from the same contexts as the original source and destination memory. The instantiation-time memory operands must be 1-dimensional. Zero-length operations are not supported.

The modifications only affect future launches of hGraphExec. Already enqueued or running launches of hGraphExec are not affected by this call. node is also not modified by this call.

Returns cudaErrorInvalidValue if the memory operands’ mappings changed or the original memory operands are multidimensional.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaGraphAddMemcpyNode, cudaGraphAddMemcpyNodeToSymbol, cudaGraphMemcpyNodeSetParams, cudaGraphMemcpyNodeSetParamsToSymbol, cudaGraphExecMemcpyNodeSetParams.
cudaGraphExecMemcpyNodeSetParamsFromSymbol,
cudaGraphExecKernelNodeSetParams, cudaGraphExecMemsetNodeSetParams,
cudaGraphExecHostNodeSetParams, cudaGraphExecChildGraphNodeSetParams,
cudaGraphExecEventRecordNodeSetEvent, cudaGraphExecEventWaitNodeSetEvent,
cudaGraphExecExternalSemaphoresSignalNodeSetParams,
cudaGraphExecExternalSemaphoresWaitNodeSetParams, cudaGraphExecUpdate,
cudaGraphInstantiate

__host__cudaError_t
cudaGraphExecMemsetNodeSetParams
(cudaGraphExec_t hGraphExec, cudaGraphNode_t node, const cudaMemsetParams *pNodeParams)
Sets the parameters for a memset node in the given graphExec.

Parameters

hGraphExec
- The executable graph in which to set the specified node
node
- Memset node from the graph which was used to instantiate graphExec
pNodeParams
- Updated Parameters to set

Returns

cudaSuccess, cudaErrorInvalidValue,

Description

Updates the work represented by node in hGraphExec as though node had contained
pNodeParams at instantiation. node must remain in the graph which was used to instantiate
hGraphExec. Changed edges to and from node are ignored.

The destination memory in pNodeParams must be allocated from the same context as the
original destination memory. Both the instantiation-time memory operand and the memory
operand in pNodeParams must be 1-dimensional. Zero-length operations are not supported.

The modifications only affect future launches of hGraphExec. Already enqueued or running
launches of hGraphExec are not affected by this call. node is also not modified by this call.

Returns cudaErrorInvalidValue if the memory operand’s mappings changed or either the
original or new memory operand are multidimensional.
Note:

- Graph objects are not thread safe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

- `cudaGraphAddMemsetNode`, `cudaGraphMemsetNodeSetParams`,
- `cudaGraphExecKernelNodeSetParams`, `cudaGraphExecMemcpyNodeSetParams`,
- `cudaGraphExecHostNodeSetParams`, `cudaGraphExecChildGraphNodeSetParams`,
- `cudaGraphExecEventRecordNodeSetEvent`, `cudaGraphExecEventWaitNodeSetEvent`,
- `cudaGraphExecExternalSemaphoresSignalNodeSetParams`,
- `cudaGraphExecExternalSemaphoresWaitNodeSetParams`, `cudaGraphExecUpdate`,
- `cudaGraphInstantiate`

```c
__host__ cudaError_t cudaGraphExecUpdate
(HWND迄今 hGraphExec, cudaGraph_t hGraph, cudaGraphNode_t *hErrorNode_out, cudaGraphExecUpdateResult *updateResult_out)
```

Check whether an executable graph can be updated with a graph and perform the update if possible.

Parameters

- **hGraphExec**
  - The instantiated graph to be updated
- **hGraph**
  - The graph containing the updated parameters
- **hErrorNode_out**
  - The node which caused the permissibility check to forbid the update, if any
- **updateResult_out**
  - Whether the graph update was permitted. If was forbidden, the reason why
Returns

cudaSuccess, cudaErrorGraphExecUpdateFailure.

Description

Updates the node parameters in the instantiated graph specified by hGraphExec with the node parameters in a topologically identical graph specified by hGraph.

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 CDP.

- **Memset and memcpy 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 contexts as the original source/destination memory.
  - Only 1D memsets can be changed.

- **Additional memcpy node restrictions:**
  - Changing either the source or destination memory type (i.e. CU_MEMORYTYPE DEVICE, CU_MEMORYTYPE ARRAY, etc.) is not supported.

Note: The API may add further restrictions in future releases. The return code should always be checked.

cudaGraphExecUpdate sets updateResult_out to cudaGraphExecUpdateErrorTopologyChanged under the following conditions:

- The count of nodes directly in hGraphExec and hGraph differ, in which case hErrorNode_out is NULL.
- A node is deleted in hGraph but not not its pair from hGraphExec, in which case hErrorNode_out is NULL.
- A node is deleted in hGraphExec but not its pair from hGraph, in which case hErrorNode_out is the pairless node from hGraph.
- The dependent nodes of a pair differ, in which case hErrorNode_out is the node from hGraph.

cudaGraphExecUpdate sets updateResult_out to:

- cudaGraphExecUpdateError if passed an invalid value.
- cudaGraphExecUpdateErrorTopologyChanged if the graph topology changed

- cudaGraphExecUpdateErrorNodeTypeChanged if the type of a node changed, in which case hErrorNode_out is set to the node from hGraph.

- cudaGraphExecUpdateErrorFunctionChanged if the function of a kernel node changed (CUDA driver < 11.2)

- cudaGraphExecUpdateErrorUnsupportedFunctionChange if the func field of a kernel changed in an unsupported way (see note above), in which case hErrorNode_out is set to the node from hGraph

- cudaGraphExecUpdateErrorParametersChanged if any parameters to a node changed in a way that is not supported, in which case hErrorNode_out is set to the node from hGraph

- cudaGraphExecUpdateErrorNotSupported if something about a node is unsupported, like the node’s type or configuration, in which case hErrorNode_out is set to the node from hGraph

If updateResult_out isn’t set in one of the situations described above, the update check passes and cudaGraphExecUpdate updates hGraphExec to match the contents of hGraph. If an error happens during the update, updateResult_out will be set to cudaGraphExecUpdateError; otherwise, updateResult_out is set to cudaGraphExecUpdateSuccess.

cudaGraphExecUpdate returns cudaSuccess when the update was performed successfully. It returns cudaErrorGraphExecUpdateFailure if the graph update was not performed because it included changes which violated constraints specific to instantiated graph update.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphInstantiate,
__host__ cudaError_t
cudaGraphExternalSemaphoresSignalNodeGetParams
(cudaGraphNode_t hNode,
cudaExternalSemaphoreSignalNodeParams
*params_out)

Returns an external semaphore signal node’s parameters.

Parameters

hNode
- Node to get the parameters for

params_out
- Pointer to return the parameters

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Returns the parameters of an external semaphore signal node hNode in params_out. The extSemArray and paramsArray returned in params_out, are owned by the node. This memory remains valid until the node is destroyed or its parameters are modified, and should not be modified directly. Use cudaGraphExternalSemaphoresSignalNodeSetParams to update the parameters of this node.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
Sets an external semaphore signal node's parameters.

**Parameters**

**hNode**
- Node to set the parameters for

**nodeParams**
- Parameters to copy

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`

**Description**

Sets the parameters of an external semaphore signal node `hNode` to `nodeParams`.

**Note:**
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**
cudaGraphAddExternalSemaphoresSignalNode, cudaGraphExternalSemaphoresSignalNodeSetParams, cudaGraphAddExternalSemaphoresWaitNode, cudaSignalExternalSemaphoresAsync, cudaWaitExternalSemaphoresAsync

__host__cudaError_t
cudaGraphExternalSemaphoresWaitNodeGetParams (cudaGraphNode_t hNode, cudaExternalSemaphoreWaitNodeParams *params_out)
Returns an external semaphore wait node’s parameters.

Parameters

hNode
- Node to get the parameters for

params_out
- Pointer to return the parameters

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Returns the parameters of an external semaphore wait node hNode in params_out. The extSemArray and paramsArray returned in params_out, are owned by the node. This memory remains valid until the node is destroyed or its parameters are modified, and should not be modified directly. Use cudaGraphExternalSemaphoresSignalNodeSetParams to update the parameters of this node.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
`cudaLaunchKernel`, `cudaGraphAddExternalSemaphoresWaitNode`, `cudaGraphExternalSemaphoresWaitNodeSetParams`, `cudaGraphAddExternalSemaphoresWaitNode`, `cudaSignalExternalSemaphoresAsync`, `cudaWaitExternalSemaphoresAsync`

```c
__host__ cudaError_t
cudaGraphExternalSemaphoresWaitNodeSetParams(
  cudaGraphNode_t hNode, const
cudaExternalSemaphoreWaitNodeParams
  *nodeParams)
```

Sets an external semaphore wait node’s parameters.

**Parameters**

- **hNode**
  - Node to set the parameters for
- **nodeParams**
  - Parameters to copy

**Returns**

- `cudaSuccess`, `cudaErrorInvalidValue`

**Description**

Sets the parameters of an external semaphore wait node hNode to nodeParams.

**Note:**

- Graph objects are not threadsafe. [More here](#).
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
- `cudaGraphAddExternalSemaphoresWaitNode`
- `cudaGraphExternalSemaphoresWaitNodeSetParams`
- `cudaGraphAddExternalSemaphoresWaitNode`
- `cudaSignalExternalSemaphoresAsync`
- `cudaWaitExternalSemaphoresAsync`

```c
__host__ cudaError_t cudaGraphGetEdges
(cudaGraph_t graph, cudaGraphNode_t *from,
cudaGraphNode_t *to, size_t *numEdges)
```

Returns a graph’s dependency edges.

Parameters

- **graph**
  - Graph to get the edges from
- **from**
  - Location to return edge endpoints
- **to**
  - Location to return edge endpoints
- **numEdges**
  - See description

Returns

- `cudaSuccess`, `cudaErrorInvalidValue`

Description

Returns a list of graph’s dependency edges. Edges are returned via corresponding indices in `from` and `to`; that is, the node in `to[i]` has a dependency on the node in `from[i]`. `from` and `to` may both be NULL, in which case this function only returns the number of edges in `numEdges`. Otherwise, `numEdges` entries will be filled in. If `numEdges` is higher than the actual number of edges, the remaining entries in `from` and `to` will be set to NULL, and the number of edges actually returned will be written to `numEdges`.

Note:

- Graph objects are not threadsafe. More here.
Note that this function may also return error codes from previous, asynchronous launches.

Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.

Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

`cudaGraphGetNodes`, `cudaGraphGetRootNodes`, `cudaGraphAddDependencies`, `cudaGraphRemoveDependencies`, `cudaGraphNodeGetDependencies`, `cudaGraphNodeGetDependentNodes`

```c
__host__cudaError_t cudaGraphGetNodes(
    cudaGraph_t graph, cudaGraphNode_t *nodes, size_t *numNodes)
```

Returns a graph’s nodes.

Parameters

- **graph**
  - Graph to query
- **nodes**
  - Pointer to return the nodes
- **numNodes**
  - See description

Returns

`cudaSuccess`, `cudaErrorInvalidValue`

Description

Returns a list of graph’s nodes. `nodes` may be NULL, in which case this function will return the number of nodes in `numNodes`. Otherwise, `numNodes` entries will be filled in. If `numNodes` is higher than the actual number of nodes, the remaining entries in `nodes` will be set to NULL, and the number of nodes actually obtained will be returned in `numNodes`.

Note:

- Graph objects are not threadsafe. [More here](#)
Note that this function may also return error codes from previous, asynchronous launches.

Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.

Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphCreate, cudaGraphGetRootNodes, cudaGraphGetEdges, cudaGraphNodeGetType, cudaGraphNodeGetDependencies, cudaGraphNodeGetDependentNodes

__host__ cudaError_t cudaGraphGetRootNodes
(cudaGraph_t graph, cudaGraphNode_t *pRootNodes, size_t *pNumRootNodes)
Returns a graph’s root nodes.

Parameters

graph
- Graph to query

pRootNodes
- Pointer to return the root nodes

pNumRootNodes
- See description

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Returns a list of graph's root nodes. pRootNodes may be NULL, in which case this function will return the number of root nodes in pNumRootNodes. Otherwise, pNumRootNodes entries will be filled in. If pNumRootNodes is higher than the actual number of root nodes, the remaining entries in pRootNodes will be set to NULL, and the number of nodes actually obtained will be returned in pNumRootNodes.

Note:
- Graph objects are not threadsafe. [More here](#).
Note that this function may also return error codes from previous, asynchronous launches.

Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.

Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphCreate, cudaGraphGetNodes, cudaGraphGetEdges, cudaGraphNodeGetType, cudaGraphNodeGetDependencies, cudaGraphNodeGetDependentNodes

__host__cudaError_t cudaGraphHostNodeGetParams (cudaGraphNode_t node, cudaHostNodeParams *pNodeParams)
Returns a host node’s parameters.

Parameters

node
  - Node to get the parameters for

pNodeParams
  - Pointer to return the parameters

Returns
cudaSuccess, cudaErrorInvalidValue

Description
Returns the parameters of host node node in pNodeParams.

Note:

Graph objects are not threadsafe. More here.

Note that this function may also return error codes from previous, asynchronous launches.

Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaLaunchHostFunc, cudaGraphAddHostNode, cudaGraphHostNodeSetParams

__host__cudaError_t cudaGraphHostNodeSetParams (cudaGraphNode_t node, const cudaHostNodeParams *pNodeParams)
Sets a host node’s parameters.

Parameters

node
- Node to set the parameters for

pNodeParams
- Parameters to copy

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Sets the parameters of host node node to nodeParams.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaLaunchHostFunc, cudaGraphAddHostNode, cudaGraphHostNodeGetParams
"__host__cudaError_t cudaGraphInstantiate\n(cudaGraphExec_t *pGraphExec, cudaGraph_t graph,\ncudaGraphNode_t *pErrorNode, char *pLogBuffer,\nsize_t bufferSize)"

Creates an executable graph from a graph.

Parameters

- **pGraphExec** - Returns instantiated graph
- **graph** - Graph to instantiate
- **pErrorNode** - In case of an instantiation error, this may be modified to indicate a node contributing to the error
- **pLogBuffer** - A character buffer to store diagnostic messages
- **bufferSize** - Size of the log buffer in bytes

Returns

- **cudaSuccess**, **cudaErrorInvalidValue**

Description

Instantiates `graph` as an executable graph. The graph is validated for any structural constraints or intra-node constraints which were not previously validated. If instantiation is successful, a handle to the instantiated graph is returned in `pGraphExec`.

If there are any errors, diagnostic information may be returned in `pErrorNode` and `pLogBuffer`. This is the primary way to inspect instantiation errors. The output will be null terminated unless the diagnostics overflow the buffer. In this case, they will be truncated, and the last byte can be inspected to determine if truncation occurred.

**Note:**

- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
- `cudaGraphInstantiateWithFlags`
- `cudaGraphCreate`
- `cudaGraphUpload`
- `cudaGraphLaunch`
- `cudaGraphExecDestroy`

```c
__host__ cudaError_t cudaGraphInstantiateWithFlags(cudaGraphExec_t *pGraphExec, cudaGraph_t graph, unsigned long long flags)
```

Creates an executable graph from a graph.

**Parameters**

- `pGraphExec` - Returns instantiated graph
- `graph` - Graph to instantiate
- `flags` - Flags to control instantiation. See `CUgraphInstantiate_flags`.

**Returns**

- `cudaSuccess`
- `cudaErrorInvalidValue`

**Description**

Instantiates `graph` as an executable graph. The graph is validated for any structural constraints or intra-node constraints which were not previously validated. If instantiation is successful, a handle to the instantiated graph is returned in `pGraphExec`.

The `flags` parameter controls the behavior of instantiation and subsequent graph launches. Valid flags are:

- `cudaGraphInstantiateFlagAutoFreeOnLaunch`, which configures a graph containing memory allocation nodes to automatically free any unfreed memory allocations before the graph is relaunched.

If `graph` contains any allocation or free nodes, there can be at most one executable graph in existence for that graph at a time.

An attempt to instantiate a second executable graph before destroying the first with `cudaGraphExecDestroy` will result in an error.
__host__ cudaError_t cudaGraphKernelNodeCopyAttributes (cudaGraphNode_t hSrc, cudaGraphNode_t hDst)
Copies attributes from source node to destination node.

Returns

cudaSuccess, cudaErrorInvalidContext

Description

Copies attributes from source node src to destination node dst. Both node must have the same context.

See also:

cudaAccessPolicyWindow

Note:

Graph objects are not threadsafe. More here.

Note that this function may also return error codes from previous, asynchronous launches.

Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.

Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaGraphInstantiate, cudaGraphCreate, cudaGraphUpload, cudaGraphLaunch, cudaGraphExecDestroy

cudaAccessPolicyWindow
cudaGraphKernelNodeGetAttribute

`cudaGraphNode_t hNode, cudaKernelNodeAttrID attr, cudaKernelNodeAttrValue *value_out)`

Queries node attribute.

**Parameters**
- **hNode**
- **attr**
- **value_out**

**Returns**
- cudaSuccess
- cudaErrorInvalidValue
- cudaErrorInvalidResourceHandle

**Description**
Queries attribute `attr` from node `hNode` and stores it in corresponding member of `value_out`.

**Note:**
Note that this function may also return error codes from previous, asynchronous launches.

**See also:**
- `cudaAccessPolicyWindow`

cudaGraphKernelNodeGetParams

`cudaGraphNode_t node, cudaKernelNodeParams *pNodeParams)`

Returns a kernel node’s parameters.

**Parameters**
- **node**
  - Node to get the parameters for
- **pNodeParams**
  - Pointer to return the parameters
Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDeviceFunction`

Description

Returns the parameters of kernel node `node` in `pNodeParams`. The `kernelParams` or `extra` array returned in `pNodeParams`, as well as the argument values it points to, are owned by the node. This memory remains valid until the node is destroyed or its parameters are modified, and should not be modified directly. Use `cudaGraphKernelNodeSetParams` to update the parameters of this node.

The params will contain either `kernelParams` or `extra`, according to which of these was most recently set on the node.

---

Note:

- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

`cudaLaunchKernel`, `cudaGraphAddKernelNode`, `cudaGraphKernelNodeSetParams`

```c
__host__ cudaError_t
cudaGraphKernelNodeSetAttribute
(cudaGraphNode_t hNode, cudaKernelNodeAttrID attr, const cudaKernelNodeAttrValue *value)
```

Sets node attribute.

Parameters

- `hNode`
- `attr`
- `value`
**Returns**

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle

**Description**

Sets attribute attr on node hNode from corresponding attribute of value.

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

cudaAccessPolicyWindow

```c
__host__ cudaError_t
cudaGraphKernelNodeSetParams (cudaGraphNode_t node, const cudaKernelNodeParams *pNodeParams)
```

Sets a kernel node’s parameters.

**Parameters**

- **node**
  - Node to set the parameters for
- **pNodeParams**
  - Parameters to copy

**Returns**

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorMemoryAllocation

**Description**

Sets the parameters of kernel node node to pNodeParams.

**Note:**

- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.
Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.

Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
`cudaLaunchKernel`, `cudaGraphAddKernelNode`, `cudaGraphKernelNodeGetParams`

```c
__host__ cudaError_t cudaGraphLaunch (cudaGraphExec_t graphExec, cudaStream_t stream)
```

Launches an executable graph in a stream.

**Parameters**

- **graphExec**
  - Executable graph to launch

- **stream**
  - Stream in which to launch the graph

**Returns**

- `cudaSuccess`
- `cudaErrorInvalidValue`

**Description**

Executes `graphExec` in `stream`. Only one instance of `graphExec` may be executing at a time. Each launch is ordered behind both any previous work in `stream` and any previous launches of `graphExec`. To execute a graph concurrently, it must be instantiated multiple times into multiple executable graphs.

If any allocations created by `graphExec` remain unfreed (from a previous launch) and `graphExec` was not instantiated with `cudaGraphInstantiateFlagAutoFreeOnLaunch`, the launch will fail with `cudaErrorInvalidValue`.

**Note:**

- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.
Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.

Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
`cudaGraphInstantiate`, `cudaGraphUpload`, `cudaGraphExecDestroy`

```c
__host__ cudaError_t cudaGraphMemAllocNodeGetParams (cudaGraphNode_t node, cudaMemAllocNodeParams *params_out) returns a memory alloc node’s parameters.
```

**Parameters**

- **node**
  - Node to get the parameters for

- **params_out**
  - Pointer to return the parameters

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`

**Description**

Returns the parameters of a memory alloc node `hNode` in `params_out`. The poolProps and accessDescs returned in `params_out`, are owned by the node. This memory remains valid until the node is destroyed. The returned parameters must not be modified.

**Note:**

- Graph objects are not threadsafe. More here.

- Note that this function may also return error codes from previous, asynchronous launches.

- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
`cudaGraphAddMemAllocNode`, `cudaGraphMemFreeNodeGetParams`

```c
__host__ cudaError_t
cudaGraphMemcpyNodeGetParams
(cudaGraphNode_t node, cudaMemcpy3DParms *pNodeParams)
```

Returns a memcpy node's parameters.

**Parameters**

- `node` - Node to get the parameters for
- `pNodeParams` - Pointer to return the parameters

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`

**Description**

Returns the parameters of memcpy node `node` in `pNodeParams`.

**Note:**

- Graph objects are not thread-safe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
__host__ cudaError_t
cudaGraphMemcpyNodeSetParams
(cudaGraphNode_t node, const cudaMemcpy3DParms *pNodeParams)

Sets a memcpy node’s parameters.

Parameters

node
  - Node to set the parameters for

pNodeParams
  - Parameters to copy

Returns
cudaSuccess, cudaErrorInvalidValue.

Description

Sets the parameters of memcpy node node to pNodeParams.

Note:

- Graph objects are not thread safe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpy3D, cudaMemcpyNodeSetParamsToSymbol, cudaMemcpyNodeSetParamsFromSymbol, cudaMemcpyNodeSetParams1D, cudaMemcpyAddMemcpyNode, cudaMemcpyNodeGetParams no CUDA function may be called from callback. cudaMemcpyNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
__host__ cudaError_t
cudaGraphMemcpyNodeSetParams1D
(cudaGraphNode_t node, void *dst, const void *src,
size_t count, cudaMemcpyKind kind)
Sets a memcpy node’s parameters to perform a 1-dimensional copy.

Parameters

node
- Node to set the parameters for
dst
- Destination memory address
src
- Source memory address
count
- Size in bytes to copy
kind
- Type of transfer

Returns
cudaSuccess, cudaErrorInvalidValue

Description
Sets the parameters of memcpy node node to the copy described by the provided parameters.

When the graph is launched, the node will copy count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing. Launching a memcpy node with dst and src pointers that do not match the direction of the copy results in an undefined behavior.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.

Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

`cudaMemcpy`, `cudaGraphMemcpyNodeSetParams`, `cudaGraphAddMemcpyNode`, `cudaGraphMemcpyNodeGetParams`  

```c
__host__ cudaError_t
cudaGraphMemcpyNodeSetParamsFromSymbol(
cudaGraphNode_t node, void *dst, const void *
symbol, size_t count, size_t offset, cudaMemcpyKind
kind)
```

Sets a memcpy node’s parameters to copy from a symbol on the device.

**Parameters**

- **node**
  - Node to set the parameters for
- **dst**
  - Destination memory address
- **symbol**
  - Device symbol address
- **count**
  - Size in bytes to copy
- **offset**
  - Offset from start of symbol in bytes
- **kind**
  - Type of transfer

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`

**Description**

Sets the parameters of memcpy node `node` to the copy described by the provided parameters.
When the graph is launched, the node will copy \texttt{count} bytes from the memory area pointed to by \texttt{offset} bytes from the start of \texttt{symbol} to the memory area pointed to by \texttt{dst}. The memory areas may not overlap. \texttt{symbol} is a variable that resides in global or constant memory space. \texttt{kind} can be either \texttt{cudaMemcpyDeviceToHost}, \texttt{cudaMemcpyDeviceToDevice}, or \texttt{cudaMemcpyDefault}. Passing \texttt{cudaMemcpyDefault} is recommended, in which case the type of transfer is inferred from the pointer values. However, \texttt{cudaMemcpyDefault} is only allowed on systems that support unified virtual addressing.

\begin{itemize}
  \item Graph objects are not threadsafe. \texttt{More here}.
  \item Note that this function may also return error codes from previous, asynchronous launches.
  \item Note that this function may also return \texttt{cudaErrorInitializationError}, \texttt{cudaErrorInsufficientDriver} or \texttt{cudaErrorNoDevice} if this call tries to initialize internal CUDA RT state.
  \item Note that as specified by \texttt{cudaStreamAddCallback} no CUDA function may be called from callback. \texttt{cudaErrorNotPermitted} may, but is not guaranteed to, be returned as a diagnostic in such case.
\end{itemize}

\textbf{See also:}
\begin{itemize}
  \item \texttt{cudaMemcpyFromSymbol}, \texttt{cudaGraphMemcpyNodeSetParams}, \texttt{cudaGraphMemcpyNodeSetParamsToSymbol}, \texttt{cudaGraphAddMemcpyNode}, \texttt{cudaGraphMemcpyNodeGetParams}
\end{itemize}

\begin{verbatim}
__host__ cudaError_t
cudaGraphMemcpyNodeSetParamsToSymbol
  (cudaGraphNode_t node, const void *symbol, const void *src, size_t count, size_t offset, cudaMemcpyKind kind)
\end{verbatim}

Sets a memcpy node’s parameters to copy to a symbol on the device.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{node} \\
  \hspace{1em} - Node to set the parameters for
  \item \texttt{symbol} \\
  \hspace{1em} - Device symbol address
  \item \texttt{src} \\
  \hspace{1em} - Source memory address
\end{itemize}
count
- Size in bytes to copy

offset
- Offset from start of symbol in bytes

kind
- Type of transfer

Returns
cudaSuccess, cudaErrorInvalidValue

Description
Sets the parameters of memcpy node node to the copy described by the provided parameters. When the graph is launched, the node will copy count bytes from the memory area pointed to by src to the memory area pointed to by offset bytes from the start of symbol symbol. The memory areas may not overlap. symbol is a variable that resides in global or constant memory space. kind can be either cudaMemcpyHostToDevice, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaMemcpyToSymbol, cudaMemcpyNodeSetParams,
cudaGraphMemcpyNodeSetParamsFromSymbol, cudaGraphAddMemcpyNode,
cudaGraphMemcpyNodeGetParams
__host__ cudaError_t
cudaGraphMemFreeNodeGetParams
cudaGraphNode_t node, void *dptr_out)

Returns a memory free node’s parameters.

Parameters

node
  - Node to get the parameters for
dptr_out
  - Pointer to return the device address

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Returns the address of a memory free node hNode in dptr_out.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphAddMemFreeNode, cudaGraphMemFreeNodeGetParams
__host__ cudaError_t cudaGraphMemsetNodeGetParams (cudaGraphNode_t node, cudaMemsetParams *pNodeParams)

Returns a memset node’s parameters.

Parameters

node
- Node to get the parameters for

pNodeParams
- Pointer to return the parameters

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Returns the parameters of memset node node in pNodeParams.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaMemset2D, cudaGraphAddMemsetNode, cudaGraphMemsetNodeSetParams
__host__ cudaError_t
.cudaGraphMemsetNodeSetParams
(cudaGraphNode_t node, const cudaMemcpyParams
*pNodeParams)
Sets a memset node’s parameters.

Parameters
node
- Node to set the parameters for

pNodeParams
- Parameters to copy

Returns
cudaSuccess, cudaErrorInvalidValue

Description
Sets the parameters of memset node node to pNodeParams.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpyCallback no CUDA function may be called from callback. cudaMemcpyNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaMemset2D, cudaGraphAddMemsetNode, cudaGraphMemsetNodeGetParams
__host__ cudaError_t cudaGraphNodeFindInClone
            (cudaGraphNode_t *pNode, cudaGraphNode_t originalNode, cudaGraph_t clonedGraph)

Finds a cloned version of a node.

Parameters

pNode
  - Returns handle to the cloned node

originalNode
  - Handle to the original node

clonedGraph
  - Cloned graph to query

Returns

cudaSuccess, cudaErrorInvalidValue

Description

This function returns the node in clonedGraph corresponding to originalNode in the
original graph.

clonedGraph must have been cloned from originalGraph via cudaMemcpy.
originalNode must have been in originalGraph at the time of the call to
cudaGraphClone, and the corresponding cloned node in clonedGraph must not have been
removed. The cloned node is then returned via pColnodedNode.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaMemcpyInitializationError, cudaMemcpyInsufficientDriver or cudaMemcpyNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpyAddCallback no CUDA function may be called from callback. cudaMemcpyNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphClone
__host__ cudaError_t
cudaGraphNodeGetDependencies (cudaGraphNode_t node, cudaGraphNode_t *pDependencies, size_t *pNumDependencies)

Returns a node’s dependencies.

Parameters

node
- Node to query

pDependencies
- Pointer to return the dependencies

pNumDependencies
- See description

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Returns a list of node's dependencies. pDependencies may be NULL, in which case this function will return the number of dependencies in pNumDependencies. Otherwise, pNumDependencies entries will be filled in. If pNumDependencies is higher than the actual number of dependencies, the remaining entries in pDependencies will be set to NULL, and the number of nodes actually obtained will be returned in pNumDependencies.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotAllowed may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphNodeGetDependentNodes, cudaGraphNodeGetNodes, cudaGraphGetRootNodes, cudaGraphGetEdges, cudaGraphAddDependencies, cudaGraphRemoveDependencies

__host__ cudaError_t
cudaGraphNodeGetDependentNodes
(cudaGraphNode_t node, cudaGraphNode_t *pDependentNodes, size_t *pNumDependentNodes)

Returns a node’s dependent nodes.

Parameters

- **node**
  - Node to query
- **pDependentNodes**
  - Pointer to return the dependent nodes
- **pNumDependentNodes**
  - See description

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Returns a list of node’s dependent nodes. pDependentNodes may be NULL, in which case this function will return the number of dependent nodes in pNumDependentNodes. Otherwise, pNumDependentNodes entries will be filled in. If pNumDependentNodes is higher than the actual number of dependent nodes, the remaining entries in pDependentNodes will be set to NULL, and the number of nodes actually obtained will be returned in pNumDependentNodes.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
__host__ cudaError_t cudaGraphNodeGetType(cudaGraphNode_t node, cudaGraphNodeType *pType)

Returns a node’s type.

Parameters

node
- Node to query

pType
- Pointer to return the node type

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Returns the node type of node in pType.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaGraphNodeGetDependencies, cudaGraphGetNodes, cudaGraphGetRootNodes,
cudaGraphGetEdges, cudaGraphAddDependencies, cudaGraphRemoveDependencies.
cudaGraphMemcpyNodeGetParams, cudaGraphMemcpyNodeSetParams,
cudaGraphMemsetNodeGetParams, cudaGraphMemsetNodeSetParams

__host__cudaError_t cudaGraphReleaseUserObject
(cudaGraph_t graph, cudaUserObject_t object,
unsigned int count)
Release a user object reference from a graph.

Parameters

graph
  - The graph that will release the reference
object
  - The user object to release a reference for
count
  - The number of references to release, typically 1. Must be nonzero and not larger than
    INT_MAX.

Returns
cudaSuccess, cudaErrorInvalidValue

Description
Releases user object references owned by a graph.
See CUDA User Objects in the CUDA C++ Programming Guide for more information on user
objects.

See also:
cudaUserObjectCreate, cudaUserObjectRetain, cudaUserObjectRelease,
cudaGraphRetainUserObject, cudaGraphCreate
__host__ cudaError_t
cudaGraphRemoveDependencies (cudaGraph_t graph, const cudaGraphNode_t *from, const cudaGraphNode_t *to, size_t numDependencies)

Removes dependency edges from a graph.

Parameters

graph
  - Graph from which to remove dependencies
from
  - Array of nodes that provide the dependencies
to
  - Array of dependent nodes
numDependencies
  - Number of dependencies to be removed

Returns
cudaSuccess, cudaErrorInvalidValue

Description

The number of dependencies to be removed is defined by numDependencies. Elements in pFrom and pTo at corresponding indices define a dependency. Each node in pFrom and pTo must belong to graph.

If numDependencies is 0, elements in pFrom and pTo will be ignored. Specifying a non-existing dependency will return an error.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
__host__ cudaError_t cudaGraphRetainUserObject (cudaGraph_t graph, cudaUserObject_t object, unsigned int count, unsigned int flags)

Retain a reference to a user object from a graph.

Parameters

- **graph**
  - The graph to associate the reference with

- **object**
  - The user object to retain a reference for

- **count**
  - The number of references to add to the graph, typically 1. Must be nonzero and not larger than INT_MAX.

- **flags**
  - The optional flag cudaGraphUserObjectMove transfers references from the calling thread, rather than create new references. Pass 0 to create new references.

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Creates or moves user object references that will be owned by a CUDA graph.

See CUDA User Objects in the CUDA C++ Programming Guide for more information on user objects.

See also:
cudaUserObjectCreate, cudaUserObjectRetain, cudaUserObjectRelease, cudaGraphReleaseUserObject, cudaGraphCreate
__host__ cudaError_t cudaGraphUpload (cudaGraphExec_t graphExec, cudaStream_t stream)

Uploads an executable graph in a stream.

Returns

cudaSuccess, cudaErrorInvalidValue,

Description

Uploads hGraphExec to the device in hStream without executing it. Uploads of the same hGraphExec will be serialized. Each upload is ordered behind both any previous work in hStream and any previous launches of hGraphExec. Uses memory cached by stream to back the allocations owned by graphExec.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.

See also:

cudaGraphInstantiate, cudaGraphLaunch, cudaGraphExecDestroy

__host__ cudaError_t cudaUserObjectCreate (cudaUserObject_t *object_out, void *ptr, cudaHostFn_t destroy, unsigned int initialRefcount, unsigned int flags)

Create a user object.

Parameters

object_out
- Location to return the user object handle

ptr
- The pointer to pass to the destroy function

destroy
- Callback to free the user object when it is no longer in use
**initialRefcount**
- The initial refcount to create the object with, typically 1. The initial references are owned by the calling thread.

**flags**
- Currently it is required to pass `cudaUserObjectNoDestructorSync`, which is the only defined flag. This indicates that the destroy callback cannot be waited on by any CUDA API. Users requiring synchronization of the callback should signal its completion manually.

**Returns**
`cudaSuccess`, `cudaErrorInvalidValue`

**Description**
Create a user object with the specified destructor callback and initial reference count. The initial references are owned by the caller.

Destructor callbacks cannot make CUDA API calls and should avoid blocking behavior, as they are executed by a shared internal thread. Another thread may be signaled to perform such actions, if it does not block forward progress of tasks scheduled through CUDA.

See CUDA User Objects in the CUDA C++ Programming Guide for more information on user objects.

**See also:**
`cudaUserObjectRetain`, `cudaUserObjectRelease`, `cudaGraphRetainUserObject`, `cudaGraphReleaseUserObject`, `cudaGraphCreate`

```c
__host__ cudaError_t cudaUserObjectRelease (cudaUserObject_t object, unsigned int count)
```
Release a reference to a user object.

**Parameters**

**object**
- The object to release

**count**
- The number of references to release, typically 1. Must be nonzero and not larger than INT_MAX.

**Returns**
`cudaSuccess`, `cudaErrorInvalidValue`
Description
Releases user object references owned by the caller. The object’s destructor is invoked if the reference count reaches zero.
It is undefined behavior to release references not owned by the caller, or to use a user object handle after all references are released.
See CUDA User Objects in the CUDA C++ Programming Guide for more information on user objects.

See also:
cudaUserObjectCreate, cudaUserObjectRetain, cudaGraphRetainUserObject, cudaGraphReleaseUserObject, cudaGraphCreate

__host__cudaError_t cudaUserObjectRetain(cudaUserObject_t object, unsigned int count)
Retain a reference to a user object.

Parameters
object
- The object to retain
count
- The number of references to retain, typically 1. Must be nonzero and not larger than INT_MAX.

Returns
cudaSuccess, cudaErrorInvalidValue

Description
Retains new references to a user object. The new references are owned by the caller.
See CUDA User Objects in the CUDA C++ Programming Guide for more information on user objects.

See also:
cudaUserObjectCreate, cudaUserObjectRelease, cudaGraphRetainUserObject, cudaGraphReleaseUserObject, cudaGraphCreate
6.31. Driver Entry Point Access

This section describes the driver entry point access functions of CUDA runtime application programming interface.

```c
__host__ cudaError_t cudaGetDriverEntryPoint (const char *symbol, void **funcPtr, unsigned long long flags)
```

Returns the requested driver API function pointer.

**Parameters**

- **symbol**
  - The base name of the driver API function to look for. As an example, for the driver API `cuMemAlloc_v2`, `symbol` would be `cuMemAlloc`. Note that the API will use the CUDA runtime version to return the address to the most recent ABI compatible driver symbol, `cuMemAlloc` or `cuMemAlloc_v2`.

- **funcPtr**
  - Location to return the function pointer to the requested driver function

- **flags**
  - Flags to specify search options.

**Returns**

- `cudaSuccess`
- `cudaErrorInvalidValue`
- `cudaErrorNotSupported`
- `cudaErrorSymbolNotFound`

**Description**

Returns in **funcPtr** the address of the CUDA driver function for the requested flags.

For a requested driver symbol, if the CUDA version in which the driver symbol was introduced is less than or equal to the CUDA runtime version, the API will return the function pointer to the corresponding versioned driver function.

The pointer returned by the API should be cast to a function pointer matching the requested driver function’s definition in the API header file. The function pointer typedef can be picked up from the corresponding typedefs header file. For example, `cudaTypedefs.h` consists of function pointer typedefs for driver APIs defined in `cuda.h`.

The API will return `cudaErrorSymbolNotFound` if the requested driver function is not supported on the platform, no ABI compatible driver function exists for the CUDA runtime version or if the driver symbol is invalid.

The requested flags can be:
 cudaEnableDefault: This is the default mode. This is equivalent to cudaEnablePerThreadDefaultStream if the code is compiled with --default-stream per-thread compilation flag or the macro CUDA_API_PER_THREAD_DEFAULT_STREAM is defined; cudaEnableLegacyStream otherwise.

 cudaEnableLegacyStream: This will enable the search for all driver symbols that match the requested driver symbol name except the corresponding per-thread versions.

 cudaEnablePerThreadDefaultStream: This will enable the search for all driver symbols that match the requested driver symbol name including the per-thread versions. If a per-thread version is not found, the API will return the legacy version of the driver function.

Note:

- Version mixing among CUDA-defined types and driver API versions is strongly discouraged and doing so can result in an undefined behavior. More here.

- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.

- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cuGetProcAddress

6.32. C++ API Routines

C++-style interface built on top of CUDA runtime API.

This section describes the C++ high level API functions of the CUDA runtime application programming interface. To use these functions, your application needs to be compiled with the nvcc compiler.

__cudaOccupancyB2DHelper
cppClassifierVisibility: visibility=public
template < class T, int dim > __host__ cudaError_t
cudaBindSurfaceToArray (const surfaceTdim surf, 
cudaArray_const_t array)

[C++ API] Binds an array to a surface

Parameters
surf
  - Surface to bind
array
  - Memory array on device

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSurface

Description
Binds the CUDA array array to the surface reference surf. The channel descriptor is
inherited from the CUDA array. Any CUDA array previously bound to surf is unbound.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaBindSurfaceToArray | C API], cudaBindSurfaceToArray | C++ API]
template < class T, int dim > __host__ cudaError_t
cudaBindSurfaceToArray (const surfaceTdim surf, cudaArray_const_t array, const
cudaChannelFormatDesc desc)

[C++ API] Binds an array to a surface

Parameters
surf
- Surface to bind
array
- Memory array on device
desc
- Channel format

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSurface

Description
Binds the CUDA array array to the surface reference surf. desc describes how the memory is interpreted when dealing with the surface. Any CUDA array previously bound to surf is unbound.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaBindSurfaceToArray | C API, cudaBindSurfaceToArray | C++ API, inherited channel descriptor
template < class T, int dim, enum cudaTextureReadMode readMode >
__host__cudaError_t cudaBindTexture (size_t *offset, const textureTdimreadMode tex, const void *devPtr, size_t size)

[C++ API] Binds a memory area to a texture

Parameters

offset
- Offset in bytes
tex
- Texture to bind
devPtr
- Memory area on device
size
- Size of the memory area pointed to by devPtr

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidTexture

Description

Binds size bytes of the memory area pointed to by devPtr to texture reference tex. The channel descriptor is inherited from the texture reference type. The offset parameter is an optional byte offset as with the low-level cudaBindTexture( size_t*, const struct textureReference*, const void*, const struct cudaChannelFormatDesc*, size_t) function. Any memory previously bound to tex is unbound.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
template < class T, int dim, enum cudaTextureReadMode readMode >
__host__cudaError_t cudaBindTexture (size_t *offset, const textureTdimreadMode tex, const void *devPtr, const cudaChannelFormatDesc desc, size_t size)

[C++ API] Binds a memory area to a texture

Parameters

offset
- Offset in bytes
tex
- Texture to bind
devPtr
- Memory area on device
desc
- Channel format
size
- Size of the memory area pointed to by devPtr

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidTexture

Description

Binds size bytes of the memory area pointed to by devPtr to texture reference tex. desc describes how the memory is interpreted when fetching values from the texture. The offset parameter is an optional byte offset as with the low-level cudaBindTexture() function. Any memory previously bound to tex is unbound.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
Note that this function may also return \texttt{cudaErrorInitializationError}, \texttt{cudaErrorInsufficientDriver} or \texttt{cudaErrorNoDevice} if this call tries to initialize internal CUDA RT state.

Note that as specified by \texttt{cudaStreamAddCallback} no CUDA function may be called from callback. \texttt{cudaErrorNotPermitted} may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
\texttt{cudaCreateChannelDesc ( C++ API)}, \texttt{cudaGetChannelDesc}, \texttt{cudaGetTextureReference}, \texttt{cudaBindTexture ( C API)}, \texttt{cudaBindTexture ( C++ API, inherited channel descriptor)}, \texttt{cudaBindTexture2D ( C++ API)}, \texttt{cudaBindTexture2D ( C++ API, inherited channel descriptor)}, \texttt{cudaBindTextureToArray ( C++ API)}, \texttt{cudaBindTextureToArray ( C++ API, inherited channel descriptor)}, \texttt{cudaUnbindTexture ( C++ API)}, \texttt{cudaGetTextureAlignmentOffset ( C++ API)}

\begin{verbatim}
template < class T, int dim, enum cudaTextureReadMode readMode >
__host__ cudaError_t cudaBindTexture2D (size_t *offset, const textureTdimreadMode tex, const void *devPtr, size_t width, size_t height, size_t pitch)
[C++ API] Binds a 2D memory area to a texture

Parameters
offset
  - Offset in bytes
tex
  - Texture reference to bind
devPtr
  - 2D memory area on device
width
  - Width in texel units
height
  - Height in texel units
pitch
  - Pitch in bytes

Returns
\texttt{cudaSuccess}, \texttt{cudaErrorInvalidValue}, \texttt{cudaErrorInvalidTexture}
\end{verbatim}
Description

Binds the 2D memory area pointed to by devPtr to the texture reference tex. The size of the area is constrained by width in texel units, height in texel units, and pitch in byte units. The channel descriptor is inherited from the texture reference type. Any memory previously bound to tex is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, cudaBindTexture2D() returns in *offset a byte offset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the tex2D() function. If the device memory pointer was returned from cudaMalloc(), the offset is guaranteed to be 0 and NULL may be passed as the offset parameter.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaCreateChannelDesc [ C++ API], cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture [ C++ API], cudaBindTexture [ C++ API, inherited channel descriptor], cudaBindTexture2D [ C API], cudaBindTexture2D [ C++ API], cudaBindTextureToArray [ C++ API], cudaBindTextureToArray [ C++ API, inherited channel descriptor], cudaUnbindTexture [ C ++ API], cudaGetTextureAlignmentOffset [ C++ API]
template < class T, int dim, enum cudaTextureReadMode readMode >
__host__ cudaError_t cudaBindTexture2D (size_t *offset, const textureTdimreadMode tex, const void *devPtr, const cudaChannelFormatDesc desc, size_t width, size_t height, size_t pitch)

[C++ API] Binds a 2D memory area to a texture

Parameters

offset
- Offset in bytes

tex
- Texture reference to bind

devPtr
- 2D memory area on device
desc
- Channel format

width
- Width in texel units

height
- Height in texel units

pitch
- Pitch in bytes

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidTexture

Description

Binds the 2D memory area pointed to by devPtr to the texture reference tex. The size of the area is constrained by width in texel units, height in texel units, and pitch in byte units. desc describes how the memory is interpreted when fetching values from the texture. Any memory previously bound to tex is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, cudaBindTexture2D() returns in *offset a byte offset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the tex2D() function. If the device memory pointer was returned from cudaMalloc(), the offset is guaranteed to be 0 and NULL may be passed as the offset parameter.
Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:


```cpp
template < class T, int dim, enum cudaTextureReadMode readMode >
__host__ cudaError_t cudaBindTextureToArray (const textureTdimreadMode tex, cudaArray_const_t array)
```

[C++ API] Binds an array to a texture

**Parameters**

- `tex` - Texture to bind
- `array` - Memory array on device

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidTexture`

**Description**

Binds the CUDA array `array` to the texture reference `tex`. The channel descriptor is inherited from the CUDA array. Any CUDA array previously bound to `tex` is unbound.
Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C++ API), cudaBindTexture (C++ API, inherited channel descriptor), cudaBindTexture2D (C++ API), cudaBindTexture2D (C++ API, inherited channel descriptor), cudaBindTextureToArray (C API), cudaBindTextureToArray (C++ API), cudaUnbindTexture (C++ API), cudaGetTextureAlignmentOffset (C++ API)

template < class T, int dim, enum cudaTextureReadMode readMode >
__host__ cudaError_t cudaBindTextureToArray (const textureTdimreadMode tex, cudaArray_const_t array, const cudaChannelFormatDesc desc)

[C++ API] Binds an array to a texture

Parameters

tex
- Texture to bind
array
- Memory array on device
desc
- Channel format

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidTexture
Description

Binds the CUDA array `array` to the texture reference `tex`. `desc` describes how the memory is interpreted when fetching values from the texture. Any CUDA array previously bound to `tex` is unbound.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:


```cpp
template <class T, int dim, enum cudaTextureReadMode readMode>
__host__cudaError_t cudaBindTextureToMipmappedArray
(const textureTdimreadMode tex,
 const mipmappedArray const_t mipmappedArray)

[C++ API] Binds a mipmapped array to a texture

Parameters

- `tex`  
  - Texture to bind
- `mipmappedArray`  
  - Memory mipmapped array on device
Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidTexture

Description

Binds the CUDA mipmapped array `mipmappedArray` to the texture reference `tex`. The channel descriptor is inherited from the CUDA array. Any CUDA mipmapped array previously bound to `tex` is unbound.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

template < class T, int dim, enum cudaTextureReadMode readMode >
__host__ cudaError_t
cudaBindTextureToMipmappedArray
(const textureTdimreadMode tex,
cudaMipmappedArray_const_t mipmappedArray,
const cudaChannelFormatDesc desc)

[C++ API] Binds a mipmapped array to a texture

Parameters

tex
- Texture to bind
mipmappedArray
- Memory mipmapped array on device
desc
- Channel format

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidTexture

Description

Binds the CUDA mipmapped array `mipmappedArray` to the texture reference `tex`. `desc` describes how the memory is interpreted when fetching values from the texture. Any CUDA mipmapped array previously bound to `tex` is unbound.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
template < class T >

__host__cudaCreateChannelDesc (void)

[C++ API] Returns a channel descriptor using the specified format

Returns
Channel descriptor with format $f$

Description

Returns a channel descriptor with format $f$ and number of bits of each component $x$, $y$, $z$, and $w$. The `cudaChannelFormatDesc` is defined as:

```
struct cudaChannelFormatDesc {
  int x, y, z, w;
  enum cudaChannelFormatKind f;
};
```

where `cudaChannelFormatKind` is one of `cudaChannelFormatKindSigned`, `cudaChannelFormatKindUnsigned`, `cudaChannelFormatKindFloat` or `cudaChannelFormatKindNV12`.

See also:

`cudaCreateChannelDesc [Low level]`, `cudaGetChannelDesc`, `cudaGetTextureReference`, `cudaBindTexture [High level]`, `cudaBindTexture [High level, inherited channel descriptor]`, `cudaBindTexture2D [High level]`, `cudaBindTextureToArray [High level]`, `cudaBindTextureToArray [High level, inherited channel descriptor]`, `cudaUnbindTexture [High level]`, `cudaGetTextureAlignmentOffset [High level]`

__host__cudaError_t cudaEventCreate (cudaEvent_t *event, unsigned int flags)

[C++ API] Creates an event object with the specified flags

Parameters

`event`
- Newly created event
flags
- Flags for new event

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorLaunchFailure, cudaErrorMemoryAllocation

Description
Creates an event object with the specified flags. Valid flags include:

- **cudaEventDefault**: Default event creation flag.
- **cudaEventBlockingSync**: Specifies that event should use blocking synchronization. A host thread that uses `cudaEventSynchronize()` to wait on an event created with this flag will block until the event actually completes.
- **cudaEventDisableTiming**: Specifies that the created event does not need to record timing data. Events created with this flag specified and the `cudaEventBlockingSync` flag not specified will provide the best performance when used with `cudaStreamWaitEvent[]` and `cudaEventQuery[]`.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaEventCreate (C API), cudaEventCreateWithFlags, cudaEventRecord, cudaEventQuery, cudaEventSynchronize, cudaEventDestroy, cudaEventElapsedTime, cudaStreamWaitEvent
template < class T > __host__ cudaError_t
cudaFuncGetAttributes (cudaFuncAttributes *attr, T *
entry)

[C++ API] Find out attributes for a given function

Parameters

attr
   - Return pointer to function’s attributes

entry
   - Function to get attributes of

Returns
cudaSuccess, cudaErrorInvalidDeviceFunction

Description

This function obtains the attributes of a function specified via entry. The parameter entry must be a pointer to a function that executes on the device. The parameter specified by entry must be declared as a __global__ function. The fetched attributes are placed in attr. If the specified function does not exist, then cudaErrorInvalidDeviceFunction is returned.

Note that some function attributes such as maxThreadsPerBlock may vary based on the device that is currently being used.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.

- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.

- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

cudaLaunchKernel (C++ API), cudaFuncSetCacheConfig (C++ API), cudaFuncGetAttributes (C API), cudaSetDoubleForDevice, cudaSetDoubleForHost
template < class T > __host__ cudaError_t cudaFuncSetAttribute (T *entry, cudaFuncAttribute attr, int value)

[C++ API] Set attributes for a given function

Parameters

entry
- Function to get attributes of

attr
- Attribute to set

value
- Value to set

Returns
cudaSuccess, cudaErrorInvalidDeviceFunction, cudaErrorInvalidValue

Description

This function sets the attributes of a function specified via entry. The parameter entry must be a pointer to a function that executes on the device. The parameter specified by entry must be declared as a __global__ function. The enumeration defined by attr is set to the value defined by value. If the specified function does not exist, then cudaErrorInvalidDeviceFunction is returned. If the specified attribute cannot be written, or if the value is incorrect, then cudaErrorInvalidValue is returned.

Valid values for attr are:

- cudaFuncAttributeMaxDynamicSharedMemorySize - The requested maximum size in bytes of dynamically-allocated shared memory. The sum of this value and the function attribute sharedSizeBytes cannot exceed the device attribute cudaDevAttrMaxSharedMemoryPerBlockOptin. The maximal size of requestable dynamic shared memory may differ by GPU architecture.

- cudaFuncAttributePreferredSharedMemoryCarveout - On devices where the L1 cache and shared memory use the same hardware resources, this sets the shared memory carveout preference, in percent of the total shared memory. See cudaDevAttrMaxSharedMemoryPerMultiprocessor. This is only a hint, and the driver can choose a different ratio if required to execute the function.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.

Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

cudaLaunchKernel (C++ API), cudaFuncSetCacheConfig (C++ API), cudaFuncGetAttributes (C API), cudaSetDoubleForDevice, cudaSetDoubleForHost

template < class T > __host__ cudaError_t cudaFuncSetCacheConfig (T *func, cudaFuncCache cacheConfig)

[C++ API] Sets the preferred cache configuration for a device function

**Parameters**

- **func**
  - device function pointer
- **cacheConfig**
  - Requested cache configuration

**Returns**

- `cudaSuccess`
- `cudaErrorInvalidDeviceFunction`

**Description**

On devices where the L1 cache and shared memory use the same hardware resources, this sets through `cacheConfig` the preferred cache configuration for the function specified via `func`. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute `func`.

`func` must be a pointer to a function that executes on the device. The parameter specified by `func` must be declared as a `__global__` function. If the specified function does not exist, then `cudaErrorInvalidDeviceFunction` is returned.

This setting does nothing on devices where the size of the L1 cache and shared memory are fixed.

Launching a kernel with a different preference than the most recent preference setting may insert a device-side synchronization point.

The supported cache configurations are:

- `cudaFuncCachePreferNone`: no preference for shared memory or L1 (default)
CUDA Runtime API

- **cudaFuncCachePreferShared**: prefer larger shared memory and smaller L1 cache
- **cudaFuncCachePreferL1**: prefer larger L1 cache and smaller shared memory

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

---

cudaLaunchKernel [ C++ API], cudaFuncSetCacheConfig [ C API], cudaFuncGetAttributes [ C++ API], cudaSetDoubleForDevice, cudaSetDoubleForHost, cudaThreadGetCacheConfig, cudaThreadSetCacheConfig

template < class T > __host__ cudaError_t cudaGetSymbolAddress (void **devPtr, const T symbol)

[C++ API] Finds the address associated with a CUDA symbol

**Parameters**

- **devPtr**
  - Return device pointer associated with symbol
- **symbol**
  - Device symbol reference

**Returns**

- `cudaSuccess`, `cudaErrorInvalidSymbol`, `cudaErrorNoKernelImageForDevice`

**Description**

Returns in *devPtr* the address of symbol `symbol` on the device. `symbol` can either be a variable that resides in global or constant memory space. If `symbol` cannot be found, or if `symbol` is not declared in the global or constant memory space, *devPtr* is unchanged and the error `cudaErrorInvalidSymbol` is returned.
Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaGetSymbolAddress [ C API], cudaGetSymbolSize [ C++ API]

template < class T > __host__cudaError_t cudaGetSymbolSize (size_t *size, const T symbol)

[C++ API] Finds the size of the object associated with a CUDA symbol

Parameters

size
  - Size of object associated with symbol

symbol
  - Device symbol reference

Returns

cudaSuccess, cudaErrorInvalidSymbol, cudaErrorNoKernelImageForDevice

Description

Returns in *size the size of symbol symbol. symbol must be a variable that resides in global or constant memory space. If symbol cannot be found, or if symbol is not declared in global or constant memory space, *size is unchanged and the error `cudaErrorInvalidSymbol` is returned.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
- `cudaGetSymbolAddress [C++ API]`, `cudaGetSymbolSize [C API]`

```cpp
template < class T, int dim, enum cudaTextureReadMode readMode >
__host__ cudaError_t cudaGetTextureAlignmentOffset(size_t *offset, const textureTdimreadMode tex)
```

[C++ API] Get the alignment offset of a texture

**Parameters**
- `offset` - Offset of texture reference in bytes
- `tex` - Texture to get offset of

**Returns**
- `cudaSuccess`, `cudaErrorInvalidTexture`, `cudaErrorInvalidTextureBinding`

**Description**
Returns in `*offset` the offset that was returned when texture reference `tex` was bound.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
- `cudaCreateChannelDesc [C++ API]`, `cudaGetChannelDesc`, `cudaGetTextureReference`, `cudaBindTexture [C++ API]`, `cudaBindTexture [C++ API, inherited channel descriptor]`
template < class T > __host__ cudaError_t cudaGraphAddMemcpyNodeFromSymbol (cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, void *dst, const T symbol, size_t count, size_t offset, cudaMemcpyKind kind)

Creates a memcpy node to copy from a symbol on the device and adds it to a graph.

Parameters

pGraphNode
  - Returns newly created node

graph
  - Graph to which to add the node

pDependencies
  - Dependencies of the node

numDependencies
  - Number of dependencies

dst
  - Destination memory address

symbol
  - Device symbol address

count
  - Size in bytes to copy

offset
  - Offset from start of symbol in bytes

kind
  - Type of transfer

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Creates a new memcpy node to copy from symbol and adds it to graph with numDependencies dependencies specified via pDependencies. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph.
pDependencies may not have any duplicate entries. A handle to the new node will be returned in pGraphNode.

When the graph is launched, the node will copy count bytes from the memory area pointed to by offset bytes from the start of symbol symbol to the memory area pointed to by dst. The memory areas may not overlap. symbol is a variable that resides in global or constant memory space. kind can be either cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing.

Memcpy nodes have some additional restrictions with regards to managed memory, if the system contains at least one device which has a zero value for the device attribute cudaDevAttrConcurrentManagedAccess.

Note:

‣ Graph objects are not threadsafe. More here.
‣ Note that this function may also return error codes from previous, asynchronous launches.
‣ Note that this function may also return cudaMemcpyInitializationError, cudaMemcpyInsufficientDriver or cudaMemcpyNoDevice if this call tries to initialize internal CUDA RT state.
‣ Note that as specified by cudaMemcpyFromSymbol no CUDA function may be called from callback. cudaMemcpyNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
template < class T > __host__ cudaError_t cudaGraphAddMemcpyNodeToSymbol(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, const T symbol, const void *src, size_t count, size_t offset, cudaMemcpyKind kind)

Creates a memcpy node to copy to a symbol on the device and adds it to a graph.

Parameters

pGraphNode
- Returns newly created node
graph
- Graph to which to add the node
pDependencies
- Dependencies of the node
numDependencies
- Number of dependencies
symbol
- Device symbol address
src
- Source memory address
count
- Size in bytes to copy
offset
- Offset from start of symbol in bytes
kind
- Type of transfer

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Creates a new memcpy node to copy to symbol and adds it to graph with numDependencies dependencies specified via pDependencies. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. pDependencies may not have any duplicate entries. A handle to the new node will be returned in pGraphNode.
When the graph is launched, the node will copy `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing.

Memcpy nodes have some additional restrictions with regards to managed memory, if the system contains at least one device which has a zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`.

Note:
- Graph objects are not threadsafe. [More here](#).
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
- `cudaMemcpyToSymbol`, `cudaGraphAddMemcpyNode`, `cudaGraphAddMemcpyNodeFromSymbol`, `cudaGraphMemcpyNodeGetParams`, `cudaGraphMemcpyNodeSetParams`, `cudaGraphMemcpyNodeSetParamsToSymbol`, `cudaGraphMemcpyNodeSetParamsFromSymbol`, `cudaGraphCreate`, `cudaGraphDestroyNode`, `cudaGraphAddChildGraphNode`, `cudaGraphAddEmptyNode`, `cudaGraphAddKernelNode`, `cudaGraphAddHostNode`, `cudaGraphAddMemsetNode`
template < class T > __host__cudaError_t
cudaGraphExecMemcpyNodeSetParamsFromSymbol
(cudaGraphExec_t hGraphExec, cudaGraphNode_t
node, void *dst, const T symbol, size_t count, size_t
offset, cudaMemcpyKind kind)

Sets the parameters for a memcpy node in the given graphExec to copy from a symbol on the
device.

Parameters

hGraphExec
- The executable graph in which to set the specified node

node
- Memcpy node from the graph which was used to instantiate graphExec
dst
- Destination memory address

symbol
- Device symbol address

count
- Size in bytes to copy

offset
- Offset from start of symbol in bytes

kind
- Type of transfer

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Updates the work represented by node in hGraphExec as though node had contained the
given params at instantiation. node must remain in the graph which was used to instantiate
hGraphExec. Changed edges to and from node are ignored.

symbol and dst must be allocated from the same contexts as the original source and
destination memory. The instantiation-time memory operands must be 1-dimensional. Zero-
length operations are not supported.

The modifications only affect future launches of hGraphExec. Already enqueued or running
launches of hGraphExec are not affected by this call. node is also not modified by this call.

Returns cudaErrorInvalidValue if the memory operands’ mappings changed or the original
memory operands are multidimensional.
Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaGraphAddMemcpyNode, cudaGraphAddMemcpyNodeFromSymbol, cudaGraphMemcpynodeSetParams, cudaGraphMemcpynodeSetParamsFromSymbol, cudaGraphInstantiate, cudaGraphExecMemcpyNodeSetParams, cudaGraphExecMemcpyNodeSetParamsToSymbol, cudaGraphExecKernelNodeSetParams, cudaGraphExecMemsetNodeSetParams, cudaGraphExecHostNodeSetParams

template < class T > __host__ cudaError_t cudaGraphExecMemcpyNodeSetParamsToSymbol(cudaGraphExec_t hGraphExec, cudaGraphNode_t node, const T symbol, const void *src, size_t count, size_t offset, cudaMemcpyKind kind)

Sets the parameters for a memcpy node in the given graphExec to copy to a symbol on the device.

Parameters

hGraphExec
- The executable graph in which to set the specified node
node
- Memcpy node from the graph which was used to instantiate graphExec
symbol
- Device symbol address
src
- Source memory address
count
- Size in bytes to copy
offset
- Offset from start of symbol in bytes

kind
- Type of transfer

Returns
cudaSuccess, cudaErrorInvalidValue

Description
Updates the work represented by node in hGraphExec as though node had contained the given params at instantiation. node must remain in the graph which was used to instantiate hGraphExec. Changed edges to and from node are ignored.

src and symbol must be allocated from the same contexts as the original source and destination memory. The instantiation-time memory operands must be 1-dimensional. Zero-length operations are not supported.

The modifications only affect future launches of hGraphExec. Already enqueued or running launches of hGraphExec are not affected by this call. node is also not modified by this call.

Returns cudaErrorInvalidValue if the memory operands’ mappings changed or the original memory operands are multidimensional.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaMemcpyNodeToSymbol, no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
template < class T > __host__ cudaError_t cudaGraphMemcpyNodeSetParamsFromSymbol (cudaGraphNode_t node, void *dst, const T symbol, size_t count, size_t offset, cudaMemcpyKind kind)

Sets a memcpy node’s parameters to copy from a symbol on the device.

Parameters

**node**
- Node to set the parameters for

**dst**
- Destination memory address

**symbol**
- Device symbol address

**count**
- Size in bytes to copy

**offset**
- Offset from start of symbol in bytes

**kind**
- Type of transfer

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Sets the parameters of memcpy node **node** to the copy described by the provided parameters.

When the graph is launched, the node will copy **count** bytes from the memory area pointed to by **offset** bytes from the start of symbol **symbol** to the memory area pointed to by **dst**. The memory areas may not overlap. **symbol** is a variable that resides in global or constant memory space. **kind** can be either cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing.

**Note:**
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
See also:

cudaMemcpyFromSymbol, cudaMemcpyNodeSetParams,
cudaGraphMemcpyNodeSetParamsToSymbol, cudaMemcpyNodeSetParamsToSymbol,
cudaGraphMemcpyNodeGetParams

template < class T > __host__cudaError_t
cudaGraphMemcpyNodeSetParamsToSymbol
cudaGraphNode_t node, const T symbol, const void *src, size_t count, size_t offset, cudaMemcpyKind kind)

Sets a memcpy node’s parameters to copy to a symbol on the device.

Parameters

node
  - Node to set the parameters for

symbol
  - Device symbol address

src
  - Source memory address

count
  - Size in bytes to copy

offset
  - Offset from start of symbol in bytes

kind
  - Type of transfer

Returns
cudaSuccess, cudaErrorInvalidValue

Description

Sets the parameters of memcpy node node to the copy described by the provided parameters.
When the graph is launched, the node will copy `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing.

**Note:**
- Graph objects are not threadsafe. [More here](#).
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**
- `cudaMemcpyToSymbol`, `cudaGraphMemcpyNodeSetParams`,
- `cudaGraphMemcpyNodeSetParamsFromSymbol`, `cudaGraphAddMemcpyNode`,
- `cudaGraphMemcpyNodeGetParams`

```cpp
template < class T > __host__ cudaError_t
cudaLaunchCooperativeKernel (const T *func, dim3 gridDim, dim3 blockDim, void **args, size_t sharedMem, cudaStream_t stream)
```

Launches a device function.

**Parameters**

- `func` - Device function symbol
- `gridDim` - Grid dimensions
- `blockDim` - Block dimensions
args
- Arguments

sharedMem
- Shared memory (defaults to 0)

stream
- Stream identifier (defaults to NULL)

Returns
cudaSuccess, cudaErrorInvalidDeviceFunction, cudaErrorInvalidConfiguration, cudaErrorLaunchFailure, cudaErrorLaunchTimeout, cudaErrorLaunchOutOfResources, cudaErrorSharedObjectInitFailed

Description

The device on which this kernel is invoked must have a non-zero value for the device attribute cudaDevAttrCooperativeLaunch.

The total number of blocks launched cannot exceed the maximum number of blocks per multiprocessor as returned by cudaOccupancyMaxActiveBlocksPerMultiprocessor (or cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags) times the number of multiprocessors as specified by the device attribute cudaDevAttrMultiProcessorCount.

The kernel cannot make use of CUDA dynamic parallelism.

If the kernel has N parameters the args should point to array of N pointers. Each pointer, from args[0] to args[N - 1], point to the region of memory from which the actual parameter will be copied.

sharedMem sets the amount of dynamic shared memory that will be available to each thread block.

stream specifies a stream the invocation is associated to.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

cudaLaunchCooperativeKernel [ C API]

template < class T > __host__ cudaError_t
cudaLaunchKernel (const T *func, dim3 gridDim, dim3 blockDim, void **args, size_t sharedMem, cudaStream_t stream)

Launches a device function.

Parameters

- **func**: Device function symbol
- **gridDim**: Grid dimensions
- **blockDim**: Block dimensions
- **args**: Arguments
- **sharedMem**: Shared memory (defaults to 0)
- **stream**: Stream identifier (defaults to NULL)

Returns


Description


If the kernel has N parameters the `args` should point to array of N pointers. Each pointer, from `args[0]` to `args[N - 1]`, point to the region of memory from which the actual parameter will be copied.
sharedMem sets the amount of dynamic shared memory that will be available to each thread block.

stream specifies a stream the invocation is associated to.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

cudaLaunchKernel (C API)

__host__ cudaError_t cudaMallocAsync (void **ptr, size_t size, cudaMemPool_t memPool, cudaStream_t stream)
Allocate from a pool.

Description
This is an alternate spelling for cudaMemcpyFromPoolAsync made available through operator overloading.

See also:
cudaMallocFromPoolAsync, cudaMemcpyAsync (C API)

__host__ cudaError_t cudaMallocHost (void **ptr, size_t size, unsigned int flags)
[C++ API] Allocates page-locked memory on the host

Parameters
ptr
- Device pointer to allocated memory
size
- Requested allocation size in bytes

flags
- Requested properties of allocated memory

Returns
cudaSuccess, cudaErrorMemoryAllocation

Description
Allocates \texttt{size} bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as \texttt{cudaMemcpy()}. Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as \texttt{malloc()}. Allocating excessive amounts of pinned memory may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to allocate staging areas for data exchange between host and device.

The \texttt{flags} parameter enables different options to be specified that affect the allocation, as follows.

- \texttt{cudaHostAllocDefault}: This flag’s value is defined to be 0.
- \texttt{cudaHostAllocPortable}: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
- \texttt{cudaHostAllocMapped}: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling \texttt{cudaHostGetDevicePointer()}. The \texttt{cudaHostAllocMapped} flag may be specified on CUDA contexts for devices that do not support mapped pinned memory. The failure is deferred to \texttt{cudaHostGetDevicePointer()} because the memory may be mapped into other CUDA contexts via the \texttt{cudaHostAllocPortable} flag.
- \texttt{cudaHostAllocWriteCombined}: Allocates the memory as write-combined (WC). WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs. WC memory is a good option for buffers that will be written by the CPU and read by the device via mapped pinned memory or host->device transfers.

All of these flags are orthogonal to one another: a developer may allocate memory that is portable, mapped and/or write-combined with no restrictions.

\texttt{cudaSetDeviceFlags()} must have been called with the \texttt{cudaDeviceMapHost} flag in order for the \texttt{cudaHostAllocMapped} flag to have any effect.

Memory allocated by this function must be freed with \texttt{cudaFreeHost()}. 
Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaSetDeviceFlags, cudaMallocHost [ C API], cudaFreeHost, cudaHostAlloc

template < class T > __host__ cudaError_t cudaMallocManaged (T **devPtr, size_t size, unsigned int flags)

Allocates memory that will be automatically managed by the Unified Memory system.

**Parameters**

- **devPtr**
  - Pointer to allocated device memory
- **size**
  - Requested allocation size in bytes
- **flags**
  - Must be either `cudaMemAttachGlobal` or `cudaMemAttachHost` (defaults to `cudaMemAttachGlobal`)

**Returns**

cudaSuccess, cudaErrorMemoryAllocation, cudaErrorNotSupported, cudaErrorInvalidValue

**Description**

Allocates size bytes of managed memory on the device and returns in *devPtr* a pointer to the allocated memory. If the device doesn’t support allocating managed memory, `cudaErrorNotSupported` is returned. Support for managed memory can be queried using the device attribute `cudaDevAttrManagedMemory`. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. If size is 0, `cudaMallocManaged` returns `cudaErrorInvalidValue`. The pointer is valid on the CPU and on all GPUs in the system that
support managed memory. All accesses to this pointer must obey the Unified Memory programming model.

flags specifies the default stream association for this allocation. flags must be one of cudaMemAttachGlobal or cudaMemAttachHost. The default value for flags is cudaMemAttachGlobal. If cudaMemAttachGlobal is specified, then this memory is accessible from any stream on any device. If cudaMemAttachHost is specified, then the allocation should not be accessed from devices that have a zero value for the device attribute cudaDevAttrConcurrentManagedAccess; an explicit call to cudaStreamAttachMemAsync will be required to enable access on such devices.

If the association is later changed via cudaStreamAttachMemAsync to a single stream, the default association, as specified during cudaMemcpyManaged, is restored when that stream is destroyed. For __managed__ variables, the default association is always cudaMemAttachGlobal. Note that destroying a stream is an asynchronous operation, and as a result, the change to default association won’t happen until all work in the stream has completed.

Memory allocated with cudaMemcpyManaged should be released with cudaFree.

Device memory oversubscription is possible for GPUs that have a non-zero value for the device attribute cudaDevAttrConcurrentManagedAccess. Managed memory on such GPUs may be evicted from device memory to host memory at any time by the Unified Memory driver in order to make room for other allocations.

In a multi-GPU system where all GPUs have a non-zero value for the device attribute cudaDevAttrConcurrentManagedAccess, managed memory may not be populated when this API returns and instead may be populated on access. In such systems, managed memory can migrate to any processor’s memory at any time. The Unified Memory driver will employ heuristics to maintain data locality and prevent excessive page faults to the extent possible. The application can also guide the driver about memory usage patterns via cudaMemAdvise. The application can also explicitly migrate memory to a desired processor’s memory via cudaMemcpyPrefetchAsync.

In a multi-GPU system where all of the GPUs have a zero value for the device attribute cudaDevAttrConcurrentManagedAccess and all the GPUs have peer-to-peer support with each other, the physical storage for managed memory is created on the GPU which is active at the time cudaMemcpyManaged is called. All other GPUs will reference the data at reduced bandwidth via peer mappings over the PCIe bus. The Unified Memory driver does not migrate memory among such GPUs.

In a multi-GPU system where not all GPUs have peer-to-peer support with each other and where the value of the device attribute cudaDevAttrConcurrentManagedAccess is zero for at least one of those GPUs, the location chosen for physical storage of managed memory is system-dependent.

- On Linux, the location chosen will be device memory as long as the current set of active contexts are on devices that either have peer-to-peer support with each other or have a
non-zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`. If there is an active context on a GPU that does not have a non-zero value for that device attribute and it does not have peer-to-peer support with the other devices that have active contexts on them, then the location for physical storage will be ‘zero-copy’ or host memory. Note that this means that managed memory that is located in device memory is migrated to host memory if a new context is created on a GPU that doesn’t have a non-zero value for the device attribute and does not support peer-to-peer with at least one of the other devices that has an active context. This in turn implies that context creation may fail if there is insufficient host memory to migrate all managed allocations.

‣ On Windows, the physical storage is always created in ‘zero-copy’ or host memory. All GPUs will reference the data at reduced bandwidth over the PCIe bus. In these circumstances, use of the environment variable CUDA_VISIBLE_DEVICES is recommended to restrict CUDA to only use those GPUs that have peer-to-peer support. Alternatively, 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 the CUDA programming guide under the “CUDA environment variables” section.

‣ On ARM, managed memory is not available on discrete gpu with Drive PX-2.

**Note:**
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**
cudaMallocPitch, cudaFree, cudaMallocArray, cudaFreeArray, cudaMalloc3D, cudaMalloc3DArray, cudaMallocHost [C API], cudaFreeHost, cudaHostAlloc, cudaDeviceGetAttribute, cudaStreamAttachMemAsync
template < class T > __host__ cudaError_t cudaMemcpyFromSymbol (void *dst, const T symbol, size_t count, size_t offset, cudaMemcpyKind kind)
[C++ API] Copies data from the given symbol on the device

Parameters

dst
- Destination memory address

symbol
- Device symbol reference

count
- Size in bytes to copy

offset
- Offset from start of symbol in bytes

kind
- Type of transfer

Returns

cudaSuccess, cudaMemcpyInvalidValue, cudaMemcpyInvalidSymbol,
cudaMemcpyInvalidMemcpyDirection, cudaMemcpyNoKernelImageForDevice

Description
Copies count bytes from the memory area offset bytes from the start of symbol symbol to the memory area pointed to by dst. The memory areas may not overlap. symbol is a variable that resides in global or constant memory space. kind can be either cudaMemcpyDeviceToHost or cudaMemcpyDeviceToDevice.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.
- Use of a string naming a variable as the symbol parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.
- Note that this function may also return cudaMemcpyInitializationError, cudaMemcpyInsufficientDriver or cudaMemcpyNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
- `cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`

```cpp
template < class T > __host__ cudaError_t
cudaMemcpyFromSymbolAsync (void *dst, const T symbol, size_t count, size_t offset, cudaMemcpyKind kind, cudaStream_t stream)
```

[C++ API] Copies data from the given symbol on the device

**Parameters**

- **dst**
  - Destination memory address
- **symbol**
  - Device symbol reference
- **count**
  - Size in bytes to copy
- **offset**
  - Offset from start of symbol in bytes
- **kind**
  - Type of transfer
- **stream**
  - Stream identifier

**Returns**
- `cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidSymbol`, `cudaErrorInvalidMemcpyDirection`, `cudaErrorNoKernelImageForDevice`

**Description**

Copies `count` bytes from the memory area `offset` bytes from the start of symbol `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either `cudaMemcpyDeviceToHost` or `cudaMemcpyDeviceToDevice`. 
`cudaMemcpyFromSymbolAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero stream argument. If `kind` is `cudaMemcpyDeviceToHost` and `stream` is non-zero, the copy may overlap with operations in other streams.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits **asynchronous** behavior for most use cases.
- Use of a string naming a variable as the `symbol` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`

template < class T > __host__ cudaError_t cudaMemcpyToSymbol (const T symbol, const void *src, size_t count, size_t offset, cudaMemcpyKind kind)

[C++ API] Copies data to the given symbol on the device

**Parameters**

- **symbol**
  - Device symbol reference
- **src**
  - Source memory address
- **count**
  - Size in bytes to copy
offset
- Offset from start of symbol in bytes

kind
- Type of transfer

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSymbol,
cudaErrorInvalidMemcpyDirection, cudaErrorNoKernelImageForDevice

Description
Copies count bytes from the memory area pointed to by src to the memory area
offset bytes from the start of symbol symbol. The memory areas may not overlap.
symbol is a variable that resides in global or constant memory space. kind can be either
cudaMemcpyHostToDevice or cudaMemcpyDeviceToDevice.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.
- Use of a string naming a variable as the symbol parameter was deprecated in CUDA 4.1
  and removed in CUDA 5.0.
- Note that this function may also return cudaErrorInitializationError,
cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal
  CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called
  from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a
  diagnostic in such case.

See also:
cudaMemcpy, cudaMemcpy2D, cudaMemcpy2DToArray, cudaMemcpy2DFromArray,
cudaMemcpy2DArrayToArray, cudaMemcpyFromSymbol, cudaMemcpyAsync,
cudaMemcpy2DAsync, cudaMemcpy2DToArrayAsync, cudaMemcpy2DFromArrayAsync,
cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync
template < class T > __host__ cudaError_t cudaMemcpyToSymbolAsync (const T symbol, const void *src, size_t count, size_t offset, cudaMemcpyKind kind, cudaStream_t stream)

[C++ API] Copies data to the given symbol on the device

Parameters
symbol
- Device symbol reference
src
- Source memory address
count
- Size in bytes to copy
offset
- Offset from start of symbol in bytes
kind
- Type of transfer
stream
- Stream identifier

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSymbol, cudaErrorInvalidMemcpyDirection, cudaErrorNoKernelImageForDevice

Description
Copies count bytes from the memory area pointed to by src to the memory area offset bytes from the start of symbol symbol. The memory areas may not overlap. symbol is a variable that resides in global or constant memory space. kind can be either cudaMemcpyHostToDevice or cudaMemcpyDeviceToDevice.

cudaMemcpyToSymbolAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is cudaMemcpyHostToDevice and stream is non-zero, the copy may overlap with operations in other streams.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
Use of a string naming a variable as the symbol parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.

Note that this function may also return cudaMemcpyInitializationError, cudaMemcpyInsufficientDriver or cudaMemcpyNoDevice if this call tries to initialize internal CUDA RT state.

Note that as specified by cudaMemcpyAddCallback no CUDA function may be called from callback. cudaMemcpyNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudamempy, cudaMemcpy2D, cudaMemcpy2DToArray, cudaMemcpy2DFromArray, cudaMemcpy2DFromArrayToArray, cudaMemcpy2DFromArrayToSymbol, cudaMemcpy2DFromArrayToSymbolAsync, cudaMemcpy2DArrayToArray, cudaMemcpy2DArrayToSymbol, cudaMemcpy2DAsync, cudaMemcpy2DAsyncToArray, cudaMemcpy2DAsyncToArrayAsync, cudaMemcpyAsync, cudaMemcpyFromSymbolAsync

template < class T > __host__cudaError_t
cudaOccupancyAvailableDynamicSMemPerBlock (size_t *dynamicSmemSize, T func, int numBlocks, int blockSize)

Returns dynamic shared memory available per block when launching numBlocks blocks on SM.

Parameters

dynamicSmemSize
- Returned maximum dynamic shared memory
func
- Kernel function for which occupancy is calculated
numBlocks
- Number of blocks to fit on SM
blockSize
- Size of the block

Returns
cudaSuccess, cudaMemcpyInitializationError, cudaMemcpyInsufficientDevice, cudaMemcpyInvalidDeviceFunction, cudaMemcpyInvalidValue, cudaMemcpyUnknown

Description

Returns in *dynamicSmemSize the maximum size of dynamic shared memory to allow numBlocks blocks per SM.
Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

- `cudaOccupancyMaxPotentialBlockSize`
- `cudaOccupancyMaxPotentialBlockSizeWithFlags`
- `cudaOccupancyMaxActiveBlocksPerMultiprocessor`
- `cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags`
- `cudaOccupancyMaxPotentialBlockSizeVariableSMem`
- `cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags`

```cpp
template < class T > __host__ cudaError_t
cudaOccupancyMaxActiveBlocksPerMultiprocessor (int *numBlocks, T func, int blockSize, size_t dynamicSMemSize)
```

Returns occupancy for a device function.

Parameters

- **numBlocks**
  - Returned occupancy
- **func**
  - Kernel function for which occupancy is calculated
- **blockSize**
  - Block size the kernel is intended to be launched with
- **dynamicSMemSize**
  - Per-block dynamic shared memory usage intended, in bytes
Returns

cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidDeviceFunction, cudaErrorInvalidValue, cudaErrorUnknown.

Description

Returns in *numBlocks the maximum number of active blocks per streaming multiprocessor for the device function.

Note:

‣ Note that this function may also return error codes from previous, asynchronous launches.

‣ Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.

‣ Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags

cudaOccupancyMaxPotentialBlockSize

cudaOccupancyMaxPotentialBlockSizeWithFlags

cudaOccupancyMaxPotentialBlockSizeVariableSMem

cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags

cudaOccupancyAvailableDynamicSMemPerBlock

template < class T > __host__cudaError_t
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
(int *numBlocks, T func, int blockSize, size_t dynamicSMemSize, unsigned int flags)

Returns occupancy for a device function with the specified flags.

Parameters

template < class T >

- numBlocks
- Returned occupancy
func
  - Kernel function for which occupancy is calculated

blockSize
  - Block size the kernel is intended to be launched with

dynamicSMemSize
  - Per-block dynamic shared memory usage intended, in bytes

flags
  - Requested behavior for the occupancy calculator

Returns
cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidDeviceFunction, cudaErrorInvalidValue, cudaErrorUnknown.

Description
Returns in *numBlocks the maximum number of active blocks per streaming multiprocessor for the device function.

The flags parameter controls how special cases are handled. Valid flags include:

- `cudaOccupancyDefault`: keeps the default behavior as `cudaOccupancyMaxActiveBlocksPerMultiprocessor`

- `cudaOccupancyDisableCachingOverride`: suppresses the default behavior on platform where global caching affects occupancy. On such platforms, if caching is enabled, but per-block SM resource usage would result in zero occupancy, the occupancy calculator will calculate the occupancy as if caching is disabled. Setting this flag makes the occupancy calculator to return 0 in such cases. More information can be found about this feature in the “Unified L1/Texture Cache” section of the Maxwell tuning guide.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaOccupancyMaxActiveBlocksPerMultiprocessor
cudaOccupancyMaxPotentialBlockSize
template < class T > __host__ cudaError_t
cudaOccupancyMaxPotentialBlockSize (int *minGridSize, int *blockSize, T func, size_t dynamicSMemSize, int blockSizeLimit)

Returns grid and block size that achieves maximum potential occupancy for a device function.

**Parameters**

- **minGridSize**
  - Returned minimum grid size needed to achieve the best potential occupancy
- **blockSize**
  - Returned block size
- **func**
  - Device function symbol
- **dynamicSMemSize**
  - Per-block dynamic shared memory usage intended, in bytes
- **blockSizeLimit**
  - The maximum block size `func` is designed to work with. 0 means no limit.

**Returns**

cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidDeviceFunction, cudaErrorInvalidValue, cudaErrorUnknown.

**Description**

Returns in *minGridSize* and *blockSize* a suggested grid / block size pair that achieves the best potential occupancy (i.e. the maximum number of active warps with the smallest number of blocks).

**Use**

**See also:**

cudaOccupancyMaxPotentialBlockSizeVariableSMem if the amount of per-block dynamic shared memory changes with different block sizes.
Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

- `cudaOccupancyMaxPotentialBlockSizeWithFlags`
- `cudaOccupancyMaxActiveBlocksPerMultiprocessor`
- `cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags`
- `cudaOccupancyMaxPotentialBlockSizeVariableSMem`
- `cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags`
- `cudaOccupancyAvailableDynamicSMemPerBlock`

```cpp
template < typename UnaryFunction,
class T > __host__cudaError_t
cudaOccupancyMaxPotentialBlockSizeVariableSMem
(int *minGridSize, int *blockSize, T func,
UnaryFunction blockSizeToDynamicSMemSize, int
blockSizeLimit)
```

Returns grid and block size that achieves maximum potential occupancy for a device function.

**Parameters**

- **minGridSize**
  - Returned minimum grid size needed to achieve the best potential occupancy
- **blockSize**
  - Returned block size
- **func**
  - Device function symbol
**blockSizeToDynamicSMemSize**
- A unary function / functor that takes block size, and returns the size, in bytes, of dynamic shared memory needed for a block

**blockSizeLimit**
- The maximum block size \( \text{func} \) is designed to work with. 0 means no limit.

**Returns**
cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidDeviceFunction, cudaErrorInvalidValue, cudaErrorUnknown.

**Description**
Returns in \*minGridSize\* and \*blocksize\* a suggested grid / block size pair that achieves the best potential occupancy (i.e. the maximum number of active warps with the smallest number of blocks).

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**
cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags
cudaOccupancyMaxActiveBlocksPerMultiprocessor
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
cudaOccupancyMaxPotentialBlockSize
cudaOccupancyMaxPotentialBlockSizeWithFlags
cudaOccupancyAvailableDynamicSMemPerBlock
template < typename UnaryFunction,
            class T > __host__ cudaError_t
cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags(
    int *minGridSize, int *blockSize, T func,
    UnaryFunction blockSizeToDynamicSMemSize, int blockSizeLimit, unsigned int flags)

Returns grid and block size that achieves maximum potential occupancy for a device function.

Parameters

minGridSize
- Returned minimum grid size needed to achieve the best potential occupancy

blockSize
- Returned block size

func
- Device function symbol

blockSizeToDynamicSMemSize
- A unary function / functor that takes block size, and returns the size, in bytes, of dynamic shared memory needed for a block

blockSizeLimit
- The maximum block size func is designed to work with. 0 means no limit.

flags
- Requested behavior for the occupancy calculator

Returns

cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidDeviceFunction,
cudaErrorInvalidValue, cudaErrorUnknown.

Description

Returns in *minGridSize and *blocksize a suggested grid / block size pair that achieves the best potential occupancy (i.e. the maximum number of active warps with the smallest number of blocks).

The flags parameter controls how special cases are handled. Valid flags include:

- **cudaOccupancyDefault**: keeps the default behavior as
cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags

- **cudaOccupancyDisableCachingOverride**: This flag suppresses the default behavior on platform where global caching affects occupancy. On such platforms, if caching is enabled, but per-block SM resource usage would result in zero occupancy, the occupancy
calculator will calculate the occupancy as if caching is disabled. Setting this flag makes the occupancy calculator to return 0 in such cases. More information can be found about this feature in the “Unified L1/Texture Cache” section of the Maxwell tuning guide.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaOccupancyMaxPotentialBlockSizeVariableSMem
cudaOccupancyMaxActiveBlocksPerMultiprocessor
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
cudaOccupancyMaxPotentialBlockSize
cudaOccupancyMaxPotentialBlockSizeWithFlags
cudaOccupancyAvailableDynamicSMemPerBlock

template < class T > __host__ cudaError_t cudaOccupancyMaxPotentialBlockSizeWithFlags (int *minGridSize, int *blockSize, T func, size_t dynamicSMemSize, int blockSizeLimit, unsigned int flags)

Returns grid and block size that achieved maximum potential occupancy for a device function with the specified flags.

Parameters

- **minGridSize**
  - Returned minimum grid size needed to achieve the best potential occupancy
- **blockSize**
  - Returned block size
**func**
- Device function symbol

**dynamicSMemSize**
- Per-block dynamic shared memory usage intended, in bytes

**blockSizeLimit**
- The maximum block size `func` is designed to work with. 0 means no limit.

**flags**
- Requested behavior for the occupancy calculator

**Returns**
cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidDeviceFunction, cudaErrorInvalidValue, cudaErrorUnknown.

**Description**
Returns in `*minGridSize` and `*blocksize` a suggested grid / block size pair that achieves the best potential occupancy (i.e. the maximum number of active warps with the smallest number of blocks).

The `flags` parameter controls how special cases are handle. Valid flags include:

- `cudaOccupancyDefault`: keeps the default behavior as `cudaOccupancyMaxPotentialBlockSize`
- `cudaOccupancyDisableCachingOverride`: This flag suppresses the default behavior on platform where global caching affects occupancy. On such platforms, if caching is enabled, but per-block SM resource usage would result in zero occupancy, the occupancy calculator will calculate the occupancy as if caching is disabled. Setting this flag makes the occupancy calculator to return 0 in such cases. More information can be found about this feature in the “Unified L1/Texture Cache” section of the Maxwell tuning guide.

**Use**

**See also:**
cudaOccupancyMaxPotentialBlockSizeVariableSMem if the amount of per-block dynamic shared memory changes with different block sizes.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:
cudaOccupancyMaxPotentialBlockSize
cudaOccupancyMaxActiveBlocksPerMultiprocessor
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
cudaOccupancyMaxPotentialBlockSizeVariableSMem
cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags
cudaOccupancyAvailableDynamicSMemPerBlock

template < class T > __host__cudaError_t cudaStreamAttachMemAsync (cudaStream_t stream, T *devPtr, size_t length, unsigned int flags)
Attach memory to a stream asynchronously.

Parameters

stream
- Stream in which to enqueue the attach operation
devPtr
- Pointer to memory (must be a pointer to managed memory or to a valid host-accessible region of system-allocated memory)
length
- Length of memory (defaults to zero)
flags
- Must be one of cudaMemAttachGlobal, cudaMemAttachHost or cudaMemAttachSingle (defaults to cudaMemAttachSingle)

Returns
cudaSuccess, cudaErrorNotReady, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle

Description

Enqueues an operation in stream to specify stream association of length bytes of memory starting from devPtr. This function is a stream-ordered operation, meaning that it is dependent on, and will only take effect when, previous work in stream has completed. Any previous association is automatically replaced.

devPtr must point to an one of the following types of memories:
managed memory declared using the __managed__ keyword or allocated with
\texttt{cudaMallocManaged}.

a valid host-accessible region of system-allocated pageable memory. This type of memory
may only be specified if the device associated with the stream reports a non-zero value for
the device attribute \texttt{cudaDevAttrPageableMemoryAccess}.

For managed allocations, \texttt{length} must be either zero or the entire allocation’s size. Both
indicate that the entire allocation’s stream association is being changed. Currently, it is not
possible to change stream association for a portion of a managed allocation.

For pageable allocations, \texttt{length} must be non-zero.

The stream association is specified using \texttt{flags} which must be one of \texttt{cudaMemAttachGlobal},
\texttt{cudaMemAttachHost} or \texttt{cudaMemAttachSingle}. The default value for \texttt{flags} is
\texttt{cudaMemAttachSingle} If the \texttt{cudaMemAttachGlobal} flag is specified, the memory
can be accessed by any stream on any device. If the \texttt{cudaMemAttachHost} flag is
specified, the program makes a guarantee that it won’t access the memory on the
device from any stream on a device that has a zero value for the device attribute
\texttt{cudaDevAttrConcurrentManagedAccess}. If the \texttt{cudaMemAttachSingle} flag is specified
and \texttt{stream} is associated with a device that has a zero value for the device attribute
\texttt{cudaDevAttrConcurrentManagedAccess}, the program makes a guarantee that it will only
access the memory on the device from \texttt{stream}. It is illegal to attach singly to the NULL
stream, because the NULL stream is a virtual global stream and not a specific stream. An
error will be returned in this case.

When memory is associated with a single stream, the Unified Memory system will allow
CPU access to this memory region so long as all operations in \texttt{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.

Accessing memory on the device from streams that are not associated with it will produce
undefined results. No error checking is performed by the Unified Memory system to ensure
that kernels launched into other streams do not access this region.

It is a program’s responsibility to order calls to \texttt{cudaStreamAttachMemAsync} via events,
synchronization or other means to ensure legal access to memory at all times. Data visibility
and coherency will be changed appropriately for all kernels which follow a stream-association
change.

If \texttt{stream} is destroyed while data is associated with it, the association is removed
and the association reverts to the default visibility of the allocation as specified at
\texttt{cudaMallocManaged}. For __managed__ variables, the default association is always
\texttt{cudaMemAttachGlobal}. Note that destroying a stream is an asynchronous operation, and
as a result, the change to default association won’t happen until all work in the stream has
completed.
template < class T, int dim, enum cudaTextureReadMode readMode >
__host__cudaError_t cudaUnbindTexture (const textureTdimreadMode tex)

[C++ API] Unbinds a texture

Parameters

tex
  - Texture to unbind

Returns

cudaSuccess, cudaErrorInvalidTexture

Description

Unbinds the texture bound to tex. If texref is not currently bound, no operation is performed.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
Modules

CUDA Runtime API

Release Version | 485

6.33. Interactions with the CUDA Driver API

This section describes the interactions between the CUDA Driver API and the CUDA Runtime API.

Primary Contexts

There exists a one to one relationship between CUDA devices in the CUDA Runtime API and CUcontexts in the CUDA Driver API within a process. The specific context which the CUDA Runtime API uses for a device is called the device’s primary context. From the perspective of the CUDA Runtime API, a device and its primary context are synonymous.

Initialization and Tear-Down

CUDA Runtime API calls operate on the CUDA Driver API CUcontext which is current to the calling host thread.

The function cudaSetDevice[] makes the primary context for the specified device current to the calling thread by calling cuCtxSetCurrent[].

The CUDA Runtime API will automatically initialize the primary context for a device at the first CUDA Runtime API call which requires an active context. If no CUcontext is current to the calling thread when a CUDA Runtime API call which requires an active context is made, then the primary context for a device will be selected, made current to the calling thread, and initialized.

The context which the CUDA Runtime API initializes will be initialized using the parameters specified by the CUDA Runtime API functions cudaSetDeviceFlags[], cudaD3D9SetDirect3DDevice[], cudaD3D10SetDirect3DDevice[], cudaD3D11SetDirect3DDevice[], cudaGLSetGLDevice[], and cudaVDPAUSetVDPAUDevice[].

Note that these functions will fail with cudaErrorSetOnActiveProcess if they are called when the primary context for the specified device has already been initialized. (or if the current device has already been initialized, in the case of cudaSetDeviceFlags[]).
Primary contexts will remain active until they are explicitly deinitialized using `cudaDeviceReset()`. The function `cudaDeviceReset()` will deinitialize the primary context for the calling thread’s current device immediately. The context will remain current to all of the threads that it was current to. The next CUDA Runtime API call on any thread which requires an active context will trigger the reinitialization of that device’s primary context.

Note that primary contexts are shared resources. It is recommended that the primary context not be reset except just before exit or to recover from an unspecified launch failure.

**Context Interoperability**

Note that the use of multiple `CUcontext`s per device within a single process will substantially degrade performance and is strongly discouraged. Instead, it is highly recommended that the implicit one-to-one device-to-context mapping for the process provided by the CUDA Runtime API be used.

If a non-primary `CUcontext` created by the CUDA Driver API is current to a thread then the CUDA Runtime API calls to that thread will operate on that `CUcontext`, with some exceptions listed below. Interoperability between data types is discussed in the following sections.

The function `cudaPointerGetAttributes()` will return the error `cudaErrorIncompatibleDriverContext` if the pointer being queried was allocated by a non-primary context. The function `cudaDeviceEnablePeerAccess()` and the rest of the peer access API may not be called when a non-primary `CUcontext` is current. To use the pointer query and peer access APIs with a context created using the CUDA Driver API, it is necessary that the CUDA Driver API be used to access these features.

All CUDA Runtime API state (e.g., global variables’ addresses and values) travels with its underlying `CUcontext`. In particular, if a `CUcontext` is moved from one thread to another then all CUDA Runtime API state will move to that thread as well.

Please note that attaching to legacy contexts (those with a version of 3010 as returned by `cuCtxGetApiVersion()`) is not possible. The CUDA Runtime will return `cudaErrorIncompatibleDriverContext` in such cases.

**Interactions between CUstream and cudaStream_t**

The types `CUstream` and `cudaStream_t` are identical and may be used interchangeably.

**Interactions between CUevent and cudaEvent_t**

The types `CUevent` and `cudaEvent_t` are identical and may be used interchangeably.

**Interactions between CUarray and cudaArray_t**

The types `CUarray` and `struct cudaArray *` represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a `CUarray` in a CUDA Runtime API function which takes a struct `cudaArray *`, it is necessary to explicitly cast the `CUarray` to a struct `cudaArray *`. 
In order to use a struct cudaArray * in a CUDA Driver API function which takes a CUarray, it is necessary to explicitly cast the struct cudaArray * to a CUarray.

Interactions between CUgraphicsResource and cudaGraphicsResource_t

The types CUgraphicsResource and cudaGraphicsResource_t represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a CUgraphicsResource in a CUDA Runtime API function which takes a cudaGraphicsResource_t, it is necessary to explicitly cast the CUgraphicsResource to a cudaGraphicsResource_t.

In order to use a cudaGraphicsResource_t in a CUDA Driver API function which takes a CUgraphicsResource, it is necessary to explicitly cast the cudaGraphicsResource_t to a CUgraphicsResource.

Interactions between CUtexObject * and cudaTextureObject_t

The types CUtexObject * and cudaTextureObject_t represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a CUtexObject * in a CUDA Runtime API function which takes a cudaTextureObject_t, it is necessary to explicitly cast the CUtexObject * to a cudaTextureObject_t.

In order to use a cudaTextureObject_t in a CUDA Driver API function which takes a CUtexObject *, it is necessary to explicitly cast the cudaTextureObject_t to a CUtexObject *.

Interactions between CUsurfObject * and cudaSurfaceObject_t

The types CUsurfObject * and cudaSurfaceObject_t represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a CUsurfObject * in a CUDA Runtime API function which takes a cudaSurfaceObject_t, it is necessary to explicitly cast the CUsurfObject * to a cudaSurfaceObject_t.

In order to use a cudaSurfaceObject_t in a CUDA Driver API function which takes a CUsurfObject *, it is necessary to explicitly cast the cudaSurfaceObject_t to a CUsurfObject *.

Interactions between CUfunction and cudaFunction_t

The types CUfunction and cudaFunction_t represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a cudaFunction_t in a CUDA Driver API function which takes a CUfunction, it is necessary to explicitly cast the cudaFunction_t to a CUfunction.
__host__cudaError_t cudaMemcpy (cudaDevice_t *devicePtr, const void *srcPtr, void *dstPtr, size_t size, cudaMemcpyKind kind)

Copy data from host to device.

Parameters

devicePtr - Pointer to device context

srcPtr - Pointer to the source host data

dstPtr - Pointer to the destination device data

size - Number of bytes to copy

kind - Indicates the type of memory copy operation

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation

description

Copy data from host to device.

6.34. Profiler Control [DEPRECATED]

This section describes the profiler control functions of the CUDA runtime application programming interface.

__host__cudaError_t cudaProfilerInitialize (const char *configFile, const char *outputFile, cudaOutputMode_t outputMode)

Initialize the CUDA profiler.

Parameters

cconfigFile - Name of the config file that lists the counters/options for profiling.

outputFile - Name of the outputFile where the profiling results will be stored.

outputMode - OutputMode, can be cudaKeyValuePair OR cudaCSV.

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorProfilerDisabled
**Description**

**Deprecated**

Using this API user can initialize the CUDA profiler by specifying the configuration file, output file and output file format. This API is generally used to profile different set of counters by looping the kernel launch. The `configFile` parameter can be used to select profiling options including profiler counters. Refer to the “Compute Command Line Profiler User Guide” for supported profiler options and counters.

Limitation: The CUDA profiler cannot be initialized with this API if another profiling tool is already active, as indicated by the `cudaErrorProfilerDisabled` return code.

Typical usage of the profiling APIs is as follows:

```c
for each set of counters/options {
    cudaProfilerInitialize(); // Initialize profiling, set the counters/options in the config file...
    cudaProfilerStart(); // code to be profiled
    cudaProfilerStop(); // code to be profiled
}
```

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

`cudaProfilerStart`, `cudaProfilerStop`, `cuProfilerInitialize`

### 6.35. Profiler Control

This section describes the profiler control functions of the CUDA runtime application programming interface.

```c
__host__cudaError_t cudaProfilerStart (void)
```

Enable profiling.

**Returns**

`cudaSuccess`

**Description**

Enables profile collection by the active profiling tool for the current context. If profiling is already enabled, then `cudaProfilerStart()` has no effect.

cudaProfilerStart and cudaProfilerStop APIs are used to programmatically control the profiling granularity by allowing profiling to be done only on selective pieces of code.
__host__cudaError_t cudaProfilerStop (void)

Disable profiling.

Returns

cudaSuccess

Description

Disables profile collection by the active profiling tool for the current context. If profiling is already disabled, then cudaProfilerStop() has no effect.

cudaProfilerStart and cudaProfilerStop APIs are used to programmatically control the profiling granularity by allowing profiling to be done only on selective pieces of code.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaProfilerInitialize, cudaProfilerStart, cuProfilerStop

6.36. Data types used by CUDA Runtime
struct cudaAccessPolicyWindow
struct cudaArraySparseProperties
struct cudaChannelFormatDesc
struct cudaDeviceProp
struct cudaEglFrame
struct cudaEglPlaneDesc
struct cudaExtent
struct cudaExternalMemoryBufferDesc
struct cudaExternalMemoryHandleDesc
struct cudaExternalMemoryMipmappedArrayDesc
struct cudaExternalSemaphoreHandleDesc
struct cudaExternalSemaphoreSignalNodeParams
struct cudaExternalSemaphoreSignalParams
struct cudaExternalSemaphoreWaitNodeParams
struct cudaExternalSemaphoreWaitParams
struct cudaFuncAttributes
struct cudaHostNodeParams
struct cuda_ipcEventHandle_t
struct cuda_ipcMemHandle_t
union cudaKernelNodeAttrValue
struct cudaKernelNodeParams
struct cudaLaunchParams
struct cudaMemAccessDesc
struct cudaMemAllocNodeParams
struct cudaMemcpy3DParms
struct cudaMemcpy3DPeerParms
struct cudaMemLocation
struct cudaMemPoolProps
struct cudaMemPoolPtrExportData
struct cudaMemcpyParams
struct cudaMemcpy3DParms
struct cudaMemcpy3DPeerParms
struct cudaMemcpyParams
struct cudaMemcpy3DParms
struct cudaMemcpy3DPeerParms
struct cudaPos
struct cudaResourceDesc
struct cudaResourceViewDesc
union cudaStreamAttrValue
struct cudaTextureDesc
struct CUuuid_st
struct surfaceReference
struct textureReference

denum cudaAccessProperty

Specifies performance hint with cudaAccessPolicyWindow for hitProp and missProp members.

Values

cudaAccessPropertyNormal = 0
    Normal cache persistence.
cudaAccessPropertyStreaming = 1
    Streaming access is less likely to persist from cache.
cudaAccessPropertyPersisting = 2
    Persisting access is more likely to persist in cache.

denum cudaCGScope

CUDA cooperative group scope

Values

cudaCGScopeInvalid = 0
    Invalid cooperative group scope
cudaCGScopeGrid = 1
    Scope represented by a grid_group
cudaCGScopeMultiGrid = 2
    Scope represented by a multi_grid_group

denum cudaChannelFormatKind

Channel format kind
Values

(cudaChannelFormatKindSigned = 0)
  Signed channel format
(cudaChannelFormatKindUnsigned = 1)
  Unsigned channel format
(cudaChannelFormatKindFloat = 2)
  Float channel format
(cudaChannelFormatKindNone = 3)
  No channel format
(cudaChannelFormatKindNV12 = 4)

enum cudaComputeMode
CUDA device compute modes

Values

(cudaComputeModeDefault = 0)
  Default compute mode [Multiple threads can use cudaSetDevice() with this device]
(cudaComputeModeExclusive = 1)
  Compute-exclusive-thread mode [Only one thread in one process will be able to use
  cudaSetDevice() with this device]
(cudaComputeModeProhibited = 2)
  Compute-prohibited mode [No threads can use cudaSetDevice() with this device]
(cudaComputeModeExclusiveProcess = 3)
  Compute-exclusive-process mode [Many threads in one process will be able to use
  cudaSetDevice() with this device]

enum cudaDeviceAttr
CUDA device attributes

Values

(cudaDevAttrMaxThreadsPerBlock = 1)
  Maximum number of threads per block
(cudaDevAttrMaxBlockDimX = 2)
  Maximum block dimension X
(cudaDevAttrMaxBlockDimY = 3)
  Maximum block dimension Y
(cudaDevAttrMaxBlockDimZ = 4)
  Maximum block dimension Z
(cudaDevAttrMaxGridDimX = 5)
  Maximum grid dimension X
cudaDevAttrMaxGridDimY = 6
    Maximum grid dimension Y
cudaDevAttrMaxGridDimZ = 7
    Maximum grid dimension Z
cudaDevAttrMaxSharedMemoryPerBlock = 8
    Maximum shared memory available per block in bytes
cudaDevAttrTotalConstantMemory = 9
    Memory available on device for __constant__ variables in a CUDA C kernel in bytes
cudaDevAttrWarpSize = 10
    Warp size in threads
cudaDevAttrMaxPitch = 11
    Maximum pitch in bytes allowed by memory copies
cudaDevAttrMaxRegistersPerBlock = 12
    Maximum number of 32-bit registers available per block
cudaDevAttrClockRate = 13
    Peak clock frequency in kilohertz
cudaDevAttrTextureAlignment = 14
    Alignment requirement for textures
cudaDevAttrGpuOverlap = 15
    Device can possibly copy memory and execute a kernel concurrently
cudaDevAttrMultiProcessorCount = 16
    Number of multiprocessors on device
cudaDevAttrKernelExecTimeout = 17
    Specifies whether there is a run time limit on kernels
cudaDevAttrIntegrated = 18
    Device is integrated with host memory
cudaDevAttrCanMapHostMemory = 19
    Device can map host memory into CUDA address space
cudaDevAttrComputeMode = 20
    Compute mode (See cudaComputeMode for details)
cudaDevAttrMaxTexture1DWidth = 21
    Maximum 1D texture width
cudaDevAttrMaxTexture2DWidth = 22
    Maximum 2D texture width
cudaDevAttrMaxTexture2DHeight = 23
    Maximum 2D texture height
cudaDevAttrMaxTexture3DWidth = 24
    Maximum 3D texture width
cudaDevAttrMaxTexture3DHeight = 25
    Maximum 3D texture height
cudaDevAttrMaxTexture3DDepth = 26
    Maximum 3D texture depth
cudaDevAttrMaxTexture2DLayeredWidth = 27
Maximum 2D layered texture width
\( \text{cudaDevAttrMaxTexture2DLayeredHeight} = 28 \)
Maximum 2D layered texture height
\( \text{cudaDevAttrMaxTexture2DLayeredLayers} = 29 \)
Maximum layers in a 2D layered texture
\( \text{cudaDevAttrSurfaceAlignment} = 30 \)
Alignment requirement for surfaces
\( \text{cudaDevAttrConcurrentKernels} = 31 \)
Device can possibly execute multiple kernels concurrently
\( \text{cudaDevAttrEccEnabled} = 32 \)
Device has ECC support enabled
\( \text{cudaDevAttrPciBusId} = 33 \)
PCI bus ID of the device
\( \text{cudaDevAttrPciDeviceId} = 34 \)
PCI device ID of the device
\( \text{cudaDevAttrTccDriver} = 35 \)
Device is using TCC driver model
\( \text{cudaDevAttrMemoryClockRate} = 36 \)
Peak memory clock frequency in kilohertz
\( \text{cudaDevAttrGlobalMemoryBusWidth} = 37 \)
Global memory bus width in bits
\( \text{cudaDevAttrL2CacheSize} = 38 \)
Size of L2 cache in bytes
\( \text{cudaDevAttrMaxThreadsPerMultiProcessor} = 39 \)
Maximum resident threads per multiprocessor
\( \text{cudaDevAttrAsyncEngineCount} = 40 \)
Number of asynchronous engines
\( \text{cudaDevAttrUnifiedAddressing} = 41 \)
Device shares a unified address space with the host
\( \text{cudaDevAttrMaxTexture1DLayeredWidth} = 42 \)
Maximum 1D layered texture width
\( \text{cudaDevAttrMaxTexture1DLayeredLayers} = 43 \)
Maximum layers in a 1D layered texture
\( \text{cudaDevAttrMaxTexture2DGatherWidth} = 45 \)
Maximum 2D texture width if cudaArrayTextureGather is set
\( \text{cudaDevAttrMaxTexture2DGatherHeight} = 46 \)
Maximum 2D texture height if cudaArrayTextureGather is set
\( \text{cudaDevAttrMaxTexture3DWidthAlt} = 47 \)
Alternate maximum 3D texture width
\( \text{cudaDevAttrMaxTexture3DHeightAlt} = 48 \)
Alternate maximum 3D texture height
\( \text{cudaDevAttrMaxTexture3DDepthAlt} = 49 \)
Alternate maximum 3D texture depth
cudaDevAttrPciDomainId = 50  
PCI domain ID of the device

cudaDevAttrTexturePitchAlignment = 51  
Pitch alignment requirement for textures

cudaDevAttrMaxTextureCubemapWidth = 52  
Maximum cubemap texture width/height

cudaDevAttrMaxTextureCubemapLayeredWidth = 53  
Maximum cubemap layered texture width/height

cudaDevAttrMaxTextureCubemapLayeredLayers = 54  
Maximum layers in a cubemap layered texture

cudaDevAttrMaxSurface1DWidth = 55  
Maximum 1D surface width

cudaDevAttrMaxSurface2DWidth = 56  
Maximum 2D surface width

cudaDevAttrMaxSurface2DHeight = 57  
Maximum 2D surface height

cudaDevAttrMaxSurface3DWidth = 58  
Maximum 3D surface width

cudaDevAttrMaxSurface3DHeight = 59  
Maximum 3D surface height

cudaDevAttrMaxSurface3DDepth = 60  
Maximum 3D surface depth

cudaDevAttrMaxSurface1DLayeredWidth = 61  
Maximum 1D layered surface width

cudaDevAttrMaxSurface1DLayeredLayers = 62  
Maximum layers in a 1D layered surface

cudaDevAttrMaxSurface2DLayeredWidth = 63  
Maximum 2D layered surface width

cudaDevAttrMaxSurface2DLayeredHeight = 64  
Maximum 2D layered surface height

cudaDevAttrMaxSurface2DLayeredLayers = 65  
Maximum layers in a 2D layered surface

cudaDevAttrMaxSurfaceCubemapWidth = 66  
Maximum cubemap surface width

cudaDevAttrMaxSurfaceCubemapLayeredWidth = 67  
Maximum cubemap layered surface width

cudaDevAttrMaxSurfaceCubemapLayeredLayers = 68  
Maximum layers in a cubemap layered surface

cudaDevAttrMaxTexture1DLinearWidth = 69  
Maximum 1D linear texture width

cudaDevAttrMaxTexture2DLinearWidth = 70  
Maximum 2D linear texture width

cudaDevAttrMaxTexture2DLinearHeight = 71  
Maximum 2D linear texture height
Maximum 2D linear texture height
\texttt{cudaDevAttrMaxTexture2DLinearPitch = 72}

Maximum 2D linear texture pitch in bytes
\texttt{cudaDevAttrMaxTexture2DMipmappedWidth = 73}

Maximum mipmapped 2D texture width
\texttt{cudaDevAttrMaxTexture2DMipmappedHeight = 74}

Maximum mipmapped 2D texture height
\texttt{cudaDevAttrComputeCapabilityMajor = 75}

Major compute capability version number
\texttt{cudaDevAttrComputeCapabilityMinor = 76}

Minor compute capability version number
\texttt{cudaDevAttrMaxTexture1DMipmappedWidth = 77}

Maximum mipmapped 1D texture width
\texttt{cudaDevAttrStreamPrioritiesSupported = 78}

Device supports stream priorities
\texttt{cudaDevAttrGlobalL1CacheSupported = 79}

Device supports caching globals in L1
\texttt{cudaDevAttrLocalL1CacheSupported = 80}

Device supports caching locals in L1
\texttt{cudaDevAttrMaxSharedMemoryPerMultiprocessor = 81}

Maximum shared memory available per multiprocessor in bytes
\texttt{cudaDevAttrMaxRegistersPerMultiprocessor = 82}

Maximum number of 32-bit registers available per multiprocessor
\texttt{cudaDevAttrManagedMemory = 83}

Device can allocate managed memory on this system
\texttt{cudaDevAttrIsMultiGpuBoard = 84}

Device is on a multi-GPU board
\texttt{cudaDevAttrMultiGpuBoardGroupID = 85}

Unique identifier for a group of devices on the same multi-GPU board
\texttt{cudaDevAttrHostNativeAtomicSupported = 86}

Link between the device and the host supports native atomic operations
\texttt{cudaDevAttrSingleToDoublePrecisionPerfRatio = 87}

Ratio of single precision performance [in floating-point operations per second] to double precision performance
\texttt{cudaDevAttrPageableMemoryAccess = 88}

Device supports coherently accessing pageable memory without calling cudaHostRegister on it
\texttt{cudaDevAttrConcurrentManagedAccess = 89}

Device can coherently access managed memory concurrently with the CPU
\texttt{cudaDevAttrComputePreemptionSupported = 90}

Device supports Compute Preemption
\texttt{cudaDevAttrCanUseHostPointerForRegisteredMem = 91}

Device can access host registered memory at the same virtual address as the CPU
cudaDevAttrReserved92 = 92
cudaDevAttrReserved93 = 93
cudaDevAttrReserved94 = 94

**cudaDevAttrCooperativeLaunch = 95**

Device supports launching cooperative kernels via [cudaLaunchCooperativeKernel](https://docs.nvidia.com/cuda/index.html).

**cudaDevAttrCooperativeMultiDeviceLaunch = 96**

Deprecated, cudaLaunchCooperativeKernelMultiDevice is deprecated.

**cudaDevAttrMaxSharedMemoryPerBlockOptin = 97**

The maximum optin shared memory per block. This value may vary by chip. See [cudaFuncSetAttribute](https://docs.nvidia.com/cuda/index.html).

**cudaDevAttrCanFlushRemoteWrites = 98**

Device supports flushing of outstanding remote writes.

**cudaDevAttrHostRegisterSupported = 99**

Device supports host memory registration via [cudaHostRegister](https://docs.nvidia.com/cuda/index.html).

**cudaDevAttrPageableMemoryAccessUsesHostPageTables = 100**

Device accesses pageable memory via the host’s page tables.

**cudaDevAttrDirectManagedMemAccessFromHost = 101**

Host can directly access managed memory on the device without migration.

**cudaDevAttrMaxBlocksPerMultiprocessor = 106**

Maximum number of blocks per multiprocessor

**cudaDevAttrMaxPersistingL2CacheSize = 108**

Maximum L2 persisting lines capacity setting in bytes.

**cudaDevAttrMaxAccessPolicyWindowSize = 109**

Maximum value of [cudaAccessPolicyWindow::num_bytes](https://docs.nvidia.com/cuda/index.html).

**cudaDevAttrReservedSharedMemoryPerBlock = 111**

Shared memory reserved by CUDA driver per block in bytes

**cudaDevAttrSparseCudaArraySupported = 112**

Device supports sparse CUDA arrays and sparse CUDA mipmapmed arrays

**cudaDevAttrHostRegisterReadOnlySupported = 113**

Device supports using the [cudaHostRegister](https://docs.nvidia.com/cuda/index.html) flag cudaHostRegisterReadOnly to register memory that must be mapped as read-only to the GPU

**cudaDevAttrMaxTimelineSemaphoreInteropSupported = 114**

External timeline semaphore interop is supported on the device

**cudaDevAttrMemoryPoolsSupported = 115**

Device supports using the [cudaMallocAsync](https://docs.nvidia.com/cuda/index.html) and [cudaMemPool](https://docs.nvidia.com/cuda/index.html) family of APIs

**cudaDevAttrGPUDirectRDMASupported = 116**

Device supports GPUDirect RDMA APIs, like nvidia_p2p_get_pages (see [https://docs.nvidia.com/cuda/gpudirect-rdma](https://docs.nvidia.com/cuda/gpudirect-rdma) for more information)

**cudaDevAttrGPUDirectRDMAFlushWritesOptions = 117**

The returned attribute shall be interpreted as a bitmask, where the individual bits are listed in the [cudaFlushGPUDirectRDMAWritesOptions](https://docs.nvidia.com/cuda/index.html) enum

**cudaDevAttrGPUDirectRDMAWritesOrdering = 118**
GPUDirect RDMA writes to the device do not need to be flushed for consumers within the scope indicated by the returned attribute. See cudaGPUDirectRDMAWritesOrdering for the numerical values returned here.

cudaDevAttrMemoryPoolSupportedHandleTypes = 119
Handle types supported with mempool based IPC

enum cudaDevAttrMax
CUDA device attr max values

Values

cudaDevAttrMemoryPoolSupportedHandleTypes = 119
Handle types supported with mempool based IPC

cudaDevAttrMarker = 120
Markers can be used to mark state changes for compelling

cudaDevAttrMax
CUDA device attr max values

enum cudaDeviceP2PAttr
CUDA device P2P attributes

Values

cudaDevP2PAttrPerformanceRank = 1
A relative value indicating the performance of the link between two devices

cudaDevP2PAttrAccessSupported = 2
Peer access is enabled

cudaDevP2PAttrNativeAtomicSupported = 3
Native atomic operation over the link supported

cudaDevP2PAttrCudaArrayAccessSupported = 4
Accessing CUDA arrays over the link supported

enum cudaEglColorFormat
CUDA EGL Color Format - The different planar and multiplanar formats currently supported for CUDA_EGL interops.

Values

cudaEglColorFormatYUV420Planar = 0
Y, U, V in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

cudaEglColorFormatYUV420SemiPlanar = 1
Y, UV in two surfaces (UV as one surface) with VU byte ordering, width, height ratio same as YUV420Planar.

cudaEglColorFormatYUV422Planar = 2
Y, U, V each in a separate surface, U/V width = 1/2 Y width, U/V height = Y height.

cudaEglColorFormatYUV422SemiPlanar = 3
Y, UV in two surfaces with VU byte ordering, width, height ratio same as YUV422Planar.

cudaEglColorFormatRGB = 4
R/G/B three channels in one surface with BGR byte ordering. Only pitch linear format supported.

cudaEglColorFormatBGR = 5
R/G/B three channels in one surface with RGB byte ordering. Only pitch linear format supported.
cudaEglColorFormatARGB = 6
   R/G/B/A four channels in one surface with BGRA byte ordering.

cudaEglColorFormatRGBA = 7
   R/G/B/A four channels in one surface with ABGR byte ordering.

cudaEglColorFormatL = 8
   single luminance channel in one surface.

cudaEglColorFormatR = 9
   single color channel in one surface.

cudaEglColorFormatYUV444Planar = 10
   Y, U, V in three surfaces, each in a separate surface, U/V width = Y width, U/V height = Y height.

cudaEglColorFormatYUV444SemiPlanar = 11
   Y, UV in two surfaces (UV as one surface) with VU byte ordering, width, height ratio same as YUV444Planar.

cudaEglColorFormatUYVY422 = 12
   Y, U, V in one surface, interleaved as UYV.

cudaEglColorFormatUYYY422 = 13
   Y, U, V in one surface, interleaved as YUY.

cudaEglColorFormatABGR = 14
   R/G/B/A four channels in one surface with RGBA byte ordering.

cudaEglColorFormatBGRA = 15
   R/G/B/A four channels in one surface with ARGB byte ordering.

cudaEglColorFormatA = 16
   Alpha color format - one channel in one surface.

cudaEglColorFormatRG = 17
   R/G color format - two channels in one surface with GR byte ordering.

cudaEglColorFormatAYUV = 18
   Y, U, V, A four channels in one surface, interleaved as VUYA.

cudaEglColorFormatYVU444SemiPlanar = 19
   Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = Y width, U/V height = Y height.

cudaEglColorFormatYVU422SemiPlanar = 20
   Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = Y height.

cudaEglColorFormatYVU420SemiPlanar = 21
   Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

cudaEglColorFormatY10V10U10_444SemiPlanar = 22
   Y10, V10U10 in two surfaces (VU as one surface) with UV byte ordering, U/V width = Y width, U/V height = Y height.

cudaEglColorFormatY10V10U10_420SemiPlanar = 23
   Y10, V10U10 in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.
cudaEglColorFormatY12V12U12_444SemiPlanar = 24
Y12, V12U12 in two surfaces (VU as one surface) with UV byte ordering, U/V width = Y width, U/V height = Y height.

cudaEglColorFormatY12V12U12_420SemiPlanar = 25
Y12, V12U12 in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

cudaEglColorFormatYYUY_ER = 26
Extended Range Y, U, V in one surface, interleaved as YYUY.

cudaEglColorFormatUYVY_ER = 27
Extended Range Y, U, V in one surface, interleaved as UYVY.

cudaEglColorFormatYUYV_ER = 28
Extended Range Y, U, V in one surface, interleaved as UYVY.

cudaEglColorFormatYVUY_ER = 29
Extended Range Y, U, V in one surface, interleaved as VYUY.

cudaEglColorFormatYUVA_ER = 30
Extended Range Y, U, V three channels in one surface, interleaved as VUY. Only pitch linear format supported.

cudaEglColorFormatAYUV_ER = 31
Extended Range Y, U, V, A four channels in one surface, interleaved as AVUY.

cudaEglColorFormatUYA_V = 32
Extended Range Y, U, V, A four channels in one surface, interleaved as VUYA.

cudaEglColorFormatYUV444Planar_ER = 33
Extended Range Y, U, V in three surfaces, U/V width = Y width, U/V height = Y height.

cudaEglColorFormatYUV422Planar_ER = 34
Extended Range Y, U, V in three surfaces, U/V width = 1/2 Y width, U/V height = Y height.

cudaEglColorFormatYUV420Planar_ER = 35
Extended Range Y, U, V in three surfaces, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

cudaEglColorFormatYUV444SemiPlanar_ER = 36
Extended Range Y, UV in two surfaces (UV as one surface) with VU byte ordering, U/V width = Y width, U/V height = Y height.

cudaEglColorFormatYUV422SemiPlanar_ER = 37
Extended Range Y, UV in two surfaces (UV as one surface) with VU byte ordering, U/V width = 1/2 Y width, U/V height = Y height.

cudaEglColorFormatYUV420SemiPlanar_ER = 38
Extended Range Y, UV in two surfaces (UV as one surface) with VU byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

cudaEglColorFormatYUV444Planar_ER = 39
Extended Range Y, V, U in three surfaces, U/V width = Y width, U/V height = Y height.

cudaEglColorFormatYUV422Planar_ER = 40
Extended Range Y, V, U in three surfaces, U/V width = 1/2 Y width, U/V height = Y height.

cudaEglColorFormatYUV420Planar_ER = 41
Extended Range Y, V, U in three surfaces, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

cudaEglColorFormatYUV444SemiPlanar_ER = 42
Extended Range Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = Y width, U/V height = Y height.

**cudaEglColorFormatYVU422SemiPlanar_ER** = 43
Extended Range Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = Y height.

**cudaEglColorFormatYVU420SemiPlanar_ER** = 44
Extended Range Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

**cudaEglColorFormatBayerRGGB** = 45
Bayer format - one channel in one surface with interleaved RGGB ordering.

**cudaEglColorFormatBayerBGGR** = 46
Bayer format - one channel in one surface with interleaved BGGR ordering.

**cudaEglColorFormatBayerGRBG** = 47
Bayer format - one channel in one surface with interleaved GRBG ordering.

**cudaEglColorFormatBayerGBRG** = 48
Bayer format - one channel in one surface with interleaved GBRG ordering.

**cudaEglColorFormatBayer10RGGB** = 49
Bayer10 format - one channel in one surface with interleaved RGGB ordering. Out of 16 bits, 10 bits used 6 bits No-op.

**cudaEglColorFormatBayer10BGGR** = 50
Bayer10 format - one channel in one surface with interleaved BGGR ordering. Out of 16 bits, 10 bits used 6 bits No-op.

**cudaEglColorFormatBayer10GRBG** = 51
Bayer10 format - one channel in one surface with interleaved GRBG ordering. Out of 16 bits, 10 bits used 6 bits No-op.

**cudaEglColorFormatBayer10GBRG** = 52
Bayer10 format - one channel in one surface with interleaved GBRG ordering. Out of 16 bits, 10 bits used 6 bits No-op.

**cudaEglColorFormatBayer12RGGB** = 53
Bayer12 format - one channel in one surface with interleaved RGGB ordering. Out of 16 bits, 12 bits used 4 bits No-op.

**cudaEglColorFormatBayer12BGGR** = 54
Bayer12 format - one channel in one surface with interleaved BGGR ordering. Out of 16 bits, 12 bits used 4 bits No-op.

**cudaEglColorFormatBayer12GRBG** = 55
Bayer12 format - one channel in one surface with interleaved GRBG ordering. Out of 16 bits, 12 bits used 4 bits No-op.

**cudaEglColorFormatBayer12GBRG** = 56
Bayer12 format - one channel in one surface with interleaved GBRG ordering. Out of 16 bits, 12 bits used 4 bits No-op.

**cudaEglColorFormatBayer14RGGB** = 57
Bayer14 format - one channel in one surface with interleaved RGGB ordering. Out of 16 bits, 14 bits used 2 bits No-op.
cudaEglColorFormatBayer14BGGR = 58
Bayer14 format - one channel in one surface with interleaved BGGR ordering. Out of 16 bits, 14 bits used 2 bits No-op.
cudaEglColorFormatBayer14GRBG = 59
Bayer14 format - one channel in one surface with interleaved GRBG ordering. Out of 16 bits, 14 bits used 2 bits No-op.
cudaEglColorFormatBayer14GBRG = 60
Bayer14 format - one channel in one surface with interleaved GBRG ordering. Out of 16 bits, 14 bits used 2 bits No-op.
cudaEglColorFormatBayer20RGGB = 61
Bayer20 format - one channel in one surface with interleaved RGGB ordering. Out of 32 bits, 20 bits used 12 bits No-op.
cudaEglColorFormatBayer20BGGR = 62
Bayer20 format - one channel in one surface with interleaved BGGR ordering. Out of 32 bits, 20 bits used 12 bits No-op.
cudaEglColorFormatBayer20GRBG = 63
Bayer20 format - one channel in one surface with interleaved GRBG ordering. Out of 32 bits, 20 bits used 12 bits No-op.
cudaEglColorFormatBayer20GBRG = 64
Bayer20 format - one channel in one surface with interleaved GBRG ordering. Out of 32 bits, 20 bits used 12 bits No-op.
cudaEglColorFormatYVU444Planar = 65
Y, V, U in three surfaces, each in a separate surface, U/V width = Y width, U/V height = Y height.
cudaEglColorFormatYVU422Planar = 66
Y, V, U in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = Y height.
cudaEglColorFormatYVU420Planar = 67
Y, V, U in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.
cudaEglColorFormatBayerIspRGGB = 68
Nvidia proprietary Bayer ISP format - one channel in one surface with interleaved RGGB ordering and mapped to opaque integer datatype.
cudaEglColorFormatBayerIspBGGR = 69
Nvidia proprietary Bayer ISP format - one channel in one surface with interleaved BGGR ordering and mapped to opaque integer datatype.
cudaEglColorFormatBayerIspGRBG = 70
Nvidia proprietary Bayer ISP format - one channel in one surface with interleaved GRBG ordering and mapped to opaque integer datatype.
cudaEglColorFormatBayerIspGBRG = 71
Nvidia proprietary Bayer ISP format - one channel in one surface with interleaved GBRG ordering and mapped to opaque integer datatype.
cudaEglColorFormatBayerBCCR = 72
Bayer format - one channel in one surface with interleaved BCCR ordering.

\texttt{cudaEglColorFormatBayerRCCB = 73}

Bayer format - one channel in one surface with interleaved RCCB ordering.

\texttt{cudaEglColorFormatBayerCRBC = 74}

Bayer format - one channel in one surface with interleaved CRBC ordering.

\texttt{cudaEglColorFormatBayerCBRC = 75}

Bayer format - one channel in one surface with interleaved CBRC ordering.

\texttt{cudaEglColorFormatBayer10CCCC = 76}

Bayer10 format - one channel in one surface with interleaved CCCC ordering. Out of 16 bits, 10 bits used 6 bits No-op.

\texttt{cudaEglColorFormatBayer12BCCR = 77}

Bayer12 format - one channel in one surface with interleaved BCCR ordering. Out of 16 bits, 12 bits used 4 bits No-op.

\texttt{cudaEglColorFormatBayer12RCCB = 78}

Bayer12 format - one channel in one surface with interleaved RCCB ordering. Out of 16 bits, 12 bits used 4 bits No-op.

\texttt{cudaEglColorFormatBayer12CRBC = 79}

Bayer12 format - one channel in one surface with interleaved CRBC ordering. Out of 16 bits, 12 bits used 4 bits No-op.

\texttt{cudaEglColorFormatBayer12CBRC = 80}

Bayer12 format - one channel in one surface with interleaved CBRC ordering. Out of 16 bits, 12 bits used 4 bits No-op.

\texttt{cudaEglColorFormatBayer12CCCC = 81}

Bayer12 format - one channel in one surface with interleaved CCCC ordering. Out of 16 bits, 12 bits used 4 bits No-op.

\texttt{cudaEglColorFormatY = 82}

Color format for single Y plane.

\texttt{cudaEglColorFormatYUV420SemiPlanar_2020 = 83}

Y, UV in two surfaces (UV as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

\texttt{cudaEglColorFormatYVU420SemiPlanar_2020 = 84}

Y, VU in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

\texttt{cudaEglColorFormatYUV420Planar_2020 = 85}

Y, U, V in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

\texttt{cudaEglColorFormatYVU420Planar_2020 = 86}

Y, V, U in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

\texttt{cudaEglColorFormatYUV420SemiPlanar_709 = 87}

Y, UV in two surfaces (UV as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

\texttt{cudaEglColorFormatYVU420SemiPlanar_709 = 88}

Y, VU in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

\texttt{cudaEglColorFormatYUV420Planar_709 = 89}
Y, U, V in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

cudaEglColorFormatYVU420Planar_709 = 90
Y, V, U in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

cudaEglColorFormatY10V10U10_420SemiPlanar_709 = 91
Y10, V10U10 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

cudaEglColorFormatY10V10U10_420SemiPlanar_2020 = 92
Y10, V10U10 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

cudaEglColorFormatY10V10U10_422SemiPlanar_2020 = 93
Y10, V10U10 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = Y height.

cudaEglColorFormatY10V10U10_422SemiPlanar = 94
Y10, V10U10 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = Y height.

cudaEglColorFormatY10V10U10_422SemiPlanar_709 = 95
Y10, V10U10 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = Y height.

cudaEglColorFormatY_ER = 96
Extended Range Color format for single Y plane.

cudaEglColorFormatY_709_ER = 97
Extended Range Color format for single Y plane.

cudaEglColorFormatY10_ER = 98
Extended Range Color format for single Y10 plane.

cudaEglColorFormatY10_709_ER = 99
Extended Range Color format for single Y10 plane.

cudaEglColorFormatY12_ER = 100
Extended Range Color format for single Y12 plane.

cudaEglColorFormatY12_709_ER = 101
Extended Range Color format for single Y12 plane.

cudaEglColorFormatYUVA = 102
Y, U, V, A four channels in one surface, interleaved as AVUY.

cudaEglColorFormatYYVU = 104
Y, U, V in one surface, interleaved as YYVU.

cudaEglColorFormatVYUY = 105
Y, U, V in one surface, interleaved as VYUY.

cudaEglColorFormatY10V10U10_420SemiPlanar_ER = 106
Extended Range Y10, V10U10 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

cudaEglColorFormatY10V10U10_420SemiPlanar_709_ER = 107
Extended Range Y10, V10U10 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

`cudaEglColorFormatY10V10U10_444SemiPlanar_ER = 108`
Extended Range Y10, V10U10 in two surfaces (VU as one surface) U/V width = Y width, U/V height = Y height.

`cudaEglColorFormatY10V10U10_444SemiPlanar_709_ER = 109`
Extended Range Y10, V10U10 in two surfaces (VU as one surface) U/V width = Y width, U/V height = Y height.

`cudaEglColorFormatY12V12U12_420SemiPlanar_ER = 110`
Extended Range Y12, V12U12 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

`cudaEglColorFormatY12V12U12_420SemiPlanar_709_ER = 111`
Extended Range Y12, V12U12 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

`cudaEglColorFormatY12V12U12_444SemiPlanar_ER = 112`
Extended Range Y12, V12U12 in two surfaces (VU as one surface) U/V width = Y width, U/V height = Y height.

`cudaEglColorFormatY12V12U12_444SemiPlanar_709_ER = 113`
Extended Range Y12, V12U12 in two surfaces (VU as one surface) U/V width = Y width, U/V height = Y height.

**enum cudaEglFrameType**

CUDA EglFrame type - array or pointer

**Values**

`cudaEglFrameTypeArray = 0`
Frame type CUDA array

`cudaEglFrameTypePitch = 1`
Frame type CUDA pointer

**enum cudaEglResourceLocationFlags**

Resource location flags- sysmem or vidmem

For CUDA context on iGPU, since video and system memory are equivalent - these flags will not have an effect on the execution.

For CUDA context on dGPU, applications can use the flag `cudaEglResourceLocationFlags` to give a hint about the desired location.

`cudaEglResourceLocationSysmem` - the frame data is made resident on the system memory to be accessed by CUDA.

`cudaEglResourceLocationVidmem` - the frame data is made resident on the dedicated video memory to be accessed by CUDA.
There may be an additional latency due to new allocation and data migration, if the frame is produced on a different memory.

**Values**

- `cudaEglResourceLocationSysmem = 0x00`  
  Resource location sysmem
- `cudaEglResourceLocationVidmem = 0x01`  
  Resource location vidmem

### enum cudaError

CUDA error types

**Values**

- `cudaSuccess = 0`  
  The API call returned with no errors. In the case of query calls, this also means that the operation being queried is complete [see `cudaEventQuery()` and `cudaStreamQuery()`].
- `cudaErrorInvalidValue = 1`  
  This indicates that one or more of the parameters passed to the API call is not within an acceptable range of values.
- `cudaErrorMemoryAllocation = 2`  
  The API call failed because it was unable to allocate enough memory to perform the requested operation.
- `cudaErrorInitializationError = 3`  
  The API call failed because the CUDA driver and runtime could not be initialized.
- `cudaErrorCudartUnloading = 4`  
  This indicates that a CUDA Runtime API call cannot be executed because it is being called during process shut down, at a point in time after CUDA driver has been unloaded.
- `cudaErrorProfilerDisabled = 5`  
  This indicates profiler is not initialized for this run. This can happen when the application is running with external profiling tools like visual profiler.
- `cudaErrorProfilerNotInitialized = 6`  
  **Deprecated** This error return is deprecated as of CUDA 5.0. It is no longer an error to attempt to enable/disable the profiling via `cudaProfilerStart` or `cudaProfilerStop` without initialization.
- `cudaErrorProfilerAlreadyStarted = 7`  
  **Deprecated** This error return is deprecated as of CUDA 5.0. It is no longer an error to call `cudaProfilerStart()` when profiling is already enabled.
- `cudaErrorProfilerAlreadyStopped = 8`  
  **Deprecated** This error return is deprecated as of CUDA 5.0. It is no longer an error to call `cudaProfilerStop()` when profiling is already disabled.
- `cudaErrorInvalidConfiguration = 9`
This indicates that a kernel launch is requesting resources that can never be satisfied by the current device. Requesting more shared memory per block than the device supports will trigger this error, as will requesting too many threads or blocks. See cudaDeviceProp for more device limitations.

cudaErrorInvalidPitchValue = 12
This indicates that one or more of the pitch-related parameters passed to the API call is not within the acceptable range for pitch.

cudaErrorInvalidSymbol = 13
This indicates that the symbol name/identifier passed to the API call is not a valid name or identifier.

cudaErrorInvalidHostPointer = 16
This indicates that at least one host pointer passed to the API call is not a valid host pointer. Deprecated This error return is deprecated as of CUDA 10.1.

cudaErrorInvalidDevicePointer = 17
This indicates that at least one device pointer passed to the API call is not a valid device pointer. Deprecated This error return is deprecated as of CUDA 10.1.

cudaErrorInvalidTexture = 18
This indicates that the texture passed to the API call is not a valid texture.

cudaErrorInvalidTextureBinding = 19
This indicates that the texture binding is not valid. This occurs if you call cudaGetTextureAlignmentOffset() with an unbound texture.

cudaErrorInvalidChannelDescriptor = 20
This indicates that the channel descriptor passed to the API call is not valid. This occurs if the format is not one of the formats specified by cudaChannelFormatKind, or if one of the dimensions is invalid.

cudaErrorInvalidMemcpyDirection = 21
This indicates that the direction of the memcpy passed to the API call is not one of the types specified by cudaMemcpyKind.

cudaErrorAddressOfConstant = 22
This indicated that the user has taken the address of a constant variable, which was forbidden up until the CUDA 3.1 release. Deprecated This error return is deprecated as of CUDA 3.1. Variables in constant memory may now have their address taken by the runtime via cudaGetSymbolAddress().

cudaErrorTextureFetchFailed = 23
This indicated that a texture fetch was not able to be performed. This was previously used for device emulation of texture operations. Deprecated This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

cudaErrorTextureNotBound = 24
This indicated that a texture was not bound for access. This was previously used for device emulation of texture operations. Deprecated This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

cudaErrorSynchronizationError = 25
This indicated that a synchronization operation had failed. This was previously used for some device emulation functions. **Deprecated** This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

**cudaErrorInvalidFilterSetting = 26**
This indicates that a non-float texture was being accessed with linear filtering. This is not supported by CUDA.

**cudaErrorInvalidNormSetting = 27**
This indicates that an attempt was made to read a non-float texture as a normalized float. This is not supported by CUDA.

**cudaErrorMixedDeviceExecution = 28**
Mixing of device and device emulation code was not allowed. **Deprecated** This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

**cudaErrorNotYetImplemented = 31**
This indicates that the API call is not yet implemented. Production releases of CUDA will never return this error. **Deprecated** This error return is deprecated as of CUDA 4.1.

**cudaErrorMemoryValueTooLarge = 32**
This indicated that an emulated device pointer exceeded the 32-bit address range. **Deprecated** This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

**cudaErrorStubLibrary = 34**
This indicates that the CUDA driver that the application has loaded is a stub library. Applications that run with the stub rather than a real driver loaded will result in CUDA API returning this error.

**cudaErrorInsufficientDriver = 35**
This indicates that the installed NVIDIA CUDA driver is older than the CUDA runtime library. This is not a supported configuration. Users should install an updated NVIDIA display driver to allow the application to run.

**cudaErrorCallRequiresNewerDriver = 36**
This indicates that the API call requires a newer CUDA driver than the one currently installed. Users should install an updated NVIDIA CUDA driver to allow the API call to succeed.

**cudaErrorInvalidSurface = 37**
This indicates that the surface passed to the API call is not a valid surface.

**cudaErrorDuplicateVariableName = 43**
This indicates that multiple global or constant variables [across separate CUDA source files in the application] share the same string name.

**cudaErrorDuplicateTextureName = 44**
This indicates that multiple textures [across separate CUDA source files in the application] share the same string name.

**cudaErrorDuplicateSurfaceName = 45**
This indicates that multiple surfaces [across separate CUDA source files in the application] share the same string name.
cudaErrorDevicesUnavailable = 46
This indicates that all CUDA devices are busy or unavailable at the current time. Devices are often busy/unavailable due to use of cudaComputeModeExclusive, cudaComputeModeProhibited or when long running CUDA kernels have filled up the GPU and are blocking new work from starting. They can also be unavailable due to memory constraints on a device that already has active CUDA work being performed.

cudaErrorIncompatibleDriverContext = 49
This indicates that the current context is not compatible with this the CUDA Runtime. This can only occur if you are using CUDA Runtime/Driver interoperability and have created an existing Driver context using the driver API. The Driver context may be incompatible either because the Driver context was created using an older version of the API, because the Runtime API call expects a primary driver context and the Driver context is not primary, or because the Driver context has been destroyed. Please see Interactions with the CUDA Driver API" for more information.

cudaErrorMissingConfiguration = 52
The device function being invoked (usually via cudaLaunchKernel()) was not previously configured via the cudaConfigureCall() function.

cudaErrorPriorLaunchFailure = 53
This indicated that a previous kernel launch failed. This was previously used for device emulation of kernel launches. Deprecated This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

cudaErrorLaunchMaxDepthExceeded = 65
This error indicates that a device runtime grid launch did not occur because the depth of the child grid would exceed the maximum supported number of nested grid launches.

cudaErrorLaunchFileScopedTex = 66
This error indicates that a grid launch did not occur because the kernel uses file-scoped textures which are unsupported by the device runtime. Kernels launched via the device runtime only support textures created with the Texture Object API’s.

cudaErrorLaunchFileScopedSurf = 67
This error indicates that a grid launch did not occur because the kernel uses file-scoped surfaces which are unsupported by the device runtime. Kernels launched via the device runtime only support surfaces created with the Surface Object API’s.

cudaErrorSyncDepthExceeded = 68
This error indicates that a call to cudaDeviceSynchronize made from the device runtime failed because the call was made at grid depth greater than than either the default (2 levels of grids) or user specified device limit cudaLimitDevRuntimeSyncDepth. To be able to synchronize on launched grids at a greater depth successfully, the maximum nested depth at which cudaDeviceSynchronize will be called must be specified with the cudaLimitDevRuntimeSyncDepth limit to the cudaDeviceSetLimit api before the host-side launch of a kernel using the device runtime. Keep in mind that additional levels of sync depth require the runtime to reserve large amounts of device memory that cannot be used for user allocations.

cudaErrorLaunchPendingCountExceeded = 69
This error indicates that a device runtime grid launch failed because the launch would exceed the limit `cudaLimitDevRuntimePendingLaunchCount`. For this launch to proceed successfully, `cudaDeviceSetLimit` must be called to set the `cudaLimitDevRuntimePendingLaunchCount` to be higher than the upper bound of outstanding launches that can be issued to the device runtime. Keep in mind that raising the limit of pending device runtime launches will require the runtime to reserve device memory that cannot be used for user allocations.

**cudaErrorInvalidDeviceFunction = 98**
The requested device function does not exist or is not compiled for the proper device architecture.

**cudaErrorNoDevice = 100**
This indicates that no CUDA-capable devices were detected by the installed CUDA driver.

**cudaErrorInvalidDevice = 101**
This indicates that the device ordinal supplied by the user does not correspond to a valid CUDA device or that the action requested is invalid for the specified device.

**cudaErrorDeviceNotLicensed = 102**
This indicates that the device doesn’t have a valid Grid License.

**cudaErrorSoftwareValidityNotEstablished = 103**
By default, the CUDA runtime may perform a minimal set of self-tests, as well as CUDA driver tests, to establish the validity of both. Introduced in CUDA 11.2, this error return indicates that at least one of these tests has failed and the validity of either the runtime or the driver could not be established.

**cudaErrorStartupFailure = 127**
This indicates an internal startup failure in the CUDA runtime.

**cudaErrorInvalidKernelImage = 200**
This indicates that the device kernel image is invalid.

**cudaErrorDeviceUninitialized = 201**
This most frequently indicates that there is no context bound to the current thread. This can also be returned if the context passed to an API call is not a valid handle (such as a context that has had `cuCtxDestroy()` invoked on it). This can also be returned if a user mixes different API versions (i.e. 3010 context with 3020 API calls). See `cuCtxGetApiVersion()` for more details.

**cudaErrorMapBufferObjectFailed = 205**
This indicates that the buffer object could not be mapped.

**cudaErrorUnmapBufferObjectFailed = 206**
This indicates that the buffer object could not be unmapped.

**cudaErrorArrayIsMapped = 207**
This indicates that the specified array is currently mapped and thus cannot be destroyed.

**cudaErrorAlreadyMapped = 208**
This indicates that the resource is already mapped.

**cudaErrorNoKernelImageForDevice = 209**
This indicates that there is no kernel image available that is suitable for the device. This can occur when a user specifies code generation options for a particular CUDA source file that do not include the corresponding device configuration.

**cudaErrorAlreadyAcquired = 210**
This indicates that a resource has already been acquired.

**cudaErrorNotMapped = 211**
This indicates that a resource is not mapped.

**cudaErrorNotMappedAsArray = 212**
This indicates that a mapped resource is not available for access as an array.

**cudaErrorNotMappedAsPointer = 213**
This indicates that a mapped resource is not available for access as a pointer.

**cudaErrorECCUncorrectable = 214**
This indicates that an uncorrectable ECC error was detected during execution.

**cudaErrorUnsupportedLimit = 215**
This indicates that the cudaLimit passed to the API call is not supported by the active device.

**cudaErrorDeviceAlreadyInUse = 216**
This indicates that a call tried to access an exclusive-thread device that is already in use by a different thread.

**cudaErrorPeerAccessUnsupported = 217**
This error indicates that P2P access is not supported across the given devices.

**cudaErrorInvalidPtx = 218**
A PTX compilation failed. The runtime may fall back to compiling PTX if an application does not contain a suitable binary for the current device.

**cudaErrorInvalidGraphicsContext = 219**
This indicates an error with the OpenGL or DirectX context.

**cudaErrorNvlinkUncorrectable = 220**
This indicates that an uncorrectable NVLink error was detected during the execution.

**cudaErrorJitCompilerNotFound = 221**
This indicates that the PTX JIT compiler library was not found. The JIT Compiler library is used for PTX compilation. The runtime may fall back to compiling PTX if an application does not contain a suitable binary for the current device.

**cudaErrorUnsupportedPtxVersion = 222**
This indicates that the provided PTX was compiled with an unsupported toolchain. The most common reason for this, is the PTX was generated by a compiler newer than what is supported by the CUDA driver and PTX JIT compiler.

**cudaErrorJitCompilationDisabled = 223**
This indicates that the JIT compilation was disabled. The JIT compilation compiles PTX. The runtime may fall back to compiling PTX if an application does not contain a suitable binary for the current device.

**cudaErrorUnsupportedExecAffinity = 224**
This indicates that the provided execution affinity is not supported by the device.

**cudaErrorInvalidSource = 300**
This indicates that the device kernel source is invalid.

```
cudaErrorFileNotFound = 301
```
This indicates that the file specified was not found.

```
cudaErrorSharedObjectSymbolNotFound = 302
```
This indicates that a link to a shared object failed to resolve.

```
cudaErrorSharedObjectInitFailed = 303
```
This indicates that initialization of a shared object failed.

```
cudaErrorOperatingSystem = 304
```
This error indicates that an OS call failed.

```
cudaErrorInvalidResourceHandle = 400
```
This indicates that a resource handle passed to the API call was not valid. Resource handles are opaque types like `cudaStream_t` and `cudaEvent_t`.

```
cudaErrorIllegalState = 401
```
This indicates that a resource required by the API call is not in a valid state to perform the requested operation.

```
cudaErrorSymbolNotFound = 500
```
This indicates that a named symbol was not found. Examples of symbols are global/constant variable names, driver function names, texture names, and surface names.

```
cudaErrorNotReady = 600
```
This indicates that asynchronous operations issued previously have not completed yet. This result is not actually an error, but must be indicated differently than `cudaSuccess` (which indicates completion). Calls that may return this value include `cudaEventQuery()` and `cudaStreamQuery()`.

```
cudaErrorIllegalAddress = 700
```
The device encountered a load or store instruction on an invalid memory address. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

```
cudaErrorLaunchOutOfResources = 701
```
This indicates that a launch did not occur because it did not have appropriate resources. Although this error is similar to `cudaErrorInvalidConfiguration`, this error usually indicates that the user has attempted to pass too many arguments to the device kernel, or the kernel launch specifies too many threads for the kernel’s register count.

```
cudaErrorLaunchTimeout = 702
```
This indicates that the device kernel took too long to execute. This can only occur if timeouts are enabled - see the device property `kernelExecTimeoutEnabled` for more information. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

```
cudaErrorLaunchIncompatibleTexturing = 703
```
This error indicates a kernel launch that uses an incompatible texturing mode.

```
cudaErrorPeerAccessAlreadyEnabled = 704
```
This error indicates that a call to `cudaDeviceEnablePeerAccess()` is trying to re-enable peer addressing on from a context which has already had peer addressing enabled.
cudaErrorPeerAccessNotEnabled = 705
This error indicates that cudaDeviceDisablePeerAccess() is trying to disable peer addressing which has not been enabled yet via cudaDeviceEnablePeerAccess().

cudaErrorSetOnActiveProcess = 708
This indicates that the user has called cudaSetValidDevices(), cudaSetDeviceFlags(), cudaD3D9SetDirect3DDevice(), cudaD3D10SetDirect3DDevice, cudaD3D11SetDirect3DDevice(), or cudaVDPAUSetVDPAUDevice() after initializing the CUDA runtime by calling non-device management operations (allocating memory and launching kernels are examples of non-device management operations). This error can also be returned if using runtime/driver interoperability and there is an existing CUcontext active on the host thread.

cudaErrorContextIsDestroyed = 709
This error indicates that the context current to the calling thread has been destroyed using cuCtxDestroy, or is a primary context which has not yet been initialized.

cudaErrorAssert = 710
An assert triggered in device code during kernel execution. The device cannot be used again. All existing allocations are invalid. To continue using CUDA, the process must be terminated and relaunched.

cudaErrorTooManyPeers = 711
This error indicates that the hardware resources required to enable peer access have been exhausted for one or more of the devices passed to cudaEnablePeerAccess().

cudaErrorHostMemoryAlreadyRegistered = 712
This error indicates that the memory range passed to cudaHostRegister() has already been registered.

cudaErrorHostMemoryNotRegistered = 713
This error indicates that the pointer passed to cudaHostUnregister() does not correspond to any currently registered memory region.

cudaErrorHardwareStackError = 714
Device encountered an error in the call stack during kernel execution, possibly due to stack corruption or exceeding the stack size limit. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

cudaErrorIllegalInstruction = 715
The device encountered an illegal instruction during kernel execution. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

cudaErrorMisalignedAddress = 716
The device encountered a load or store instruction on a memory address which is not aligned. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

cudaErrorInvalidAddressSpace = 717
While executing a kernel, the device encountered an instruction which can only operate on memory locations in certain address spaces (global, shared, or local), but was supplied a memory address not belonging to an allowed address space. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

**cudaErrorInvalidPc = 718**

The device encountered an invalid program counter. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

**cudaErrorLaunchFailure = 719**

An exception occurred on the device while executing a kernel. Common causes include dereferencing an invalid device pointer and accessing out of bounds shared memory. Less common cases can be system specific - more information about these cases can be found in the system specific user guide. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

**cudaErrorCooperativeLaunchTooLarge = 720**

This error indicates that the number of blocks launched per grid for a kernel that was launched via either `cudaLaunchCooperativeKernel` or `cudaLaunchCooperativeKernelMultiDevice` exceeds the maximum number of blocks as allowed by `cudaOccupancyMaxActiveBlocksPerMultiprocessor` or `cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags` times the number of multiprocessors as specified by the device attribute `cudaDevAttrMultiProcessorCount`.

**cudaErrorNotPermitted = 800**

This error indicates the attempted operation is not permitted.

**cudaErrorNotSupported = 801**

This error indicates the attempted operation is not supported on the current system or device.

**cudaErrorSystemNotReady = 802**

This error indicates that the system is not yet ready to start any CUDA work. To continue using CUDA, verify the system configuration is in a valid state and all required driver daemons are actively running. More information about this error can be found in the system specific user guide.

**cudaErrorSystemDriverMismatch = 803**

This error indicates that there is a mismatch between the versions of the display driver and the CUDA driver. Refer to the compatibility documentation for supported versions.

**cudaErrorCompatNotSupportedOnDevice = 804**

This error indicates that the system was upgraded to run with forward compatibility but the visible hardware detected by CUDA does not support this configuration. Refer to the compatibility documentation for the supported hardware matrix or ensure that only supported hardware is visible during initialization via the CUDA_VISIBLE_DEVICES environment variable.

**cudaErrorMpsConnectionFailed = 805**
This error indicates that the MPS client failed to connect to the MPS control daemon or the MPS server.

\texttt{cudaErrorMpsRpcFailure = 806}

This error indicates that the remote procedural call between the MPS server and the MPS client failed.

\texttt{cudaErrorMpsServerNotReady = 807}

This error indicates that the MPS server is not ready to accept new MPS client requests. This error can be returned when the MPS server is in the process of recovering from a fatal failure.

\texttt{cudaErrorMpsMaxClientsReached = 808}

This error indicates that the hardware resources required to create MPS client have been exhausted.

\texttt{cudaErrorMpsMaxConnectionsReached = 809}

This error indicates that the hardware resources required to device connections have been exhausted.

\texttt{cudaErrorStreamCaptureUnsupported = 900}

The operation is not permitted when the stream is capturing.

\texttt{cudaErrorStreamCaptureInvalidated = 901}

The current capture sequence on the stream has been invalidated due to a previous error.

\texttt{cudaErrorStreamCaptureMerge = 902}

The operation would have resulted in a merge of two independent capture sequences.

\texttt{cudaErrorStreamCaptureUnmatched = 903}

The capture was not initiated in this stream.

\texttt{cudaErrorStreamCaptureUnjoined = 904}

The capture sequence contains a fork that was not joined to the primary stream.

\texttt{cudaErrorStreamCaptureIsolation = 905}

A dependency would have been created which crosses the capture sequence boundary. Only implicit in-stream ordering dependencies are allowed to cross the boundary.

\texttt{cudaErrorStreamCaptureImplicit = 906}

The operation would have resulted in a disallowed implicit dependency on a current capture sequence from cudaStreamLegacy.

\texttt{cudaErrorCapturedEvent = 907}

The operation is not permitted on an event which was last recorded in a capturing stream.

\texttt{cudaErrorStreamCaptureWrongThread = 908}

A stream capture sequence not initiated with the cudaStreamCaptureModeRelaxed argument to \texttt{cudaStreamBeginCapture} was passed to \texttt{cudaStreamEndCapture} in a different thread.

\texttt{cudaErrorTimeout = 909}

This indicates that the wait operation has timed out.

\texttt{cudaErrorGraphExecUpdateFailure = 910}

This error indicates that the graph update was not performed because it included changes which violated constraints specific to instantiated graph update.

\texttt{cudaErrorExternalDevice = 911}
This indicates that an async error has occurred in a device outside of CUDA. If CUDA was waiting for an external device’s signal before consuming shared data, the external device signaled an error indicating that the data is not valid for consumption. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

\texttt{cudaErrorUnknown = 999}

This indicates that an unknown internal error has occurred.

\texttt{cudaErrorApiFailureBase = 10000}

Any unhandled CUDA driver error is added to this value and returned via the runtime. Production releases of CUDA should not return such errors. \texttt{Deprecated} This error return is deprecated as of CUDA 4.1.

\textbf{enum cudaExternalMemoryHandleType}

External memory handle types

\textbf{Values}

\texttt{cudaExternalMemoryHandleTypeOpaqueFd = 1}

Handle is an opaque file descriptor

\texttt{cudaExternalMemoryHandleTypeOpaqueWin32 = 2}

Handle is an opaque shared NT handle

\texttt{cudaExternalMemoryHandleTypeOpaqueWin32Kmt = 3}

Handle is an opaque, globally shared handle

\texttt{cudaExternalMemoryHandleTypeD3D12Heap = 4}

Handle is a D3D12 heap object

\texttt{cudaExternalMemoryHandleTypeD3D12Resource = 5}

Handle is a D3D12 committed resource

\texttt{cudaExternalMemoryHandleTypeD3D11Resource = 6}

Handle is a shared NT handle to a D3D11 resource

\texttt{cudaExternalMemoryHandleTypeD3D11ResourceKmt = 7}

Handle is a globally shared handle to a D3D11 resource

\texttt{cudaExternalMemoryHandleTypeNvSciBuf = 8}

Handle is an NvSciBuf object

\textbf{enum cudaExternalSemaphoreHandleType}

External semaphore handle types

\textbf{Values}

\texttt{cudaExternalSemaphoreHandleTypeOpaqueFd = 1}

Handle is an opaque file descriptor

\texttt{cudaExternalSemaphoreHandleTypeOpaqueWin32 = 2}

Handle is an opaque shared NT handle
cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt = 3
Handle is an opaque, globally shared handle.

cudaExternalSemaphoreHandleTypeD3D12Fence = 4
Handle is a shared NT handle referencing a D3D12 fence object.

cudaExternalSemaphoreHandleTypeD3D11Fence = 5
Handle is a shared NT handle referencing a D3D11 fence object.

cudaExternalSemaphoreHandleTypeNvSciSync = 6
Opaque handle to NvSciSync Object.

cudaExternalSemaphoreHandleTypeKeyedMutex = 7
Handle is a shared NT handle referencing a D3D11 keyed mutex object.

cudaExternalSemaphoreHandleTypeKeyedMutexKmt = 8
Handle is a shared KMT handle referencing a D3D11 keyed mutex object.

cudaExternalSemaphoreHandleTypeTimelineSemaphoreFd = 9
Handle is an opaque handle file descriptor referencing a timeline semaphore.

cudaExternalSemaphoreHandleTypeTimelineSemaphoreWin32 = 10
Handle is an opaque handle file descriptor referencing a timeline semaphore.

enum cudaFlushGPUDirectRDMAWritesOptions
CUDA GPUDirect RDMA flush writes APIs supported on the device.

Values

cudaFlushGPUDirectRDMAWritesOptionHost = 1<<0
  cudaDeviceFlushGPUDirectRDMAWrites() and its CUDA Driver API counterpart are supported on the device.

cudaFlushGPUDirectRDMAWritesOptionMemOps = 1<<1
  The CU_STREAM_WAIT_VALUE_FLUSH flag and the CU_STREAM_MEM_OP_FLUSH_REMOTE_WRITES MemOp are supported on the CUDA device.

enum cudaFlushGPUDirectRDMAWritesScope
CUDA GPUDirect RDMA flush writes scopes.

Values

cudaFlushGPUDirectRDMAWritesToOwner = 100
  Blocks until remote writes are visible to the CUDA device context owning the data.

cudaFlushGPUDirectRDMAWritesToAllDevices = 200
  Blocks until remote writes are visible to all CUDA device contexts.

enum cudaFlushGPUDirectRDMAWritesTarget
CUDA GPUDirect RDMA flush writes targets.
Values

**cudaFlushGPUDirectRDMAWritesTargetCurrentDevice**
Sets the target for [cudaDeviceFlushGPUDirectRDMAWrites](https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDA__runtime.html) to the currently active CUDA device context.

**enum cudaFuncAttribute**
CUDA function attributes that can be set using [cudaFuncSetAttribute](https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDA__runtime.html)

Values

**cudaFuncAttributeMaxDynamicSharedMemorySize = 8**
Maximum dynamic shared memory size

**cudaFuncAttributePreferredSharedMemoryCarveout = 9**
Preferred shared memory-L1 cache split

**enum cudaFuncCache**
CUDA function cache configurations

Values

**cudaFuncCachePreferNone = 0**
Default function cache configuration, no preference

**cudaFuncCachePreferShared = 1**
Prefer larger shared memory and smaller L1 cache

**cudaFuncCachePreferL1 = 2**
Prefer larger L1 cache and smaller shared memory

**cudaFuncCachePreferEqual = 3**
Prefer equal size L1 cache and shared memory

**enum cudaGetDriverEntryPointFlags**
Flags to specify search options to be used with [cudaGetDriverEntryPoint](https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDA__driver.html#gga51d115e06831f5f300a9510a5050aa4f) For more details see [cuGetProcAddress](https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDA__runtime.html)

Values

**cudaEnableDefault = 0x0**
Default search mode for driver symbols.

**cudaEnableLegacyStream = 0x1**
Search for legacy versions of driver symbols.

**cudaEnablePerThreadDefaultStream = 0x2**
Search for per-thread versions of driver symbols.
enum cudaGPUDirectRDMAWritesOrdering

CUDA GPUDirect RDMA flush writes ordering features of the device

Values

cudaGPUDirectRDMAWritesOrderingNone = 0
    The device does not natively support ordering of GPUDirect RDMA writes.
    cudaFlushGPUDirectRDMAWrites() can be leveraged if supported.
cudaGPUDirectRDMAWritesOrderingOwner = 100
    Natively, the device can consistently consume GPUDirect RDMA writes, although other
    CUDA devices may not.
cudaGPUDirectRDMAWritesOrderingAllDevices = 200
    Any CUDA device in the system can consistently consume GPUDirect RDMA writes to this
    device.

enum cudaGraphDebugDotFlags

CUDA Graph debug write options

Values

cudaGraphDebugDotFlagsVerbose = 1<<0
    Output all debug data as if every debug flag is enabled

cudaGraphDebugDotFlagsMemcpyNodeParams = 1<<2
    Adds cudaMemcpy3DParms to output

cudaGraphDebugDotFlagsMemsetNodeParams = 1<<3
    Adds cudaMemcpy3DParms to output

cudaGraphDebugDotFlagsMemcpyNodeParams = 1<<4
    Adds cudaMemcpy3DParms to output

cudaGraphDebugDotFlagsMemcpyNodeParams = 1<<5
    Adds cudaMemcpy3DParms to output

cudaGraphDebugDotFlagsMemcpyNodeParams = 1<<6
    Adds cudaMemcpy3DParms to output

cudaGraphDebugDotFlagsMemcpyNodeParams = 1<<7
    Adds cudaMemcpy3DParms to output

cudaGraphDebugDotFlagsMemcpyNodeParams = 1<<8
    Adds cudaMemcpy3DParms to output

cudaGraphDebugDotFlagsMemcpyNodeParams = 1<<9
    Adds cudaMemcpy3DParms to output

cudaGraphDebugDotFlagsMemcpyNodeParams = 1<<10
    Adds cudaMemcpy3DParms to output
enum cudaGraphExecUpdateResult

CUDA Graph Update error types

Values

cudaGraphExecUpdateSuccess = 0x0
  The update succeeded

cudaGraphExecUpdateError = 0x1
  The update failed for an unexpected reason which is described in the return value of the function

cudaGraphExecUpdateErrorTopologyChanged = 0x2
  The update failed because the topology changed

cudaGraphExecUpdateErrorNodeTypeChanged = 0x3
  The update failed because a node type changed

cudaGraphExecUpdateErrorFunctionChanged = 0x4
  The update failed because the function of a kernel node changed (CUDA driver < 11.2)

cudaGraphExecUpdateErrorParametersChanged = 0x5
  The update failed because the parameters changed in a way that is not supported

cudaGraphExecUpdateErrorNotSupported = 0x6
  The update failed because something about the node is not supported

cudaGraphExecUpdateErrorUnsupportedFunctionChange = 0x7
  The update failed because the function of a kernel node changed in an unsupported way

enum cudaGraphicsCubeFace

CUDA graphics interop array indices for cube maps

Values

cudaGraphicsCubeFacePositiveX = 0x00
  Positive X face of cubemap

cudaGraphicsCubeFaceNegativeX = 0x01
  Negative X face of cubemap

cudaGraphicsCubeFacePositiveY = 0x02
  Positive Y face of cubemap

cudaGraphicsCubeFaceNegativeY = 0x03
  Negative Y face of cubemap

cudaGraphicsCubeFacePositiveZ = 0x04
  Positive Z face of cubemap

cudaGraphicsCubeFaceNegativeZ = 0x05
  Negative Z face of cubemap
enum cudaGraphicsMapFlags
CUDA graphics interop map flags

Values

cudaGraphicsMapFlagsNone = 0
  Default; Assume resource can be read/written

cudaGraphicsMapFlagsReadOnly = 1
  CUDA will not write to this resource

cudaGraphicsMapFlagsWriteDiscard = 2
  CUDA will only write to and will not read from this resource

enum cudaGraphicsRegisterFlags
CUDA graphics interop register flags

Values

cudaGraphicsRegisterFlagsNone = 0
  Default

cudaGraphicsRegisterFlagsReadOnly = 1
  CUDA will not write to this resource

cudaGraphicsRegisterFlagsWriteDiscard = 2
  CUDA will only write to and will not read from this resource

cudaGraphicsRegisterFlagsSurfaceLoadStore = 4
  CUDA will bind this resource to a surface reference

cudaGraphicsRegisterFlagsTextureGather = 8
  CUDA will perform texture gather operations on this resource

enum cudaGraphInstantiateFlags
Flags for instantiating a graph

Values

cudaGraphInstantiateFlagAutoFreeOnLaunch = 1
  Automatically free memory allocated in a graph before relaunching.

enum cudaGraphMemAttributeType
Graph memory attributes

Values

cudaGraphMemAttrUsedMemCurrent = 0x1
cudaGraphMemAttrUsedMemHigh = 0x2
(value type = cuuint64_t) Amount of memory, in bytes, currently associated with graphs.

cudaGraphMemAttrReservedMemCurrent = 0x3
(value type = cuuint64_t) Amount of memory, in bytes, currently allocated for use by the CUDA graphs asynchronous allocator.

cudaGraphMemAttrReservedMemHigh = 0x4
(value type = cuuint64_t) High watermark of memory, in bytes, currently allocated for use by the CUDA graphs asynchronous allocator.

enum cudaGraphNodeType
CUDA Graph node types

Values

cudaGraphNodeTypeKernel = 0x00
GPU kernel node

cudaGraphNodeTypeMemcpy = 0x01
Memcpy node

cudaGraphNodeTypeMemset = 0x02
Memset node

cudaGraphNodeTypeHost = 0x03
Host (executable) node

cudaGraphNodeTypeGraph = 0x04
Node which executes an embedded graph

cudaGraphNodeTypeEmpty = 0x05
Empty (no-op) node

cudaGraphNodeTypeWaitEvent = 0x06
External event wait node

cudaGraphNodeTypeEventRecord = 0x07
External event record node

cudaGraphNodeTypeExtSemaphoreSignal = 0x08
External semaphore signal node

cudaGraphNodeTypeExtSemaphoreWait = 0x09
External semaphore wait node

cudaGraphNodeTypeMemAlloc = 0x0a
Memory allocation node

cudaGraphNodeTypeMemFree = 0x0b
Memory free node

cudaGraphNodeTypeCount
enum cudaKernelNodeAttrID

Graph kernel node Attributes

Values

cudaKernelNodeAttributeAccessPolicyWindow = 1
    Identifier for cudaKernelNodeAttrValue::accessPolicyWindow.
cudaKernelNodeAttributeCooperative = 2
    Allows a kernel node to be cooperative [see cudaLaunchCooperativeKernel].

enum cudaLimit

CUDA Limits

Values

cudaLimitStackSize = 0x00
    GPU thread stack size
cudaLimitPrintfFifoSize = 0x01
    GPU printf FIFO size
cudaLimitMallocHeapSize = 0x02
    GPU malloc heap size
cudaLimitDevRuntimeSyncDepth = 0x03
    GPU device runtime synchronize depth
cudaLimitDevRuntimePendingLaunchCount = 0x04
    GPU device runtime pending launch count
cudaLimitMaxL2FetchGranularity = 0x05
    A value between 0 and 128 that indicates the maximum fetch granularity of L2 (in Bytes). This is a hint
cudaLimitPersistingL2CacheSize = 0x06
    A size in bytes for L2 persisting lines cache size

enum cudaMemAccessFlags

Specifies the memory protection flags for mapping.

Values

cudaMemAccessFlagsProtNone = 0
    Default, make the address range not accessible
cudaMemAccessFlagsProtRead = 1
    Make the address range read accessible
cudaMemAccessFlagsProtReadWrite = 3
    Make the address range read-write accessible
**enum cudaMemAllocationHandleType**

Flags for specifying particular handle types

**Values**

- **cudaMemHandleTypeNone = 0x0**
  - Does not allow any export mechanism.
- **cudaMemHandleTypePosixFileDescriptor = 0x1**
  - Allows a file descriptor to be used for exporting. Permitted only on POSIX systems. (int)
- **cudaMemHandleTypeWin32 = 0x2**
  - Allows a Win32 NT handle to be used for exporting. (HANDLE)
- **cudaMemHandleTypeWin32Kmt = 0x4**
  - Allows a Win32 KMT handle to be used for exporting. (D3DKMT_HANDLE)

**enum cudaMemAllocationType**

Defines the allocation types available

**Values**

- **cudaMemAllocationTypeInvalid = 0x0**
- **cudaMemAllocationTypePinned = 0x1**
  - This allocation type is 'pinned', i.e. cannot migrate from its current location while the application is actively using it
- **cudaMemAllocationTypeMax = 0x7FFFFFFF**

**enum cudaMemcpyKind**

CUDA memory copy types

**Values**

- **cudaMemcpyHostToHost = 0**
  - Host -> Host
- **cudaMemcpyHostToDevice = 1**
  - Host -> Device
- **cudaMemcpyDeviceToHost = 2**
  - Device -> Host
- **cudaMemcpyDeviceToDevice = 3**
  - Device -> Device
- **cudaMemcpyDefault = 4**
  - Direction of the transfer is inferred from the pointer values. Requires unified virtual addressing
enum cudaMemLocationType

Specifies the type of location

Values

cudaMemLocationTypeInvalid = 0

cudaMemLocationTypeDevice = 1
    Location is a device location, thus id is a device ordinal

enum cudaMemoryAdvise

CUDA Memory Advise values

Values

cudaMemAdviseSetReadMostly = 1
    Data will mostly be read and only ocassionally be written to

cudaMemAdviseUnsetReadMostly = 2
    Undo the effect of cudaMemAdviseSetReadMostly

cudaMemAdviseSetPreferredLocation = 3
    Set the preferred location for the data as the specified device

cudaMemAdviseUnsetPreferredLocation = 4
    Clear the preferred location for the data

cudaMemAdviseSetAccessedBy = 5
    Data will be accessed by the specified device, so prevent page faults as much as possible

cudaMemAdviseUnsetAccessedBy = 6
    Let the Unified Memory subsystem decide on the page faulting policy for the specified device

enum cudaMemoryType

CUDA memory types

Values

cudaMemoryTypeUnregistered = 0
    Unregistered memory

cudaMemoryTypeHost = 1
    Host memory

cudaMemoryTypeDevice = 2
    Device memory

cudaMemoryTypeManaged = 3
    Managed memory
enum cudaMemPoolAttr

CUDA memory pool attributes

Values

cudaMemPoolReuseFollowEventDependencies = 0x1
  (value type = int) Allow cuMemAllocAsync to use memory asynchronously freed in another
  streams as long as a stream ordering dependency of the allocating stream on the free
  action exists. Cuda events and null stream interactions can create the required stream
  ordered dependencies. (default enabled)
cudaMemPoolReuseAllowOpportunistic = 0x2
  (value type = int) Allow reuse of already completed frees when there is no dependency
  between the free and allocation. (default enabled)
cudaMemPoolReuseAllowInternalDependencies = 0x3
  (value type = int) Allow cuMemAllocAsync to insert new stream dependencies in order
  to establish the stream ordering required to reuse a piece of memory released by
  cuFreeAsync (default enabled).
cudaMemPoolAttrReleaseThreshold = 0x4
  (value type = cuuint64_t) Amount of reserved memory in bytes to 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 context synchronize. (default 0)
cudaMemPoolAttrReservedMemCurrent = 0x5
  (value type = cuuint64_t) Amount of backing memory currently allocated for the mempool.
cudaMemPoolAttrReservedMemHigh = 0x6
  (value type = cuuint64_t) High watermark of backing memory allocated for the mempool
  since the last time it was reset. High watermark can only be reset to zero.
cudaMemPoolAttrUsedMemCurrent = 0x7
  (value type = cuuint64_t) Amount of memory from the pool that is currently in use by the
  application.
cudaMemPoolAttrUsedMemHigh = 0x8
  (value type = cuuint64_t) High watermark of the amount of memory from the pool that was
  in use by the application since the last time it was reset. High watermark can only be reset
to zero.

enum cudaMemRangeAttribute

CUDA range attributes

Values

cudaMemRangeAttributeReadMostly = 1
  Whether the range will mostly be read and only occasionally be written to
cudaMemRangeAttributePreferredLocation = 2
  The preferred location of the range
cudaMemRangeAttributeAccessedBy = 3
  Memory range has cudaMemAdviseSetAccessedBy set for specified device
cudaMemRangeAttributeLastPrefetchLocation = 4
  The last location to which the range was prefetched

enum cudaOutputMode
  CUDA Profiler Output modes

Values
  cudaKeyValuePair = 0x00
    Output mode Key-Value pair format.
  cudaCSV = 0x01
    Output mode Comma separated values format.

enum cudaResourceType
  CUDA resource types

Values
  cudaResourceTypeArray = 0x00
    Array resource
  cudaResourceTypeMipmappedArray = 0x01
    Mipmapped array resource
  cudaResourceTypeLinear = 0x02
    Linear resource
  cudaResourceTypePitch2D = 0x03
    Pitch 2D resource

enum cudaResourceViewFormat
  CUDA texture resource view formats

Values
  cudaResViewFormatNone = 0x00
    No resource view format (use underlying resource format)
  cudaResViewFormatUnsignedChar1 = 0x01
    1 channel unsigned 8-bit integers
  cudaResViewFormatUnsignedChar2 = 0x02
    2 channel unsigned 8-bit integers
  cudaResViewFormatUnsignedChar4 = 0x03
4 channel unsigned 8-bit integers

`cudaResViewFormatSignedChar1 = 0x04`  
1 channel signed 8-bit integers

`cudaResViewFormatSignedChar2 = 0x05`  
2 channel signed 8-bit integers

`cudaResViewFormatSignedChar4 = 0x06`  
4 channel signed 8-bit integers

`cudaResViewFormatUnsignedShort1 = 0x07`  
1 channel unsigned 16-bit integers

`cudaResViewFormatUnsignedShort2 = 0x08`  
2 channel unsigned 16-bit integers

`cudaResViewFormatUnsignedShort4 = 0x09`  
4 channel unsigned 16-bit integers

`cudaResViewFormatSignedShort1 = 0x0a`  
1 channel signed 16-bit integers

`cudaResViewFormatSignedShort2 = 0x0b`  
2 channel signed 16-bit integers

`cudaResViewFormatSignedShort4 = 0x0c`  
4 channel signed 16-bit integers

`cudaResViewFormatUnsignedInt1 = 0x0d`  
1 channel unsigned 32-bit integers

`cudaResViewFormatUnsignedInt2 = 0x0e`  
2 channel unsigned 32-bit integers

`cudaResViewFormatUnsignedInt4 = 0x0f`  
4 channel unsigned 32-bit integers

`cudaResViewFormatSignedInt1 = 0x10`  
1 channel signed 32-bit integers

`cudaResViewFormatSignedInt2 = 0x11`  
2 channel signed 32-bit integers

`cudaResViewFormatSignedInt4 = 0x12`  
4 channel signed 32-bit integers

`cudaResViewFormatHalf1 = 0x13`  
1 channel 16-bit floating point

`cudaResViewFormatHalf2 = 0x14`  
2 channel 16-bit floating point

`cudaResViewFormatHalf4 = 0x15`  
4 channel 16-bit floating point

`cudaResViewFormatFloat1 = 0x16`  
1 channel 32-bit floating point

`cudaResViewFormatFloat2 = 0x17`  
2 channel 32-bit floating point

`cudaResViewFormatFloat4 = 0x18`  
4 channel 32-bit floating point
cudaResViewFormatUnsignedBlockCompressed1 = 0x19
    Block compressed 1
cudaResViewFormatUnsignedBlockCompressed2 = 0x1a
    Block compressed 2
cudaResViewFormatUnsignedBlockCompressed3 = 0x1b
    Block compressed 3
cudaResViewFormatUnsignedBlockCompressed4 = 0x1c
    Block compressed 4 unsigned
cudaResViewFormatSignedBlockCompressed4 = 0x1d
    Block compressed 4 signed
cudaResViewFormatUnsignedBlockCompressed5 = 0x1e
    Block compressed 5 unsigned
cudaResViewFormatSignedBlockCompressed5 = 0x1f
    Block compressed 5 signed
cudaResViewFormatUnsignedBlockCompressed6H = 0x20
    Block compressed 6 unsigned half-float
cudaResViewFormatSignedBlockCompressed6H = 0x21
    Block compressed 6 signed half-float
cudaResViewFormatUnsignedBlockCompressed7 = 0x22
    Block compressed 7

enum cudaSharedCarveout

Shared memory carveout configurations. These may be passed to cudaFuncSetAttribute

Values

cudaSharedmemCarveoutDefault = -1
    No preference for shared memory or L1 (default)
cudaSharedmemCarveoutMaxShared = 100
    Prefer maximum available shared memory, minimum L1 cache
cudaSharedmemCarveoutMaxL1 = 0
    Prefer maximum available L1 cache, minimum shared memory

enum cudaSharedMemConfig

CUDA shared memory configuration

Values

cudaSharedMemBankSizeDefault = 0
cudaSharedMemBankSizeFourByte = 1
cudaSharedMemBankSizeEightByte = 2
enum cudaStreamAttrID
Stream Attributes

Values
- cudaStreamAttributeAccessPolicyWindow = 1
  Identifier for cudaStreamAttrValue::accessPolicyWindow.
- cudaStreamAttributeSynchronizationPolicy = 3
  cudaSynchronizationPolicy for work queued up in this stream

enum cudaStreamCaptureMode
Possible modes for stream capture thread interactions. For more details see cudaStreamBeginCapture and cudaThreadExchangeStreamCaptureMode

Values
- cudaStreamCaptureModeGlobal = 0
- cudaStreamCaptureModeThreadLocal = 1
- cudaStreamCaptureModeRelaxed = 2

enum cudaStreamCaptureStatus
Possible stream capture statuses returned by cudaStreamIsCapturing

Values
- cudaStreamCaptureStatusNone = 0
  Stream is not capturing
- cudaStreamCaptureStatusActive = 1
  Stream is actively capturing
- cudaStreamCaptureStatusInvalidated = 2
  Stream is part of a capture sequence that has been invalidated, but not terminated

enum cudaStreamUpdateCaptureDependenciesFlags
Flags for cudaStreamUpdateCaptureDependencies

Values
- cudaStreamAddCaptureDependencies = 0x0
  Add new nodes to the dependency set
- cudaStreamSetCaptureDependencies = 0x1
  Replace the dependency set with the new nodes
enum cudaSurfaceBoundaryMode

CUDA Surface boundary modes

Values

cudaBoundaryModeZero = 0
    Zero boundary mode
cudaBoundaryModeClamp = 1
    Clamp boundary mode
cudaBoundaryModeTrap = 2
    Trap boundary mode

enum cudaSurfaceFormatMode

CUDA Surface format modes

Values

cudaFormatModeForced = 0
    Forced format mode
cudaFormatModeAuto = 1
    Auto format mode

enum cudaTextureAddressMode

CUDA texture address modes

Values

cudaAddressModeWrap = 0
    Wrapping address mode
cudaAddressModeClamp = 1
    Clamp to edge address mode
cudaAddressModeMirror = 2
    Mirror address mode
cudaAddressModeBorder = 3
    Border address mode

enum cudaTextureFilterMode

CUDA texture filter modes

Values

cudaFilterModePoint = 0
Point filter mode
\texttt{cudaFilterModeLinear} = 1
Linear filter mode

\textbf{enum cudaTextureReadMode}
CUDA texture read modes

\textbf{Values}
\texttt{cudaReadModeElementType} = 0
Read texture as specified element type
\texttt{cudaReadModeNormalizedFloat} = 1
Read texture as normalized float

\textbf{enum cudaUserObjectFlags}
Flags for user objects for graphs

\textbf{Values}
\texttt{cudaUserObjectNoDestructorSync} = 0x1
Indicates the destructor execution is not synchronized by any CUDA handle.

\textbf{enum cudaUserObjectRetainFlags}
Flags for retaining user object references for graphs

\textbf{Values}
\texttt{cudaGraphUserObjectMove} = 0x1
Transfer references from the caller rather than creating new references.

\textbf{typedef cudaArray *cudaArray\_const\_t}
CUDA array (as source copy argument)

\textbf{typedef cudaArray *cudaArray\_t}
CUDA array

\textbf{typedef struct CUeglStreamConnection\_st}
\*cudaEglStreamConnection
CUDA EGLStream Connection
typedef cudaError_t
CUDA Error types

typedef struct CUevent_st *cudaEvent_t
CUDA event types

typedef struct CUexternalMemory_st
 *cudaExternalMemory_t
CUDA external memory

typedef struct CUexternalSemaphore_st
 *cudaExternalSemaphore_t
CUDA external semaphore

typedef struct CUfunc_st *cudaFunction_t
CUDA function

typedef struct CUgraph_st *cudaGraph_t
CUDA graph

typedef struct CUgraphExec_st *cudaGraphExec_t
CUDA executable (launchable) graph

typedef cudaGraphicsResource
 *cudaGraphicsResource_t
CUDA graphics resource types

typedef struct CUgraphNode_st *cudaGraphNode_t
CUDA graph node.
typedef void (CUDART_CB *cudaHostFn_t) (void* userData)
CUDA host function

typedef struct CUmemPoolHandle_st *cudaMemPool_t
CUDA memory pool

typedef cudaMipmappedArray *cudaMipmappedArray_const_t
CUDA mipmapped array (as source argument)

typedef cudaMipmappedArray *cudaMipmappedArray_t
CUDA mipmapped array

typedef cudaOutputMode_t
CUDA output file modes

typedef struct CUstream_st *cudaStream_t
CUDA stream

typedef unsigned long long cudaSurfaceObject_t
An opaque value that represents a CUDA Surface object

typedef unsigned long long cudaTextureObject_t
An opaque value that represents a CUDA texture object

typedef struct CUuserObject_st *cudaUserObject_t
CUDA user object for graphs
#define CUDA_EGL_MAX_PLANES 3
Maximum number of planes per frame

#define CUDA_IPC_HANDLE_SIZE 64
CUDA IPC Handle Size

#define cudaArrayColorAttachment 0x20
Must be set in cudaMemcpyGetMappedMipmappedArray if the mipmapped array is used as a color target in a graphics API

#define cudaArrayCubemap 0x04
Must be set in cudaMemcpy3DArray to create a cubemap CUDA array

#define cudaArrayDefault 0x00
Default CUDA array allocation flag

#define cudaArrayLayered 0x01
Must be set in cudaMemcpy3DArray to create a layered CUDA array

#define cudaArraySparse 0x40
Must be set in cudaMemcpyArray, cudaMemcpy3DArray or cudaMemcpyMipmappedArray in order to create a sparse CUDA array or CUDA mipmapped array

#define cudaArraySparsePropertiesSingleMipTail 0x1
Indicates that the layered sparse CUDA array or CUDA mipmapped array has a single mip tail region for all layers

#define cudaArraySurfaceLoadStore 0x02
Must be set in cudaMemcpyArray or cudaMemcpy3DArray in order to bind surfaces to the CUDA array
#define cudaArrayTextureGather 0x08
Must be set in cudaMallocArray or cudaMemcpy3DArray in order to perform texture gather operations on the CUDA array

#define cudaCooperativeLaunchMultiDeviceNoPostSync 0x02
If set, any subsequent work pushed in a stream that participated in a call to cudaLaunchCooperativeKernelMultiDevice will only wait for the kernel launched on the GPU corresponding to that stream to complete before it begins execution.

#define cudaCooperativeLaunchMultiDeviceNoPreSync 0x01
If set, each kernel launched as part of cudaLaunchCooperativeKernelMultiDevice only waits for prior work in the stream corresponding to that GPU to complete before the kernel begins execution.

#define cudaCpuDeviceId ((int)-1)
Device id that represents the CPU

#define cudaDeviceBlockingSync 0x04
Deprecated This flag was deprecated as of CUDA 4.0 and replaced with cudaDeviceScheduleBlockingSync.
Device flag - Use blocking synchronization

#define cudaDeviceLmemResizeToMax 0x10
Device flag - Keep local memory allocation after launch

#define cudaDeviceMapHost 0x08
Device flag - Support mapped pinned allocations

#define cudaDeviceMask 0x1f
Device flags mask
```c
#define cudaDevicePropDontCare
Empty device properties

#define cudaDeviceScheduleAuto 0x00
Device flag - Automatic scheduling

#define cudaDeviceScheduleBlockingSync 0x04
Device flag - Use blocking synchronization

#define cudaDeviceScheduleMask 0x07
Device schedule flags mask

#define cudaDeviceScheduleSpin 0x01
Device flag - Spin default scheduling

#define cudaDeviceScheduleYield 0x02
Device flag - Yield default scheduling

#define cudaEventBlockingSync 0x01
Event uses blocking synchronization

#define cudaEventDefault 0x00
Default event flag

#define cudaEventDisableTiming 0x02
Event will not record timing data

#define cudaEventInterprocess 0x04
Event is suitable for interprocess use. cudaEventDisableTiming must be set

#define cudaEventRecordDefault 0x00
Default event record flag
```
#define cudaEventRecordExternal 0x01
Event is captured in the graph as an external event node when performing stream capture

#define cudaEventWaitDefault 0x00
Default event wait flag

#define cudaEventWaitExternal 0x01
Event is captured in the graph as an external event node when performing stream capture

#define cudaExternalMemoryDedicated 0x1
Indicates that the external memory object is a dedicated resource

#define cudaExternalSemaphoreSignalSkipNvSciBufMemSync 0x01
When the /p flags parameter of cudaExternalSemaphoreSignalParams contains this flag, it indicates that signaling an external semaphore object should skip performing appropriate memory synchronization operations over all the external memory objects that are imported as cudaExternalMemoryHandleTypeNvSciBuf, which otherwise are performed by default to ensure data coherency with other importers of the same NvSciBuf memory objects.

#define cudaExternalSemaphoreWaitSkipNvSciBufMemSync 0x02
When the /p flags parameter of cudaExternalSemaphoreWaitParams contains this flag, it indicates that waiting an external semaphore object should skip performing appropriate memory synchronization operations over all the external memory objects that are imported as cudaExternalMemoryHandleTypeNvSciBuf, which otherwise are performed by default to ensure data coherency with other importers of the same NvSciBuf memory objects.

#define(cudaHostAllocDefault 0x00
Default page-locked allocation flag
#define cudaHostAllocMapped 0x02
Map allocation into device space

#define cudaHostAllocPortable 0x01
Pinned memory accessible by all CUDA contexts

#define cudaHostAllocWriteCombined 0x04
Write-combined memory

#define cudaHostRegisterDefault 0x00
Default host memory registration flag

#define cudaHostRegisterIoMemory 0x04
Memory-mapped I/O space

#define cudaHostRegisterMapped 0x02
Map registered memory into device space

#define cudaHostRegisterPortable 0x01
Pinned memory accessible by all CUDA contexts

#define cudaHostRegisterReadOnly 0x08
Memory-mapped read-only

#define cudaInvalidDeviceId ((int)-2)
Device id that represents an invalid device

#define cudaIpcMemLazyEnablePeerAccess 0x01
Automatically enable peer access between remote devices as needed

#define cudaMemAttachGlobal 0x01
Memory can be accessed by any stream on any device
#define cudaMemAttachHost 0x02
Memory cannot be accessed by any stream on any device

#define cudaMemAttachSingle 0x04
Memory can only be accessed by a single stream on the associated device

#define cudaNvSciSyncAttrSignal 0x1
When /p flags of cudaDeviceGetNvSciSyncAttributes is set to this, it indicates that application need signaler specific NvSciSyncAttr to be filled by cudaDeviceGetNvSciSyncAttributes.

#define cudaNvSciSyncAttrWait 0x2
When /p flags of cudaDeviceGetNvSciSyncAttributes is set to this, it indicates that application need waiter specific NvSciSyncAttr to be filled by cudaDeviceGetNvSciSyncAttributes.

#define cudaOccupancyDefault 0x00
Default behavior

#define cudaOccupancyDisableCachingOverride 0x01
Assume global caching is enabled and cannot be automatically turned off

#define cudaPeerAccessDefault 0x00
Default peer addressing enable flag

#define cudaStreamDefault 0x00
Default stream flag

#define cudaStreamLegacy ((cudaStream_t)0x1)
Legacy stream handle
Stream handle that can be passed as a cudaStream_t to use an implicit stream with legacy synchronization behavior.
See details of the synchronization behavior.


#define cudaStreamNonBlocking 0x01
Stream does not synchronize with stream 0 (the NULL stream)

#define cudaStreamPerThread ((cudaStream_t)0x2)
Per-thread stream handle
Stream handle that can be passed as a cudaStream_t to use an implicit stream with per-thread synchronization behavior.
See details of the synchronization behavior.
Chapter 7. Data Structures

Here are the data structures with brief descriptions:

__cudaOccupancyB2DHelper
cudaAccessPolicyWindow
cudaArraySparseProperties
cudaChannelFormatDesc
cudaDeviceProp
cudaEglFrame
cudaEglPlaneDesc
cudaExtent
cudaExternalMemoryBufferDesc
cudaExternalMemoryHandleDesc
cudaExternalMemoryMipmappedArrayDesc
cudaExternalSemaphoreHandleDesc
cudaExternalSemaphoreSignalNodeParams
cudaExternalSemaphoreSignalParams
cudaExternalSemaphoreWaitNodeParams
cudaExternalSemaphoreWaitParams
cudaFuncAttributes
cudaHostNodeParams

cudalpcEventHandle_t
cudalpcMemHandle_t
cudaKernelNodeAttrValue
cudaKernelNodeParams
cudaLaunchParams
cudaMemAccessDesc
cudaMemAllocNodeParams
cudaMemcpy3DParms
cudaMemcpy3DPeerParms
cudaMemLocation
cudaMemPoolProps
cudaMemPoolPtrExportData
cudaMemsetParams
cudaPitchedPtr
cudaPointerAttributes
cudaPos
cudaResourceDesc
cudaResourceViewDesc
cudaStreamAttrValue
cudaTextureDesc
cudaUUID_t
surfaceReference
textureReference

7.1.  __cudaOccupancyB2DHelper

C++ API Routines cppClassifierVisibility: visibility=public  cppClassifierTemplateModel: =
Helper functor for cudaOccupancyMaxPotentialBlockSize

7.2.  cudaAccessPolicyWindow Struct

Reference

Specifies an access policy for a window, a contiguous extent of memory beginning at base_ptr and ending at base_ptr + num_bytes. Partition into many segments and assign segments such that. sum of “hit segments” / window == approx. ratio. sum of “miss segments” / window == approx 1-ratio. Segments and ratio specifications are fitted to the capabilities of the architecture. Accesses in a hit segment apply the hitProp access policy. Accesses in a miss segment apply the missProp access policy.

void *cudaAccessPolicyWindow::base_ptr

Starting address of the access policy window. CUDA driver may align it.

enum cudaAccessProperty
cudaAccessPolicyWindow::hitProp

CUaccessProperty set for hit.

float cudaAccessPolicyWindow::hitRatio

hitRatio specifies percentage of lines assigned hitProp, rest are assigned missProp.
enum cudaAccessProperty
cudaAccessPolicyWindow::missProp

CUaccessProperty set for miss. Must be either NORMAL or STREAMING.

size_t cudaAccessPolicyWindow::num_bytes
Size in bytes of the window policy. CUDA driver may restrict the maximum size and alignment.

7.3. cudaArraySparseProperties Struct Reference
Sparse CUDA array and CUDA mipmapped array properties

unsigned int cudaArraySparseProperties::depth
Tile depth in elements

unsigned int cudaArraySparseProperties::flags
Flags will either be zero or cudaArraySparsePropertiesSingleMipTail

unsigned int cudaArraySparseProperties::height
Tile height in elements

unsigned int cudaArraySparseProperties::miptailFirstLevel
First mip level at which the mip tail begins

unsigned long long cudaArraySparseProperties::miptailSize
Total size of the mip tail.

unsigned int cudaArraySparseProperties::width
Tile width in elements
7.4. **cudaChannelFormatDesc Struct Reference**

CUDA Channel format descriptor

```cpp
eenum cudaChannelFormatKind
```

Channel format kind

```cpp
int cudaChannelFormatDesc::f
```

```cpp
w
```

```cpp
int cudaChannelFormatDesc::x
```

```cpp
x
```

```cpp
int cudaChannelFormatDesc::y
```

```cpp
y
```

```cpp
int cudaChannelFormatDesc::z
```

```cpp
z
```

7.5. **cudaDeviceProp Struct Reference**

CUDA device properties

```cpp
int cudaDeviceProp::accessPolicyMaxWindowSize
```

The maximum value of `cudaAccessPolicyWindow::num_bytes`.

```cpp
int cudaDeviceProp::asyncEngineCount
```

Number of asynchronous engines
int cudaDeviceProp::canMapHostMemory
Device can map host memory with cudaHostAlloc/cudaHostGetDevicePointer

int cudaDeviceProp::canUseHostPointerForRegisteredMem
Device can access host registered memory at the same virtual address as the CPU

int cudaDeviceProp::clockRate
Clock frequency in kilohertz

int cudaDeviceProp::computeMode
Compute mode (See cudaComputeMode)

int cudaDeviceProp::computePreemptionSupported
Device supports Compute Preemption

int cudaDeviceProp::concurrentKernels
Device can possibly execute multiple kernels concurrently

int cudaDeviceProp::concurrentManagedAccess
Device can coherently access managed memory concurrently with the CPU

int cudaDeviceProp::cooperativeLaunch
Device supports launching cooperative kernels via cudaLaunchCooperativeKernel

int cudaDeviceProp::cooperativeMultiDeviceLaunch
Deprecated, cudaLaunchCooperativeKernelMultiDevice is deprecated.

int cudaDeviceProp::deviceOverlap
Device can concurrently copy memory and execute a kernel. Deprecated. Use instead asyncEngineCount.
int cudaDeviceProp::directManagedMemAccessFromHost
Host can directly access managed memory on the device without migration.

int cudaDeviceProp::ECCEnabled
Device has ECC support enabled

int cudaDeviceProp::globalL1CacheSupported
Device supports caching globals in L1

int cudaDeviceProp::hostNativeAtomicSupported
Link between the device and the host supports native atomic operations

int cudaDeviceProp::integrated
Device is integrated as opposed to discrete

int cudaDeviceProp::isMultiGpuBoard
Device is on a multi-GPU board

int cudaDeviceProp::kernelExecTimeoutEnabled
Specified whether there is a run time limit on kernels

int cudaDeviceProp::l2CacheSize
Size of L2 cache in bytes

int cudaDeviceProp::localL1CacheSupported
Device supports caching locals in L1

char cudaDeviceProp::luid
8-byte locally unique identifier. Value is undefined on TCC and non-Windows platforms
unsigned int cudaDeviceProp::luidDeviceNodeMask
LUID device node mask. Value is undefined on TCC and non-Windows platforms

int cudaDeviceProp::major
Major compute capability

int cudaDeviceProp::managedMemory
Device supports allocating managed memory on this system

int cudaDeviceProp::maxBlocksPerMultiProcessor
Maximum number of resident blocks per multiprocessor

int cudaDeviceProp::maxGridSize
Maximum size of each dimension of a grid

int cudaDeviceProp::maxSurface1D
Maximum 1D surface size

int cudaDeviceProp::maxSurface1DLayered
Maximum 1D layered surface dimensions

int cudaDeviceProp::maxSurface2D
Maximum 2D surface dimensions

int cudaDeviceProp::maxSurface2DLayered
Maximum 2D layered surface dimensions

int cudaDeviceProp::maxSurface3D
Maximum 3D surface dimensions

int cudaDeviceProp::maxSurfaceCubemap
Maximum Cubemap surface dimensions
int cudaDeviceProp::maxSurfaceCubemapLayered
Maximum Cubemap layered surface dimensions

int cudaDeviceProp::maxTexture1D
Maximum 1D texture size

int cudaDeviceProp::maxTexture1DLayered
Maximum 1D layered texture dimensions

int cudaDeviceProp::maxTexture1DLinear
Deprecated, do not use. Use cudaDeviceGetTexture1DLinearMaxWidth() or cuDeviceGetTexture1DLinearMaxWidth() instead.

int cudaDeviceProp::maxTexture1DMipmap
Maximum 1D mipmapped texture size

int cudaDeviceProp::maxTexture2D
Maximum 2D texture dimensions

int cudaDeviceProp::maxTexture2DGather
Maximum 2D texture dimensions if texture gather operations have to be performed

int cudaDeviceProp::maxTexture2DLayered
Maximum 2D layered texture dimensions

int cudaDeviceProp::maxTexture2DLinear
Maximum dimensions [width, height, pitch] for 2D textures bound to pitched memory

int cudaDeviceProp::maxTexture2DMipmap
Maximum 2D mipmapped texture dimensions
int cudaDeviceProp::maxTexture3D
Maximum 3D texture dimensions

int cudaDeviceProp::maxTexture3DAlt
Maximum alternate 3D texture dimensions

int cudaDeviceProp::maxTextureCubemap
Maximum Cubemap texture dimensions

int cudaDeviceProp::maxTextureCubemapLayered
Maximum Cubemap layered texture dimensions

int cudaDeviceProp::maxThreadsDim
Maximum size of each dimension of a block

int cudaDeviceProp::maxThreadsPerBlock
Maximum number of threads per block

int cudaDeviceProp::maxThreadsPerMultiProcessor
Maximum resident threads per multiprocessor

int cudaDeviceProp::memoryBusWidth
Global memory bus width in bits

int cudaDeviceProp::memoryClockRate
Peak memory clock frequency in kilohertz

size_t cudaDeviceProp::memPitch
Maximum pitch in bytes allowed by memory copies

int cudaDeviceProp::minor
Minor compute capability
int cudaDeviceProp::multiGpuBoardGroupID
Unique identifier for a group of devices on the same multi-GPU board

int cudaDeviceProp::multiProcessorCount
Number of multiprocessors on device

char cudaDeviceProp::name
ASCII string identifying device

int cudaDeviceProp::pageableMemoryAccess
Device supports coherently accessing pageable memory without calling cudaHostRegister on it

int cudaDeviceProp::pageableMemoryAccessUsesHostPageTables
Device accesses pageable memory via the host’s page tables

int cudaDeviceProp::pciBusID
PCI bus ID of the device

int cudaDeviceProp::pciDeviceID
PCI device ID of the device

int cudaDeviceProp::pciDomainID
PCI domain ID of the device

int cudaDeviceProp::persistingL2CacheMaxSize
Device’s maximum L2 persisting lines capacity setting in bytes

int cudaDeviceProp::regsPerBlock
32-bit registers available per block
int cudaDeviceProp::regsPerMultiprocessor
32-bit registers available per multiprocessor

size_t cudaDeviceProp::reservedSharedMemPerBlock
Shared memory reserved by CUDA driver per block in bytes

size_t cudaDeviceProp::sharedMemPerBlock
Shared memory available per block in bytes

size_t cudaDeviceProp::sharedMemPerBlockOptin
Per device maximum shared memory per block usable by special opt in

size_t cudaDeviceProp::sharedMemPerMultiprocessor
Shared memory available per multiprocessor in bytes

int cudaDeviceProp::singleToDoublePrecisionPerfRatio
Ratio of single precision performance (in floating-point operations per second) to double precision performance

int cudaDeviceProp::streamPrioritiesSupported
Device supports stream priorities

size_t cudaDeviceProp::surfaceAlignment
Alignment requirements for surfaces

int cudaDeviceProp::tccDriver
1 if device is a Tesla device using TCC driver, 0 otherwise

size_t cudaDeviceProp::textureAlignment
Alignment requirement for textures
size_t cudaDeviceProp::texturePitchAlignment
Pitch alignment requirement for texture references bound to pitched memory

size_t cudaDeviceProp::totalConstMem
Constant memory available on device in bytes

size_t cudaDeviceProp::totalGlobalMem
Global memory available on device in bytes

int cudaDeviceProp::unifiedAddressing
Device shares a unified address space with the host

cudaUUID_t cudaDeviceProp::uuid
16-byte unique identifier

int cudaDeviceProp::warpSize
Warp size in threads

7.6. cudaEglFrame Struct Reference

CUDA EGLFrame Descriptor - structure defining one frame of EGL.
Each frame may contain one or more planes depending on whether the surface is Multiplanar or not. Each plane of EGLFrame is represented by cudaEglPlaneDesc which is defined as:

```c
typedef struct cudaEglPlaneDesc_st {
  unsigned int width;
  unsigned int height;
  unsigned int depth;
  unsigned int pitch;
  unsigned int numChannels;
  struct cudaChannelFormatDesc channelDesc;
} cudaEglPlaneDesc;
```

cudaEglColorFormat cudaEglFrame::eglColorFormat
CUDA EGL Color Format
cudaEglFrameType cudaEglFrame::frameType
Array or Pitch

cudaArray_t cudaEglFrame::pArray
Array of CUDA arrays corresponding to each plane

unsigned int cudaEglFrame::planeCount
Number of planes

struct cudaEglPlaneDesc cudaEglFrame::planeDesc
CUDA EGL Plane Descriptor - structure defining each plane of a CUDA EGLFrame

struct cudaPitchedPtr cudaEglFrame::pPitch
Array of Pointers corresponding to each plane

7.7. cudaEglPlaneDesc Struct Reference
CUDA EGL Plane Descriptor - structure defining each plane of a CUDA EGLFrame

struct cudaChannelFormatDesc
cudaEglPlaneDesc::channelDesc
Channel Format Descriptor

unsigned int cudaEglPlaneDesc::depth
Depth of plane

unsigned int cudaEglPlaneDesc::height
Height of plane

unsigned int cudaEglPlaneDesc::numChannels
Number of channels for the plane
unsigned int cudaEglPlaneDesc::pitch
Pitch of plane

unsigned int cudaEglPlaneDesc::reserved
Reserved for future use

unsigned int cudaEglPlaneDesc::width
Width of plane

7.8. cudaExtent Struct Reference
CUDA extent
See also:
make_cudaExtent

size_t cudaExtent::depth
Depth in elements

size_t cudaExtent::height
Height in elements

size_t cudaExtent::width
Width in elements when referring to array memory, in bytes when referring to linear memory

7.9. cudaExternalMemoryBufferDesc Struct Reference
External memory buffer descriptor

unsigned int cudaExternalMemoryBufferDesc::flags
Flags reserved for future use. Must be zero.
unsigned long long
cudaExternalMemoryBufferDesc::offset
Offset into the memory object where the buffer’s base is

unsigned long long
cudaExternalMemoryBufferDesc::size
Size of the buffer

7.10. cudaExternalMemoryHandleDesc
Struct Reference
External memory handle descriptor

int cudaExternalMemoryHandleDesc::fd
File descriptor referencing the memory object. Valid when type is
cudaExternalMemoryHandleTypeOpaqueFd

unsigned int cudaExternalMemoryHandleDesc::flags
Flags must either be zero or cudaExternalMemoryDedicated

void *cudaExternalMemoryHandleDesc::handle
Valid NT handle. Must be NULL if ‘name’ is non-NULL

const void *cudaExternalMemoryHandleDesc::name
Name of a valid memory object. Must be NULL if ‘handle’ is non-NULL.

const void
*cudaExternalMemoryHandleDesc::nvSciBufObject
A handle representing NvSciBuf Object. Valid when type is
cudaExternalMemoryHandleTypeNvSciBuf
unsigned long long
cudaExternalMemoryHandleDesc::size

Size of the memory allocation

enum cudaExternalMemoryHandleType
cudaExternalMemoryHandleDesc::type

Type of the handle

cudaExternalMemoryHandleDesc::win32

Win32 handle referencing the semaphore object. Valid when type is one of the following:

- cudaExternalMemoryHandleTypeOpaqueWin32
- cudaExternalMemoryHandleTypeOpaqueWin32Kmt
- cudaExternalMemoryHandleTypeD3D12Heap
- cudaExternalMemoryHandleTypeD3D12Resource
- cudaExternalMemoryHandleTypeD3D11Resource
- cudaExternalMemoryHandleTypeD3D11ResourceKmt

Exactly one of ‘handle’ and ‘name’ must be non-NULL. If type is one of the following: cudaExternalMemoryHandleTypeOpaqueWin32Kmt cudaExternalMemoryHandleTypeD3D11ResourceKmt then ‘name’ must be NULL.

7.11. cudaExternalMemoryMipmappedArrayDesc Struct Reference

External memory mipmap descriptor

struct cudaExtent
cudaExternalMemoryMipmappedArrayDesc::extent

Dimensions of base level of the mipmap chain
unsigned int
cudaExternalMemoryMipmappedArrayDesc::flags
Flags associated with CUDA mipmapped arrays. See `cudaMallocMipmappedArray`

struct cudaChannelFormatDesc
cudaExternalMemoryMipmappedArrayDesc::formatDesc
Format of base level of the mipmap chain

unsigned int
cudaExternalMemoryMipmappedArrayDesc::numLevels
Total number of levels in the mipmap chain

unsigned long long
cudaExternalMemoryMipmappedArrayDesc::offset
Offset into the memory object where the base level of the mipmap chain is.

7.12. `cudaExternalSemaphoreHandleDesc` Struct Reference
External semaphore handle descriptor

int cudaExternalSemaphoreHandleDesc::fd
File descriptor referencing the semaphore object. Valid when type is one of the following:

- `cudaExternalSemaphoreHandleTypeOpaqueFd`
- `cudaExternalSemaphoreHandleTypeTimelineSemaphoreFd`

unsigned int
cudaExternalSemaphoreHandleDesc::flags
Flags reserved for the future. Must be zero.
void *cudaExternalSemaphoreHandleDesc::handle

Valid NT handle. Must be NULL if ‘name’ is non-NULL

const void
*cudaExternalSemaphoreHandleDesc::name

Name of a valid synchronization primitive. Must be NULL if ‘handle’ is non-NULL.

const void
*cudaExternalSemaphoreHandleDesc::nvSciSyncObj

Valid NvSciSyncObj. Must be non NULL

enum cudaExternalSemaphoreHandleType
cudaExternalSemaphoreHandleDesc::type

Type of the handle

cudaExternalSemaphoreHandleDesc::@9::@10
cudaExternalSemaphoreHandleDesc::win32

Win32 handle referencing the semaphore object. Valid when type is one of the following:

- cudaExternalSemaphoreHandleTypeOpaqueWin32
- cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt
- cudaExternalSemaphoreHandleTypeD3D12Fence
- cudaExternalSemaphoreHandleTypeD3D11Fence
- cudaExternalSemaphoreHandleTypeKeyedMutex
- cudaExternalSemaphoreHandleTypeTimelineSemaphoreWin32

Exactly one of ‘handle’ and ‘name’ must be non-NULL. If type is one of the following: cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt cudaExternalSemaphoreHandleTypeKeyedMutexKmt then ‘name’ must be NULL.
7.13. **cudaExternalSemaphoreSignalNodeParams**

Struct Reference

External semaphore signal node parameters

cudaExternalSemaphore_t

*cudaExternalSemaphoreSignalNodeParams::extSemArray

Array of external semaphore handles.

unsigned int

*cudaExternalSemaphoreSignalNodeParams::numExtSems

Number of handles and parameters supplied in extSemArray and paramsArray.


Struct Reference

External semaphore signal parameters, compatible with driver type

void *

*cudaExternalSemaphoreSignalParams::fence

Pointer to NvSciSyncFence. Valid if cudaExternalSemaphoreHandleType is of type cudaExternalSemaphoreHandleTypeNvSciSync.

cudaExternalSemaphoreSignalParams::@11::@12

cudaExternalSemaphoreSignalParams::fence

Parameters for fence objects
unsigned int

cudaExternalSemaphoreSignalParams::flags

Only when `cudaExternalSemaphoreSignalParams` is used to signal a `cudaExternalSemaphore_t` of type `cudaExternalSemaphoreHandleTypeNvSciSync`, the valid flag is `cudaExternalSemaphoreSignalSkipNvSciBufMemSync`, which indicates that while signaling the `cudaExternalSemaphore_t`, no memory synchronization operations should be performed for any external memory object imported as `cudaExternalMemoryHandleTypeNvSciBuf`. For all other types of `cudaExternalSemaphore_t`, flags must be zero.

`cudaExternalSemaphoreSignalParams::@11::@14`

`cudaExternalSemaphoreSignalParams::keyedMutex`

Parameters for keyed mutex objects

unsigned long long

cudaExternalSemaphoreSignalParams::value

Value of fence to be signaled

7.15. `cudaExternalSemaphoreWaitNodeParams`

Struct Reference

External semaphore wait node parameters

`cudaExternalSemaphore_t`

*`cudaExternalSemaphoreWaitNodeParams::extSemArray`

Array of external semaphore handles.

unsigned int

`cudaExternalSemaphoreWaitNodeParams::numExtSems`

Number of handles and parameters supplied in extSemArray and paramsArray.
cudaExternalSemaphoreWaitParams
*cudaExternalSemaphoreWaitNodeParams::paramsArray

Array of external semaphore wait parameters.

7.16. cudaExternalSemaphoreWaitParams
Struct Reference

External semaphore wait parameters, compatible with driver type

void *cudaExternalSemaphoreWaitParams::fence

Pointer to NvSciSyncFence. Valid if cudaExternalSemaphoreHandleType is of type cudaExternalSemaphoreHandleTypeNvSciSync.

cudaExternalSemaphoreWaitParams::@15::@16
cudaExternalSemaphoreWaitParams::fence

Parameters for fence objects

unsigned int
cudaExternalSemaphoreWaitParams::flags

Only when cudaExternalSemaphoreSignalParams is used to signal a cudaExternalSemaphore_t of type cudaExternalSemaphoreHandleTypeNvSciSync, the valid flag is cudaExternalSemaphoreSignalSkipNvSciBufMemSync; which indicates that while waiting for the cudaExternalSemaphore_t, no memory synchronization operations should be performed for any external memory object imported as cudaExternalMemoryHandleTypeNvSciBuf. For all other types of cudaExternalSemaphore_t, flags must be zero.

unsigned long long
cudaExternalSemaphoreWaitParams::key

Value of key to acquire the mutex with
cudaExternalSemaphoreWaitParams::timeoutMs
Timeout in milliseconds to wait to acquire the mutex

unsigned long long
cudaExternalSemaphoreWaitParams::value
Value of fence to be waited on

7.17. cudaFuncAttributes Struct Reference
CUDA function attributes

int cudaFuncAttributes::binaryVersion
The binary architecture version for which the function was compiled. This value is the major binary version * 10 + the minor binary version, so a binary version 1.3 function would return the value 13.

int cudaFuncAttributes::cacheModeCA
The attribute to indicate whether the function has been compiled with user specified option “-Xptxas --dlcm=ca” set.

size_t cudaFuncAttributes::constSizeBytes
The size in bytes of user-allocated constant memory required by this function.

size_t cudaFuncAttributes::localSizeBytes
The size in bytes of local memory used by each thread of this function.
int cudaFuncAttributes::maxDynamicSharedSizeBytes
The maximum size in bytes of dynamic shared memory per block for this function. Any launch must have a dynamic shared memory size smaller than this value.

int cudaFuncAttributes::maxThreadsPerBlock
The maximum number of threads per block, beyond which a launch of the function would fail. This number depends on both the function and the device on which the function is currently loaded.

int cudaFuncAttributes::numRegs
The number of registers used by each thread of this function.

int cudaFuncAttributes::preferredShmemCarveout
On devices where the L1 cache and shared memory use the same hardware resources, this sets the shared memory carveout preference, in percent of the maximum shared memory. Refer to cudaDevAttrMaxSharedMemoryPerMultiprocessor. This is only a hint, and the driver can choose a different ratio if required to execute the function. See cudaFuncSetAttribute

int cudaFuncAttributes::ptxVersion
The PTX virtual architecture version for which the function was compiled. This value is the major PTX version * 10 + the minor PTX version, so a PTX version 1.3 function would return the value 13.

size_t cudaFuncAttributes::sharedSizeBytes
The size in bytes of statically-allocated shared memory per block required by this function. This does not include dynamically-allocated shared memory requested by the user at runtime.

7.18. cudaHostNodeParams Struct
Reference
CUDA host node parameters

cudaHostFn_t cudaHostNodeParams::fn
The function to call when the node executes
void *cudaHostNodeParams::userData

Argument to pass to the function

7.19. cudaIpcEventHandle_t Struct
Reference

CUDA IPC event handle

7.20. cudaIpcMemHandle_t Struct
Reference

CUDA IPC memory handle

7.21. cudaKernelNodeAttrValue Union
Reference

Graph kernel node attributes union, used with
cudaGraphKernelNodeSetAttribute/cudaGraphKernelNodeGetAttribute

struct cudaAccessPolicyWindow

cudaKernelNodeAttrValue::accessPolicyWindow

Attribute CUaccessPolicyWindow.

7.22. cudaKernelNodeParams Struct
Reference

CUDA GPU kernel node parameters

dim3 cudaKernelNodeParams::blockDim

Block dimensions
**cudaKernelNodeParams::extra**

Pointer to kernel arguments in the “extra” format

```c
void *cudaKernelNodeParams::func
```

Kernel to launch

```c
dim3 cudaKernelNodeParams::gridDim
```

Grid dimensions

**cudaKernelNodeParams::kernelParams**

Array of pointers to individual kernel arguments

```c
unsigned int cudaKernelNodeParams::sharedMemBytes
```

Dynamic shared-memory size per thread block in bytes

### 7.23. cudaLaunchParams Struct Reference

CUDA launch parameters

**cudaLaunchParams::args**

Arguments

```c
dim3 cudaLaunchParams::blockDim
```

Block dimensions

```c
void *cudaLaunchParams::func
```

Device function symbol

```c
dim3 cudaLaunchParams::gridDim
```

Grid dimensions
size_t cudaLaunchParams::sharedMem
Shared memory

cudaStream_t cudaLaunchParams::stream
Stream identifier

7.24. cudaMemAccessDesc Struct Reference
Memory access descriptor

enum cudaMemcpyFlags cudaMemAccessDesc::flags
CUmemProt accessibility flags to set on the request

struct cudaMemLocation cudaMemAccessDesc::location
Location on which the request is to change its accessibility

7.25. cudaMemAllocNodeParams Struct Reference
Memory allocation node parameters

size_t cudaMemAllocNodeParams::accessDescCount
in: Number of `accessDescs`'s

cudaMemAccessDesc *cudaMemAllocNodeParams::accessDescs
in: number of memory access descriptors. Must not exceed the number of GPUs.
size_t cudaMemAllocNodeParams::bytesize
in: size in bytes of the requested allocation

void *cudaMemAllocNodeParams::dptr
out: address of the allocation returned by CUDA

struct cudaMemPoolProps
cudaMemAllocNodeParams::poolProps
in: location where the allocation should reside (specified in location). handleTypes must be cudaMemHandleTypeNone. IPC is not supported. in: array of memory access descriptors. Used to describe peer GPU access

7.26. cudaMemcpy3DParms Struct Reference
CUDA 3D memory copying parameters

cudaArray_t cudaMemcpy3DParms::dstArray
Destination memory address

struct cudaPos cudaMemcpy3DParms::dstPos
Destination position offset

struct cudaPitchedPtr cudaMemcpy3DParms::dstPtr
Pitched destination memory address

struct cudaExtent cudaMemcpy3DParms::extent
Requested memory copy size

enum cudaMemcpyKind cudaMemcpy3DParms::kind
Type of transfer
cudaArray_t cudaMemcpy3DParms::srcArray
Source memory address

struct cudaPos cudaMemcpy3DParms::srcPos
Source position offset

struct cudaPitchedPtr cudaMemcpy3DParms::srcPtr
Pitched source memory address

7.27. cudaMemcpy3DPeerParms Struct Reference
CUDA 3D cross-device memory copying parameters

cudaArray_t cudaMemcpy3DPeerParms::dstArray
Destination memory address

int cudaMemcpy3DPeerParms::dstDevice
Destination device

struct cudaPos cudaMemcpy3DPeerParms::dstPos
Destination position offset

struct cudaPitchedPtr cudaMemcpy3DPeerParms::dstPtr
Pitched destination memory address

struct cudaExtent cudaMemcpy3DPeerParms::extent
Requested memory copy size

cudaArray_t cudaMemcpy3DPeerParms::srcArray
Source memory address
int cudaMemcpy3DPeerParms::srcDevice
Source device

struct cudaPos cudaMemcpy3DPeerParms::srcPos
Source position offset

struct cudaMemcpy3DPeerParms::srcPtr
Pitched source memory address

7.28.  cudaMemcpy MemLocation Struct Reference

Specifies a memory location.
To specify a gpu, set type = cudaMemcpyMemLocationTypeDevice and set id = the gpu’s device ordinal.

int cudaMemcpyMemLocation::id
identifier for a given this location’s CUMemLocationType.

enum cudaMemcpyMemLocationType cudaMemcpyMemLocation::type
Specifies the location type, which modifies the meaning of id.

7.29.  cudaMemcpy MemPoolProps Struct Reference

Specifies the properties of allocations made from the pool.

enum cudaMemcpyMemAllocationType cudaMemcpyMemPoolProps::allocType
Allocation type. Currently must be specified as cudaMemcpyMemAllocationTypePinned

enum cudaMemcpyMemAllocationHandleType cudaMemcpyMemPoolProps::handleTypes
Handle types that will be supported by allocations from the pool.
struct cudaMemLocation

cudaMemPoolProps::location

Location allocations should reside.

unsigned char cudaMemPoolProps::reserved

reserved for future use, must be 0

void *cudaMemPoolProps::win32SecurityAttributes

Windows-specific LPSECURITYATTRIBUTES required when cudaMemHandleTypeWin32 is specified. This security attribute defines the scope of which exported allocations may be transferred to other processes. In all other cases, this field is required to be zero.

7.30. cudaMemPoolPtrExportData Struct Reference

Opaque data for exporting a pool allocation

7.31. cudaMemsetParams Struct Reference

CUDA Memset node parameters

void *cudaMemsetParams::dst

Destination device pointer

unsigned int cudaMemsetParams::elementSize

Size of each element in bytes. Must be 1, 2, or 4.

size_t cudaMemsetParams::height

Number of rows
size_t cudaMemsetParams::pitch
Pitch of destination device pointer. Unused if height is 1

unsigned int cudaMemsetParams::value
Value to be set

size_t cudaMemsetParams::width
Width of the row in elements

7.32. cudaPitchedPtr Struct Reference
CUDA Pitched memory pointer

See also:
make_cudaPitchedPtr

size_t cudaPitchedPtr::pitch
Pitch of allocated memory in bytes

void *cudaPitchedPtr::ptr
Pointer to allocated memory

size_t cudaPitchedPtr::xsize
Logical width of allocation in elements

size_t cudaPitchedPtr::ysize
Logical height of allocation in elements

7.33. cudaPointerAttributes Struct Reference
CUDA pointer attributes
int cudaPointerAttributes::device

The device against which the memory was allocated or registered. If the memory type is `cudaMemoryTypeDevice` then this identifies the device on which the memory referred physically resides. If the memory type is `cudaMemoryTypeHost` or `::cudaMemoryTypeManaged` then this identifies the device which was current when the memory was allocated or registered (and if that device is deinitialized then this allocation will vanish with that device’s state).

`void *cudaPointerAttributes::devicePointer`

The address which may be dereferenced on the current device to access the memory or NULL if no such address exists.

`void *cudaPointerAttributes::hostPointer`

The address which may be dereferenced on the host to access the memory or NULL if no such address exists.

```
// Note:
// CUDA doesn’t check if unregistered memory is allocated so this field may contain invalid
// pointer if an invalid pointer has been passed to CUDA.
```

`enum cudaMemoryType cudaPointerAttributes::type`

The type of memory - `cudaMemoryTypeUnregistered`, `cudaMemoryTypeHost`, `cudaMemoryTypeDevice` or `cudaMemoryTypeManaged`.

### 7.34. cudaPos Struct Reference

CUDA 3D position

**See also:**

`make_cudaPos`

`size_t cudaPos::x`

`x`
size_t cudaPos::y
y
size_t cudaPos::z
z

7.35. cudaResourceDesc Struct Reference
CUDA resource descriptor

cudaArray_t cudaResourceDesc::array
CUDA array

struct cudaChannelFormatDesc
cudaResourceDesc::desc
Channel descriptor

void *cudaResourceDesc::devPtr
Device pointer

size_t cudaResourceDesc::height
Height of the array in elements

cudaMipmappedArray_t cudaResourceDesc::mipmap
CUDA mipmapped array

size_t cudaResourceDesc::pitchInBytes
Pitch between two rows in bytes

enum cudaResourceType cudaResourceDesc::resType
Resource type
size_t cudaResourceDesc::sizeInBytes
Size in bytes

size_t cudaResourceDesc::width
Width of the array in elements

7.36.  cudaResourceViewDesc Struct Reference
CUDA resource view descriptor

size_t cudaResourceViewDesc::depth
Depth of the resource view

unsigned int cudaResourceViewDesc::firstLayer
First layer index

unsigned int cudaResourceViewDesc::firstMipmapLevel
First defined mipmap level

cudaResourceViewFormat
Resource view format

size_t cudaResourceViewDesc::height
Height of the resource view

unsigned int cudaResourceViewDesc::lastLayer
Last layer index
unsigned int
cudaResourceViewDesc::lastMipmapLevel
Last defined mipmap level

size_t cudaResourceViewDesc::width
Width of the resource view

7.37. cudaStreamAttrValue Union Reference
Stream attributes union used with cudaStreamSetAttribute/cudaStreamGetAttribute

7.38. cudaTextureDesc Struct Reference
CUDA texture descriptor

cudaTextureDesc::addressMode
Texture address mode for up to 3 dimensions

cudaTextureDesc::borderColor
Texture Border Color

cudaTextureDesc::disableTrilinearOptimization
Disable any trilinear filtering optimizations.

cudaTextureDesc::filterMode
Texture filter mode
unsigned int cudaTextureDesc::maxAnisotropy
Limit to the anisotropy ratio

float cudaTextureDesc::maxMipmapLevelClamp
Upper end of the mipmap level range to clamp access to

float cudaTextureDesc::minMipmapLevelClamp
Lower end of the mipmap level range to clamp access to

enum cudaTextureFilterMode
cudaTextureDesc::mipmapFilterMode
Mipmap filter mode

float cudaTextureDesc::mipmapLevelBias
Offset applied to the supplied mipmap level

int cudaTextureDesc::normalizedCoords
Indicates whether texture reads are normalized or not

enum cudaTextureReadMode
cudaTextureDesc::readMode
Texture read mode

int cudaTextureDesc::sRGB
Perform sRGB->linear conversion during texture read

7.39. CUuuid_st Struct Reference
CUDA UUID types

char CUuuid_st::bytes
< CUDA definition of UUID
7.40. surfaceReference Struct Reference

CUDA Surface reference

```c
struct cudaChannelFormatDesc
surfaceReference::channelDesc
```
Channel descriptor for surface reference

7.41. textureReference Struct Reference

CUDA texture reference

```c
eenumcudaTextureAddressMode
textureReference::addressMode
```
Texture address mode for up to 3 dimensions

```c
struct cudaChannelFormatDesc
textureReference::channelDesc
```
Channel descriptor for the texture reference

```c
int textureReference::disableTrilinearOptimization
```
Disable any trilinear filtering optimizations.

```c
eenumcudaTextureFilterMode
textureReference::filterMode
```
Texture filter mode

```c
unsigned int textureReference::maxAnisotropy
```
Limit to the anisotropy ratio

```c
float textureReference::maxMipmapLevelClamp
```
Upper end of the mipmap level range to clamp access to
float textureReference::minMipmapLevelClamp
Lower end of the mipmap level range to clamp access to

d_enumcudaTextureFilterMode
textureReference::mipmapFilterMode
Mipmap filter mode

float textureReference::mipmapLevelBias
Offset applied to the supplied mipmap level

int textureReference::normalized
Indicates whether texture reads are normalized or not

int textureReference::sRGB
Perform sRGB->linear conversion during texture read
Chapter 8. Data Fields

Here is a list of all documented struct and union fields with links to the struct/union documentation for each field:

A

- **accessDescCount**  
  [cudaMemAllocNodeParams](#)
- **accessDescs**  
  [cudaMemAllocNodeParams](#)
- **accessPolicyMaxWindowSize**  
  [cudaDeviceProp](#)
- **accessPolicyWindow**  
  [cudaKernelNodeAttrValue](#)
- **addressMode**  
  [textureReference](#)  
  [cudaTextureDesc](#)
- **allocType**  
  [cudaMemPoolProps](#)
- **args**  
  [cudaLaunchParams](#)
- **array**  
  [cudaResourceDesc](#)
- **asyncEngineCount**  
  [cudaDeviceProp](#)

B

- **base_ptr**  
  [cudaAccessPolicyWindow](#)
- **binaryVersion**  
  [cudaFuncAttributes](#)
- **blockDim**  
  [cudaKernelNodeParams](#)  
  [cudaLaunchParams](#)
Data Fields

borderColor
  cudaTextureDesc
bytes
  cudaUUID_t
bytesize
  cudaMemAllocNodeParams

cacheModeCA
  cudaFuncAttributes
canMapHostMemory
  cudaDeviceProp
canUseHostPointerForRegisteredMem
  cudaDeviceProp
channelDesc
  cudaEglPlaneDesc
  surfaceReference
textureReference
clockRate
  cudaDeviceProp
computeMode
  cudaDeviceProp
computePreemptionSupported
  cudaDeviceProp
concurrentKernels
  cudaDeviceProp
concurrentManagedAccess
  cudaDeviceProp
constSizeBytes
  cudaFuncAttributes
cooperativeLaunch
  cudaDeviceProp
cooperativeMultiDeviceLaunch
  cudaDeviceProp
depth
  cudaEglPlaneDesc
cudaArraySparseProperties
cudaResourceViewDesc
cudaExtent
desc
  cudaResourceDesc
Data Fields

device
cudaPointerAttributes
deviceOverlap
cudaDeviceProp
devicePointer
cudaPointerAttributes
devPtr
cudaResourceDesc
directManagedMemAccessFromHost
cudaDeviceProp
disableTrilinearOptimization
textureReference
cudaTextureDesc
dptr
cudaMemAllocNodeParams
dst
cudaMemsetParams
dstArray
cudaMemcpy3DPrams
cudaMemcpy3DPeerParms
dstDevice
cudaMemcpy3DPeerParms
dstPos
cudaMemcpy3DPeerParms
cudaMemcpy3DParms
dstPtr
cudaMemcpy3DPeerParms
cudaMemcpy3DParms

E

ECCEnabled
cudaDeviceProp
eglColorFormat
cudaEglFrame
elementSize
cudaMemsetParams
extent
cudaMemcpy3DPrams
cudaMemcpy3DPeerParms
cudaExternalMemoryMipmappedArrayDesc
extra
cudaKernelNodeParams
extSemArray
  cudaExternalSemaphoreSignalNodeParams
  cudaExternalSemaphoreWaitNodeParams

F
f
  cudaChannelFormatDesc
fd
  cudaExternalMemoryHandleDesc
  cudaExternalSemaphoreHandleDesc
fence
  cudaExternalSemaphoreSignalParams
  cudaExternalSemaphoreWaitParams
filterMode
  textureReference
  cudaTextureDesc
firstLayer
  cudaResourceViewDesc
firstMipmapLevel
  cudaResourceViewDesc
flags
  cudaExternalMemoryBufferDesc
  cudaExternalSemaphoreWaitParams
  cudaExternalMemoryMipmappedArrayDesc
  cudaExternalSemaphoreSignalParams
  cudaExternalSemaphoreHandleDesc
  cudaArraySparseProperties
  cudaMemAccessDesc
  cudaExternalMemoryHandleDesc
fn
  cudaHostNodeParams
format
  cudaResourceViewDesc
formatDesc
  cudaExternalMemoryMipmappedArrayDesc
frameType
  cudaEglFrame
func
  cudaKernelNodeParams
  cudaLaunchParams
**G**
globalL1CacheSupported
cudaDeviceProp
gridDim
cudaKernelNodeParams
cudaLaunchParams

**H**
handle
cudaExternalMemoryHandleDesc
cudaExternalSemaphoreHandleDesc
handleTypes
cudaMemPoolProps
height
cudaEglPlaneDesc
cudaResourceDesc
cudaResourceViewDesc
cudaArraySparseProperties
cudaMemsetParams
cudaExtent
hitProp
cudaAccessPolicyWindow
hitRatio
cudaAccessPolicyWindow
hostNativeAtomicSupported
cudaDeviceProp
hostPointer
cudaPointerAttributes

**I**
id
cudaMemLocation
integrated
cudaDeviceProp
isMultiGpuBoard
cudaDeviceProp

**K**
kernelExecTimeoutEnabled
cudaDeviceProp
kernelParams
cudaKernelNodeParams
key
  cudaExternalSemaphoreWaitParams
keyedMutex
  cudaExternalSemaphoreSignalParams
cudaExternalSemaphoreWaitParams
kind
  cudaMemcpy3DParms
L
l2CacheSize
  cudaDeviceProp
lastLayer
  cudaResourceViewDesc
lastMipmapLevel
  cudaResourceViewDesc
localL1CacheSupported
  cudaDeviceProp
localSizeBytes
  cudaFuncAttributes
location
  cudaMemPoolProps
cudaMemAccessDesc
luid
  cudaDeviceProp
luidDeviceNodeMask
  cudaDeviceProp
M
major
  cudaDeviceProp
managedMemory
  cudaDeviceProp
maxAnisotropy
  cudaTextureDesc
textureReference
maxBlocksPerMultiProcessor
  cudaDeviceProp
maxDynamicSharedSizeBytes
  cudaFuncAttributes
maxGridSize
  cudaDeviceProp
maxMipmapLevelClamp
textureReference
cudaTextureDesc
maxSurface1D
    cudaDeviceProp
maxSurface1DLayered
    cudaDeviceProp
maxSurface2D
    cudaDeviceProp
maxSurface2DLayered
    cudaDeviceProp
maxSurface3D
    cudaDeviceProp
maxSurfaceCubemap
    cudaDeviceProp
maxSurfaceCubemapLayered
    cudaDeviceProp
maxTexture1D
    cudaDeviceProp
maxTexture1DLayered
    cudaDeviceProp
maxTexture1DLinear
    cudaDeviceProp
maxTexture1DMipmap
    cudaDeviceProp
maxTexture2D
    cudaDeviceProp
maxTexture2DGather
    cudaDeviceProp
maxTexture2DLayered
    cudaDeviceProp
maxTexture2DLinear
    cudaDeviceProp
maxTexture2DMipmap
    cudaDeviceProp
maxTexture3D
    cudaDeviceProp
maxTexture3DAlt
    cudaDeviceProp
maxTextureCubemap
    cudaDeviceProp
maxTextureCubemapLayered
    cudaDeviceProp
maxThreadsDim
    cudaDeviceProp
Data Fields

- maxThreadsPerBlock
  - cudaDeviceProp
  - cudaFuncAttributes
- maxThreadsPerMultiProcessor
  - cudaDeviceProp
- memoryBusWidth
  - cudaDeviceProp
- memoryClockRate
  - cudaDeviceProp
- memPitch
  - cudaDeviceProp
- minMipmapLevelClamp
  - textureReference
  - cudaTextureDesc
- minor
  - cudaDeviceProp
- mipmap
  - cudaResourceDesc
- mipmapFilterMode
  - cudaTextureDesc
  - textureReference
- mipmapLevelBias
  - textureReference
  - cudaTextureDesc
- miptailFirstLevel
  - cudaArraySparseProperties
- miptailSize
  - cudaArraySparseProperties
- missProp
  - cudaAccessPolicyWindow
- multiGpuBoardGroupID
  - cudaDeviceProp
- multiProcessorCount
  - cudaDeviceProp
- N
- name
  - cudaDeviceProp
  - cudaExternalMemoryHandleDesc
  - cudaExternalSemaphoreHandleDesc
- normalized
  - textureReference
normalizedCoors
    cudaTextureDesc
num_bytes
    cudaAccessPolicyWindow
numChannels
    cudaEglPlaneDesc
numExtSems
    cudaExternalSemaphoreSignalNodeParams
    cudaExternalSemaphoreWaitNodeParams
numLevels
    cudaExternalMemoryMipmappedArrayDesc
numRegs
    cudaFuncAttributes
nvSciBufObject
    cudaExternalMemoryHandleDesc
nvSciSyncObj
    cudaExternalSemaphoreHandleDesc
O
offset
    cudaExternalMemoryBufferDesc
    cudaExternalMemoryMipmappedArrayDesc
P
pageableMemoryAccess
    cudaDeviceProp
pageTitleusAccessUsesHostPageTables
    cudaDeviceProp
paramsArray
    cudaExternalSemaphoreWaitNodeParams
    cudaExternalSemaphoreSignalNodeParams
pArray
    cudaEglFrame
pciBusID
    cudaDeviceProp
pciDeviceID
    cudaDeviceProp
pciDomainID
    cudaDeviceProp
persistingL2CacheMaxSize
    cudaDeviceProp
pitch
    cudaEglPlaneDesc
Data Fields

- `cudaPitchedPtr`
- `cudaMemsetParams`
- `pitchInBytes`
- `cudaResourceDesc`
- `planeCount`
- `cudaEglFrame`
- `planeDesc`
- `cudaEglFrame`
- `poolProps`
- `cudaMemAllocNodeParams`
- `pPitch`
- `cudaEglFrame`
- `preferredShmemCarveout`
- `cudaFuncAttributes`
- `ptr`
- `cudaPitchedPtr`
- `ptxVersion`
- `cudaFuncAttributes`

R

- `readMode`
- `cudaTextureDesc`
- `regsPerBlock`
- `cudaDeviceProp`
- `regsPerMultiprocessor`
- `cudaDeviceProp`
- `reserved`
- `cudaMemPoolProps`
- `cudaEglPlaneDesc`
- `reservedSharedMemPerBlock`
- `cudaDeviceProp`
- `resType`
- `cudaResourceDesc`

S

- `sharedMem`
- `cudaLaunchParams`
- `sharedMemBytes`
- `cudaKernelNodeParams`
- `sharedMemPerBlock`
- `cudaDeviceProp`
- `sharedMemPerBlockOptin`
- `cudaDeviceProp`
sharedMemPerMultiprocessor
    cudaDeviceProp
sharedSizeBytes
    cudaFuncAttributes
singleToDoublePrecisionPerfRatio
    cudaDeviceProp
size
    cudaExternalMemoryHandleDesc
    cudaExternalMemoryBufferDesc
szelnBytes
    cudaResourceDesc
srcArray
    cudaMemcpy3DParms
    cudaMemcpy3DPeerParms
srcDevice
    cudaMemcpy3DPeerParms
srcPos
    cudaMemcpy3DPeerParms
    cudaMemcpy3DParms
srcPtr
    cudaMemcpy3DParms
    cudaMemcpy3DPeerParms
sRGB
    cudaTextureDesc
    textureReference
stream
    cudaLaunchParams
streamPrioritiesSupported
    cudaDeviceProp
surfaceAlignment
    cudaDeviceProp
T
    tccDriver
    cudaMemcpy3DParms
textureAlignment
    cudaMemcpy3DParms
texturePitchAlignment
    cudaMemcpy3DParms
timeoutMs
    cudaExternalSemaphoreWaitParams
totalConstMem
    cudaDeviceProp
Data Fields

**totalGlobalMem**
- cudaDeviceProp

**type**
- cudaMemLocation
- cudaExternalSemaphoreHandleDesc
- cudaExternalMemoryHandleDesc
- cudaPointerAttributes

**unifiedAddressing**
- cudaDeviceProp

**userData**
- cudaHostNodeParams

**uuid**
- cudaDeviceProp

**value**
- cudaMemcpyParams
- cudaMemcpyWaitParams
- cudaMemcpySignalParams

**warpSize**
- cudaDeviceProp

**width**
- cudaArraySparseProperties
- cudaResourceViewDesc
- cudaExtent
- cudaResourceDesc
- cudaMemcpyParams
- cudaEglPlaneDesc

**win32**
- cudaExternalSemaphoreHandleDesc
- cudaExternalMemoryHandleDesc

**win32SecurityAttributes**
- cudaMemcpyPoolProps

**x**
- cudaChannelFormatDesc
Data Fields

- `cudaPos`
- `xsize`
- `cudaPitchedPtr`
- `Y`
- `y`
- `cudaChannelFormatDesc`
- `cudaPos`
- `ysize`
- `cudaPitchedPtr`
- `Z`
- `z`
- `cudaChannelFormatDesc`
- `cudaPos`
Chapter 9. Deprecated List

- Global `cudaThreadExit`
- Global `cudaThreadGetCacheConfig`
- Global `cudaThreadGetLimit`
- Global `cudaThreadSetCacheConfig`
- Global `cudaThreadSetLimit`
- Global `cudaThreadSynchronize`

- Global `cudaLaunchCooperativeKernelMultiDevice`
  
  This function is deprecated as of CUDA 11.3.

- Global `cudaSetDoubleForDevice`
  
  This function is deprecated as of CUDA 7.5

- Global `cudaSetDoubleForHost`
  
  This function is deprecated as of CUDA 7.5
Global cudaMemcpyFromArray

Global cudaMemcpyFromArrayAsync

Global cudaMemcpyToArray

Global cudaMemcpyToArrayAsync

Global cudaMemcpyArrayToArray

Global cudaMemcpyFromArray

Global cudaMemcpyFromArrayAsync

Global cudaGLMapBufferObject

This function is deprecated as of CUDA 3.0.

Global cudaGLMapBufferObjectAsync

This function is deprecated as of CUDA 3.0.

Global cudaGLRegisterBufferObject

This function is deprecated as of CUDA 3.0.

Global cudaGLSetBufferObjectMapFlags

This function is deprecated as of CUDA 3.0.

Global cudaGLSetGLDevice

This function is deprecated as of CUDA 5.0.

Global cudaGLUnmapBufferObject

This function is deprecated as of CUDA 3.0.

Global cudaGLUnmapBufferObjectAsync

This function is deprecated as of CUDA 3.0.
Global `cudaGLUnregisterBufferObject`  
This function is deprecated as of CUDA 3.0.

Global `cudaD3D9MapResources`  
This function is deprecated as of CUDA 3.0.

Global `cudaD3D9RegisterResource`  
This function is deprecated as of CUDA 3.0.

Global `cudaD3D9ResourceGetMappedArray`  
This function is deprecated as of CUDA 3.0.

Global `cudaD3D9ResourceGetMappedPitch`  
This function is deprecated as of CUDA 3.0.

Global `cudaD3D9ResourceGetMappedPointer`  
This function is deprecated as of CUDA 3.0.

Global `cudaD3D9ResourceGetMappedSize`  
This function is deprecated as of CUDA 3.0.

Global `cudaD3D9ResourceGetSurfaceDimensions`  
This function is deprecated as of CUDA 3.0.

Global `cudaD3D9ResourceSetMapFlags`  
This function is deprecated as of CUDA 3.0.

Global `cudaD3D9UnmapResources`  
This function is deprecated as of CUDA 3.0.
Global `cudaD3D9UnregisterResource`
   This function is deprecated as of CUDA 3.0.

Global `cudaD3D10GetDirect3DDevice`
   This function is deprecated as of CUDA 5.0.

Global `cudaD3D10MapResources`
   This function is deprecated as of CUDA 3.0.

Global `cudaD3D10RegisterResource`
   This function is deprecated as of CUDA 3.0.

Global `cudaD3D10ResourceGetMappedArray`
   This function is deprecated as of CUDA 3.0.

Global `cudaD3D10ResourceGetMappedPitch`
   This function is deprecated as of CUDA 3.0.

Global `cudaD3D10ResourceGetMappedPointer`
   This function is deprecated as of CUDA 3.0.

Global `cudaD3D10ResourceGetMappedSize`
   This function is deprecated as of CUDA 3.0.

Global `cudaD3D10ResourceGetSurfaceDimensions`
   This function is deprecated as of CUDA 3.0.

Global `cudaD3D10ResourceSetMapFlags`
   This function is deprecated as of CUDA 3.0.
Global cudaD3D10SetDirect3DDevice
   This function is deprecated as of CUDA 5.0.

Global cudaD3D10UnmapResources
   This function is deprecated as of CUDA 3.0.

Global cudaD3D10UnregisterResource
   This function is deprecated as of CUDA 3.0.

Global cudaD3D11GetDirect3DDevice
   This function is deprecated as of CUDA 5.0.

Global cudaD3D11SetDirect3DDevice
   This function is deprecated as of CUDA 5.0.

Global cudaBindTexture

Global cudaBindTexture2D

Global cudaBindTextureToArray

Global cudaBindTextureToMipmappedArray

Global cudaGetTextureAlignmentOffset

Global cudaGetTextureReference

Global cudaUnbindTexture
Global cudaBindSurfaceToArray

Global cudaGetSurfaceReference

Global cudaProfilerInitialize

Global cudaErrorProfilerNotInitialized
This error return is deprecated as of CUDA 5.0. It is no longer an error to attempt to enable/disable the profiling via cudaProfilerStart or cudaProfilerStop without initialization.

Global cudaErrorProfilerAlreadyStarted
This error return is deprecated as of CUDA 5.0. It is no longer an error to call cudaProfilerStart() when profiling is already enabled.

Global cudaErrorProfilerAlreadyStopped
This error return is deprecated as of CUDA 5.0. It is no longer an error to call cudaProfilerStop() when profiling is already disabled.

Global cudaErrorInvalidHostPointer
This error return is deprecated as of CUDA 10.1.

Global cudaErrorInvalidDevicePointer
This error return is deprecated as of CUDA 10.1.

Global cudaErrorAddressOfConstant
This error return is deprecated as of CUDA 3.1. Variables in constant memory may now have their address taken by the runtime via cudaGetSymbolAddress().

Global cudaErrorTextureFetchFailed
This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.
Global cudaErrorTextureNotBound
This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Global cudaErrorSynchronizationError
This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Global cudaErrorMixedDeviceExecution
This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Global cudaErrorNotYetImplemented
This error return is deprecated as of CUDA 4.1.

Global cudaErrorMemoryValueTooLarge
This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Global cudaErrorPriorLaunchFailure
This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Global cudaErrorApiFailureBase
This error return is deprecated as of CUDA 4.1.

Global cudaDeviceBlockingSync
This flag was deprecated as of CUDA 4.0 and replaced with cudaDeviceScheduleBlockingSync.
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.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

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

© 2007-2023 NVIDIA Corporation & affiliates. All rights reserved.