CUDA Runtime API

API Reference Manual

vRelease Version | January 2024

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
6.2. Device Management [DEPRECATED]..10
cudaDeviceGetSharedMemConfig...10
cudaDeviceSetSharedMemConfig.. 11
6.3. Thread Management [DEPRECATED]..12
cudaThreadExit.. 12
cudaThreadGetCacheConfig... 13
cudaThreadGetLimit.. 14
cudaThreadSetCacheConfig..15
cudaThreadSetLimit...16
cudaThreadSynchronize... 17
6.4. Error Handling..18
cudaGetErrorName..18
cudaGetErrorString... 18
cudaGetLastError.. 19
cudaPeekAtLastError... 20
6.5. Stream Management..21
cudaStreamCallback_t..21
cudaCtxResetPersistingL2Cache.. 21
cudaStreamAddCallback...21
cudaStreamAttachMemAsync...23
cudaStreamBeginCapture..25
cudaStreamBeginCaptureToGraph.. 26
cudaStreamCopyAttributes... 27
cudaStreamCreate...28
cudaStreamCreateWithFlags..29
cudaStreamCreateWithPriority.. 30
cudaStreamDestroy... 31
cudaStreamEndCapture...32

cudaStreamGetAttribute..32
cudaStreamGetCaptureInfo..33
cudaStreamGetCaptureInfo_v3...35
cudaStreamGetDevice..36
cudaStreamGetFlags.. 37
cudaStreamGetId..38
cudaStreamGetPriority...39
cudaStreamIsCapturing...40
cudaStreamQuery...41
cudaStreamSetAttribute.. 41
cudaStreamSynchronize..42
cudaStreamUpdateCaptureDependencies.. 43
cudaStreamUpdateCaptureDependencies_v2... 44
cudaStreamWaitEvent...45
cudaThreadExchangeStreamCaptureMode...46
6.6. Event Management...47
cudaEventCreate...47
cudaEventCreateWithFlags..48
cudaEventDestroy.. 49
cudaEventElapsedTime..50
cudaEventElapsedTime_v2...51
cudaEventQuery..52
cudaEventRecord... 53
cudaEventRecordWithFlags...54
cudaEventSynchronize...55
6.7. External Resource Interoperability.. 56
cudaDestroyExternalMemory... 56
cudaDestroyExternalSemaphore..57
cudaExternalMemoryGetMappedBuffer..58
cudaExternalMemoryGetMappedMipmappedArray... 59
cudaImportExternalMemory..60
cudaImportExternalSemaphore..63
cudaSignalExternalSemaphoresAsync..66
cudaWaitExternalSemaphoresAsync.. 68
6.8. Execution Control.. 70
cudaFuncGetAttributes... 70
cudaFuncGetName..71
cudaFuncGetParamInfo..72

cudaFuncSetAttribute....73
cudaFuncSetCacheConfig....75
cudaGetParameterBuffer....76
cudaGridDependencySynchronize....77
cudaLaunchCooperativeKernel....77
cudaLaunchCooperativeKernelMultiDevice....79
cudaLaunchDevice....81
cudaLaunchHostFunc....82
cudaLaunchKernel....84
cudaLaunchKernelExC....85
cudaSetDoubleForDevice....87
cudaSetDoubleForHost....87
cudaTriggerProgrammaticLaunchCompletion....88
6.9. Execution Control [DEPRECATED]....88
cudaFuncSetSharedMemConfig....89
6.10. Occupancy....90
cudaOccupancyAvailableDynamicSMemPerBlock....90
cudaOccupancyMaxActiveBlocksPerMultiprocessor....92
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags....93
cudaOccupancyMaxActiveClusters....94
cudaOccupancyMaxPotentialClusterSize....96
6.11. Memory Management....97
cudaArrayGetInfo....97
cudaArrayGetMemoryRequirements....98
cudaArrayGetPlane....99
cudaArrayGetSparseProperties....100
cudaFree....100
cudaFreeArray....101
cudaFreeHost....102
cudaFreeMipmappedArray....103
cudaGetMipmappedArrayLevel....104
cudaGetSymbolAddress....105
cudaGetSymbolSize....106
cudaHostAlloc....107
cudaHostGetDevicePointer....108
cudaHostGetFlags....109
cudaHostRegister....110
cudaHostUnregister....112

cudaMalloc....113
cudaMalloc3D....114
cudaMalloc3DArray....115
cudaMallocArray....117
cudaMallocHost....119
cudaMallocManaged....120
cudaMallocMipmappedArray....122
cudaMallocPitch....125
cudaMemAdvise....126
cudaMemAdvise_v2....129
cudaMemcpy....133
cudaMemcpy2D....134
cudaMemcpy2DArrayToArray....135
cudaMemcpy2DAsync....137
cudaMemcpy2DFromArray....139
cudaMemcpy2DFromArrayAsync....140
cudaMemcpy2DToArray....142
cudaMemcpy2DToArrayAsync....143
cudaMemcpy3D....145
cudaMemcpy3DAsync....147
cudaMemcpy3DBatchAsync....149
cudaMemcpy3DPeer....151
cudaMemcpy3DPeerAsync....152
cudaMemcpyAsync....153
cudaMemcpyBatchAsync....155
cudaMemcpyFromSymbol....157
cudaMemcpyFromSymbolAsync....158
cudaMemcpyPeer....159
cudaMemcpyPeerAsync....160
cudaMemcpyToSymbol....162
cudaMemcpyToSymbolAsync....163
cudaMemGetInfo....164
cudaMemPrefetchAsync....165
cudaMemPrefetchAsync_v2....167
cudaMemRangeGetAttribute....169
cudaMemRangeGetAttributes....172
cudaMemset....173
cudaMemset2D....174

cudaMemset2DAsync 175
cudaMemset3D 176
cudaMemset3DAsync 177
cudaMemsetAsync 179
cudaMipmappedArrayGetMemoryRequirements 180
cudaMipmappedArrayGetSparseProperties 181
make_cudaExtent 181
make_cudaPitchedPtr 182
make_cudaPos 183
6.12. Memory Management [DEPRECATED] 183
cudaMemcpyArrayToArray 183
cudaMemcpyFromArray 185
cudaMemcpyFromArrayAsync 186
cudaMemcpyToArray 187
cudaMemcpyToArrayAsync 189
6.13. Stream Ordered Memory Allocator 190
cudaFreeAsync 191
cudaMallocAsync 192
cudaMallocFromPoolAsync 193
cudaMemPoolCreate 194
cudaMemPoolDestroy 195
cudaMemPoolExportPointer 196
cudaMemPoolExportToShareableHandle 196
cudaMemPoolGetAccess 197
cudaMemPoolGetAttribute 198
cudaMemPoolImportFromShareableHandle 199
cudaMemPoolImportPointer 200
cudaMemPoolSetAccess 200
cudaMemPoolSetAttribute 201
cudaMemPoolTrimTo 202
6.14. Unified Addressing 203
cudaPointerGetAttributes 204
6.15. Peer Device Memory Access 206
cudaDeviceCanAccessPeer 206
cudaDeviceDisablePeerAccess 207
cudaDeviceEnablePeerAccess 207
6.16. OpenGL Interoperability 208
cudaGLDeviceList 208

cudaGLGetDevices....... 209
cudaGraphicsGLRegisterBuffer....... 210
cudaGraphicsGLRegisterImage....... 211
cudaWGLGetDevice....... 212
6.17. OpenGL Interoperability [DEPRECATED]....... 213
cudaGLMapFlags....... 213
cudaGLMapBufferObject....... 213
cudaGLMapBufferObjectAsync....... 214
cudaGLRegisterBufferObject....... 215
cudaGLSetBufferObjectMapFlags....... 216
cudaGLSetGLDevice....... 217
cudaGLUnmapBufferObject....... 217
cudaGLUnmapBufferObjectAsync....... 218
cudaGLUnregisterBufferObject....... 219
6.18. Direct3D 9 Interoperability....... 219
cudaD3D9DeviceList....... 219
cudaD3D9GetDevice....... 220
cudaD3D9GetDevices....... 221
cudaD3D9GetDirect3DDevice....... 222
cudaD3D9SetDirect3DDevice....... 222
cudaGraphicsD3D9RegisterResource....... 223
6.19. Direct3D 9 Interoperability [DEPRECATED]....... 225
cudaD3D9MapFlags....... 225
cudaD3D9RegisterFlags....... 226
cudaD3D9MapResources....... 226
cudaD3D9RegisterResource....... 227
cudaD3D9ResourceGetMappedArray....... 229
cudaD3D9ResourceGetMappedPitch....... 230
cudaD3D9ResourceGetMappedPointer....... 231
cudaD3D9ResourceGetMappedSize....... 232
cudaD3D9ResourceGetSurfaceDimensions....... 233
cudaD3D9ResourceSetMapFlags....... 234
cudaD3D9UnmapResources....... 235
cudaD3D9UnregisterResource....... 236
6.20. Direct3D 10 Interoperability....... 236
cudaD3D10DeviceList....... 237
cudaD3D10GetDevice....... 237
cudaD3D10GetDevices....... 238

cudaGraphicsD3D10RegisterResource......239
6.21. Direct3D 10 Interoperability [DEPRECATED]......241
cudaD3D10MapFlags......241
cudaD3D10RegisterFlags......241
cudaD3D10GetDirect3DDevice......241
cudaD3D10MapResources......242
cudaD3D10RegisterResource......243
cudaD3D10ResourceGetMappedArray......245
cudaD3D10ResourceGetMappedPitch......246
cudaD3D10ResourceGetMappedPointer......247
cudaD3D10ResourceGetMappedSize......248
cudaD3D10ResourceGetSurfaceDimensions......249
cudaD3D10ResourceSetMapFlags......250
cudaD3D10SetDirect3DDevice......251
cudaD3D10UnmapResources......252
cudaD3D10UnregisterResource......252
6.22. Direct3D 11 Interoperability......253
cudaD3D11DeviceList......253
cudaD3D11GetDevice......254
cudaD3D11GetDevices......254
cudaGraphicsD3D11RegisterResource......256
6.23. Direct3D 11 Interoperability [DEPRECATED]......258
cudaD3D11GetDirect3DDevice......258
cudaD3D11SetDirect3DDevice......258
6.24. VDPAU Interoperability......259
cudaGraphicsVDPAURegisterOutputSurface......259
cudaGraphicsVDPAURegisterVideoSurface......260
cudaVDPAUGetDevice......261
cudaVDPAUSetVDPAUDevice......262
6.25. EGL Interoperability......263
cudaEGLStreamConsumerAcquireFrame......263
cudaEGLStreamConsumerConnect......264
cudaEGLStreamConsumerConnectWithFlags......265
cudaEGLStreamConsumerDisconnect......265
cudaEGLStreamConsumerReleaseFrame......266
cudaEGLStreamProducerConnect......267
cudaEGLStreamProducerDisconnect......267
cudaEGLStreamProducerPresentFrame......268

cudaEGLStreamProducerReturnFrame......269
cudaEventCreateFromEGLSync......269
cudaGraphicsEGLRegisterImage......270
cudaGraphicsResourceGetMappedEglFrame......271
6.26. Graphics Interoperability......272
cudaGraphicsMapResources......273
cudaGraphicsResourceGetMappedMipmappedArray......274
cudaGraphicsResourceGetMappedPointer......275
cudaGraphicsResourceSetMapFlags......276
cudaGraphicsSubResourceGetMappedArray......277
cudaGraphicsUnmapResources......278
cudaGraphicsUnregisterResource......279
6.27. Texture Object Management......279
cudaCreateChannelDesc......280
cudaCreateTextureObject......281
cudaDestroyTextureObject......286
cudaGetChannelDesc......286
cudaGetTextureObjectResourceDesc......287
cudaGetTextureObjectResourceViewDesc......288
cudaGetTextureObjectTextureDesc......289
6.28. Surface Object Management......289
cudaCreateSurfaceObject......290
cudaDestroySurfaceObject......291
cudaGetSurfaceObjectResourceDesc......291
6.29. Version Management......292
cudaDriverGetVersion......292
cudaRuntimeGetVersion......293
6.30. Graph Management......294
cudaDeviceGetGraphMemAttribute......294
cudaDeviceGraphMemTrim......295
cudaDeviceSetGraphMemAttribute......296
cudaGetCurrentGraphExec......297
cudaGraphAddChildGraphNode......297
cudaGraphAddDependencies......298
cudaGraphAddDependencies_v2......299
cudaGraphAddEmptyNode......300
cudaGraphAddEventRecordNode......302
cudaGraphAddEventWaitNode......303

cudaGraphAddExternalSemaphoresSignalNode....... 304
cudaGraphAddExternalSemaphoresWaitNode....... 305
cudaGraphAddHostNode....... 307
cudaGraphAddKernelNode....... 308
cudaGraphAddMemAllocNode....... 310
cudaGraphAddMemcpyNode....... 312
cudaGraphAddMemcpyNode1D....... 313
cudaGraphAddMemcpyNodeFromSymbol....... 315
cudaGraphAddMemcpyNodeToSymbol....... 317
cudaGraphAddMemFreeNode....... 318
cudaGraphAddMemsetNode....... 320
cudaGraphAddNode....... 321
cudaGraphAddNode_v2....... 322
cudaGraphChildGraphNodeGetGraph....... 323
cudaGraphClone....... 324
cudaGraphConditionalHandleCreate....... 325
cudaGraphCreate....... 326
cudaGraphDebugDotPrint....... 327
cudaGraphDestroy....... 327
cudaGraphDestroyNode....... 328
cudaGraphEventRecordNodeGetEvent....... 329
cudaGraphEventRecordNodeSetEvent....... 330
cudaGraphEventWaitNodeGetEvent....... 330
cudaGraphEventWaitNodeSetEvent....... 331
cudaGraphExecChildGraphNodeSetParams....... 332
cudaGraphExecDestroy....... 333
cudaGraphExecEventRecordNodeSetEvent....... 334
cudaGraphExecEventWaitNodeSetEvent....... 335
cudaGraphExecExternalSemaphoresSignalNodeSetParams....... 336
cudaGraphExecExternalSemaphoresWaitNodeSetParams....... 338
cudaGraphExecGetFlags....... 339
cudaGraphExecHostNodeSetParams....... 340
cudaGraphExecKernelNodeSetParams....... 341
cudaGraphExecMemcpyNodeSetParams....... 342
cudaGraphExecMemcpyNodeSetParams1D....... 344
cudaGraphExecMemcpyNodeSetParamsFromSymbol....... 345
cudaGraphExecMemcpyNodeSetParamsToSymbol....... 347
cudaGraphExecMemsetNodeSetParams....... 348

cudaGraphExecNodeSetParams....350
cudaGraphExecUpdate....351
cudaGraphExternalSemaphoresSignalNodeGetParams....354
cudaGraphExternalSemaphoresSignalNodeSetParams....355
cudaGraphExternalSemaphoresWaitNodeGetParams....356
cudaGraphExternalSemaphoresWaitNodeSetParams....357
cudaGraphGetEdges....358
cudaGraphGetEdges_v2....359
cudaGraphGetNodes....360
cudaGraphGetRootNodes....361
cudaGraphHostNodeGetParams....362
cudaGraphHostNodeSetParams....363
cudaGraphInstantiate....364
cudaGraphInstantiateWithFlags....366
cudaGraphInstantiateWithParams....368
cudaGraphKernelNodeCopyAttributes....370
cudaGraphKernelNodeGetAttribute....371
cudaGraphKernelNodeGetParams....371
cudaGraphKernelNodeSetAttribute....372
cudaGraphKernelNodeSetEnabled....373
cudaGraphKernelNodeSetGridDim....374
cudaGraphKernelNodeSetParam....374
cudaGraphKernelNodeSetParam....375
cudaGraphKernelNodeSetParams....376
cudaGraphKernelNodeUpdatesApply....377
cudaGraphLaunch....378
cudaGraphMemAllocNodeGetParams....379
cudaGraphMemcpyNodeGetParams....380
cudaGraphMemcpyNodeSetParams....381
cudaGraphMemcpyNodeSetParams1D....382
cudaGraphMemcpyNodeSetParamsFromSymbol....383
cudaGraphMemcpyNodeSetParamsToSymbol....384
cudaGraphMemFreeNodeGetParams....385
cudaGraphMemsetNodeGetParams....386
cudaGraphMemsetNodeSetParams....387
cudaGraphNodeFindInClone....388
cudaGraphNodeGetDependencies....389
cudaGraphNodeGetDependencies_v2....390

cudaGraphNodeGetDependentNodes.. 391
cudaGraphNodeGetDependentNodes_v2.. 392
cudaGraphNodeGetEnabled...393
cudaGraphNodeGetType..394
cudaGraphNodeSetEnabled... 395
cudaGraphNodeSetParams..396
cudaGraphReleaseUserObject...397
cudaGraphRemoveDependencies...397
cudaGraphRemoveDependencies_v2...398
cudaGraphRetainUserObject...399
cudaGraphSetConditional... 400
cudaGraphUpload..400
cudaUserObjectCreate..401
cudaUserObjectRelease..402
cudaUserObjectRetain..403
6.31. Driver Entry Point Access... 403
cudaGetDriverEntryPoint..403
cudaGetDriverEntryPointByVersion.. 405
6.32. Library Management... 407
cudaKernelSetAttributeForDevice..407
cudaLibraryEnumerateKernels..409
cudaLibraryGetGlobal...410
cudaLibraryGetKernel...410
cudaLibraryGetKernelCount...411
cudaLibraryGetManaged..412
cudaLibraryGetUnifiedFunction... 413
cudaLibraryLoadData...413
cudaLibraryLoadFromFile... 415
cudaLibraryUnload...416
6.33. C++ API Routines..417
__cudaOccupancyB2DHelper...417
cudaCreateChannelDesc..417
cudaEventCreate..418
cudaFuncGetAttributes.. 419
cudaFuncGetName..420
cudaFuncSetAttribute...421
cudaFuncSetCacheConfig... 423
cudaGetKernel..424

cudaGetSymbolAddress....424
cudaGetSymbolSize....425
cudaGraphAddMemcpyNodeFromSymbol....426
cudaGraphAddMemcpyNodeToSymbol....428
cudaGraphExecMemcpyNodeSetParamsFromSymbol....429
cudaGraphExecMemcpyNodeSetParamsToSymbol....431
cudaGraphInstantiate....432
cudaGraphMemcpyNodeSetParamsFromSymbol....433
cudaGraphMemcpyNodeSetParamsToSymbol....435
cudaLaunchCooperativeKernel....436
cudaLaunchKernel....438
cudaLaunchKernelEx....439
cudaLaunchKernelEx....440
cudaLibraryGetGlobal....442
cudaLibraryGetManaged....443
cudaLibraryGetUnifiedFunction....443
cudaMallocAsync....444
cudaMallocHost....444
cudaMallocManaged....446
cudaMemAdvise....448
cudaMemcpyBatchAsync....449
cudaMemcpyBatchAsync....449
cudaMemcpyFromSymbol....450
cudaMemcpyFromSymbolAsync....451
cudaMemcpyToSymbol....452
cudaMemcpyToSymbolAsync....453
cudaOccupancyAvailableDynamicSMemPerBlock....455
cudaOccupancyMaxActiveBlocksPerMultiprocessor....456
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags....457
cudaOccupancyMaxActiveClusters....459
cudaOccupancyMaxPotentialBlockSize....460
cudaOccupancyMaxPotentialBlockSizeVariableSMem....461
cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags....463
cudaOccupancyMaxPotentialBlockSizeWithFlags....464
cudaOccupancyMaxPotentialClusterSize....466
cudaStreamAttachMemAsync....467
6.34. Interactions with the CUDA Driver API....469
cudaGetFuncBySymbol....471

cudaGetKernel....472
6.35. Profiler Control....472
cudaProfilerStart....473
cudaProfilerStop....473
6.36. Data types used by CUDA Runtime....474
cudaAccessPolicyWindow....475
cudaArrayMemoryRequirements....475
cudaArraySparseProperties....475
cudaAsyncNotificationInfo_t....475
cudaChannelFormatDesc....475
cudaChildGraphNodeParams....475
cudaConditionalNodeParams....475
cudaDeviceProp....475
cudaEglFrame....475
cudaEglPlaneDesc....475
cudaEventRecordNodeParams....475
cudaEventWaitNodeParams....475
cudaExtent....475
cudaExternalMemoryBufferDesc....475
cudaExternalMemoryHandleDesc....475
cudaExternalMemoryMipmappedArrayDesc....475
cudaExternalSemaphoreHandleDesc....475
cudaExternalSemaphoreSignalNodeParams....476
cudaExternalSemaphoreSignalNodeParamsV2....476
cudaExternalSemaphoreSignalParams....476
cudaExternalSemaphoreSignalParams_v1....476
cudaExternalSemaphoreWaitNodeParams....476
cudaExternalSemaphoreWaitNodeParamsV2....476
cudaExternalSemaphoreWaitParams....476
cudaExternalSemaphoreWaitParams_v1....476
cudaFuncAttributes....476
cudaGraphEdgeData....476
cudaGraphExecUpdateResultInfo....476
cudaGraphInstantiateParams....476
cudaGraphKernelNodeUpdate....476
cudaGraphNodeParams....476
cudaHostNodeParams....476
cudaHostNodeParamsV2....476

cudaIpcEventHandle_t.......... 476
cudaIpcMemHandle_t.......... 477
cudaKernelNodeParams.......... 477
cudaKernelNodeParamsV2.......... 477
cudaLaunchAttribute.......... 477
cudaLaunchAttributeValue.......... 477
cudaLaunchConfig_t.......... 477
cudaLaunchMemSyncDomainMap.......... 477
cudaLaunchParams.......... 477
cudaMemAccessDesc.......... 477
cudaMemAllocNodeParams.......... 477
cudaMemAllocNodeParamsV2.......... 477
cudaMemcpy3DOperand.......... 477
cudaMemcpy3DParms.......... 477
cudaMemcpy3DPeerParms.......... 477
cudaMemcpyAttributes.......... 477
cudaMemcpyNodeParams.......... 477
cudaMemFreeNodeParams.......... 477
cudaMemLocation.......... 478
cudaMemPoolProps.......... 478
cudaMemPoolPtrExportData.......... 478
cudaMemsetParams.......... 478
cudaMemsetParamsV2.......... 478
cudaOffset3D.......... 478
cudaPitchedPtr.......... 478
cudaPointerAttributes.......... 478
cudaPos.......... 478
cudaResourceDesc.......... 478
cudaResourceViewDesc.......... 478
cudaTextureDesc.......... 478
CUuuid_st.......... 478
cudaAccessProperty.......... 478
cudaAsyncNotificationType.......... 479
cudaCGScope.......... 479
cudaChannelFormatKind.......... 479
cudaClusterSchedulingPolicy.......... 481
cudaComputeMode.......... 481
cudaDeviceAttr.......... 481

cudaDeviceNumaConfig........488
cudaDeviceP2PAttr........488
cudaDriverEntryPointQueryResult........488
cudaEglColorFormat........489
cudaEglFrameType........495
cudaEglResourceLocationFlags........496
cudaError........496
cudaExternalMemoryHandleType........506
cudaExternalSemaphoreHandleType........507
cudaFlushGPUDirectRDMAWritesOptions........508
cudaFlushGPUDirectRDMAWritesScope........508
cudaFlushGPUDirectRDMAWritesTarget........508
cudaFuncAttribute........508
cudaFuncCache........509
cudaGetDriverEntryPointFlags........509
cudaGPUDirectRDMAWritesOrdering........510
cudaGraphChildGraphNodeOwnership........510
cudaGraphConditionalNodeType........510
cudaGraphDebugDotFlags........511
cudaGraphDependencyType........511
cudaGraphExecUpdateResult........512
cudaGraphicsCubeFace........512
cudaGraphicsMapFlags........513
cudaGraphicsRegisterFlags........513
cudaGraphInstantiateFlags........513
cudaGraphInstantiateResult........514
cudaGraphKernelNodeField........514
cudaGraphMemAttributeType........514
cudaGraphNodeType........515
cudaJit_CacheMode........516
cudaJit_Fallback........516
cudaJitOption........516
cudaLaunchAttributeID........518
cudaLaunchMemSyncDomain........520
cudaLibraryOption........521
cudaLimit........521
cudaMemAccessFlags........522
cudaMemAllocationHandleType........522

cudaMemAllocationType....522
cudaMemcpy3DOperandType....523
cudaMemcpyFlags....523
cudaMemcpyKind....523
cudaMemLocationType....524
cudaMemoryAdvise....524
cudaMemoryType....524
cudaMemPoolAttr....525
cudaMemRangeAttribute....525
cudaResourceType....526
cudaResourceViewFormat....526
cudaSharedCarveout....528
cudaSharedMemConfig....528
cudaStreamCaptureMode....529
cudaStreamCaptureStatus....529
cudaStreamUpdateCaptureDependenciesFlags....529
cudaSurfaceBoundaryMode....529
cudaSurfaceFormatMode....530
cudaTextureAddressMode....530
cudaTextureFilterMode....530
cudaTextureReadMode....530
cudaUserObjectFlags....531
cudaUserObjectRetainFlags....531
cudaArray_const_t....531
cudaArray_t....531
cudaAsyncCallbackHandle_t....531
cudaEglStreamConnection....531
cudaError_t....531
cudaEvent_t....532
cudaExternalMemory_t....532
cudaExternalSemaphore_t....532
cudaFunction_t....532
cudaGraph_t....532
cudaGraphConditionalHandle....532
cudaGraphDeviceNode_t....532
cudaGraphExec_t....532
cudaGraphicsResource_t....532
cudaGraphNode_t....532

cudaHostFn_t........ 533
cudaKernel_t........ 533
cudaLibrary_t........ 533
cudaMemPool_t........ 533
cudaMipmappedArray_const_t........ 533
cudaMipmappedArray_t........ 533
cudaStream_t........ 533
cudaSurfaceObject_t........ 533
cudaTextureObject_t........ 533
cudaUserObject_t........ 533
CUDA_EGL_MAX_PLANES........ 534
CUDA_IPC_HANDLE_SIZE........ 534
cudaArrayColorAttachment........ 534
cudaArrayCubemap........ 534
cudaArrayDefault........ 534
cudaArrayDeferredMapping........ 534
cudaArrayLayered........ 534
cudaArraySparse........ 534
cudaArraySparsePropertiesSingleMipTail........ 534
cudaArraySurfaceLoadStore........ 534
cudaArrayTextureGather........ 535
cudaCooperativeLaunchMultiDeviceNoPostSync........ 535
cudaCooperativeLaunchMultiDeviceNoPreSync........ 535
cudaCpuDeviceId........ 535
cudaDeviceBlockingSync........ 535
cudaDeviceLmemResizeToMax........ 535
cudaDeviceMapHost........ 535
cudaDeviceMask........ 535
cudaDeviceScheduleAuto........ 536
cudaDeviceScheduleBlockingSync........ 536
cudaDeviceScheduleMask........ 536
cudaDeviceScheduleSpin........ 536
cudaDeviceScheduleYield........ 536
cudaDeviceSyncMemops........ 536
cudaEventBlockingSync........ 536
cudaEventDefault........ 536
cudaEventDisableTiming........ 536
cudaEventInterprocess........ 536

cudaEventRecordDefault.......... 536
cudaEventRecordExternal.......... 537
cudaEventWaitDefault.......... 537
cudaEventWaitExternal.......... 537
cudaExternalMemoryDedicated.......... 537
cudaExternalSemaphoreSignalSkipNvSciBufMemSync.......... 537
cudaExternalSemaphoreWaitSkipNvSciBufMemSync.......... 537
cudaGraphKernelNodePortDefault.......... 537
cudaGraphKernelNodePortLaunchCompletion.......... 538
cudaGraphKernelNodePortProgrammatic.......... 538
cudaHostAllocDefault.......... 538
cudaHostAllocMapped.......... 538
cudaHostAllocPortable.......... 538
cudaHostAllocWriteCombined.......... 538
cudaHostRegisterDefault.......... 538
cudaHostRegisterIoMemory.......... 538
cudaHostRegisterMapped.......... 538
cudaHostRegisterPortable.......... 538
cudaHostRegisterReadOnly.......... 539
cudaInitDeviceFlagsAreValid.......... 539
cudaInvalidDeviceId.......... 539
cudaIpcMemLazyEnablePeerAccess.......... 539
cudaMemAttachGlobal.......... 539
cudaMemAttachHost.......... 539
cudaMemAttachSingle.......... 539
cudaMemPoolCreateUsageHwDecompress.......... 539
cudaNvSciSyncAttrSignal.......... 539
cudaNvSciSyncAttrWait.......... 539
cudaOccupancyDefault.......... 540
cudaOccupancyDisableCachingOverride.......... 540
cudaPeerAccessDefault.......... 540
cudaStreamDefault.......... 540
cudaStreamLegacy.......... 540
cudaStreamNonBlocking.......... 540
cudaStreamPerThread.......... 540
6.1. Difference between the driver and runtime APIs.......... 540
Chapter 7. Data Structures.......... 542
__cudaOccupancyB2DHelper.......... 543

cudaAccessPolicyWindow....544
base_ptr....544
hitProp....544
hitRatio....544
missProp....544
num_bytes....544
cudaArrayMemoryRequirements....544
alignment....545
size....545
cudaArraySparseProperties....545
depth....545
flags....545
height....545
miptailFirstLevel....545
miptailSize....545
width....545
cudaAsyncNotificationInfo_t....546
bytesOverBudget....546
info....546
overBudget....546
type....546
cudaChannelFormatDesc....546
f....546
w....546
x....547
y....547
z....547
cudaChildGraphNodeParams....547
graph....547
ownership....547
cudaConditionalNodeParams....547
handle....547
phGraph_out....548
size....548
type....548
cudaDeviceProp....548
accessPolicyMaxWindowSize....548
asyncEngineCount....548

canMapHostMemory.....549
canUseHostPointerForRegisteredMem.....549
clockRate.....549
clusterLaunch.....549
computeMode.....549
computePreemptionSupported.....549
concurrentKernels.....549
concurrentManagedAccess.....549
cooperativeLaunch.....549
cooperativeMultiDeviceLaunch.....549
deferredMappingCudaArraySupported.....549
deviceOverlap.....550
directManagedMemAccessFromHost.....550
ECCEnabled.....550
globalL1CacheSupported.....550
gpuDirectRDMAFlushWritesOptions.....550
gpuDirectRDMASupported.....550
gpuDirectRDMAWritesOrdering.....550
hostNativeAtomicSupported.....550
hostRegisterReadOnlySupported.....550
hostRegisterSupported.....550
integrated.....551
ipcEventSupported.....551
isMultiGpuBoard.....551
kernelExecTimeoutEnabled.....551
l2CacheSize.....551
localL1CacheSupported.....551
luid.....551
luidDeviceNodeMask.....551
major.....551
managedMemory.....551
maxBlocksPerMultiProcessor.....551
maxGridSize.....552
maxSurface1D.....552
maxSurface1DLayered.....552
maxSurface2D.....552
maxSurface2DLayered.....552
maxSurface3D.....552

maxSurfaceCubemap.. 552
maxSurfaceCubemapLayered.. 552
maxTexture1D..552
maxTexture1DLayered..552
maxTexture1DLinear.. 553
maxTexture1DMipmap.. 553
maxTexture2D..553
maxTexture2DGather..553
maxTexture2DLayered..553
maxTexture2DLinear.. 553
maxTexture2DMipmap.. 553
maxTexture3D..553
maxTexture3DAlt.. 553
maxTextureCubemap.. 553
maxTextureCubemapLayered.. 554
maxThreadsDim.. 554
maxThreadsPerBlock.. 554
maxThreadsPerMultiProcessor.. 554
memoryBusWidth.. 554
memoryClockRate..554
memoryPoolsSupported.. 554
memoryPoolSupportedHandleTypes.. 554
memPitch..554
minor.. 554
multiGpuBoardGroupID.. 555
multiProcessorCount.. 555
name.. 555
pageableMemoryAccess..555
pageableMemoryAccessUsesHostPageTables..555
pciBusID.. 555
pciDeviceID.. 555
pciDomainID..555
persistingL2CacheMaxSize..555
regsPerBlock.. 555
regsPerMultiprocessor..556
reserved.. 556
reservedSharedMemPerBlock..556
sharedMemPerBlock.. 556

sharedMemPerBlockOptin....556
sharedMemPerMultiprocessor....556
singleToDoublePrecisionPerfRatio....556
sparseCudaArraySupported....556
streamPrioritiesSupported....556
surfaceAlignment....556
tccDriver....557
textureAlignment....557
texturePitchAlignment....557
timelineSemaphoreInteropSupported....557
totalConstMem....557
totalGlobalMem....557
unifiedAddressing....557
unifiedFunctionPointers....557
uuid....557
warpSize....557
cudaEglFrame....558
eglColorFormat....558
frameType....558
pArray....558
planeCount....558
planeDesc....558
pPitch....558
cudaEglPlaneDesc....558
channelDesc....559
depth....559
height....559
numChannels....559
pitch....559
reserved....559
width....559
cudaEventRecordNodeParams....559
event....559
cudaEventWaitNodeParams....560
event....560
cudaExtent....560
depth....560
height....560

width......560
cudaExternalMemoryBufferDesc......560
flags......560
offset......561
size......561
cudaExternalMemoryHandleDesc......561
fd......561
flags......561
handle......561
name......561
nvSciBufObject......561
size......561
type......562
win32......562
cudaExternalMemoryMipmappedArrayDesc......562
extent......562
flags......562
formatDesc......562
numLevels......563
offset......563
cudaExternalSemaphoreHandleDesc......563
fd......563
flags......563
handle......563
name......563
nvSciSyncObj......563
type......564
win32......564
cudaExternalSemaphoreSignalNodeParams......564
extSemArray......564
numExtSems......564
paramsArray......565
cudaExternalSemaphoreSignalNodeParamsV2......565
extSemArray......565
numExtSems......565
paramsArray......565
cudaExternalSemaphoreSignalParams......565
fence......565

fence....566
flags....566
keyedMutex....566
value....566
cudaExternalSemaphoreSignalParams_v1....566
fence....566
fence....566
flags....567
keyedMutex....567
value....567
cudaExternalSemaphoreWaitNodeParams....567
extSemArray....567
numExtSems....567
paramsArray....568
cudaExternalSemaphoreWaitNodeParamsV2....568
extSemArray....568
numExtSems....568
paramsArray....568
cudaExternalSemaphoreWaitParams....568
fence....568
fence....569
flags....569
key....569
keyedMutex....569
timeoutMs....569
value....569
cudaExternalSemaphoreWaitParams_v1....569
fence....570
fence....570
flags....570
key....570
keyedMutex....570
timeoutMs....570
value....570
cudaFuncAttributes....571
binaryVersion....571
cacheModeCA....571
clusterDimMustBeSet....571

clusterSchedulingPolicyPreference....571
constSizeBytes....571
localSizeBytes....571
maxDynamicSharedSizeBytes....571
maxThreadsPerBlock....571
nonPortableClusterSizeAllowed....572
numRegs....572
preferredShmemCarveout....572
ptxVersion....572
requiredClusterWidth....572
reserved....572
sharedSizeBytes....573
cudaGraphEdgeData....573
from_port....573
reserved....573
to_port....573
type....573
cudaGraphExecUpdateResultInfo....574
errorFromNode....574
errorNode....574
result....574
cudaGraphInstantiateParams....574
errNode_out....574
flags....574
result_out....575
uploadStream....575
cudaGraphKernelNodeUpdate....575
field....575
gridDim....575
isEnabled....575
node....575
offset....575
param....576
pValue....576
size....576
updateData....576
cudaGraphNodeParams....576
alloc....576

conditional....576
eventRecord....576
eventWait....577
extSemSignal....577
extSemWait....577
free....577
graph....577
host....577
kernel....577
memcpy....577
memset....578
reserved0....578
reserved1....578
reserved2....578
type....578
cudaHostNodeParams....578
fn....578
userData....578
cudaHostNodeParamsV2....578
fn....579
userData....579
cudaIpcEventHandle_t....579
cudaIpcMemHandle_t....579
cudaKernelNodeParams....579
blockDim....579
extra....579
func....579
gridDim....579
kernelParams....579
sharedMemBytes....580
cudaKernelNodeParamsV2....580
blockDim....580
extra....580
func....580
gridDim....580
kernelParams....580
sharedMemBytes....580
cudaLaunchAttribute....580

id....580
val....581
cudaLaunchAttributeValue....581
accessPolicyWindow....581
clusterDim....581
clusterSchedulingPolicyPreference....581
cooperative....581
deviceUpdatableKernelNode....581
launchCompletionEvent....582
memSyncDomain....582
memSyncDomainMap....582
preferredClusterDim....582
priority....582
programmaticEvent....583
programmaticStreamSerializationAllowed....583
sharedMemCarveout....583
syncPolicy....583
cudaLaunchConfig_t....583
attrs....583
blockDim....583
dynamicSmemBytes....584
gridDim....584
numAttrs....584
stream....584
cudaLaunchMemSyncDomainMap....584
default_....584
remote....584
cudaLaunchParams....584
args....585
blockDim....585
func....585
gridDim....585
sharedMem....585
stream....585
cudaMemAccessDesc....585
flags....585
location....585
cudaMemAllocNodeParams....585

accessDescCount....586
accessDescs....586
bytesize....586
dptr....586
poolProps....586
cudaMemAllocNodeParamsV2....586
accessDescCount....586
accessDescs....586
bytesize....587
dptr....587
poolProps....587
cudaMemcpy3DOperand....587
array....587
layerHeight....587
locHint....587
ptr....588
rowLength....588
cudaMemcpy3DParms....588
dstArray....588
dstPos....588
dstPtr....588
extent....588
kind....588
srcArray....588
srcPos....588
srcPtr....589
cudaMemcpy3DPeerParms....589
dstArray....589
dstDevice....589
dstPos....589
dstPtr....589
extent....589
srcArray....589
srcDevice....589
srcPos....589
srcPtr....590
cudaMemcpyAttributes....590
dstLocHint....590

flags....590
srcAccessOrder....590
srcLocHint....590
cudaMemcpyNodeParams....590
copyParams....590
flags....591
reserved....591
cudaMemFreeNodeParams....591
dptr....591
cudaMemLocation....591
id....591
type....591
cudaMemPoolProps....591
allocType....592
handleTypes....592
location....592
maxSize....592
reserved....592
usage....592
win32SecurityAttributes....592
cudaMemPoolPtrExportData....592
cudaMemsetParams....593
dst....593
elementSize....593
height....593
pitch....593
value....593
width....593
cudaMemsetParamsV2....593
dst....593
elementSize....593
height....594
pitch....594
value....594
width....594
cudaOffset3D....594
cudaPitchedPtr....594
pitch....594

ptr... 594
xsize...594
ysize...595
cudaPointerAttributes...595
device.. 595
devicePointer..595
hostPointer..595
type..595
cudaPos.. 595
x..596
y..596
z..596
cudaResourceDesc.. 596
array..596
desc..596
devPtr.. 596
height...596
mipmap..596
pitchInBytes.. 596
resType..597
sizeInBytes.. 597
width..597
cudaResourceViewDesc..597
depth..597
firstLayer..597
firstMipmapLevel..597
format.. 597
height...597
lastLayer...597
lastMipmapLevel..598
width..598
cudaTextureDesc..598
addressMode.. 598
borderColor.. 598
disableTrilinearOptimization.. 598
filterMode...598
maxAnisotropy...598
maxMipmapLevelClamp..598

minMipmapLevelClamp.......... 598
mipmapFilterMode.......... 599
mipmapLevelBias.......... 599
normalizedCoords.......... 599
readMode.......... 599
seamlessCubemap.......... 599
sRGB.......... 599
CUuuid_st.......... 599
bytes.......... 599
Chapter 8. Data Fields.......... 600
Chapter 9. Deprecated List.......... 618

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. The synchronous forms of these APIs issue these copies through the default stream.

Any CUDA API call may block or synchronize for various reasons such as contention for or unavailability of internal resources. Such behavior is subject to change and undocumented behavior should not be relied upon.

Memcpy

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

Synchronous

1. 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.
2. For transfers from pinned host memory to device memory, the function is synchronous with respect to the host.
3. For transfers from device to either pageable or pinned host memory, the function returns only once the copy has completed.
4. For transfers from device memory to device memory, no host-side synchronization is performed.
5. For transfers from any host memory to any host memory, the function is fully synchronous with respect to the host.

Asynchronous

1. For transfers between device memory and pageable host memory, the function might be synchronous with respect to host.
2. For transfers from any host memory to any host memory, the function is fully synchronous with respect to the host.

3. If pageable memory must first be staged to pinned memory, the driver may synchronize with the stream and stage the copy into pinned memory.
4. For all other transfers, the function should be fully asynchronous.

Memset

The cudaMemset functions are asynchronous with respect to the host except when the target memory is pinned host memory. 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:

```
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 CUstream (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 `CUstream` (`cudaStream_t`) handle `CU_STREAM_PER_THREAD` (`cudaStreamPerThread`).

Chapter 4. Graph object thread safety

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
- Device Management [DEPRECATED]
- Thread Management [DEPRECATED]
- Error Handling
- Stream Management
- Event Management
- External Resource Interoperability
- Execution Control
- Execution Control [DEPRECATED]
- 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
- EGL Interoperability
- Graphics Interoperability

- Texture Object Management
- Surface Object Management
- Version Management
- Graph Management
- Driver Entry Point Access
- Library Management
- C++ API Routines
- Interactions with the CUDA Driver API
- Profiler Control
- Data types used by CUDA Runtime

6.2. Device Management [DEPRECATED]

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

__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

Deprecated

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 cudaDeviceSetSharedMemConfig (cudaSharedMemConfig config)

Sets the shared memory configuration for the current device.

Parameters

config

- Requested cache configuration

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Deprecated

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

6.3. 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.

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 cudaThreadGetLimit (size_t *pValue, cudaLimit limit)

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.
- 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

__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 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.

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

6.4. 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

`char*` 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

`char*` 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 instance of the CUDA Runtime library in the host thread and resets it to cudaSuccess.

Note: Multiple instances of the CUDA Runtime library can be present in an application when using a library that statically links the CUDA Runtime.

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

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 instance of the CUDA Runtime library in the host thread. This call does not reset the error to cudaSuccess like cudaGetLastError().

Note: Multiple instances of the CUDA Runtime library can be present in an application when using a library that statically links the CUDA Runtime.

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

6.5. 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 an 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.

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 cudaStreamBeginCaptureToGraph (cudaStream_t stream, cudaGraph_t graph, const cudaGraphNode_t *dependencies, const cudaGraphEdgeData *dependencyData, size_t numDependencies, cudaStreamCaptureMode mode)

Begins graph capture on a stream to an existing graph.

Parameters

stream

- Stream in which to initiate capture.

graph

- Graph to capture into.

dependencies

- Dependencies of the first node captured in the stream. Can be NULL if numDependencies is 0.

dependencyData

- Optional array of data associated with each dependency.

numDependencies

- Number of dependencies.

mode

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

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 `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 `src` to destination stream `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

pStream

- Pointer to new stream identifier

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Creates a new asynchronous stream on the context that is current to the calling host thread. If no context is current to the calling host thread, then the primary context for a device is selected, made current to the calling thread, and initialized before creating a stream on it.

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, cudaStreamCreateWithFlags, cudaStreamGetPriority, cudaStreamGetFlags, cudaStreamGetDevice, cudaStreamQuery, cudaStreamSynchronize, cudaStreamWaitEvent, cudaStreamAddCallback, cudaSetDevice, cudaStreamDestroy, cuStreamCreate

__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 on the context that is current to the calling host thread. If no context is current to the calling host thread, then the primary context for a device is selected, made current to the calling thread, and initialized before creating a stream on it. 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.

See also:

cudaStreamCreate, cudaStreamCreateWithPriority, cudaStreamGetFlags, cudaStreamGetDevice, cudaStreamQuery, cudaStreamSynchronize, cudaStreamWaitEvent, cudaStreamAddCallback, cudaSetDevice, cudaStreamDestroy, cuStreamCreate

__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`. The stream is created on the context that is current to the calling host thread. If no context is current to the calling host thread, then the primary context for a device is selected, made current to the calling thread, and initialized before creating a stream on it. This affects the scheduling priority of work in the stream. Priorities provide a hint to preferentially run work with higher priority when possible, but do not preempt already-running work or provide any other functional guarantee on execution order.

`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:

cudaStreamCreate, cudaStreamCreateWithFlags, cudaDeviceGetStreamPriorityRange, cudaStreamGetPriority, cudaStreamQuery, cudaStreamWaitEvent, cudaStreamAddCallback, cudaStreamSynchronize, cudaSetDevice, cudaStreamDestroy, cuStreamCreateWithPriority

__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.
- 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:

cudaStreamCreate, cudaStreamCreateWithFlags, cudaStreamQuery, cudaStreamWaitEvent, cudaStreamSynchronize, cudaStreamAddCallback, cuStreamDestroy

__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.

Note:

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

See also:

cudaStreamCreate, cudaStreamBeginCapture, cudaStreamIsCapturing, cudaGraphDestroy

__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 *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.

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 the capture is terminated. 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

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaStreamGetCaptureInfo_v3, cudaStreamBeginCapture, cudaStreamIsCapturing, cudaStreamUpdateCaptureDependencies

__host__cudaError_t cudaStreamGetCaptureInfo_v3 (cudaStream_t stream, cudaStreamCaptureStatus *captureStatus_out, unsigned long long *id_out, cudaGraph_t *graph_out, const cudaGraphNode_t **dependencies_out, const cudaGraphEdgeData **edgeData_out, size_t *numDependencies_out)

Query a stream's capture state (12.3+).

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 the capture is terminated. 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.

edgeData_out

- Optional location to store a pointer to an array of graph edge data. This array parallels `dependencies_out`; the next node to be added has an edge to `dependencies_out`[i] with annotation `edgeData_out`[i] for each `i`. The array pointer is valid until the next API call which operates on the stream or until the capture is terminated.

numDependencies_out

- Optional location to store the size of the array returned in dependencies_out.

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorStreamCaptureImplicit, cudaErrorLossyQuery

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

If `edgeData_out` is non-NULL then `dependencies_out` must be as well. If `dependencies_out` is non-NULL and `edgeData_out` is NULL, but there is non-zero edge data for one or more of the current stream dependencies, the call will return cudaErrorLossyQuery.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaStreamBeginCapture, cudaStreamIsCapturing, cudaStreamUpdateCaptureDependencies

__host__cudaError_t cudaStreamGetDevice (cudaStream_t hStream, int *device)

Query the device of a stream.

Parameters

hStream
- Handle to the stream to be queried

device
- Returns the device to which the stream belongs

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorDeviceUnavailable,

Description

Returns in `*device` the device 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:

cudaSetDevice, cudaGetDevice, cudaStreamCreate, cudaStreamGetPriority, cudaStreamGetFlags, cuStreamGetId

__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, cudaStreamGetDevice, cuStreamGetFlags

__host__cudaError_t cudaStreamGetId (cudaStream_t hStream, unsigned long long *streamId)

Query the Id of a stream.

Parameters

hStream

- Handle to the stream to be queried

streamId

- Pointer to an unsigned long long in which the stream Id is returned

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle

Description

Query the Id of a stream. The Id is returned in `streamId`. The Id is unique for the life of the program.

The stream handle `hStream` can refer to any of the following:

- a stream created via any of the CUDA runtime APIs such as cudaStreamCreate, cudaStreamCreateWithFlags and cudaStreamCreateWithPriority, or their driver API equivalents such as cuStreamCreate or cuStreamCreateWithPriority. Passing an invalid handle will result in undefined behavior.
- any of the special streams such as the NULL stream, cudaStreamLegacy and cudaStreamPerThread respectively. The driver API equivalents of these are also accepted which are NULL, CU_STREAM_LEGACY and CU_STREAM_PER_THREAD.

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, cudaStreamGetFlags, cuStreamGetId

__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 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, cudaStreamGetDevice, 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
value

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+).

Parameters

stream
  - The stream to update

dependencies
  - The set of dependencies to add

numDependencies
  - The size of the dependencies array

flags
  - See above

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,

__host__cudaError_t cudaStreamUpdateCaptureDependencies_v2 (cudaStream_t stream, cudaGraphNode_t *dependencies, const cudaGraphEdgeData *dependencyData, size_t numDependencies, unsigned int flags)

Update the set of dependencies in a capturing stream (12.3+).

Parameters

stream

- The stream to update

dependencies

- The set of dependencies to add

dependencyData

- Optional array of data associated with each dependency.

numDependencies

- The size of the dependencies array

flags

- See above

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.

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

See also:

cudaStreamBeginCapture, cudaStreamGetCaptureInfo,

__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:

- 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

__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:

```
     cudaStreamCaptureMode mode = desiredMode;
          cudaThreadExchangeStreamCaptureMode(&mode);
          ...
          cudaThreadExchangeStreamCaptureMode(&mode); // restore previous mode
```

During stream capture (see cudaStreamBeginCapture), some actions, such as a call to cudaMalloc, may be unsafe. In the case of 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 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 cudaStreamBeginCapture-cudaStreamEndCapture sequence. This behavior can be controlled via this API and flags to cudaStreamBeginCapture.

A thread's mode is one of the following:

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

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

cudaStreamBeginCapture

6.6. 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, cudaErrorUnknown

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

__host__cudaError_t cudaEventElapsedTime_v2 (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, cudaErrorUnknown

Description

Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds). Note this API is not guaranteed to return the latest errors for pending work. As such this API is intended to serve as a elapsed time calculation only and polling for completion on the events to be compared should be done with cudaEventQuery instead.

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

__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

__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 CUDA context. 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.
- 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, 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 CUDA context. 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,

__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

6.7. 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:

cudaImportExternalSemaphore, 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

__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:

```
        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, cudaErrorOperatingSystem

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:

```
        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:

```
        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 approprriate 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.

See also:

cudaDestroyExternalMemory, cudaExternalMemoryGetMappedBuffer, cudaExternalMemoryGetMappedMipmappedArray

__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, cudaErrorOperatingSystem

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:

```
        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:

```
        typedef enum cudaExternalSemaphoreHandleType_enum {
                cudaExternalSemaphoreHandleTypeOpaqueFd                = 1,
                cudaExternalSemaphoreHandleTypeOpaqueWin32             = 2,
                cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt          = 3,
                cudaExternalSemaphoreHandleTypeD3D12Fence              = 4,
                cudaExternalSemaphoreHandleTypeD3D11Fence              = 5,
                cudaExternalSemaphoreHandleTypeNvSciSync               = 6,
                cudaExternalSemaphoreHandleTypeKeyedMutex              = 7,
                cudaExternalSemaphoreHandleTypeKeyedMutexKmt           = 8,
                cudaExternalSemaphoreHandleTypeTimelineSemaphoreFd     = 9,
                cudaExternalSemaphoreHandleTypeTimelineSemaphoreWin32  = 10
            } cudaExternalSemaphoreHandleType;
```

If cudaExternalSemaphoreHandleDesc::type is cudaExternalSemaphoreHandleTypeOpaqueFd, 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 cudaExternalSemaphoreHandleTypeOpaqueWin32, 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.

If cudaExternalSemaphoreHandleDesc::type is cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt, then cudaExternalSemaphoreHandleDesc::handle::win32::handle must be non-NULL and 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 cudaExternalSemaphoreHandleDesc::type is cudaExternalSemaphoreHandleTypeD3D12Fence, 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 is returned by ID3D12Device::CreateSharedHandle when referring to a ID3D12Fence object. This handle holds a reference to the underlying object. If cudaExternalSemaphoreHandleDesc::handle::win32::name is not NULL, then it must name a valid synchronization object that refers to a valid ID3D12Fence object.

If cudaExternalSemaphoreHandleDesc::type is cudaExternalSemaphoreHandleTypeD3D11Fence, 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 is returned by 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

__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.

cudaExternalSemaphoreSignalParams::params::nvSciSync::fence associated with semaphore object of the type cudaExternalSemaphoreHandleTypeNvSciSync can be deterministic. For this the NvSciSyncAttrList used to create the semaphore object must have value of NvSciSyncAttrKey_RequireDeterministicFences key set to true. Deterministic fences allow users to enqueue a wait over the semaphore object even before corresponding signal is enqueued. For such a semaphore object, CUDA guarantees that each signal operation will increment the fence value by '1'. Users are expected to track count of signals enqueued on the semaphore object and insert waits accordingly. When such a semaphore object is signaled from multiple streams, due to concurrent stream execution, it is possible that the order in which the semaphore gets signaled is indeterministic. This could lead to waiters of the semaphore getting unblocked incorrectly. Users are expected to handle such situations, either by not using the same semaphore object with deterministic fence support enabled in different streams or by adding explicit dependency amongst such streams so that the semaphore is signaled in order.

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 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

6.8. 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 it is assumed to be a cudaKernel_t and used as is. 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 cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

See also:

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

__host__cudaError_t cudaFuncGetName (const char **name, const void *func)

Returns the function name for a device entry function pointer.

Parameters

name

- The returned name of the function

func

- The function pointer to retrieve name for

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDeviceFunction

Description

Returns in `**name` the function name associated with the symbol `func` . The function name is returned as a null-terminated string. This API may return a mangled name if the function is not declared as having C linkage. If `**name` is NULL, cudaErrorInvalidValue is returned. If `func` is not a device entry function, then it is assumed to be a cudaKernel_t and used as is.

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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

cudaFuncGetName (C++ API)

__host__cudaError_t cudaFuncGetParamInfo (const void *func, size_t paramIndex, size_t *paramOffset, size_t *paramSize)

Returns the offset and size of a kernel parameter in the device-side parameter layout.

Parameters

func

- The function to query

paramIndex

- The parameter index to query

paramOffset

- The offset into the device-side parameter layout at which the parameter resides

paramSize

- The size of the parameter in the device-side parameter layout

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE,

Description

Queries the kernel parameter at `paramIndex` in `func's` list of parameters and returns parameter information via `paramOffset` and `paramSize`. `paramOffset` returns the offset of the parameter in the device-side parameter layout. `paramSize` returns the size in bytes of the parameter. This information can be used to update kernel node parameters from the device via cudaGraphKernelNodeSetParam() and cudaGraphKernelNodeUpdatesApply(). `paramIndex` must be less than the number of parameters that `func` takes.

Note:

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

> - The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
> - Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

__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 it is assumed to be a cudaKernel_t and used as is. 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.

- cudaFuncAttributeRequiredClusterWidth: The required cluster width in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.
- cudaFuncAttributeRequiredClusterHeight: The required cluster height in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.
- cudaFuncAttributeRequiredClusterDepth: The required cluster depth in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.
- cudaFuncAttributeNonPortableClusterSizeAllowed: Indicates whether the function can be launched with non-portable cluster size. 1 is allowed, 0 is disallowed.
- cudaFuncAttributeClusterSchedulingPolicyPreference: The block scheduling policy of a function. The value type is cudaClusterSchedulingPolicy.

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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

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

__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:

- 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.
- 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.
- This API does not accept a cudaKernel_t casted as void*. If cache config modification is required for a cudaKernel_t (or a __global__ function), it can be replaced with a call to cudaFuncSetAttributes with the attribute cudaFuncAttributePreferredSharedMemoryCarveout to specify a more granular L1 cache and shared memory split configuration.

See also:

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

__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 cudaGridDependencySynchronize (void)

Programmatic grid dependency synchronization.

Description

This device function will block the thread until all direct grid dependencies have completed. This API is intended to use in conjuncture with programmatic / launch event / dependency. See cudaLaunchAttributeID::cudaLaunchAttributeProgrammaticStreamSerialization and cudaLaunchAttributeID::cudaLaunchAttributeProgrammaticEvent for more information.

__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 dimentions

blockDim
- Block dimentions

args
- Arguments

sharedMem
- Shared memory

stream
- Stream identifier

Returns

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

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.

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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

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 responsiblity 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:

```
        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 functions 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

__device__cudaError_t cudaLaunchDevice (void *func, void *parameterBuffer, dim3 gridDimension, dim3 blockDimension, unsigned int sharedMemSize, cudaStream_t stream)

Launches a specified kernel.

Parameters

func

- Pointer to the kernel to be launched

parameterBuffer

- Holds the parameters to the launched kernel. parameterBuffer can be NULL. (Optional)

gridDimension

- Specifies grid dimensions

blockDimension

- Specifies block dimensions

sharedMemSize

- Specifies size of shared memory

stream

- Specifies the stream to be used

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 cudaLaunchHostFunc (cudaStream_t stream, cudaHostFn_t fn, void *userData)

Enqueues a host function call in a stream.

Parameters

stream

fn

- The function to call once preceding stream operations are complete

userData

- User-specified data to be passed to the function

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 constrast 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 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, cudaStreamQuery, cudaStreamSynchronize, cudaStreamWaitEvent, cudaStreamDestroy, cudaMallocManaged, cudaStreamAttachMemAsync, cudaStreamAddCallback, cuLaunchHostFunc

__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 dimentions

blockDim
- Block dimentions

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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

See also:

cudaLaunchKernel (C++ API), cuLaunchKernel

__host__cudaError_t cudaLaunchKernelExC (const cudaLaunchConfig_t *config, const void *func, void **args)

Launches a CUDA function with launch-time configuration.

Parameters

config
- Launch configuration

func
- Kernel to launch

args
- Array of pointers to kernel parameters

Returns

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

Description

Note that the functionally equivalent variadic template cudaLaunchKernelEx is available for C++11 and newer.

Invokes the kernel `func` on `config->gridDim` (`config->gridDim.x` `config->gridDim.y` `config->gridDim.z`) grid of blocks. Each block contains `config->blockDim` (`config->blockDim.x` `config->blockDim.y` `config->blockDim.z`) threads.

`config->dynamicSmemBytes` sets the amount of dynamic shared memory that will be available to each thread block.

`config->stream` specifies a stream the invocation is associated to.

Configuration beyond grid and block dimensions, dynamic shared memory size, and stream can be provided with the following two fields of `config`:

`config->attrs` is an array of `config->numAttrs` contiguous cudaLaunchAttribute elements. The value of this pointer is not considered if `config->numAttrs` is zero. However, in that case, it is recommended to set the pointer to NULL. `config->numAttrs` is the number of attributes populating the first `config->numAttrs` positions of the `config->attrs` array.

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.

N.B. This function is so named to avoid unintentionally invoking the templated version, `cudaLaunchKernelEx`, for kernels taking a single void** or void* parameter.

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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

See also:

cudaLaunchKernelEx(const cudaLaunchConfig_t *config, void (*kernel)(ExpTypes...), ActTypes &&... args) "cudaLaunchKernelEx (C++ API)", cuLaunchKernelEx

__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 cudaSetDoubleForHost (double *d)

Converts a double argument after execution 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` from a potentially 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), cudaSetDoubleForDevice

__device__ void cudaTriggerProgrammaticLaunchCompletion (void)

Programmatic dependency trigger.

Description

This device function ensures the programmatic launch completion edges / events are fulfilled. See cudaLaunchAttributeID::cudaLaunchAttributeProgrammaticStreamSerialization and cudaLaunchAttributeID::cudaLaunchAttributeProgrammaticEvent for more information. The event / edge kick off only happens when every CTAs in the grid has either exited or called this function at least once, otherwise the kick off happens automatically after all warps finishes execution but before the grid completes. The kick off only enables scheduling of the secondary kernel. It provides no memory visibility guarantee itself. The user could enforce memory visibility by inserting a memory fence of the correct scope.

6.9. Execution Control [DEPRECATED]

This section describes the deprecated 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__cudaError_t cudaFuncSetSharedMemConfig (const void *func, cudaSharedMemConfig config)

Sets the shared memory configuration for a device function.

Parameters

func

- Device function symbol

config

- Requested shared memory configuration

Returns

cudaSuccess, cudaErrorInvalidDeviceFunction, cudaErrorInvalidValue,

Description

Deprecated

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

6.10. 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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags, cudaOccupancyMaxPotentialBlockSize (C++ API), cudaOccupancyMaxPotentialBlockSizeWithFlags (C++ API), cudaOccupancyMaxPotentialBlockSizeVariableSMem (C++ API), cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags (C++ API), cudaOccupancyAvailableDynamicSMemPerBlock

__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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags, cudaOccupancyMaxPotentialBlockSize (C++ API), cudaOccupancyMaxPotentialBlockSizeWithFlags (C++ API), cudaOccupancyMaxPotentialBlockSizeVariableSMem (C++ API), cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags (C++ API), cudaOccupancyAvailableDynamicSMemPerBlock (C++ API), cuOccupancyMaxActiveBlocksPerMultiprocessor

__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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessor, cudaOccupancyMaxPotentialBlockSize (C++ API), cudaOccupancyMaxPotentialBlockSizeWithFlags (C++ API), cudaOccupancyMaxPotentialBlockSizeVariableSMem (C++ API), cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags (C+ + API), cudaOccupancyAvailableDynamicSMemPerBlock (C++ API), cuOccupancyMaxActiveBlocksPerMultiprocessorWithFlags

__host__cudaError_t cudaOccupancyMaxActiveClusters (int *numClusters, const void *func, const cudaLaunchConfig_t *launchConfig)

Given the kernel function (`func`) and launch configuration (`config`), return the maximum number of clusters that could co-exist on the target device in `*numClusters`.

Parameters

numClusters

- Returned maximum number of clusters that could co-exist on the target device

func

- Kernel function for which maximum number of clusters are calculated

launchConfig

Returns

cudaSuccess, cudaErrorInvalidDeviceFunction, cudaErrorInvalidValue, cudaErrorInvalidClusterSize, cudaErrorUnknown,

Description

If the function has required cluster size already set (see cudaFuncGetAttributes), the cluster size from config must either be unspecified or match the required size. Without required sizes, the cluster size must be specified in config, else the function will return an error.

Note that various attributes of the kernel function may affect occupancy calculation. Runtime environment may affect how the hardware schedules the clusters, so the calculated occupancy is not guaranteed to be achievable.

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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

See also:

cudaFuncGetAttributes cudaOccupancyMaxActiveClusters (C++ API), cuOccupancyMaxActiveClusters

__host__cudaError_t cudaOccupancyMaxPotentialClusterSize (int *clusterSize, const void *func, const cudaLaunchConfig_t *launchConfig)

Given the kernel function (`func`) and launch configuration (`config`), return the maximum cluster size in `*clusterSize`.

Parameters

clusterSize

- Returned maximum cluster size that can be launched for the given kernel function and launch configuration

func

- Kernel function for which maximum cluster size is calculated

launchConfig

Returns

cudaSuccess, cudaErrorInvalidDeviceFunction, cudaErrorInvalidValue, cudaErrorUnknown,

Description

The cluster dimensions in `config` are ignored. If func has a required cluster size set (see cudaFuncGetAttributes),`*clusterSize` will reflect the required cluster size.

By default this function will always return a value that's portable on future hardware. A higher value may be returned if the kernel function allows non-portable cluster sizes.

This function will respect the compile time launch bounds.

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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.

- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

See also:

cudaFuncGetAttributes cudaOccupancyMaxPotentialClusterSize (C++ API), cuOccupancyMaxPotentialClusterSize

6.11. 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 (cudaChannelFormatDesc *desc, cudaExtent *extent, unsigned int *flags, cudaArray_t array)

Gets 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 cudaArrayGetMemoryRequirements (cudaArrayMemoryRequirements *memoryRequirements, cudaArray_t array, int device)

Returns the memory requirements of a CUDA array.

Parameters

memoryRequirements
- Pointer to cudaArrayMemoryRequirements

array
- CUDA array to get the memory requirements of

device
- Device to get the memory requirements for

Returns

cudaSuccess cudaErrorInvalidValue

Description

Returns the memory requirements of a CUDA array in `memoryRequirements` If the CUDA array is not allocated with flag cudaArrayDeferredMapping cudaErrorInvalidValue will be returned.

The returned value in cudaArrayMemoryRequirements::size represents the total size of the CUDA array. The returned value in cudaArrayMemoryRequirements::alignment represents the alignment necessary for mapping the CUDA array.

See also:

cudaMipmappedArrayGetMemoryRequirements

__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 one of the following memory allocation APIs - cudaMalloc(), cudaMallocPitch(), cudaMallocManaged(), cudaMallocAsync(), cudaMallocFromPoolAsync().

Note - This API will not perform any implicit synchronization when the pointer was allocated with cudaMallocAsync or cudaMallocFromPoolAsync. Callers must ensure that all accesses to these pointer have completed before invoking cudaFree. For best performance and memory reuse, users should use cudaFreeAsync to free memory allocated via the stream ordered memory allocator. For all other pointers, this API may perform implicit synchronization.

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, cudaMallocPitch, cudaMallocManaged, cudaMallocArray, cudaFreeArray, cudaMallocAsync, cudaMallocFromPoolAsync cudaMallocHost (C API), cudaFreeHost, cudaMalloc3D, cudaMalloc3DArray, cudaFreeAsync cudaHostAlloc, 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

__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:

cudaMalloc, cudaMallocPitch, cudaFree, cudaMallocArray, cudaMallocHost (C API), cudaFreeHost, cudaHostAlloc, cuMipmappedArrayDestroy

__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. 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, cuMipmappedArrayGetLevel

__host__cudaError_t cudaGetSymbolAddress (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.

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 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), cuModuleGetGlobal

__host__cudaError_t cudaGetSymbolSize (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 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), 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 cudaHostAlloc() to emulate cudaMallocHost().
- 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 cudaHostGetDevicePointer().
- 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

__host__cudaError_t cudaHostGetDevicePointer (void **pDevice, void *pHost, unsigned int flags)

Passes back device pointer of mapped host memory allocated by cudaHostAlloc or registered by cudaHostRegister.

Parameters

pDevice
- Returned device pointer for mapped memory

pHost
- Requested host pointer mapping

flags
- Flags for extensions (must be 0 for now)

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.

On systems where pageableMemoryAccessUsesHostPageTables is true, cudaHostRegister will not page-lock the memory range specified by `ptr` but only populate unpopulated pages.

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 cudaDevAttrHostRegisterReadOnlySupported. 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 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, 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` * `height` * `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

__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 (`width` field in elements)

flags

- Flags for extensions

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.

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 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 onto a physical memory allocation 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.
- cudaArrayDeferredMapping: Allocates a CUDA array without physical backing memory. The entire array can later be mapped onto a physical memory allocation by calling cuMemMapArrayAsync. 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,maxTexture2DGather[0]), (1,maxTexture2DGather[1]), 0).

CUDA array type	**Valid extents that must always be met {(width range in elements), (height range), (depth range)}**	**Valid extents with cudaArraySurfaceLoadStore set {(width range in elements), (height range), (depth range)}**
1D	{ (1,maxTexture1D), 0, 0 }	{ (1,maxSurface1D), 0, 0 }
2D	{ (1,maxTexture2D[0]), (1,maxTexture2D[1]), 0 }	{ (1,maxSurface2D[0]), (1,maxSurface2D[1]), 0 }
3D	{ (1,maxTexture3D[0]), (1,maxTexture3D[1]), (1,maxTexture3D[2]) } OR { (1,maxTexture3DAlt[0]),	{ (1,maxSurface3D[0]), (1,maxSurface3D[1]), (1,maxSurface3D[2]) }

CUDA array type	Valid extents that must always be met {(width range in elements), (height range), (depth range)}	Valid extents with cudaArraySurfaceLoadStore set {(width range in elements), (height range), (depth range)}
	(1,maxTexture3DAlt[1]), (1,maxTexture3DAlt[2]) }	
1D Layered	{ (1,maxTexture1DLayered[0]), 0, (1,maxTexture1DLayered[1]) }	{ (1,maxSurface1DLayered[0]), 0, (1,maxSurface1DLayered[1]) }
2D Layered	{ (1,maxTexture2DLayered[0]), (1,maxTexture2DLayered[1]), (1,maxTexture2DLayered[2]) }	{ (1,maxSurface2DLayered[0]), (1,maxSurface2DLayered[1]), (1,maxSurface2DLayered[2]) }
Cubemap	{ (1,maxTextureCubemap), (1,maxTextureCubemap), 6 }	{ (1,maxSurfaceCubemap), (1,maxSurfaceCubemap), 6 }
Cubemap Layered	{ (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[1]) }	{ (1,maxSurfaceCubemapLayered[0]), (1,maxSurfaceCubemapLayered[0]), (1,maxSurfaceCubemapLayered[1]) }

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

__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 onto a physical memory allocation by calling cuMemMapArrayAsync. The physical backing memory must be allocated via cuMemCreate.
- cudaArrayDeferredMapping: Allocates a CUDA array without physical backing memory. The entire array can later be mapped onto a physical memory allocation 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

__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().

On systems where pageableMemoryAccessUsesHostPageTables is true, cudaMallocHost may not page-lock the allocated memory.

Page-locking 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

__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 specifed 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 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 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 cudaMallocManaged 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.

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:

```
    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 mipmapped array without physical backing memory. The subregions within this sparse array can later be mapped onto a physical memory allocation 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.
- cudaArrayDeferredMapping: Allocates a CUDA mipmapped array without physical backing memory. The entire array can later be mapped onto a physical memory allocation by calling cuMemMapArrayAsync. 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.

CUDA array type	**Valid extents that must always be met {(width range in elements), (height range), (depth range)}**	**Valid extents with cudaArraySurfaceLoadStore set {(width range in elements), (height range), (depth range)}**
1D	{ (1,maxTexture1DMipmap), 0, 0 }	{ (1,maxSurface1D), 0, 0 }
2D	{ (1,maxTexture2DMipmap[0]), (1,maxTexture2DMipmap[1]), 0 }	{ (1,maxSurface2D[0]), (1,maxSurface2D[1]), 0 }
3D	{ (1,maxTexture3D[0]), (1,maxTexture3D[1]), (1,maxTexture3D[2]) } OR { (1,maxTexture3DAlt[0]), (1,maxTexture3DAlt[1]), (1,maxTexture3DAlt[2]) }	{ (1,maxSurface3D[0]), (1,maxSurface3D[1]), (1,maxSurface3D[2]) }
1D Layered	{ (1,maxTexture1DLayered[0]), 0, (1,maxTexture1DLayered[1]) }	{ (1,maxSurface1DLayered[0]), 0, (1,maxSurface1DLayered[1]) }
2D Layered	{ (1,maxTexture2DLayered[0]), (1,maxTexture2DLayered[1]), (1,maxTexture2DLayered[2]) }	{ (1,maxSurface2DLayered[0]), (1,maxSurface2DLayered[1]), (1,maxSurface2DLayered[2]) }
Cubemap	{ (1,maxTextureCubemap), (1,maxTextureCubemap), 6 }	{ (1,maxSurfaceCubemap), (1,maxSurfaceCubemap), 6 }

CUDA array type	Valid extents that must always be met {(width range in elements), (height range), (depth range)}	Valid extents with cudaArraySurfaceLoadStore set {(width range in elements), (height range), (depth range)}
Cubemap Layered	{ (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[1]) }	{ (1,maxSurfaceCubemapLayered[0]), (1,maxSurfaceCubemapLayered[0]), (1,maxSurfaceCubemapLayered[1]) }

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, cuMipmappedArrayCreate

__host__cudaError_t cudaMallocPitch (void **devPtr, size_t *pitch, size_t width, size_t height)

Allocates pitched memory on the device.

Parameters

devPtr

- Pointer to allocated pitched device memory

pitch

- Pitch for allocation

width

- Requested pitched allocation width (in bytes)

height

- Requested pitched allocation height

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation

Description

Allocates at least `width` (in bytes) * `height` bytes of linear memory on the device and returns in `*devPtr` 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 cudaMallocPitch() 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 cudaMallocPitch(). 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 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, cudaMallocArray, cudaFreeArray, cudaMallocHost (C API), cudaFreeHost, cudaMalloc3D, cudaMalloc3DArray, cudaHostAlloc, cuMemAllocPitch

__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.

- 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 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, cudaMemcpyPeer, cudaMemcpyAsync, cudaMemcpy3DPeerAsync, cudaMemPrefetchAsync, cuMemAdvise

__host__cudaError_t cudaMemAdvise_v2 (const void *devPtr, size_t count, cudaMemoryAdvise advice, cudaMemLocation location)

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

location

- location 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 or cudaMemPrefetchAsync_v2 is called on this region, it will create a read-only copy of the data on the destination processor. If the target location for cudaMemPrefetchAsync_v2 is a host NUMA node and a read-only copy already exists on another host NUMA node, that copy will be migrated to the targeted host NUMA node. If any processor writes to this region, all copies of the corresponding page will be invalidated except for the one where the write occurred. If the writing processor is the CPU and the preferred location of the page is a host NUMA node, then the page will also be migrated to that host NUMA node. The location 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 cudaMemAdviseSetReadMostly 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. Note: The location argument is ignored for this advice.
- cudaMemAdviseSetPreferredLocation: This advice sets the preferred location for the data to be the memory belonging to location. When cudaMemLocation::type is cudaMemLocationTypeHost, cudaMemLocation::id is ignored and the preferred location is set to be host memory. To set the preferred location to a specific host NUMA node, applications must set cudaMemLocation::type to

cudaMemLocationTypeHostNuma and cudaMemLocation::id must specify the NUMA ID of the host NUMA node. If cudaMemLocation::type is set to cudaMemLocationTypeHostNumaCurrent, cudaMemLocation::id will be ignored and the host NUMA node closest to the calling thread's CPU will be used as the preferred location. If cudaMemLocation::type is a cudaMemLocationTypeDevice, then cudaMemLocation::id must be a valid device ordinal and the device 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 `location` will not result in a read-only copy being created on that procesor as outlined in description for the advice cudaMemAdviseSetReadMostly. If the memory region refers to valid system-allocated pageable memory, and cudaMemLocation::type is cudaMemLocationTypeDevice then cudaMemLocation::id must be a valid device that has a non-zero alue for the device attribute cudaDevAttrPageableMemoryAccess.

- cudaMemAdviseUnsetPreferredLocation: Undoes the effect of cudaMemAdviseSetPreferredLocation and changes the preferred location to none. The `location` argument is ignored for this advice.
- cudaMemAdviseSetAccessedBy: This advice implies that the data will be accessed by processor `location`. The cudaMemLocation::type must be either cudaMemLocationTypeDevice with cudaMemLocation::id representing a valid device ordinal or cudaMemLocationTypeHost and cudaMemLocation::id will be ignored. All other location types are invalid. If cudaMemLocation::id 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 cudaMemAdviseSetAccessedBy 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 `location`, then the policies associated with CU_MEM_ADVISE_SET_PREFERRED_LOCATION will override the policies of this advice. If the memory region refers to valid system-allocated pageable memory, and cudaMemLocation::type is cudaMemLocationTypeDevice then device in cudaMemLocation::id must have a non-zero value for the device attribute cudaDevAttrPageableMemoryAccess. Additionally, if cudaMemLocation::id has a non-zero value for the device attribute cudaDevAttrPageableMemoryAccessUsesHostPageTables, then this call has no effect.

- CU_MEM_ADVISE_UNSET_ACCESSED_BY: Undoes the effect of cudaMemAdviseSetAccessedBy. Any mappings to the data from `location` 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, and cudaMemLocation::type is cudaMemLocationTypeDevice then device in cudaMemLocation::id must have a non-zero value for the device attribute cudaDevAttrPageableMemoryAccess. Additionally, if cudaMemLocation::id 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 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, cudaMemcpyPeer, cudaMemcpyAsync, cudaMemcpy3DPeerAsync, cudaMemPrefetchAsync, cuMemAdvise, cuMemAdvise_v2

__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, 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. 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, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DToArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, cuMemcpyDtoH, cuMemcpyHtoD, cuMemcpyDtoD, cuMemcpy

__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 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, cudaMemcpy2DToArray, cudaMemcpy2DFromArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DToArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, cuMemcpy2D, cuMemcpy2DUnaligned

__host__cudaError_t cudaMemcpy2DArrayToArray (cudaArray_t dst, size_t wOffsetDst, size_t hOffsetDst, cudaArray_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 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, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DToArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, cuMemcpy2D, cuMemcpy2DUnaligned

__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

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 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, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DToArrayAsync, cudaMemcpy2DFromArrayAsync, 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

src
- 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, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DToArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, cuMemcpy2D, cuMemcpy2DUnaligned

__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)

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

__host__cudaError_t cudaMemcpy2DToArray (cudaArray_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 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, cudaMemcpy2DFromArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DToArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, cuMemcpy2D, cuMemcpy2DUnaligned

__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

```
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 cudaPitchedPtr
                srcPtr;
        cudaArray_t
                dstArray;
        struct cudaPos
                dstPos;
        struct cudaPitchedPtr
                dstPtr;
```

```
        struct cudaExtent
              extent;
        enum cudaMemcpyKind
              kind;
    };
```

cudaMemcpy3D() copies data betwen 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:

```
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

```
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 cudaPitchedPtr
                  srcPtr;
        cudaArray_t
                  dstArray;
        struct cudaPos
                  dstPos;
        struct cudaPitchedPtr
                  dstPtr;
        struct cudaExtent
                  extent;
        enum cudaMemcpyKind
                  kind;
      };
```

cudaMemcpy3DAsync() copies data betwen 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:

```
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 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, 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.

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:

cudaMalloc3D, cudaMalloc3DArray, cudaMemset3D, cudaMemcpy3D, cudaMemcpy, cudaMemcpy2D, cudaMemcpy2DToArray, :cudaMemcpy2DFromArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DToArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, make_cudaExtent, make_cudaPos, cuMemcpy3DAsync

__host__cudaError_t cudaMemcpy3DBatchAsync (size_t numOps, cudaMemcpy3DBatchOp *opList, size_t *failIdx, unsigned long long flags, cudaStream_t stream)

Performs a batch of 3D memory copies asynchronously.

Parameters

numOps
: - Total number of memcpy operations.

opList
: - Array of size `numOps` containing the actual memcpy operations.

failIdx
: - Pointer to a location to return the index of the copy where a failure was encountered. The value will be SIZE_MAX if the error doesn't pertain to any specific copy.

flags

- Flags for future use, must be zero now.

stream

Returns

cudaSuccess cudaErrorInvalidValue

Description

Performs a batch of memory copies. The batch as a whole executes in stream order but copies within a batch are not guaranteed to execute in any specific order. Note that this means specifying any dependent copies within a batch will result in undefined behavior.

Performs memory copies as specified in the `opList` array. The length of this array is specified in `numOps`. Each entry in this array describes a copy operation. This includes among other things, the source and destination operands for the copy as specified in cudaMemcpy3DBatchOp::src and cudaMemcpy3DBatchOp::dst respectively. The source and destination operands of a copy can either be a pointer or a CUDA array. The width, height and depth of a copy is specified in cudaMemcpy3DBatchOp::extent. The width, height and depth of a copy are specified in elements and must not be zero. For pointer-to-pointer copies, the element size is considered to be 1. For pointer to CUDA array or vice versa copies, the element size is determined by the CUDA array. For CUDA array to CUDA array copies, the element size of the two CUDA arrays must match.

For a given operand, if cudaMemcpy3DOperand::type is specified as cudaMemcpyOperandTypePointer, then cudaMemcpy3DOperand::op::ptr will be used. The cudaMemcpy3DOperand::op::ptr::ptr field must contain the pointer where the copy should begin. The cudaMemcpy3DOperand::op::ptr::rowLength field specifies the length of each row in elements and must either be zero or be greater than or equal to the width of the copy specified in cudaMemcpy3DBatchOp::extent::width. The cudaMemcpy3DOperand::op::ptr::layerHeight field specifies the height of each layer and must either be zero or be greater than or equal to the height of the copy specified in cudaMemcpy3DBatchOp::extent::height. When either of these values is zero, that aspect of the operand is considered to be tightly packed according to the copy extent. For managed memory pointers on devices where cudaDevAttrConcurrentManagedAccess is true or system-allocated pageable memory on devices where cudaDevAttrPageableMemoryAccess is true, the cudaMemcpy3DOperand::op::ptr::locHint field can be used to hint the location of the operand.

If an operand's type is specified as cudaMemcpyOperandTypeArray, then cudaMemcpy3DOperand::op::array will be used. The cudaMemcpy3DOperand::op::array::array field specifies the CUDA array and cudaMemcpy3DOperand::op::array::offset specifies the 3D offset into that array where the copy begins.

The cudaMemcpyAttributes::srcAccessOrder indicates the source access ordering to be observed for copies associated with the attribute. If the source access order is set to cudaMemcpySrcAccessOrderStream, then the source will be accessed in stream order. If the source access order is set to cudaMemcpySrcAccessOrderDuringApiCall then it indicates that access to the source pointer can be out of stream order and all accesses must be complete before the API call returns. This flag is suited for ephemeral sources (ex., stack variables) when it's known that no prior operations in the stream can be accessing the memory and also that the lifetime of the memory is limited to the scope that the source variable was declared in. Specifying this flag allows the driver to optimize the copy and removes the need for the user to synchronize the stream after the API call. If the source access order is set to cudaMemcpySrcAccessOrderAny then it indicates that access to the source pointer can be out of stream order and the accesses can happen even after the API call returns. This flag is suited for host pointers allocated outside CUDA (ex., via malloc) when it's known that no prior operations in the stream can be accessing the memory. Specifying this flag allows the driver to optimize the copy on certain platforms. Each memcopy operation in `opList` must have a valid srcAccessOrder setting, otherwise this API will return cudaErrorInvalidValue.

The cudaMemcpyAttributes::flags field can be used to specify certain flags for copies. Setting the cudaMemcpyFlagPreferOverlapWithCompute flag indicates that the associated copies should preferably overlap with any compute work. Note that this flag is a hint and can be ignored depending on the platform and other parameters of the copy.

If any error is encountered while parsing the batch, the index within the batch where the error was encountered will be returned in `failIdx`.

Note:

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

__host__cudaError_t cudaMemcpy3DPeer (const cudaMemcpy3DPeerParms *p)

Copies memory between devices.

Parameters

p

- Parameters for the memory copy

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevice, cudaErrorInvalidPitchValue

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 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, 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, cudaErrorInvalidValue, cudaErrorInvalidDevice, cudaErrorInvalidPitchValue

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. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaMemcpy, cudaMemcpyPeer, cudaMemcpyAsync, cudaMemcpyPeerAsync, cudaMemcpy3DPeerAsync, cuMemcpy3DPeerAsync

__host____device__cudaError_t 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, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, cuMemcpyAsync, cuMemcpyDtoHAsync, cuMemcpyHtoDAsync, cuMemcpyDtoDAsync

__host__cudaError_t cudaMemcpyBatchAsync (void **dsts, void **srcs, size_t *sizes, size_t count, cudaMemcpyAttributes *attrs, size_t *attrsIdxs, size_t numAttrs, size_t *failIdx, cudaStream_t stream)

Performs a batch of memory copies asynchronously.

Parameters

dsts

- Array of destination pointers.

srcs

- Array of memcpy source pointers.

sizes

- Array of sizes for memcpy operations.

count

- Size of `dsts`, `srcs` and `sizes` arrays

attrs

- Array of memcpy attributes.

attrsIdxs

- Array of indices to specify which copies each entry in the `attrs` array applies to. The attributes specified in attrs[k] will be applied to copies starting from attrsIdxs[k] through attrsIdxs[k+1] - 1. Also attrs[numAttrs-1] will apply to copies starting from attrsIdxs[numAttrs-1] through count - 1.

numAttrs

- Size of `attrs` and `attrsIdxs` arrays.

failIdx

- Pointer to a location to return the index of the copy where a failure was encountered. The value will be SIZE_MAX if the error doesn't pertain to any specific copy.

stream

Returns

cudaSuccess cudaErrorInvalidValue

Description

Performs a batch of memory copies. The batch as a whole executes in stream order but copies within a batch are not guaranteed to execute in any specific order. This API only supports pointer-to-pointer copies. For copies involving CUDA arrays, please see cudaMemcpy3DBatchAsync.

Performs memory copies from source buffers specified in `srcs` to destination buffers specified in `dsts`. The size of each copy is specified in `sizes`. All three arrays must be of the same length as specified by `count`. Since there are no ordering guarantees for copies within a batch, specifying any dependent copies within a batch will result in undefined behavior.

Every copy in the batch has to be associated with a set of attributes specified in the `attrs` array. Each entry in this array can apply to more than one copy. This can be done by specifying in the `attrsIdxs` array, the index of the first copy that the corresponding entry in the `attrs` array applies to. Both `attrs` and `attrsIdxs` must be of the same length as specified by `numAttrs`. For example, if a batch has 10 copies listed in dst/src/sizes, the first 6 of which have one set of attributes and the remaining 4 another, then `numAttrs` will be 2, `attrsIdxs` will be {0, 6} and `attrs` will contains the two sets of attributes. Note that the first entry in `attrsIdxs` must always be 0. Also, each entry must be greater than the previous entry and the last entry should be less than `count`. Furthermore, `numAttrs` must be lesser than or equal to `count`.

The cudaMemcpyAttributes::srcAccessOrder indicates the source access ordering to be observed for copies associated with the attribute. If the source access order is set to cudaMemcpySrcAccessOrderStream, then the source will be accessed in stream order. If the source access order is set to cudaMemcpySrcAccessOrderDuringApiCall then it indicates that access to the source pointer can be out of stream order and all accesses must be complete before the API call returns. This flag is suited for ephemeral sources (ex., stack variables) when it's known that no prior operations in the stream can be accessing the memory and also that the lifetime of the memory is limited to the scope that the source variable was declared in. Specifying this flag allows the driver to optimize the copy and removes the need for the user to synchronize the stream after the API call. If the source access order is set to cudaMemcpySrcAccessOrderAny then it indicates that access to the source pointer can be out of stream order and the accesses can happen even after the API call returns. This flag is suited for host pointers allocated outside CUDA (ex., via malloc) when it's known that no prior operations in the stream can be accessing the memory. Specifying this flag allows the driver to optimize the copy on certain platforms. Each memcpy operation in the batch must have a valid cudaMemcpyAttributes corresponding to it including the appropriate srcAccessOrder setting, otherwise the API will return cudaErrorInvalidValue.

The cudaMemcpyAttributes::srcLocHint and cudaMemcpyAttributes::dstLocHint allows applications to specify hint locations for operands of a copy when the operand doesn't have a fixed location. That is, these hints are only applicable for managed memory pointers on devices where cudaDevAttrConcurrentManagedAccess is true or system-allocated pageable memory on devices where cudaDevAttrPageableMemoryAccess is true. For other cases, these hints are ignored.

The cudaMemcpyAttributes::flags field can be used to specify certain flags for copies. Setting the cudaMemcpyFlagPreferOverlapWithCompute flag indicates that the associated copies should preferably overlap with any compute work. Note that this flag is a hint and can be ignored depending on the platform and other parameters of the copy.

If any error is encountered while parsing the batch, the index within the batch where the error was encountered will be returned in `failIdx`.

Note:

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

__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, cudaErrorInvalidMemcpyDirection, cudaErrorNoKernelImageForDevice

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 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, 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, cudaErrorInvalidValue, cudaErrorInvalidSymbol, cudaErrorInvalidMemcpyDirection, cudaErrorNoKernelImageForDevice

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 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, cuMemcpyAsync, cuMemcpyDtoHAsync, cuMemcpyDtoDAsync

__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

src

- Source device pointer

srcDevice
- 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 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, 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:

cudaMemcpy, 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, 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 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, cuMemcpy, cuMemcpyHtoD, cuMemcpyDtoD

__host__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 `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 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, cudaMemcpyFromSymbolAsync, cuMemcpyAsync, cuMemcpyHtoDAsync, cuMemcpyDtoDAsync

__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 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-tenet 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 cudaMallocManaged or declared via __managed__ variables, or it may also refer to system-allocated memory on systems with non-zero cudaDevAttrPageableMemoryAccess.

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 cudaMallocManaged allocations to host memory in order to make room. Device memory allocated using cudaMalloc or cudaMallocArray 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 `dstDevice`. The exact behavior however also depends on the settings applied to this memory range via cudaMemAdvise as described below:

If cudaMemAdviseSetReadMostly was set on any subset of this memory range, then that subset will create a read-only copy of the pages on `dstDevice`.

If cudaMemAdviseSetPreferredLocation was called on any subset of this memory range, then the pages will be migrated to `dstDevice` even if `dstDevice` is not the preferred location of any pages in the memory range.

If 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.

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, cudaMemcpyPeer, cudaMemcpyAsync, cudaMemcpy3DPeerAsync, cudaMemAdvise, cudaMemAdvise_v2 cuMemPrefetchAsync

__host__cudaError_t cudaMemPrefetchAsync_v2 (const void *devPtr, size_t count, cudaMemLocation location, unsigned int flags, cudaStream_t stream)

Prefetches memory to the specified destination location.

Parameters

devPtr

- Pointer to be prefetched

count

- Size in bytes

location

- location to prefetch to

flags

- flags for future use, must be zero now.

stream

- Stream to enqueue prefetch operation

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevice

Description

Prefetches memory to the specified destination location. `devPtr` is the base device pointer of the memory to be prefetched and `location` specifies the destination location. `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 cudaMallocManaged or declared via __managed__ variables, or it may also refer to system-allocated memory on systems with non-zero cudaDevAttrPageableMemoryAccess.

Specifying cudaMemLocationTypeDevice for cudaMemLocation::type will prefetch memory to GPU specified by device ordinal cudaMemLocation::id which must have non-zero value for the device attribute concurrentManagedAccess. Additionally, `stream` must be associated with a device that has a non-zero value for the device attribute concurrentManagedAccess. Specifying cudaMemLocationTypeHost as cudaMemLocation::type will prefetch data to host memory. Applications can request prefetching memory to a specific host NUMA node by specifying cudaMemLocationTypeHostNuma for cudaMemLocation::type and a valid host NUMA node id in cudaMemLocation::id Users can also request prefetching memory to the host NUMA node closest to the current thread's CPU by specifying cudaMemLocationTypeHostNumaCurrent for cudaMemLocation::type. Note when cudaMemLocation::type is etiher cudaMemLocationTypeHost OR cudaMemLocationTypeHostNumaCurrent, cudaMemLocation::id will be ignored.

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 cudaMallocManaged allocations to host memory in order to make room. Device memory allocated using cudaMalloc or cudaMallocArray 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 the destination location. The exact behavior however also depends on the settings applied to this memory range via cuMemAdvise as described below:

If cudaMemAdviseSetReadMostly was set on any subset of this memory range, then that subset will create a read-only copy of the pages on destination location. If however the destination location is a host NUMA node, then any pages of that subset that are already in another host NUMA node will be transferred to the destination.

If cudaMemAdviseSetPreferredLocation was called on any subset of this memory range, then the pages will be migrated to `location` even if `location` is not the preferred location of any pages in the memory range.

If 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.

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, cudaMemcpyPeer, cudaMemcpyAsync, cudaMemcpy3DPeerAsync, cudaMemAdvise, cudaMemAdvise_v2 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 pointers 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, cudaInvalidDeviceId 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, cudaInvalidDeviceId, cudaInvalidDeviceId }. 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, cudaInvalidDeviceId will be returned. Note that this simply returns the last location that the applicaton requested to prefetch the memory range to. It gives no indication as to whether the prefetch operation to that location has completed or even begun.
- cudaMemRangeAttributePreferredLocationType: If this attribute is specified, `data` will be interpreted as a cudaMemLocationType, and `dataSize` must be sizeof(cudaMemLocationType). The cudaMemLocationType returned will be cudaMemLocationTypeDevice if all pages in the memory range have the same GPU as their preferred location, or cudaMemLocationType will be cudaMemLocationTypeHost if all pages in the memory range have the CPU as their preferred location, or or it will be cudaMemLocationTypeHostNuma if all the pages in the memory range have the same host NUMA node ID as their preferred location or it will be cudaMemLocationTypeInvalid if either all the pages don't have the same preferred location or some of the pages don't have a preferred location at all. Note that the actual location type of the pages in the memory range at the time of the query may be different from the preferred location type.

- cudaMemRangeAttributePreferredLocationId: If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. If the cudaMemRangeAttributePreferredLocationType query for the same address range returns cudaMemLocationTypeDevice, it will be a valid device ordinal or if it returns cudaMemLocationTypeHostNuma, it will be a valid host NUMA node ID or if it returns any other location type, the id should be ignored.

- cudaMemRangeAttributeLastPrefetchLocationType: If this attribute is specified, `data` will be interpreted as a cudaMemLocationType, and `dataSize` must be sizeof(cudaMemLocationType). The result returned will be the last location type to which all pages in the memory range were prefetched explicitly via cuMemPrefetchAsync. The cudaMemLocationType returned will be cudaMemLocationTypeDevice if the last prefetch location was the GPU or cudaMemLocationTypeHost if it was the CPU or cudaMemLocationTypeHostNuma if the last prefetch location was a specific host NUMA node. If any page in the memory range was never explicitly prefetched or if all pages were not prefetched to the same location, CUmemLocationType will be cudaMemLocationTypeInvalid. Note that this simply returns the last location type 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.

  - cudaMemRangeAttributeLastPrefetchLocationId: If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. If the cudaMemRangeAttributeLastPrefetchLocationType query for the same address range returns cudaMemLocationTypeDevice, it will be a valid device ordinal or if it returns cudaMemLocationTypeHostNuma, it will be a valid host NUMA node ID or if it returns any other location type, the id should be ignored.

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:

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
- :: cudaMemRangeAttributePreferredLocationType

- :: cudaMemRangeAttributePreferredLocationId
- :: cudaMemRangeAttributeLastPrefetchLocationType
- :: cudaMemRangeAttributeLastPrefetchLocationId

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:

cudaMemRangeGetAttribute, cudaMemAdvise, cudaMemPrefetchAsync, cuMemRangeGetAttributes

__host__cudaError_t cudaMemset (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.

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:

cuMemsetD8, cuMemsetD16, cuMemsetD32

__host__cudaError_t cudaMemset2D (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 `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 cudaMallocPitch().

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

__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 to 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 cudaMallocPitch().

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 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, cudaMemset3DAsync, cuMemsetD2D8Async, cuMemsetD2D16Async, cuMemsetD2D32Async

__host__cudaError_t cudaMemset3D (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 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, cudaMemsetAsync, cudaMemset2DAsync, cudaMemset3DAsync, cudaMalloc3D, make_cudaPitchedPtr, make_cudaExtent

__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, cudaMemset2DAsync, cudaMalloc3D, make_cudaPitchedPtr, make_cudaExtent

__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

__host__cudaError_t cudaMipmappedArrayGetMemoryRequirements (cudaArrayMemoryRequirements *memoryRequirements, cudaMipmappedArray_t mipmap, int device)

Returns the memory requirements of a CUDA mipmapped array.

Parameters

memoryRequirements
- Pointer to cudaArrayMemoryRequirements

mipmap
- CUDA mipmapped array to get the memory requirements of

device
- Device to get the memory requirements for

Returns

cudaSuccess cudaErrorInvalidValue

Description

Returns the memory requirements of a CUDA mipmapped array in `memoryRequirements` If the CUDA mipmapped array is not allocated with flag cudaArrayDeferredMapping cudaErrorInvalidValue will be returned.

The returned value in cudaArrayMemoryRequirements::size represents the total size of the CUDA mipmapped array. The returned value in cudaArrayMemoryRequirements::alignment represents the alignment necessary for mapping the CUDA mipmapped array.

See also:

cudaArrayGetMemoryRequirements

__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:

cudaArrayGetSparseProperties, 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

6.12. 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.

__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 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, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, cuMemcpyAtoA

__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 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, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, 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 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.

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, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, 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

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

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, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, cuMemcpyHtoA, cuMemcpyDtoA

__host__cudaError_t cudaMemcpyToArrayAsync (cudaArray_t dst, size_t wOffset, size_t hOffset, const void *src, size_t count, 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

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 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.

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 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, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, cuMemcpyHtoAAsync, cuMemcpy2DAsync

6.13. Stream Ordered Memory Allocator

overview

The asynchronous allocator allows the user to allocate and free in stream order. All asynchronous accesses of the allocation must happen between the stream executions of the 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.

__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

__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:

cuMemAllocAsync, cudaMallocAsync (C++ API), cudaMallocFromPoolAsync, cudaFreeAsync, cudaDeviceSetMemPool, cudaDeviceGetDefaultMemPool, cudaDeviceGetMemPool, cudaMemPoolSetAccess, cudaMemPoolSetAttribute, cudaMemPoolGetAttribute

__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

memPool
- The pool to allocate from

stream
- 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), cudaMallocAsync, 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.

To create a memory pool targeting a specific host NUMA node, applications must set cudaMemPoolProps::cudaMemLocation::type to cudaMemLocationTypeHostNuma and cudaMemPoolProps::cudaMemLocation::id must specify the NUMA ID of the host memory node. Specifying cudaMemLocationTypeHostNumaCurrent or cudaMemLocationTypeHost as the cudaMemPoolProps::cudaMemLocation::type will result in cudaErrorInvalidValue. By default, the pool's memory will be accessible from the device it is allocated on. In the case of pools created with cudaMemLocationTypeHostNuma, their default accessibility will be from the host CPU. Applications can control the maximum size of the pool by specifying a non-zero value for cudaMemPoolProps::maxSize. If set to 0, the maximum size of the pool will default to a system dependent value.

Applications that intend to use CU_MEM_HANDLE_TYPE_FABRIC based memory sharing must ensure: (1) `nvidia-caps-imex-channels` character device is created by the driver and is listed under /proc/devices (2) have at least one IMEX channel file accessible by the user launching the application.

When exporter and importer CUDA processes have been granted access to the same IMEX channel, they can securely share memory.

The IMEX channel security model works on a per user basis. Which means all processes under a user can share memory if the user has access to a valid IMEX channel. When multi-user isolation is desired, a separate IMEX channel is required for each user.

These channel files exist in /dev/nvidia-caps-imex-channels/channel* and can be created using standard OS native calls like mknod on Linux. For example: To create channel0 with the major number from /

proc/devices users can execute the following command: `mknod /dev/nvidia-caps-imex-channels/ channel0 c <major number>=""> 0`

> 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 *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

memPool

attr

- The attribute to get

value

- 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).
- cudaMemPoolAttrReservedMemCurrent: (value type = cuuint64_t) Amount of backing memory currently allocated for the mempool.
- cudaMemPoolAttrReservedMemHigh: (value type = cuuint64_t) High watermark of backing memory allocated for the mempool since the last time it was reset.
- cudaMemPoolAttrUsedMemCurrent: (value type = cuuint64_t) Amount of memory from the pool that is currently in use by the application.
- cudaMemPoolAttrUsedMemHigh: (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.

> 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:

cuMemPoolGetAttribute, cudaMallocAsync, cudaFreeAsync, cudaDeviceGetDefaultMemPool, cudaDeviceGetMemPool, cudaMemPoolCreate

__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 (cudaMemPool_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).

- cudaMemPoolAttrReservedMemHigh: (value type = cuuint64_t) Reset the high watermark that tracks the amount of backing memory that was allocated for the memory pool. It is illegal to set this attribute to a non-zero value.
- cudaMemPoolAttrUsedMemHigh: (value type = cuuint64_t) Reset the high watermark that tracks the amount of used memory that was allocated for the memory pool. It is illegal to set this attribute to a non-zero value.

> 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.

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:

cuMemPoolTrimTo, cudaMallocAsync, cudaFreeAsync, cudaDeviceGetDefaultMemPool, cudaDeviceGetMemPool, cudaMemPoolCreate

6.14. 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 cudaMallocHost() and cudaHostAlloc() 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.

__host__cudaError_t cudaPointerGetAttributes (cudaPointerAttributes *attributes, const void *ptr)

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:

```
    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, cudaInitDevice, cuPointerGetAttributes

6.15. Peer Device Memory Access

This section describes the peer device memory access functions of the CUDA runtime application programming interface.

__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

__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.
- 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, cudaDeviceDisablePeerAccess, cuCtxEnablePeerAccess

6.16. OpenGL Interoperability

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.

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

__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, cudaErrorOperatingSystem, 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:

cudaGraphicsUnregisterResource, cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray, cudaGraphicsResourceGetMappedPointer, cuGLGetDevices

__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, cudaErrorOperatingSystem, 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:

cudaGraphicsUnregisterResource, cudaGraphicsMapResources, cudaGraphicsResourceGetMappedPointer, cuGraphicsGLRegisterBuffer

__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, cudaErrorOperatingSystem, 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:

cudaGraphicsUnregisterResource, cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray, cuGraphicsGLRegisterImage

__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

6.17. 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.

See also:

cudaGraphicsMapResources

__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.

This function will immediately initialize the primary context on `device` if needed.

> 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

6.18. 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
cudaD3D9DeviceListCurrentFrame = 2
The CUDA devices for the GPUs used by a D3D9 device in its currently rendering frame
cudaD3D9DeviceListNextFrame = 3
The CUDA devices for the GPUs to be used by a D3D9 device in the next frame

__host__cudaError_t cudaD3D9GetDevice (int *device, const char *pszAdapterName)

Gets the device number for an adapter.

Parameters

device
- Returns the device corresponding to pszAdapterName
pszAdapterName
- D3D9 adapter to get device for

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown

Description

Returns in `*device` the CUDA-compatible device corresponding to the adapter name `pszAdapterName` obtained from EnumDisplayDevices or IDirect3D9::GetAdapterIdentifier(). If no device on the adapter with name `pszAdapterName` is CUDA-compatible then the call will fail.

Note:

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

See also:

cudaD3D9SetDirect3DDevice, cudaGraphicsD3D9RegisterResource, 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.

This function will immediately initialize the primary context on `device` if needed.

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 cirumstances, 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 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:

cudaD3D9SetDirect3DDevice, cudaGraphicsUnregisterResource, cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray, cudaGraphicsResourceGetMappedPointer, cuGraphicsD3D9RegisterResource

6.19. 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 througa 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, is 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

__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 `*pArray` 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 * pitch + (bytes per pixel) * 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* slicePitch + y * pitch + (bytes per pixel) * 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 `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 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 cudaD3D9ResourceGetMappedPointer.

> 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.

Note:

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

See also:

cudaInteropResourceSetMapFlags

__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

6.20. 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
cudaD3D10DeviceListCurrentFrame = 2
The CUDA devices for the GPUs used by a D3D10 device in its currently rendering frame
cudaD3D10DeviceListNextFrame = 3
The CUDA devices for the GPUs to be used by a D3D10 device in the next frame

__host__cudaError_t cudaD3D10GetDevice (int *device, IDXGIAdapter *pAdapter)

Gets the device number for an adapter.

Parameters

device
- Returns the device corresponding to pAdapter
pAdapter
- D3D10 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:

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:

cudaGraphicsUnregisterResource, cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray, cudaGraphicsResourceGetMappedPointer, cuD3D10GetDevices

__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

- 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, cuGraphicsD3D10RegisterResource

6.21. 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

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 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

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 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.

Note:

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

See also:

cudaGraphicsD3D10RegisterResource

__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

__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 * pitch + (bytes per pixel) * 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* slicePitch + y * pitch + (bytes per pixel) * x

Both parameters `pPitch` and `pPitchSlice` are optional and may be set to NULL.

If `pResource` is not of type ID3D10Texture1D, ID3D10Texture2D, or ID3D10Texture3D, or if `pResource` has not been registered for use with CUDA, then cudaErrorInvalidResourceHandle 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:

cudaGraphicsSubResourceGetMappedArray

__host__cudaError_t cudaD3D10ResourceGetMappedPointer (void **pPointer, ID3D10Resource *pResource, unsigned int subResource)

Gets 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

subResource
- Subresource of pResource 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 `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

__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

__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.

This function will immediately initialize the primary context on `device` if needed.

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

6.22. 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.

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:

cudaGraphicsUnregisterResource, cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray, cudaGraphicsResourceGetMappedPointer, cuD3D11GetDevice

__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:

cudaGraphicsUnregisterResource, cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray, cudaGraphicsResourceGetMappedPointer, cuD3D11GetDevices

__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

6.23. Direct3D 11 Interoperability [DEPRECATED]

This section describes deprecated Direct3D 11 interoperability functions.

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

See also:

cudaD3D11SetDirect3DDevice

__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.

This function will immediately initialize the primary context on `device` if needed.

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

See also:

cudaD3D11GetDevice, cudaGraphicsD3D11RegisterResource, cudaDeviceReset

6.24. 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.

This function will immediately initialize the primary context on `device` if needed.

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

6.25. EGL Interoperability

This section describes the EGL interoperability functions of the CUDA runtime application programming interface.

__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

__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.

See also:

cudaEGLStreamConsumerDisconnect, cudaEGLStreamConsumerAcquireFrame, cudaEGLStreamConsumerReleaseFrame, cuEGLStreamConsumerConnect

__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

- Conection 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

conn

- Pointer to the returned connection handle

eglStream

- EGLStreamKHR handle

width

- width of the image to be submitted to the stream

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

- Conection 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 Proucer 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:

```
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

__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 flages 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 accesing 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:

cudaGraphicsUnregisterResource, cudaGraphicsResourceGetMappedEglFrame, cuGraphicsEGLRegisterImage

__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 `resource` may be accessed. This API can only be called for EGL graphics resources.

The cudaEglFrame is defined as

```
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;
```

Note:

Note that in case of multiplanar `*eglFrame`, pitch of only first plane (unsigned int cudaEglPlaneDesc::pitch) is to be considered by the application.

See also:

cudaGraphicsSubResourceGetMappedArray, cudaGraphicsResourceGetMappedPointer, cuGraphicsResourceGetMappedEglFrame

6.26. 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:

cudaGraphicsResourceGetMappedPointer, cudaGraphicsSubResourceGetMappedArray, cudaGraphicsUnmapResources, cuGraphicsMapResources

__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 `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:

cudaGraphicsResourceGetMappedPointer, cuGraphicsResourceGetMappedMipmappedArray

__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:

cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray, cuGraphicsResourceGetMappedPointer

__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:

cudaGraphicsResourceGetMappedPointer, cuGraphicsSubResourceGetMappedArray

__host__cudaError_t cudaGraphicsUnmapResources (int count, cudaGraphicsResource_t *resources, cudaStream_t stream)

Unmap graphics resources.

Parameters

count

- Number of resources to unmap

resources

- Resources to unmap

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.
- 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, cuGraphicsUnmapResources

__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 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:

cudaGraphicsD3D9RegisterResource, cudaGraphicsD3D10RegisterResource, cudaGraphicsD3D11RegisterResource, cudaGraphicsGLRegisterBuffer, cudaGraphicsGLRegisterImage, cuGraphicsUnregisterResource

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:

```
  struct cudaChannelFormatDesc {
          int x, y, z, w;
          enum cudaChannelFormatKind
                  f;
        };
```

where cudaChannelFormatKind is one of cudaChannelFormatKindSigned, cudaChannelFormatKindUnsigned, or cudaChannelFormatKindFloat.

See also:

cudaCreateChannelDesc (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 not in a block compressed format.

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 cudaResourceDesc structure is defined as:

```
        struct cudaResourceDesc {
                  enum cudaResourceType
                  resType;

                  union {
                      struct {
                          cudaArray_t
                  array;
                      } array;
                      struct {
                          cudaMipmappedArray_t
                  mipmap;
                      } mipmap;
                      struct {
```

```
                        void *devPtr;
                        struct cudaChannelFormatDesc
                desc;
                        size_t sizeInBytes;
                    } linear;
                    struct {
                        void *devPtr;
                        struct cudaChannelFormatDesc
                desc;
                        size_t width;
                        size_t height;
                        size_t pitchInBytes;
                    } pitch2D;
                } res;
            };
```

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

```
        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;
                int                         seamlessCubemap;
            };
```

where

- cudaTextureDesc::addressMode specifies the addressing mode for each dimension of the texture data. cudaTextureAddressMode is defined as:

  ```
          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:

  ```
          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:

  ```
          enum cudaTextureReadMode {
                  cudaReadModeElementType     = 0,
                  cudaReadModeNormalizedFloat = 1
              };
  ```

  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.
- cudaTextureDesc::seamlessCubemap specifies whether seamless cube map filtering is enabled. This flag can only be specified if the underlying resource is a CUDA array or a CUDA mipmapped array that was created with the flag cudaArrayCubemap. When seamless cube map filtering is enabled, texture address modes specified by cudaTextureDesc::addressMode are ignored. Instead, if the cudaTextureDesc::filterMode is set to cudaFilterModePoint the address mode cudaAddressModeClamp will be applied for all dimensions. If the cudaTextureDesc::filterMode is set to cudaFilterModeLinear seamless cube map filtering will be performed when sampling along the cube face borders.

The cudaResourceViewDesc struct is defined as

```
        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.

As of CUDA 12.0, this function no longer initializes CUDA. The purpose of this API is solely to return a compile-time constant stating the CUDA Toolkit version in the above format.

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.

__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:

cudaDeviceSetGraphMemAttribute, cudaGraphAddMemAllocNode, cudaGraphAddMemFreeNode, cudaDeviceGraphMemTrim, 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:

cudaDeviceGetGraphMemAttribute, cudaGraphAddMemAllocNode, cudaGraphAddMemFreeNode, cudaDeviceGraphMemTrim, cudaMallocAsync, cudaFreeAsync

__device__cudaGraphExec_t cudaGetCurrentGraphExec (void)

Get the currently running device graph id.

Returns

Returns the current device graph id, 0 if the call is outside of a device graph.

Description

Get the currently running device graph id.

See also:

cudaGraphLaunch

__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 `childGraph` contains allocation nodes, free nodes, or conditional 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:

cudaGraphAddNode, 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

__host__cudaError_t cudaGraphAddDependencies_v2 (cudaGraph_t graph, const cudaGraphNode_t *from, const cudaGraphNode_t *to, const cudaGraphEdgeData *edgeData, size_t numDependencies)

Adds dependency edges to a graph. (12.3+).

Parameters

graph

- Graph to which dependencies are added

from

- Array of nodes that provide the dependencies

to

- Array of dependent nodes

edgeData

- Optional array of edge data. If NULL, default (zeroed) edge data is assumed.

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

__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:

cudaGraphAddNode, 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

- Number of dependencies

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 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:

cudaGraphAddNode, 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
pDependencies
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 `phGraphNode`.

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:

cudaGraphAddNode, 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:

cudaGraphAddNode, 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:

cudaGraphAddNode, cudaGraphExternalSemaphoresWaitNodeGetParams, cudaGraphExternalSemaphoresWaitNodeSetParams, cudaGraphExecExternalSemaphoresWaitNodeSetParams, cudaGraphAddExternalSemaphoresSignalNode, cudaImportExternalSemaphore, cudaSignalExternalSemaphoresAsync, cudaWaitExternalSemaphoresAsync, cudaGraphCreate, cudaGraphDestroyNode, cudaGraphAddEventRecordNode, cudaGraphAddEventWaitNode, cudaGraphAddChildGraphNode, cudaGraphAddEmptyNode, cudaGraphAddKernelNode, cudaGraphAddMemcpyNode, cudaGraphAddMemsetNode

__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:

cudaGraphAddNode, 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:

```
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` x `gridDim.y` x `gridDim.z`) grid of blocks. Each block contains (`blockDim.x` x `blockDim.y` x `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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

See also:

cudaGraphAddNode, 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 `nodeParams.dptr`. The allocation's address remains fixed across instantiations and launches.

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 can only be used in a child node if the ownership is moved to the parent.
- 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:

cudaGraphAddNode, cudaGraphAddMemFreeNode, cudaGraphMemAllocNodeGetParams, cudaDeviceGraphMemTrim, cudaDeviceGetGraphMemAttribute, cudaDeviceSetGraphMemAttribute, cudaMallocAsync, cudaFreeAsync, cudaGraphCreate, cudaGraphDestroyNode, cudaGraphAddChildGraphNode, cudaGraphAddEmptyNode, cudaGraphAddEventRecordNode, cudaGraphAddEventWaitNode, cudaGraphAddExternalSemaphoresSignalNode, cudaGraphAddExternalSemaphoresWaitNode, cudaGraphAddKernelNode, cudaGraphAddMemcpyNode, cudaGraphAddMemsetNode

__host__cudaError_t 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:

cudaGraphAddNode, cudaMemcpy3D, cudaGraphAddMemcpyNodeToSymbol, cudaGraphAddMemcpyNodeFromSymbol, cudaGraphAddMemcpyNode1D, cudaGraphMemcpyNodeGetParams, cudaGraphMemcpyNodeSetParams, cudaGraphCreate, cudaGraphDestroyNode, cudaGraphAddChildGraphNode, cudaGraphAddEmptyNode, cudaGraphAddKernelNode, cudaGraphAddHostNode, cudaGraphAddMemsetNode

__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, cudaGraphAddMemcpyNode, 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 `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 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:

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 can only be used in a child node if the ownership is moved to the parent.
- 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:

cudaGraphAddNode, 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:

cudaGraphAddNode, cudaMemset2D, cudaGraphMemsetNodeGetParams, cudaGraphMemsetNodeSetParams, cudaGraphCreate, cudaGraphDestroyNode, cudaGraphAddChildGraphNode, cudaGraphAddEmptyNode, cudaGraphAddKernelNode, cudaGraphAddHostNode, cudaGraphAddMemcpyNode

__host__cudaError_t cudaGraphAddNode (cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, cudaGraphNodeParams *nodeParams)

Adds a node of arbitrary type 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
: - Specification of the node

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDeviceFunction, cudaErrorNotSupported

Description

Creates a new node in `graph` described by `nodeParams` with `numDependencies` dependencies specified via `pDependencies`. `numDependencies` may be 0. `pDependencies` may be null if `numDependencies` is 0. `pDependencies` may not have any duplicate entries.

`nodeParams` is a tagged union. The node type should be specified in the `type` field, and type-specific parameters in the corresponding union member. All unused bytes - that is, `reserved0` and all bytes past the utilized union member - must be set to zero. It is recommended to use brace initialization or memset to ensure all bytes are initialized.

Note that for some node types, `nodeParams` may contain "out parameters" which are modified during the call, such as `nodeParams->alloc.dptr`.

A handle to the new node will be returned in `phGraphNode`.

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, cudaGraphNodeSetParams, cudaGraphExecNodeSetParams

__host__cudaError_t cudaGraphAddNode_v2 (cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, const cudaGraphEdgeData *dependencyData, size_t numDependencies, cudaGraphNodeParams *nodeParams)

Adds a node of arbitrary type to a graph (12.3+).

Parameters

pGraphNode
- Returns newly created node

graph
- Graph to which to add the node

pDependencies
- Dependencies of the node

dependencyData
- Optional edge data for the dependencies. If NULL, the data is assumed to be default (zeroed) for all dependencies.

numDependencies
- Number of dependencies

nodeParams
- Specification of the node

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDeviceFunction, cudaErrorNotSupported

Description

Creates a new node in `graph` described by `nodeParams` with `numDependencies` dependencies specified via `pDependencies`. `numDependencies` may be 0. `pDependencies` may be null if `numDependencies` is 0. `pDependencies` may not have any duplicate entries.

`nodeParams` is a tagged union. The node type should be specified in the `type` field, and type-specific parameters in the corresponding union member. All unused bytes - that is, `reserved0` and all bytes past the utilized union member - must be set to zero. It is recommended to use brace initialization or memset to ensure all bytes are initialized.

Note that for some node types, `nodeParams` may contain "out parameters" which are modified during the call, such as `nodeParams->alloc.dptr`.

A handle to the new node will be returned in `phGraphNode`.

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, cudaGraphNodeSetParams, cudaGraphExecNodeSetParams

__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:

: Cloning is not supported for graphs which contain memory allocation nodes, memory free nodes, or conditional 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.

See also:

cudaGraphCreate, cudaGraphNodeFindInClone

__host__cudaError_t cudaGraphConditionalHandleCreate (cudaGraphConditionalHandle *pHandle_out, cudaGraph_t graph, unsigned int defaultLaunchValue, unsigned int flags)

Create a conditional handle.

Parameters

pHandle_out

- Pointer used to return the handle to the caller.

graph

defaultLaunchValue

- Optional initial value for the conditional variable. Applied at the beginning of each graph execution if cudaGraphCondAssignDefault is set in `flags`.

flags

- Currently must be cudaGraphCondAssignDefault or 0.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_SUPPORTED

Description

Creates a conditional handle associated with `hGraph`.

The conditional handle must be associated with a conditional node in this graph or one of its children.

Handles not associated with a conditional node may cause graph instantiation to fail.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphAddNode,

__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 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, cudaGraphAddEmptyNode, cudaGraphAddKernelNode, cudaGraphAddHostNode, cudaGraphAddMemcpyNode, cudaGraphAddMemsetNode, cudaGraphInstantiate, cudaGraphDestroy, cudaGraphGetNodes, cudaGraphGetRootNodes, cudaGraphGetEdges, cudaGraphClone

__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.

__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 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:

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

event_out

- Pointer to return the event

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:

cudaGraphNodeSetParams, 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, cudaGraphEventRecordNodeGetEvent, 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:

cudaGraphNodeSetParams, 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:

cudaGraphExecNodeSetParams, cudaGraphAddChildGraphNode, cudaGraphChildGraphNodeGetGraph, cudaGraphExecKernelNodeSetParams, cudaGraphExecMemcpyNodeSetParams, cudaGraphExecMemsetNodeSetParams, cudaGraphExecHostNodeSetParams, cudaGraphExecEventRecordNodeSetEvent, cudaGraphExecEventWaitNodeSetEvent, cudaGraphExecExternalSemaphoresSignalNodeSetParams, cudaGraphExecExternalSemaphoresWaitNodeSetParams, cudaGraphExecUpdate, cudaGraphInstantiate

__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.

See also:

cudaGraphInstantiate, cudaGraphUpload, cudaGraphLaunch

__host__cudaError_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

cudaSuccess, cudaErrorInvalidValue,

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 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:

cudaGraphExecNodeSetParams, cudaGraphAddEventRecordNode, cudaGraphEventRecordNodeGetEvent, cudaGraphEventWaitNodeSetEvent, cudaEventRecordWithFlags, cudaStreamWaitEvent, cudaGraphExecKernelNodeSetParams, cudaGraphExecMemcpyNodeSetParams, cudaGraphExecMemsetNodeSetParams, cudaGraphExecHostNodeSetParams, cudaGraphExecChildGraphNodeSetParams, cudaGraphExecEventWaitNodeSetEvent, cudaGraphExecExternalSemaphoresSignalNodeSetParams, cudaGraphExecExternalSemaphoresWaitNodeSetParams, cudaGraphExecUpdate, cudaGraphInstantiate

__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:

cudaGraphExecNodeSetParams, cudaGraphAddEventWaitNode, cudaGraphEventWaitNodeGetEvent, cudaGraphEventRecordNodeSetEvent, cudaEventRecordWithFlags, cudaStreamWaitEvent, cudaGraphExecKernelNodeSetParams, cudaGraphExecMemcpyNodeSetParams, cudaGraphExecMemsetNodeSetParams, cudaGraphExecHostNodeSetParams, cudaGraphExecChildGraphNodeSetParams, cudaGraphExecEventRecordNodeSetEvent, cudaGraphExecExternalSemaphoresSignalNodeSetParams, cudaGraphExecExternalSemaphoresWaitNodeSetParams, cudaGraphExecUpdate, cudaGraphInstantiate

__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,

Description

Sets the parameters of an external semaphore signal 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:

cudaGraphExecNodeSetParams, cudaGraphAddExternalSemaphoresSignalNode, cudaImportExternalSemaphore, cudaSignalExternalSemaphoresAsync, cudaWaitExternalSemaphoresAsync, cudaGraphExecKernelNodeSetParams, cudaGraphExecMemcpyNodeSetParams, cudaGraphExecMemsetNodeSetParams, cudaGraphExecHostNodeSetParams, cudaGraphExecChildGraphNodeSetParams, cudaGraphExecEventRecordNodeSetEvent, cudaGraphExecEventWaitNodeSetEvent, cudaGraphExecExternalSemaphoresWaitNodeSetParams, cudaGraphExecUpdate, 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:

cudaGraphExecNodeSetParams, cudaGraphAddExternalSemaphoresWaitNode, cudaImportExternalSemaphore, cudaSignalExternalSemaphoresAsync, cudaWaitExternalSemaphoresAsync, cudaGraphExecKernelNodeSetParams, cudaGraphExecMemcpyNodeSetParams, cudaGraphExecMemsetNodeSetParams, cudaGraphExecHostNodeSetParams, cudaGraphExecChildGraphNodeSetParams, cudaGraphExecEventRecordNodeSetEvent, cudaGraphExecEventWaitNodeSetEvent, cudaGraphExecExternalSemaphoresSignalNodeSetParams, cudaGraphExecUpdate, cudaGraphInstantiate

__host__cudaError_t cudaGraphExecGetFlags (cudaGraphExec_t graphExec, unsigned long long *flags)

Query the instantiation flags of an executable graph.

Parameters

graphExec
- The executable graph to query

flags
- Returns the instantiation flags

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Returns the flags that were passed to instantiation for the given executable graph. cudaGraphInstantiateFlagUpload will not be returned by this API as it does not affect the resulting executable graph.

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, cudaGraphInstantiateWithFlags, cudaGraphInstantiateWithParams

__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 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:

cudaGraphExecNodeSetParams, 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 `node` in the non-executable graph, from which the executable graph was instantiated.

`node` must not have been removed from the original graph. All `nodeParams` fields may change, but the following restrictions apply to `func` updates:

- The owning device 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
- A node whose function originally did not make device-side update calls cannot be updated to a function which makes device-side update calls.
- If `hGraphExec` was not instantiated for device launch, a node whose function originally did not use device-side cudaGraphLaunch() cannot be updated to a function which uses device-side cudaGraphLaunch() unless the node resides on the same device as nodes which contained such calls at instantiate-time. If no such calls were present at instantiation, these updates cannot be performed at all.

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.

If `node` is a device-updatable kernel node, the next upload/launch of `hGraphExec` will overwrite any previous device-side updates. Additionally, applying host updates to a device-updatable kernel node while it is being updated from the device will result in 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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

See also:

cudaGraphExecNodeSetParams, 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:

cudaGraphExecNodeSetParams, cudaGraphAddMemcpyNode, cudaGraphMemcpyNodeSetParams, cudaGraphExecMemcpyNodeSetParamsToSymbol, cudaGraphExecMemcpyNodeSetParamsFromSymbol, cudaGraphExecMemcpyNodeSetParams1D, cudaGraphExecKernelNodeSetParams, cudaGraphExecMemsetNodeSetParams, cudaGraphExecHostNodeSetParams, cudaGraphExecChildGraphNodeSetParams, cudaGraphExecEventRecordNodeSetEvent, cudaGraphExecEventWaitNodeSetEvent, cudaGraphExecExternalSemaphoresSignalNodeSetParams, cudaGraphExecExternalSemaphoresWaitNodeSetParams, cudaGraphExecUpdate, cudaGraphInstantiate

__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, cudaGraphMemcpyNodeSetParams, cudaGraphMemcpyNodeSetParams1D, 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

__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 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, 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.

Zero sized operations are not supported.

The new destination pointer in `pNodeParams` must be to the same kind of allocation as the original destination pointer and have the same context association and device mapping as the original destination pointer.

Both the value and pointer address may be updated. Changing other aspects of the memset (width, height, element size or pitch) may cause the update to be rejected. Specifically, for 2d memsets, all dimension changes are rejected. For 1d memsets, changes in height are explicitly rejected and other changes are opportunistically allowed if the resulting work maps onto the work resources already allocated for the node.

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:

cudaGraphExecNodeSetParams, cudaGraphAddMemsetNode, cudaGraphMemsetNodeSetParams, cudaGraphExecKernelNodeSetParams, cudaGraphExecMemcpyNodeSetParams, cudaGraphExecHostNodeSetParams, cudaGraphExecChildGraphNodeSetParams, cudaGraphExecEventRecordNodeSetEvent, cudaGraphExecEventWaitNodeSetEvent, cudaGraphExecExternalSemaphoresSignalNodeSetParams, cudaGraphExecExternalSemaphoresWaitNodeSetParams, cudaGraphExecUpdate, cudaGraphInstantiate

__host__cudaError_t cudaGraphExecNodeSetParams (cudaGraphExec_t graphExec, cudaGraphNode_t node, cudaGraphNodeParams *nodeParams)

Update's a graph node's parameters in an instantiated graph.

Parameters

graphExec
- The executable graph in which to update the specified node

node
- Corresponding node from the graph from which graphExec was instantiated

nodeParams
- Updated Parameters to set

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDeviceFunction, cudaErrorNotSupported

Description

Sets the parameters of a node in an executable graph `graphExec`. The node is identified by the corresponding node `node` in the non-executable graph from which the executable graph was instantiated. `node` must not have been removed from the original graph.

The modifications only affect future launches of `graphExec`. Already enqueued or running launches of `graphExec` are not affected by this call. `node` is also not modified by this call.

Allowed changes to parameters on executable graphs are as follows:

Node type	Allowed changes
kernel	See cudaGraphExecKernelNodeSetParams
memcpy	Addresses for 1-dimensional copies if allocated in same context; see cudaGraphExecMemcpyNodeSetParams
memset	Addresses for 1-dimensional memsets if allocated in same context; see cudaGraphExecMemsetNodeSetParams
host	Unrestricted
child graph	Topology must match and restrictions apply recursively; see cudaGraphExecUpdate
event wait	Unrestricted

Node type	Allowed changes
event record	Unrestricted
external semaphore signal	Number of semaphore operations cannot change
external semaphore wait	Number of semaphore operations cannot change
memory allocation	API unsupported
memory free	API unsupported

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:

cudaGraphAddNode, cudaGraphNodeSetParams cudaGraphExecUpdate, cudaGraphInstantiate

__host__cudaError_t cudaGraphExecUpdate (cudaGraphExec_t hGraphExec, cudaGraph_t hGraph, cudaGraphExecUpdateResultInfo *resultInfo)

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

resultInfo
: the error info structure

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.
  - A node whose function originally did not make device-side update calls cannot be updated to a function which makes device-side update calls.
  - A cooperative node cannot be updated to a non-cooperative node, and vice-versa.
  - If the graph was instantiated with cudaGraphInstantiateFlagUseNodePriority, the priority attribute cannot change. Equality is checked on the originally requested priority values, before they are clamped to the device's supported range.
  - If `hGraphExec` was not instantiated for device launch, a node whose function originally did not use device-side cudaGraphLaunch() cannot be updated to a function which uses device-side cudaGraphLaunch() unless the node resides on the same device as nodes which contained such calls at instantiate-time. If no such calls were present at instantiation, these updates cannot be performed at all.
  - Neither `hGraph` nor `hGraphExec` may contain device-updatable kernel nodes.
- 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.
  - For 2d memsets, only address and assigned value may be updated.
  - For 1d memsets, updating dimensions is also allowed, but may fail if the resulting operation doesn't map onto the work resources already allocated for the node.
- Additional memcpy node restrictions:
  - Changing either the source or destination memory type(i.e. CU_MEMORYTYPE_DEVICE, CU_MEMORYTYPE_ARRAY, etc.) is not supported.
- Conditional nodes:
  - Changing node parameters is not supported.
  - Changing parameters of nodes within the conditional body graph is subject to the rules above.
  - Conditional handle flags and default values are updated as part of the graph update.

Note: The API may add further restrictions in future releases. The return code should always be checked.

cudaGraphExecUpdate sets the result member of `resultInfo` to cudaGraphExecUpdateErrorTopologyChanged under the following conditions:

- The count of nodes directly in `hGraphExec` and `hGraph` differ, in which case resultInfo->errorNode is set to NULL.
- `hGraph` has more exit nodes than `hGraph`, in which case resultInfo->errorNode is set to one of the exit nodes in hGraph.
- A node in `hGraph` has a different number of dependencies than the node from `hGraphExec` it is paired with, in which case resultInfo->errorNode is set to the node from `hGraph`.
- A node in `hGraph` has a dependency that does not match with the corresponding dependency of the paired node from `hGraphExec`. resultInfo->errorNode will be set to the node from `hGraph`. resultInfo->errorFromNode will be set to the mismatched dependency. The dependencies are paired based on edge order and a dependency does not match when the nodes are already paired based on other edges examined in the graph.

cudaGraphExecUpdate sets `the` result member of `resultInfo` 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`
- cudaGraphExecUpdateErrorAttributesChanged if any attributes of 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 the update fails for a reason not listed above, the result member of `resultInfo` will be set to cudaGraphExecUpdateError. If the update succeeds, the result member will be set to cudaGraphExecUpdateSuccess.

cudaGraphExecUpdate returns cudaSuccess when the updated 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:

cudaLaunchKernel, cudaGraphAddExternalSemaphoresSignalNode, cudaGraphExternalSemaphoresSignalNodeSetParams, cudaGraphAddExternalSemaphoresWaitNode, cudaSignalExternalSemaphoresAsync, cudaWaitExternalSemaphoresAsync

__host__cudaError_t cudaGraphExternalSemaphoresSignalNodeSetParams (cudaGraphNode_t hNode, const cudaExternalSemaphoreSignalNodeParams *nodeParams)

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:

cudaGraphNodeSetParams, 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

__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:

cudaGraphNodeSetParams, cudaGraphAddExternalSemaphoresWaitNode, cudaGraphExternalSemaphoresWaitNodeSetParams, cudaGraphAddExternalSemaphoresWaitNode, cudaSignalExternalSemaphoresAsync, cudaWaitExternalSemaphoresAsync

__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

__host__cudaError_t cudaGraphGetEdges_v2 (cudaGraph_t graph, cudaGraphNode_t *from, cudaGraphNode_t *to, cudaGraphEdgeData *edgeData, size_t *numEdges)

Returns a graph's dependency edges (12.3+).

Parameters

graph
- Graph to get the edges from

from
- Location to return edge endpoints

to
- Location to return edge endpoints

edgeData
- Optional location to return edge data

numEdges
- See description

Returns

cudaSuccess, cudaErrorLossyQuery, cudaErrorInvalidValue

Description

Returns a list of `graph's` dependency edges. Edges are returned via corresponding indices in `from`, `to` and `edgeData`; that is, the node in `to`[i] has a dependency on the node in `from`[i] with data `edgeData`[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`. `edgeData` may alone be NULL, in which case the edges must all have default (zeroed) edge data. Attempting a losst query via NULL `edgeData` will result in cudaErrorLossyQuery. If `edgeData` is non-NULL then `from` and `to` must be as well.

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

__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:

cudaGraphNodeSetParams, cudaLaunchHostFunc, cudaGraphAddHostNode, cudaGraphHostNodeGetParams

__host__cudaError_t cudaGraphInstantiate (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.
- cudaGraphInstantiateFlagDeviceLaunch, which configures the graph for launch from the device. If this flag is passed, the executable graph handle returned can be used to launch the graph from both the host and device. This flag cannot be used in conjunction with cudaGraphInstantiateFlagAutoFreeOnLaunch.
- cudaGraphInstantiateFlagUseNodePriority, which causes the graph to use the priorities from the per-node attributes rather than the priority of the launch stream during execution. Note that priorities are only available on kernel nodes, and are copied from stream priority during stream capture.

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. The same also applies if `graph` contains any device-updatable kernel nodes.

Graphs instantiated for launch on the device have additional restrictions which do not apply to host graphs:

- The graph's nodes must reside on a single device.
- The graph can only contain kernel nodes, memcpy nodes, memset nodes, and child graph nodes.
- The graph cannot be empty and must contain at least one kernel, memcpy, or memset node. Operation-specific restrictions are outlined below.
- Kernel nodes:
  - Use of CUDA Dynamic Parallelism is not permitted.
  - Cooperative launches are permitted as long as MPS is not in use.
- Memcpy nodes:
  - Only copies involving device memory and/or pinned device-mapped host memory are permitted.
  - Copies involving CUDA arrays are not permitted.
  - Both operands must be accessible from the current device, and the current device must match the device of other nodes in the graph.

If `graph` is not instantiated for launch on the device but contains kernels which call device-side cudaGraphLaunch() from multiple devices, this will result in 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:

cudaGraphInstantiateWithFlags, cudaGraphCreate, cudaGraphUpload, cudaGraphLaunch, cudaGraphExecDestroy

__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.
- cudaGraphInstantiateFlagDeviceLaunch, which configures the graph for launch from the device. If this flag is passed, the executable graph handle returned can be used to launch the graph from both the host and device. This flag can only be used on platforms which support unified addressing. This flag cannot be used in conjunction with cudaGraphInstantiateFlagAutoFreeOnLaunch.
- cudaGraphInstantiateFlagUseNodePriority, which causes the graph to use the priorities from the per-node attributes rather than the priority of the launch stream during execution. Note that priorities are only available on kernel nodes, and are copied from stream priority during stream capture.

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. The same also applies if `graph` contains any device-updatable kernel nodes.

If `graph` contains kernels which call device-side cudaGraphLaunch() from multiple devices, this will result in an error.

Graphs instantiated for launch on the device have additional restrictions which do not apply to host graphs:

- The graph's nodes must reside on a single device.
- The graph can only contain kernel nodes, memcpy nodes, memset nodes, and child graph nodes.
- The graph cannot be empty and must contain at least one kernel, memcpy, or memset node. Operation-specific restrictions are outlined below.
- Kernel nodes:
  - Use of CUDA Dynamic Parallelism is not permitted.
  - Cooperative launches are permitted as long as MPS is not in use.
- Memcpy nodes:
  - Only copies involving device memory and/or pinned device-mapped host memory are permitted.
  - Copies involving CUDA arrays are not permitted.
  - Both operands must be accessible from the current device, and the current device must match the device of other nodes in the graph.

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

__host__cudaError_t cudaGraphInstantiateWithParams (cudaGraphExec_t *pGraphExec, cudaGraph_t graph, cudaGraphInstantiateParams *instantiateParams)

Creates an executable graph from a graph.

Parameters

pGraphExec
- Returns instantiated graph

graph
- Graph to instantiate

instantiateParams
- Instantiation parameters

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Instantiates `graph` as an executable graph according to the `instantiateParams` structure. 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`.

`instantiateParams` controls the behavior of instantiation and subsequent graph launches, as well as returning more detailed information in the event of an error. cudaGraphInstantiateParams is defined as:

```
    typedef struct {
                unsigned long long flags;
                cudaStream_t uploadStream;
                cudaGraphNode_t errNode_out;
                cudaGraphInstantiateResult result_out;
            } cudaGraphInstantiateParams;
```

The `flags` field 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.
- cudaGraphInstantiateFlagUpload, which will perform an upload of the graph into `uploadStream` once the graph has been instantiated.
- cudaGraphInstantiateFlagDeviceLaunch, which configures the graph for launch from the device. If this flag is passed, the executable graph handle returned can be used to launch the graph from both the host and device. This flag can only be used on platforms which support unified addressing. This flag cannot be used in conjunction with cudaGraphInstantiateFlagAutoFreeOnLaunch.

- cudaGraphInstantiateFlagUseNodePriority, which causes the graph to use the priorities from the per-node attributes rather than the priority of the launch stream during execution. Note that priorities are only available on kernel nodes, and are copied from stream priority during stream capture.

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. The same also applies if `graph` contains any device-updatable kernel nodes.

If `graph` contains kernels which call device-side cudaGraphLaunch() from multiple devices, this will result in an error.

Graphs instantiated for launch on the device have additional restrictions which do not apply to host graphs:

- The graph's nodes must reside on a single device.
- The graph can only contain kernel nodes, memcpy nodes, memset nodes, and child graph nodes.
- The graph cannot be empty and must contain at least one kernel, memcpy, or memset node. Operation-specific restrictions are outlined below.
- Kernel nodes:
  - Use of CUDA Dynamic Parallelism is not permitted.
  - Cooperative launches are permitted as long as MPS is not in use.
- Memcpy nodes:
  - Only copies involving device memory and/or pinned device-mapped host memory are permitted.
  - Copies involving CUDA arrays are not permitted.
  - Both operands must be accessible from the current device, and the current device must match the device of other nodes in the graph.

In the event of an error, the `result_out` and `errNode_out` fields will contain more information about the nature of the error. Possible error reporting includes:

- cudaGraphInstantiateError, if passed an invalid value or if an unexpected error occurred which is described by the return value of the function. `errNode_out` will be set to NULL.
- cudaGraphInstantiateInvalidStructure, if the graph structure is invalid. `errNode_out` will be set to one of the offending nodes.
- cudaGraphInstantiateNodeOperationNotSupported, if the graph is instantiated for device launch but contains a node of an unsupported node type, or a node which performs unsupported operations, such as use of CUDA dynamic parallelism within a kernel node. `errNode_out` will be set to this node.
- cudaGraphInstantiateMultipleDevicesNotSupported, if the graph is instantiated for device launch but a node's device differs from that of another node. This error can also be returned

if a graph is not instantiated for device launch and it contains kernels which call device-side cudaGraphLaunch() from multiple devices. `errNode_out` will be set to this node.

If instantiation is successful, `result_out` will be set to cudaGraphInstantiateSuccess, and `hErrNode_out` will be set to NULL.

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, cudaGraphInstantiate, cudaGraphInstantiateWithFlags, cudaGraphExecDestroy

__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.

Note:

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

See also:

cudaAccessPolicyWindow

__host__cudaError_t 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

__host__cudaError_t 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

__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

__device__cudaError_t cudaGraphKernelNodeSetEnabled (cudaGraphDeviceNode_t node, bool enable)

Enables or disables the given kernel node.

Parameters

node
- The node to update

enable
- Whether to enable or disable the node

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Enables or disables `node` based upon `enable`. If `enable` is true, the node will be enabled; if it is false, the node will be disabled. Disabled nodes will act as a NOP during execution. `node` must be device-updatable, and must reside upon the same device as the calling kernel.

If this function is called for the node's immediate dependent and that dependent is configured for programmatic dependent launch, then a memory fence must be invoked via __threadfence() before kickoff of the dependent is triggered via cudaTriggerProgrammaticLaunchCompletion() to ensure that the update is visible to that dependent node before it is launched.

Note:

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

See also:

cudaGraphKernelNodeSetParam, cudaGraphKernelNodeSetGridDim, cudaGraphKernelNodeUpdatesApply

__device__cudaError_t cudaGraphKernelNodeSetGridDim (cudaGraphDeviceNode_t node, dim3 gridDim)

Updates the grid dimensions of the given kernel node.

Parameters

node

- The node to update

gridDim

- The grid dimensions to set

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Sets the grid dimensions of `node` to `gridDim`. `node` must be device-updatable, and must reside upon the same device as thecalling kernel.

If this function is called for the node's immediate dependent and that dependent is configured for programmatic dependent launch, then a memory fence must be invoked via __threadfence() before kickoff of the dependent is triggered via cudaTriggerProgrammaticLaunchCompletion() to ensure that the update is visible to that dependent node before it is launched.

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

See also:

cudaGraphKernelNodeSetParam, cudaGraphKernelNodeSetEnabled, cudaGraphKernelNodeUpdatesApply

template < typename T > __device__cudaError_t cudaGraphKernelNodeSetParam (cudaGraphDeviceNode_t node, size_t offset, const T value)

Updates the kernel parameters of the given kernel node.

Parameters

node

- The node to update

offset

- The offset into the params at which to make the update

value

- Parameter value to write

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Updates the kernel parameters of `node` at `offset` to `value`. `node` must be device-updatable, and must reside upon the same device as the calling kernel.

If this function is called for the node's immediate dependent and that dependent is configured for programmatic dependent launch, then a memory fence must be invoked via __threadfence() before kickoff of the dependent is triggered via cudaTriggerProgrammaticLaunchCompletion() to ensure that the update is visible to that dependent node before it is launched.

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

See also:

cudaGraphKernelNodeSetEnabled, cudaGraphKernelNodeSetGridDim, cudaGraphKernelNodeUpdatesApply

__device__cudaError_t cudaGraphKernelNodeSetParam (cudaGraphDeviceNode_t node, size_t offset, const void *value, size_t size)

Updates the kernel parameters of the given kernel node.

Parameters

node

- The node to update

offset

- The offset into the params at which to make the update

value

- Buffer containing the params to write

size

- Size in bytes to update

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Updates `size` bytes in the kernel parameters of `node` at `offset` to the contents of `value`. `node` must be device-updatable, and must reside upon the same device as the calling kernel.

If this function is called for the node's immediate dependent and that dependent is configured for programmatic dependent launch, then a memory fence must be invoked via __threadfence() before kickoff of the dependent is triggered via cudaTriggerProgrammaticLaunchCompletion() to ensure that the update is visible to that dependent node before it is launched.

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

See also:

cudaGraphKernelNodeSetEnabled, cudaGraphKernelNodeSetGridDim, cudaGraphKernelNodeUpdatesApply

__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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

See also:

cudaGraphNodeSetParams, cudaLaunchKernel, cudaGraphAddKernelNode, cudaGraphKernelNodeGetParams

__device__cudaError_t cudaGraphKernelNodeUpdatesApply (const cudaGraphKernelNodeUpdate *updates, size_t updateCount)

Batch applies multiple kernel node updates.

Parameters

updates

- The updates to apply

updateCount

- The number of updates to apply

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Batch applies one or more kernel node updates based on the information provided in `updates`. `updateCount` specifies the number of updates to apply. Each entry in `updates` must specify a node to update, the type of update to apply, and the parameters for that type of update. See the documentation for cudaGraphKernelNodeUpdate for more detail.

If this function is called for the node's immediate dependent and that dependent is configured for programmatic dependent launch, then a memory fence must be invoked via __threadfence() before kickoff of the dependent is triggered via cudaTriggerProgrammaticLaunchCompletion() to ensure that the update is visible to that dependent node before it is launched.

Note:

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

See also:

cudaGraphKernelNodeSetParam, cudaGraphKernelNodeSetEnabled, cudaGraphKernelNodeSetGridDim

__host____device__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

__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

__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 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, cudaGraphAddMemcpyNode, cudaGraphMemcpyNodeSetParams

__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 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:

cudaGraphNodeSetParams, cudaMemcpy3D, cudaGraphMemcpyNodeSetParamsToSymbol, cudaGraphMemcpyNodeSetParamsFromSymbol, cudaGraphMemcpyNodeSetParams1D, cudaGraphAddMemcpyNode, cudaGraphMemcpyNodeGetParams

__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

__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 `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.
- 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:

cudaMemcpyFromSymbol, cudaGraphMemcpyNodeSetParams, cudaGraphMemcpyNodeSetParamsToSymbol, cudaGraphAddMemcpyNode, cudaGraphMemcpyNodeGetParams

__host__cudaError_t cudaGraphMemcpyNodeSetParamsToSymbol (cudaGraphNode_t node, const void *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 `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

__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 cudaMemsetParams *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 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:

cudaGraphNodeSetParams, 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 cudaGraphClone. `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 `pClonedNode`.

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:

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. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

cudaGraphNodeGetDependentNodes, cudaGraphGetNodes, cudaGraphGetRootNodes, cudaGraphGetEdges, cudaGraphAddDependencies, cudaGraphRemoveDependencies

__host__cudaError_t cudaGraphNodeGetDependencies_v2 (cudaGraphNode_t node, cudaGraphNode_t *pDependencies, cudaGraphEdgeData *edgeData, size_t *pNumDependencies)

Returns a node's dependencies (12.3+).

Parameters

node
- Node to query

pDependencies
- Pointer to return the dependencies

edgeData
- Optional array to return edge data for each dependency

pNumDependencies
- See description

Returns

cudaSuccess, cudaErrorLossyQuery, 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 that if an edge has non-zero (non-default) edge data and `edgeData` is NULL, this API will return cudaErrorLossyQuery. If `edgeData` is non-NULL, then `pDependencies` must be as well.

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:

cudaGraphNodeGetDependentNodes, cudaGraphGetNodes, 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.

See also:

cudaGraphNodeGetDependencies, cudaGraphGetNodes, cudaGraphGetRootNodes, cudaGraphGetEdges, cudaGraphAddDependencies, cudaGraphRemoveDependencies

__host__cudaError_t cudaGraphNodeGetDependentNodes_v2 (cudaGraphNode_t node, cudaGraphNode_t *pDependentNodes, cudaGraphEdgeData *edgeData, size_t *pNumDependentNodes)

Returns a node's dependent nodes (12.3+).

Parameters

node
    - Node to query

pDependentNodes
    - Pointer to return the dependent nodes

edgeData
    - Optional pointer to return edge data for dependent nodes

pNumDependentNodes
    - See description

Returns

cudaSuccess, cudaErrorLossyQuery, 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 that if an edge has non-zero (non-default) edge data and `edgeData` is NULL, this API will return cudaErrorLossyQuery. If `edgeData` is non-NULL, then `pDependentNodes` must be as well.

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

__host__cudaError_t cudaGraphNodeGetEnabled (cudaGraphExec_t hGraphExec, cudaGraphNode_t hNode, unsigned int *isEnabled)

Query whether a node in the given graphExec is enabled.

Parameters

hGraphExec
- The executable graph in which to set the specified node

hNode
- Node from the graph from which graphExec was instantiated

isEnabled
- Location to return the enabled status of the node

Returns

cudaSuccess, cudaErrorInvalidValue,

Description

Sets isEnabled to 1 if `hNode` is enabled, or 0 if `hNode` is disabled.

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.

Note:

Currently only kernel, memset and memcpy nodes are 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:

cudaGraphNodeSetEnabled, cudaGraphExecUpdate, cudaGraphInstantiate cudaGraphLaunch

__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:

cudaGraphGetNodes, cudaGraphGetRootNodes, cudaGraphChildGraphNodeGetGraph, cudaGraphKernelNodeGetParams, cudaGraphKernelNodeSetParams, cudaGraphHostNodeGetParams, cudaGraphHostNodeSetParams, cudaGraphMemcpyNodeGetParams, cudaGraphMemcpyNodeSetParams, cudaGraphMemsetNodeGetParams, cudaGraphMemsetNodeSetParams

__host__cudaError_t cudaGraphNodeSetEnabled (cudaGraphExec_t hGraphExec, cudaGraphNode_t hNode, unsigned int isEnabled)

Enables or disables the specified node in the given graphExec.

Parameters

hGraphExec
- The executable graph in which to set the specified node

hNode
- Node from the graph from which graphExec was instantiated

isEnabled
- Node is enabled if != 0, otherwise the node is disabled

Returns

cudaSuccess, cudaErrorInvalidValue,

Description

Sets `hNode` to be either enabled or disabled. Disabled nodes are functionally equivalent to empty nodes until they are reenabled. Existing node parameters are not affected by disabling/enabling the node.

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.

Note:

Currently only kernel, memset and memcpy nodes are 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:

cudaGraphNodeGetEnabled, cudaGraphExecUpdate, cudaGraphInstantiate cudaGraphLaunch

__host__cudaError_t cudaGraphNodeSetParams (cudaGraphNode_t node, cudaGraphNodeParams *nodeParams)

Update's a graph node's parameters.

Parameters

node

- Node to set the parameters for

nodeParams

- Parameters to copy

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDeviceFunction, cudaErrorNotSupported

Description

Sets the parameters of graph node `node` to `nodeParams`. The node type specified by `nodeParams->type` must match the type of `node`. `nodeParams` must be fully initialized and all unused bytes (reserved, padding) zeroed.

Modifying parameters is not supported for node types cudaGraphNodeTypeMemAlloc and cudaGraphNodeTypeMemFree.

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:

cudaGraphAddNode, cudaGraphExecNodeSetParams

__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 `pDependencies` 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.

See also:

cudaGraphAddDependencies, cudaGraphGetEdges, cudaGraphNodeGetDependencies, cudaGraphNodeGetDependentNodes

__host__cudaError_t cudaGraphRemoveDependencies_v2 (cudaGraph_t graph, const cudaGraphNode_t *from, const cudaGraphNode_t *to, const cudaGraphEdgeData *edgeData, size_t numDependencies)

Removes dependency edges from a graph. (12.3+).

Parameters

graph

- Graph from which to remove dependencies

from

- Array of nodes that provide the dependencies

to

- Array of dependent nodes

edgeData

- Optional array of edge data. If NULL, edge data is assumed to be default (zeroed).

numDependencies

- Number of dependencies to be removed

Returns

cudaSuccess, cudaErrorInvalidValue

Description

The number of `pDependencies` 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 an edge that does not exist in the graph, with data matching `edgeData`, results in an error. `edgeData` is nullable, which is equivalent to passing default (zeroed) data for each edge.

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:

cudaGraphAddDependencies, cudaGraphGetEdges, cudaGraphNodeGetDependencies, cudaGraphNodeGetDependentNodes

__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

__device__ void cudaGraphSetConditional (cudaGraphConditionalHandle handle, unsigned int value)

Sets the condition value associated with a conditional node.

Description

Sets the condition value associated with a conditional node.

Note: `handle` must be associated with the same context as the kernel calling this function.

See also:

cudaGraphConditionalHandleCreate

__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

__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.

__host__cudaError_t cudaGetDriverEntryPoint (const char *symbol, void **funcPtr, unsigned long long flags, cudaDriverEntryPointQueryResult *driverStatus)

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.

driverStatus

- Optional location to store the status of finding the symbol from the driver. See cudaDriverEntryPointQueryResult for possible values.

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorNotSupported

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 cudaSuccess and set the returned `funcPtr` if the requested driver function is valid and supported on the platform.

The API will return cudaSuccess and set the returned `funcPtr` to NULL 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.

It will also set the optional `driverStatus` to one of the values in cudaDriverEntryPointQueryResult with the following meanings:

- cudaDriverEntryPointSuccess - The requested symbol was succesfully found based on input arguments and `pfn` is valid
- cudaDriverEntryPointSymbolNotFound - The requested symbol was not found
- cudaDriverEntryPointVersionNotSufficent - The requested symbol was found but is not supported by the current runtime version (CUDART_VERSION)

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

__host__cudaError_t cudaGetDriverEntryPointByVersion (const char *symbol, void **funcPtr, unsigned int cudaVersion, unsigned long long flags, cudaDriverEntryPointQueryResult *driverStatus)

Returns the requested driver API function pointer by CUDA version.

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.

funcPtr

- Location to return the function pointer to the requested driver function

cudaVersion

- The CUDA version to look for the requested driver symbol

flags

- Flags to specify search options.

driverStatus

- Optional location to store the status of finding the symbol from the driver. See cudaDriverEntryPointQueryResult for possible values.

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorNotSupported

Description

Returns in `**funcPtr` the address of the CUDA driver function for the requested flags and CUDA driver version.

The CUDA version is specified as (1000 * major + 10 * minor), so CUDA 11.2 should be specified as 11020. For a requested driver symbol, if the specified CUDA version is greater than or equal to the CUDA version in which the driver symbol was introduced, this API will return the function pointer to the corresponding versioned 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.

For the case where the CUDA version requested is greater than the CUDA Toolkit installed, there may not be an appropriate function pointer typedef in the corresponding header file and may need a custom typedef to match the driver function signature returned. This can be done by getting the typedefs from a later toolkit or creating appropriately matching custom function typedefs.

The API will return cudaSuccess and set the returned `funcPtr` if the requested driver function is valid and supported on the platform.

The API will return cudaSuccess and set the returned `funcPtr` to NULL if the requested driver function is not supported on the platform, no ABI compatible driver function exists for the requested version or if the driver symbol is invalid.

It will also set the optional `driverStatus` to one of the values in cudaDriverEntryPointQueryResult with the following meanings:

- cudaDriverEntryPointSuccess - The requested symbol was succesfully found based on input arguments and `pfn` is valid
- cudaDriverEntryPointSymbolNotFound - The requested symbol was not found
- cudaDriverEntryPointVersionNotSufficent - The requested symbol was found but is not supported by the specified version `cudaVersion`

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. Library Management

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

__host__cudaError_t cudaKernelSetAttributeForDevice (cudaKernel_t kernel, cudaFuncAttribute attr, int value, int device)

Sets information about a kernel.

Parameters

kernel

- Kernel to set attribute of

attr

- Attribute requested

value

- Value to set

device

- Device to set attribute of

Returns

cudaSuccess, cudaErrorInvalidDeviceFunction, cudaErrorInvalidValue

Description

This call sets the value of a specified attribute `attr` on the kernel `kernel` for the requested device `device` to an integer value specified by `value`. This function returns cudaSuccess if the new value of the attribute could be successfully set. If the set fails, this call will return an error. Not all attributes can have values set. Attempting to set a value on a read-only attribute will result in an error (cudaErrorInvalidValue)

Note that attributes set using cudaFuncSetAttribute() will override the attribute set by this API irrespective of whether the call to cudaFuncSetAttribute() is made before or after this API call. Because of this and the stricter locking requirements mentioned below it is suggested that this call be used during the initialization path and not on each thread accessing `kernel` such as on kernel launches or on the critical path.

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.
- cudaFuncAttributeRequiredClusterWidth: The required cluster width in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.
- cudaFuncAttributeRequiredClusterHeight: The required cluster height in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.
- cudaFuncAttributeRequiredClusterDepth: The required cluster depth in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.
- cudaFuncAttributeNonPortableClusterSizeAllowed: Indicates whether the function can be launched with non-portable cluster size. 1 is allowed, 0 is disallowed.
- cudaFuncAttributeClusterSchedulingPolicyPreference: The block scheduling policy of a function. The value type is cudaClusterSchedulingPolicy.

Note:

> The API has stricter locking requirements in comparison to its legacy counterpart cudaFuncSetAttribute() due to device-wide semantics. If multiple threads are trying to set the same attribute on the same device simultaneously, the attribute setting will depend on the interleavings chosen by the OS scheduler and memory consistency.

See also:

cudaLibraryLoadData, cudaLibraryLoadFromFile, cudaLibraryUnload, cudaLibraryGetKernel, cudaLaunchKernel, cudaFuncSetAttribute, cuKernelSetAttribute

__host__cudaError_t cudaLibraryEnumerateKernels (cudaKernel_t *kernels, unsigned int numKernels, cudaLibrary_t lib)

Retrieve the kernel handles within a library.

Parameters

kernels
- Buffer where the kernel handles are returned to

numKernels
- Maximum number of kernel handles may be returned to the buffer

lib
- Library to query from

Returns

cudaSuccess, cudaErrorCudartUnloading, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle

Description

Returns in `kernels` a maximum number of `numKernels` kernel handles within `lib`. The returned kernel handle becomes invalid when the library is unloaded.

See also:

cudaLibraryGetKernelCount, cuLibraryEnumerateKernels

__host__cudaError_t cudaLibraryGetGlobal (void **dptr, size_t *bytes, cudaLibrary_t library, const char *name)

Returns a global device pointer.

Parameters

dptr
- Returned global device pointer for the requested library

bytes
- Returned global size in bytes

library
- Library to retrieve global from

name
- Name of global to retrieve

Returns

cudaSuccess, cudaErrorCudartUnloading, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorSymbolNotFound cudaErrorDeviceUninitialized, cudaErrorContextIsDestroyed

Description

Returns in `*dptr` and `*bytes` the base pointer and size of the global with name `name` for the requested library `library` and the current device. If no global for the requested name `name` exists, the call returns cudaErrorSymbolNotFound. One of the parameters `dptr` or `bytes` (not both) can be NULL in which case it is ignored. The returned `dptr` cannot be passed to the Symbol APIs such as cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaGetSymbolAddress, or cudaGetSymbolSize.

See also:

cudaLibraryLoadData, cudaLibraryLoadFromFile, cudaLibraryUnload, cudaLibraryGetManaged, cuLibraryGetGlobal

__host__cudaError_t cudaLibraryGetKernel (cudaKernel_t *pKernel, cudaLibrary_t library, const char *name)

Returns a kernel handle.

Parameters

pKernel
- Returned kernel handle

library
- Library to retrieve kernel from

name

- Name of kernel to retrieve

Returns

cudaSuccess, cudaErrorCudartUnloading, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorSymbolNotFound

Description

Returns in `pKernel` the handle of the kernel with name `name` located in library `library`. If kernel handle is not found, the call returns cudaErrorSymbolNotFound.

See also:

cudaLibraryLoadData, cudaLibraryLoadFromFile, cudaLibraryUnload, cuLibraryGetKernel

__host__cudaError_t cudaLibraryGetKernelCount (unsigned int *count, cudaLibrary_t lib)

Returns the number of kernels within a library.

Parameters

count

- Number of kernels found within the library

lib

- Library to query

Returns

cudaSuccess, cudaErrorCudartUnloading, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle

Description

Returns in `count` the number of kernels in `lib`.

See also:

cudaLibraryEnumerateKernels, cudaLibraryLoadFromFile, cudaLibraryLoadData, cuLibraryGetKernelCount

__host__cudaError_t cudaLibraryGetManaged (void **dptr, size_t *bytes, cudaLibrary_t library, const char *name)

Returns a pointer to managed memory.

Parameters

dptr
- Returned pointer to the managed memory

bytes
- Returned memory size in bytes

library
- Library to retrieve managed memory from

name
- Name of managed memory to retrieve

Returns

cudaSuccess, cudaErrorCudartUnloading, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorSymbolNotFound

Description

Returns in `*dptr` and `*bytes` the base pointer and size of the managed memory with name `name` for the requested library `library`. If no managed memory with the requested name `name` exists, the call returns cudaErrorSymbolNotFound. One of the parameters `dptr` or `bytes` (not both) can be NULL in which case it is ignored. Note that managed memory for library `library` is shared across devices and is registered when the library is loaded. The returned `dptr` cannot be passed to the Symbol APIs such as cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaGetSymbolAddress, or cudaGetSymbolSize.

See also:

cudaLibraryLoadData, cudaLibraryLoadFromFile, cudaLibraryUnload, cudaLibraryGetGlobal, cuLibraryGetManaged

__host__cudaError_t cudaLibraryGetUnifiedFunction (void **fptr, cudaLibrary_t library, const char *symbol)

Returns a pointer to a unified function.

Parameters

fptr
- Returned pointer to a unified function

library
- Library to retrieve function pointer memory from

symbol
- Name of function pointer to retrieve

Returns

cudaSuccess, cudaErrorCudartUnloading, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorSymbolNotFound

Description

Returns in `*fptr` the function pointer to a unified function denoted by `symbol`. If no unified function with name `symbol` exists, the call returns cudaErrorSymbolNotFound. If there is no device with attribute cudaDeviceProp::unifiedFunctionPointers present in the system, the call may return cudaErrorSymbolNotFound.

See also:

cudaLibraryLoadData, cudaLibraryLoadFromFile, cudaLibraryUnload, cuLibraryGetUnifiedFunction

__host__cudaError_t cudaLibraryLoadData (cudaLibrary_t *library, const void *code, cudaJitOption *jitOptions, void **jitOptionsValues, unsigned int numJitOptions, cudaLibraryOption *libraryOptions, void **libraryOptionValues, unsigned int numLibraryOptions)

Load a library with specified code and options.

Parameters

library
- Returned library

code
- Code to load

jitOptions

- Options for JIT

jitOptionsValues

- Option values for JIT

numJitOptions

- Number of options

libraryOptions

- Options for loading

libraryOptionValues

- Option values for loading

numLibraryOptions

- Number of options for loading

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation, cudaErrorInitializationError, cudaErrorCudartUnloading, cudaErrorInvalidPtx, cudaErrorUnsupportedPtxVersion, cudaErrorNoKernelImageForDevice, cudaErrorSharedObjectSymbolNotFound, cudaErrorSharedObjectInitFailed, cudaErrorJitCompilerNotFound

Description

Takes a pointer `code` and loads the corresponding library `library` based on the application defined library loading mode:

- If module loading is set to EAGER, via the environment variables described in "Module loading", `library` is loaded eagerly into all contexts at the time of the call and future contexts at the time of creation until the library is unloaded with cudaLibraryUnload().
- If the environment variables are set to LAZY, `library` is not immediately loaded onto all existent contexts and will only be loaded when a function is needed for that context, such as a kernel launch.

These environment variables are described in the CUDA programming guide under the "CUDA environment variables" section.

The `code` may be a cubin or fatbin as output by nvcc, or a NULL-terminated PTX, either as output by nvcc or hand-written. A fatbin should also contain relocatable code when doing separate compilation. Please also see the documentation for nvrtc (https://docs.nvidia.com/cuda/nvrtc/index.html), nvjitlink (https://docs.nvidia.com/cuda/nvjitlink/index.html), and nvfatbin (https://docs.nvidia.com/cuda/nvfatbin/index.html) for more information on generating loadable code at runtime.

Options are passed as an array via `jitOptions` and any corresponding parameters are passed in `jitOptionsValues`. The number of total JIT options is supplied via `numJitOptions`. Any outputs will be returned via `jitOptionsValues`.

Library load options are passed as an array via `libraryOptions` and any corresponding parameters are passed in `libraryOptionValues`. The number of total library load options is supplied via `numLibraryOptions`.

See also:

cudaLibraryLoadFromFile, cudaLibraryUnload, cuLibraryLoadData

__host__cudaError_t cudaLibraryLoadFromFile (cudaLibrary_t *library, const char *fileName, cudaJitOption *jitOptions, void **jitOptionsValues, unsigned int numJitOptions, cudaLibraryOption *libraryOptions, void **libraryOptionValues, unsigned int numLibraryOptions)

Load a library with specified file and options.

Parameters

library
- Returned library

fileName
- File to load from

jitOptions
- Options for JIT

jitOptionsValues
- Option values for JIT

numJitOptions
- Number of options

libraryOptions
- Options for loading

libraryOptionValues
- Option values for loading

numLibraryOptions
- Number of options for loading

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation, cudaErrorInitializationError, cudaErrorCudartUnloading, cudaErrorInvalidPtx, cudaErrorUnsupportedPtxVersion, cudaErrorNoKernelImageForDevice, cudaErrorSharedObjectSymbolNotFound, cudaErrorSharedObjectInitFailed, cudaErrorJitCompilerNotFound

Description

Takes a pointer `code` and loads the corresponding library `library` based on the application defined library loading mode:

- If module loading is set to EAGER, via the environment variables described in "Module loading", `library` is loaded eagerly into all contexts at the time of the call and future contexts at the time of creation until the library is unloaded with cudaLibraryUnload().
- If the environment variables are set to LAZY, `library` is not immediately loaded onto all existent contexts and will only be loaded when a function is needed for that context, such as a kernel launch.

These environment variables are described in the CUDA programming guide under the "CUDA environment variables" section.

The file should be a cubin file as output by nvcc, or a PTX file either as output by nvcc or handwritten, or a fatbin file as output by nvcc. A fatbin should also contain relocatable code when doing separate compilation. Please also see the documentation for nvrtc (https://docs.nvidia.com/cuda/nvrtc/index.html), nvjitlink (https://docs.nvidia.com/cuda/nvjitlink/index.html), and nvfatbin (https://docs.nvidia.com/cuda/nvfatbin/index.html) for more information on generating loadable code at runtime.

Options are passed as an array via `jitOptions` and any corresponding parameters are passed in `jitOptionsValues`. The number of total options is supplied via `numJitOptions`. Any outputs will be returned via `jitOptionsValues`.

Library load options are passed as an array via `libraryOptions` and any corresponding parameters are passed in `libraryOptionValues`. The number of total library load options is supplied via `numLibraryOptions`.

See also:

cudaLibraryLoadData, cudaLibraryUnload, cuLibraryLoadFromFile

__host__cudaError_t cudaLibraryUnload (cudaLibrary_t library)

Unloads a library.

Parameters

library

- Library to unload

Returns

cudaSuccess, cudaErrorCudartUnloading, cudaErrorInitializationError, cudaErrorInvalidValue

Description

Unloads the library specified with `library`

See also:

cudaLibraryLoadData, cudaLibraryLoadFromFile, cuLibraryUnload

6.33. 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 > __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, cudaChannelFormatKindSignedNormalized8X1, cudaChannelFormatKindSignedNormalized8X2, cudaChannelFormatKindSignedNormalized8X4, cudaChannelFormatKindUnsignedNormalized8X1, cudaChannelFormatKindUnsignedNormalized8X2, cudaChannelFormatKindUnsignedNormalized8X4, cudaChannelFormatKindSignedNormalized16X1, cudaChannelFormatKindSignedNormalized16X2, cudaChannelFormatKindSignedNormalized16X4, cudaChannelFormatKindUnsignedNormalized16X1, cudaChannelFormatKindUnsignedNormalized16X2, cudaChannelFormatKindUnsignedNormalized16X4, cudaChannelFormatKindUnsignedNormalized1010102 or cudaChannelFormatKindNV12.

The format is specified by the template specialization.

The template function specializes for the following scalar types: char, signed char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, and float. The template function specializes for the following vector types: char{1|2|4}, uchar{1|2|4}, short{1|2|4}, ushort{1|2|4}, int{1|2|4}, uint{1|2|4}, long{1|2|4}, ulong{1|2|4}, float{1|2|4}. The template function specializes for following cudaChannelFormatKind enum values: cudaChannelFormatKind{Uns|S}ignedNormalized{8|16}X{1|2|4}, cudaChannelFormatKindUnsignedNormalized1010102 and cudaChannelFormatKindNV12.

Invoking the function on a type without a specialization defaults to creating a channel format of kind cudaChannelFormatKindNone

See also:

cudaCreateChannelDesc (Low level), cudaGetChannelDesc,

__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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

cudaLaunchKernel (C++ API), cudaFuncSetCacheConfig (C++ API), cudaFuncGetAttributes (C API), cudaSetDoubleForDevice, cudaSetDoubleForHost

template < class T > __host__cudaError_t cudaFuncGetName (const char **name, T *func)

Returns the function name for a device entry function pointer.

Parameters

name

- The returned name of the function

func

- The function pointer to retrieve name for

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDeviceFunction

Description

Returns in `**name` the function name associated with the symbol `func` . The function name is returned as a null-terminated string. This API may return a mangled name if the function is not declared as having C linkage. If `**name` is NULL, cudaErrorInvalidValue is returned. If `func` is not a device entry function, cudaErrorInvalidDeviceFunction 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.

> - The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
> - Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

cudaFuncGetName (C API)

template < class T > __host__cudaError_t cudaFuncSetAttribute (T *func, cudaFuncAttribute attr, int value)

[C++ API] Set attributes for a given function

Parameters

func

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.

- cudaFuncAttributeRequiredClusterWidth: The required cluster width in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.
- cudaFuncAttributeRequiredClusterHeight: The required cluster height in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.
- cudaFuncAttributeRequiredClusterDepth: The required cluster depth in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.
- cudaFuncAttributeNonPortableClusterSizeAllowed: Indicates whether the function can be launched with non-portable cluster size. 1 is allowed, 0 is disallowed.
- cudaFuncAttributeClusterSchedulingPolicyPreference: The block scheduling policy of a function. The value type is cudaClusterSchedulingPolicy.

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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

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)
- 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 cudaGetKernel (cudaKernel_t *kernelPtr, T *func)

Get pointer to device kernel that matches entry function `entryFuncAddr`.

Parameters

kernelPtr
- Returns the device kernel

func

Returns

cudaSuccess

Description

Returns in `kernelPtr` the device kernel corresponding to the entry function `entryFuncAddr`.

See also:

cudaGetKernel (C API)

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)

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 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:

cudaMemcpyFromSymbol, cudaGraphAddMemcpyNode, cudaGraphAddMemcpyNodeToSymbol, cudaGraphMemcpyNodeGetParams, cudaGraphMemcpyNodeSetParams, cudaGraphMemcpyNodeSetParamsFromSymbol, cudaGraphMemcpyNodeSetParamsToSymbol, cudaGraphCreate, cudaGraphDestroyNode, cudaGraphAddChildGraphNode, cudaGraphAddEmptyNode, cudaGraphAddKernelNode, cudaGraphAddHostNode, cudaGraphAddMemsetNode

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 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, cudaGraphInstantiate, cudaGraphExecMemcpyNodeSetParams, cudaGraphExecMemcpyNodeSetParamsFromSymbol, cudaGraphExecKernelNodeSetParams, cudaGraphExecMemsetNodeSetParams, cudaGraphExecHostNodeSetParams

__host__cudaError_t cudaGraphInstantiate (cudaGraphExec_t *pGraphExec, cudaGraph_t graph, cudaGraphNode_t *pErrorNode, char *pLogBuffer, size_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

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.
- 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:

cudaMemcpyFromSymbol, cudaGraphMemcpyNodeSetParams, cudaGraphMemcpyNodeSetParamsToSymbol, cudaGraphAddMemcpyNode, 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 `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

template < class T > __host__cudaError_t cudaLaunchCooperativeKernel (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 dimentions

blockDim
- Block dimentions

args
- Arguments

sharedMem
- Shared memory (defaults to 0)

stream
- Stream identifier (defaults to NULL)

Returns

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

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.

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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

cudaLaunchCooperativeKernel (C API)

template < class T > __host__cudaError_t cudaLaunchKernel (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 dimentions

blockDim
- Block dimentions

args
- Arguments

sharedMem
- Shared memory (defaults to 0)

stream
- Stream identifier (defaults to NULL)

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.

`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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

cudaLaunchKernel (C API)

template < typename... ActTypes > __host__cudaError_t cudaLaunchKernelEx (const cudaLaunchConfig_t *config, const cudaKernel_t kernel, ActTypes &&... args)

Launches a CUDA function with launch-time configuration.

Parameters

config
  - Launch configuration

kernel

args
  - Parameter pack of kernel parameters

Returns

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

Description

Invokes the kernel `kernel` on `config->gridDim` (`config->gridDim.x` `config->gridDim.y` `config->gridDim.z`) grid of blocks. Each block contains `config->blockDim` (`config->blockDim.x` `config->blockDim.y` `config->blockDim.z`) threads.

`config->dynamicSmemBytes` sets the amount of dynamic shared memory that will be available to each thread block.

`config->stream` specifies a stream the invocation is associated to.

Configuration beyond grid and block dimensions, dynamic shared memory size, and stream can be provided with the following two fields of `config`:

`config->attrs` is an array of `config->numAttrs` contiguous cudaLaunchAttribute elements. The value of this pointer is not considered if `config->numAttrs` is zero. However, in that case, it is recommended to set the pointer to NULL. `config->numAttrs` is the number of attributes populating the first `config->numAttrs` positions of the `config->attrs` array.

The kernel arguments should be passed as arguments to this function via the `args` parameter pack.

The C API version of this function, `cudaLaunchKernelExC`, is also available for pre-C++11 compilers and for use cases where the ability to pass kernel parameters via void* array is preferable.

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:

cudaLaunchKernelEx (C API), cuLaunchKernelEx

template < typename... ExpTypes, typename... ActTypes > __host__cudaError_t cudaLaunchKernelEx (const cudaLaunchConfig_t *config, void(*)(ExpTypes...) kernel, ActTypes &&... args)

Launches a CUDA function with launch-time configuration.

Parameters

config
- Launch configuration

kernel
- Kernel to launch

args

- Parameter pack of kernel parameters

Returns

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

Description

Invokes the kernel `kernel` on `config->gridDim` (`config->gridDim.x` `config->gridDim.y` `config->gridDim.z`) grid of blocks. Each block contains `config->blockDim` (`config->blockDim.x` `config->blockDim.y` `config->blockDim.z`) threads.

`config->dynamicSmemBytes` sets the amount of dynamic shared memory that will be available to each thread block.

`config->stream` specifies a stream the invocation is associated to.

Configuration beyond grid and block dimensions, dynamic shared memory size, and stream can be provided with the following two fields of `config`:

`config->attrs` is an array of `config->numAttrs` contiguous cudaLaunchAttribute elements. The value of this pointer is not considered if `config->numAttrs` is zero. However, in that case, it is recommended to set the pointer to NULL. `config->numAttrs` is the number of attributes populating the first `config->numAttrs` positions of the `config->attrs` array.

The kernel arguments should be passed as arguments to this function via the `args` parameter pack.

The C API version of this function, `cudaLaunchKernelExC`, is also available for pre-C++11 compilers and for use cases where the ability to pass kernel parameters via void* array is preferable.

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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.

- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

See also:

cudaLaunchKernelEx (C API), cuLaunchKernelEx

template < class T > __host__cudaError_t cudaLibraryGetGlobal (T **dptr, size_t *bytes, cudaLibrary_t library, const char *name)

Returns a global device pointer.

Parameters

dptr
- Returned global device pointer for the requested library

bytes
- Returned global size in bytes

library
- Library to retrieve global from

name
- Name of global to retrieve

Returns

cudaSuccess, cudaErrorCudartUnloading, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorSymbolNotFound cudaErrorDeviceUninitialized, cudaErrorContextIsDestroyed

Description

Returns in `*dptr` and `*bytes` the base pointer and size of the global with name `name` for the requested library `library` and the current device. If no global for the requested name `name` exists, the call returns cudaErrorSymbolNotFound. One of the parameters `dptr` or `bytes` (not both) can be NULL in which case it is ignored.

See also:

cudaLibraryLoadData, cudaLibraryLoadFromFile, cudaLibraryUnload, cudaLibraryGetManaged

template < class T > __host__cudaError_t cudaLibraryGetManaged (T **dptr, size_t *bytes, cudaLibrary_t library, const char *name)

Returns a pointer to managed memory.

Parameters

dptr

- Returned pointer to the managed memory

bytes

- Returned memory size in bytes

library

- Library to retrieve managed memory from

name

- Name of managed memory to retrieve

Returns

cudaSuccess, cudaErrorCudartUnloading, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorSymbolNotFound

Description

Returns in `*dptr` and `*bytes` the base pointer and size of the managed memory with name `name` for the requested library `library`. If no managed memory with the requested name `name` exists, the call returns cudaErrorSymbolNotFound. One of the parameters `dptr` or `bytes` (not both) can be NULL in which case it is ignored. Note that managed memory for library `library` is shared across devices and is registered when the library is loaded.

See also:

cudaLibraryLoadData, cudaLibraryLoadFromFile, cudaLibraryUnload, cudaLibraryGetGlobal

template < class T > __host__cudaError_t cudaLibraryGetUnifiedFunction (T **fptr, cudaLibrary_t library, const char *symbol)

Returns a pointer to a unified function.

Parameters

fptr

- Returned pointer to a unified function

library

- Library to retrieve function pointer memory from

symbol

- Name of function pointer to retrieve

Returns

cudaSuccess, cudaErrorCudartUnloading, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorSymbolNotFound

Description

Returns in `*fptr` the function pointer to a unified function denoted by `symbol`. If no unified function with name `symbol` exists, the call returns cudaErrorSymbolNotFound. If there is no device with attribute cudaDeviceProp::unifiedFunctionPointers present in the system, the call may return cudaErrorSymbolNotFound.

See also:

cudaLibraryLoadData, cudaLibraryLoadFromFile, cudaLibraryUnload

__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 cudaMallocFromPoolAsync made available through function overloading.

See also:

cudaMallocFromPoolAsync, cudaMallocAsync (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 `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.
- 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 cudaHostGetDevicePointer().
- 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.

cudaSetDeviceFlags() must have been called with the cudaDeviceMapHost flag in order for the cudaHostAllocMapped flag to have any effect.

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, 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 specifed 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 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 cudaMallocManaged 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 > cudaError_t cudaMemAdvise (T *devPtr, size_t count, cudaMemoryAdvise advice, cudaMemLocation location)

Advise about the usage of a given memory range.

Description

This is an alternate spelling for cudaMemAdvise made available through function overloading.

See also:

cudaMemAdvise, cudaMemAdvise (C API)

template < typename T, typename U > __host__cudaError_t cudaMemcpyBatchAsync (T **dsts, U **srcs, size_t *sizes, size_t count, cudaMemcpyAttributes attr, size_t *failIdx, cudaStream_t hStream)

Performs a batch of memory copies asynchronously.

Description

This is an alternate spelling for cudaMemcpyBatchAsync made available through function overloading.

The cudaMemcpyAttributes specified by `attr` are applicable for all the copies specified in the batch.

See also:

cudaMemcpyBatchAsync

template < typename T, typename U > __host__cudaError_t cudaMemcpyBatchAsync (T **dsts, U **srcs, size_t *sizes, size_t count, cudaMemcpyAttributes *attrs, size_t *attrsIdxs, size_t numAttrs, size_t *failIdx, cudaStream_t hStream)

Performs a batch of memory copies asynchronously.

Description

This is an alternate spelling for cudaMemcpyBatchAsync made available through function overloading.

See also:

cudaMemcpyBatchAsync

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, 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.

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, cudaMemcpyToSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DToArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

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 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, 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, 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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

See also:

cudaOccupancyMaxPotentialBlockSize

cudaOccupancyMaxPotentialBlockSizeWithFlags

cudaOccupancyMaxActiveBlocksPerMultiprocessor

cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags

cudaOccupancyMaxPotentialBlockSizeVariableSMem

cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags

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 calulated

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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

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

numBlocks
- Returned occupancy

func
- Kernel function for which occupancy is calulated

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.
> - The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.
> - Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessor

cudaOccupancyMaxPotentialBlockSize

cudaOccupancyMaxPotentialBlockSizeWithFlags

cudaOccupancyMaxPotentialBlockSizeVariableSMem

cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags

cudaOccupancyAvailableDynamicSMemPerBlock

template < class T > __host__cudaError_t cudaOccupancyMaxActiveClusters (int *numClusters, T *func, const cudaLaunchConfig_t *config)

Given the kernel function (`func`) and launch configuration (`config`), return the maximum number of clusters that could co-exist on the target device in `*numClusters`.

Parameters

numClusters

- Returned maximum number of clusters that could co-exist on the target device

func

- Kernel function for which maximum number of clusters are calculated

config

- Launch configuration for the given kernel function

Returns

cudaSuccess, cudaErrorInvalidDeviceFunction, cudaErrorInvalidValue, cudaErrorInvalidClusterSize, cudaErrorUnknown,

Description

If the function has required cluster size already set (see cudaFuncGetAttributes), the cluster size from config must either be unspecified or match the required size. Without required sizes, the cluster size must be specified in config, else the function will return an error.

Note that various attributes of the kernel function may affect occupancy calculation. Runtime environment may affect how the hardware schedules the clusters, so the calculated occupancy is not guaranteed to be achievable.

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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.

- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

See also:

cudaFuncGetAttributes

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

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 `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 achived 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 cudaOccupancyMaxPotentialClusterSize (int *clusterSize, T *func, const cudaLaunchConfig_t *config)

Given the kernel function (`func`) and launch configuration (`config`), return the maximum cluster size in `*clusterSize`.

Parameters

clusterSize

- Returned maximum cluster size that can be launched for the given kernel function and launch configuration

func

- Kernel function for which maximum cluster size is calculated

config

- Launch configuration for the given kernel function

Returns

cudaSuccess, cudaErrorInvalidDeviceFunction, cudaErrorInvalidValue, cudaErrorUnknown,

Description

The cluster dimensions in `config` are ignored. If func has a required cluster size set (see cudaFuncGetAttributes),`*clusterSize` will reflect the required cluster size.

By default this function will always return a value that's portable on future hardware. A higher value may be returned if the kernel function allows non-portable cluster sizes.

This function will respect the compile time launch bounds.

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.
- The API can also be used with a kernel cudaKernel_t by querying the handle using cudaLibraryGetKernel() or cudaGetKernel and then passing it to the API by casting to void*. The

symbol `entryFuncAddr` passed to cudaGetKernel should be a symbol that is registered with the same CUDA Runtime instance.

- Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is cudaKernel_t

See also:

cudaFuncGetAttributes

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 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

6.34. 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 CUcontext s 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 to the calling host thread.

The function cudaInitDevice() ensures that the primary context is initialized for the requested device but does not make it current to the calling thread.

The function cudaSetDevice() initializes the primary context for the specified device and makes it 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, except for cudaSetDeviceFlags() which will simply overwrite the previous settings.

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.

Interactions between CUkernel and cudaKernel_t

The types CUkernel and cudaKernel_t represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a cudaKernel_t in a CUDA Driver API function which takes a CUkernel, it is necessary to explicitly cast the cudaKernel_t to a CUkernel.

__host__cudaError_t cudaGetFuncBySymbol (cudaFunction_t *functionPtr, const void *symbolPtr)

Get pointer to device entry function that matches entry function `symbolPtr`.

Parameters

functionPtr

- Returns the device entry function

symbolPtr

- Pointer to device entry function to search for

Returns

cudaSuccess

Description

Returns in `functionPtr` the device entry function corresponding to the symbol `symbolPtr`.

__host__cudaError_t cudaGetKernel (cudaKernel_t *kernelPtr, const void *entryFuncAddr)

Get pointer to device kernel that matches entry function `entryFuncAddr`.

Parameters

kernelPtr
- Returns the device kernel

entryFuncAddr
- Address of device entry function to search kernel for

Returns

cudaSuccess

Description

Returns in `kernelPtr` the device kernel corresponding to the entry function `entryFuncAddr`.

Note that it is possible that there are multiple symbols belonging to different translation units with the same `entryFuncAddr` registered with this CUDA Runtime and so the order which the translation units are loaded and registered with the CUDA Runtime can lead to differing return pointers in `kernelPtr` . Suggested methods of ensuring uniqueness are to limit visibility of __global__ device functions by using static or hidden visibility attribute in the respective translation units.

See also:

cudaGetKernel (C++ API)

6.35. Profiler Control

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

__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.

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

See also:

cudaProfilerStop, cuProfilerStart

__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:

cudaProfilerStart, cuProfilerStop

6.36. Data types used by CUDA Runtime

struct cudaAccessPolicyWindow

struct cudaArrayMemoryRequirements

struct cudaArraySparseProperties

struct cudaAsyncNotificationInfo_t

struct cudaChannelFormatDesc

struct cudaChildGraphNodeParams

struct cudaConditionalNodeParams

struct cudaDeviceProp

struct cudaEglFrame

struct cudaEglPlaneDesc

struct cudaEventRecordNodeParams

struct cudaEventWaitNodeParams

struct cudaExtent

struct cudaExternalMemoryBufferDesc

struct cudaExternalMemoryHandleDesc

struct cudaExternalMemoryMipmappedArrayDesc

struct cudaExternalSemaphoreHandleDesc

struct cudaExternalSemaphoreSignalNodeParams

struct cudaExternalSemaphoreSignalNodeParamsV2

struct cudaExternalSemaphoreSignalParams

struct cudaExternalSemaphoreSignalParams_v1

struct cudaExternalSemaphoreWaitNodeParams

struct cudaExternalSemaphoreWaitNodeParamsV2

struct cudaExternalSemaphoreWaitParams

struct cudaExternalSemaphoreWaitParams_v1

struct cudaFuncAttributes

struct cudaGraphEdgeData

struct cudaGraphExecUpdateResultInfo

struct cudaGraphInstantiateParams

struct cudaGraphKernelNodeUpdate

struct cudaGraphNodeParams

struct cudaHostNodeParams

struct cudaHostNodeParamsV2

struct cudaIpcEventHandle_t

struct cudaIpcMemHandle_t

struct cudaKernelNodeParams

struct cudaKernelNodeParamsV2

struct cudaLaunchAttribute

union cudaLaunchAttributeValue

struct cudaLaunchConfig_t

struct cudaLaunchMemSyncDomainMap

struct cudaLaunchParams

struct cudaMemAccessDesc

struct cudaMemAllocNodeParams

struct cudaMemAllocNodeParamsV2

struct cudaMemcpy3DOperand

struct cudaMemcpy3DParms

struct cudaMemcpy3DPeerParms

struct cudaMemcpyAttributes

struct cudaMemcpyNodeParams

struct cudaMemFreeNodeParams

struct cudaMemLocation

struct cudaMemPoolProps

struct cudaMemPoolPtrExportData

struct cudaMemsetParams

struct cudaMemsetParamsV2

struct cudaOffset3D

struct cudaPitchedPtr

struct cudaPointerAttributes

struct cudaPos

struct cudaResourceDesc

struct cudaResourceViewDesc

struct cudaTextureDesc

struct CUuuid_st

enum 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 persit from cache.
cudaAccessPropertyPersisting = 2

Persisting access is more likely to persist in cache.

enum cudaAsyncNotificationType

Types of async notification that can occur

Values

cudaAsyncNotificationTypeOverBudget = 0x1
Sent when the process has exceeded its device memory budget

enum 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

enum 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
Unsigned 8-bit integers, planar 4:2:0 YUV format
cudaChannelFormatKindUnsignedNormalized8X1 = 5
1 channel unsigned 8-bit normalized integer
cudaChannelFormatKindUnsignedNormalized8X2 = 6
2 channel unsigned 8-bit normalized integer
cudaChannelFormatKindUnsignedNormalized8X4 = 7
4 channel unsigned 8-bit normalized integer

cudaChannelFormatKindUnsignedNormalized16X1 = 8
1 channel unsigned 16-bit normalized integer
cudaChannelFormatKindUnsignedNormalized16X2 = 9
2 channel unsigned 16-bit normalized integer
cudaChannelFormatKindUnsignedNormalized16X4 = 10
4 channel unsigned 16-bit normalized integer
cudaChannelFormatKindSignedNormalized8X1 = 11
1 channel signed 8-bit normalized integer
cudaChannelFormatKindSignedNormalized8X2 = 12
2 channel signed 8-bit normalized integer
cudaChannelFormatKindSignedNormalized8X4 = 13
4 channel signed 8-bit normalized integer
cudaChannelFormatKindSignedNormalized16X1 = 14
1 channel signed 16-bit normalized integer
cudaChannelFormatKindSignedNormalized16X2 = 15
2 channel signed 16-bit normalized integer
cudaChannelFormatKindSignedNormalized16X4 = 16
4 channel signed 16-bit normalized integer
cudaChannelFormatKindUnsignedBlockCompressed1 = 17
4 channel unsigned normalized block-compressed (BC1 compression) format
cudaChannelFormatKindUnsignedBlockCompressed1SRGB = 18
4 channel unsigned normalized block-compressed (BC1 compression) format with sRGB encoding
cudaChannelFormatKindUnsignedBlockCompressed2 = 19
4 channel unsigned normalized block-compressed (BC2 compression) format
cudaChannelFormatKindUnsignedBlockCompressed2SRGB = 20
4 channel unsigned normalized block-compressed (BC2 compression) format with sRGB encoding
cudaChannelFormatKindUnsignedBlockCompressed3 = 21
4 channel unsigned normalized block-compressed (BC3 compression) format
cudaChannelFormatKindUnsignedBlockCompressed3SRGB = 22
4 channel unsigned normalized block-compressed (BC3 compression) format with sRGB encoding
cudaChannelFormatKindUnsignedBlockCompressed4 = 23
1 channel unsigned normalized block-compressed (BC4 compression) format
cudaChannelFormatKindSignedBlockCompressed4 = 24
1 channel signed normalized block-compressed (BC4 compression) format
cudaChannelFormatKindUnsignedBlockCompressed5 = 25
2 channel unsigned normalized block-compressed (BC5 compression) format
cudaChannelFormatKindSignedBlockCompressed5 = 26
2 channel signed normalized block-compressed (BC5 compression) format
cudaChannelFormatKindUnsignedBlockCompressed6H = 27
3 channel unsigned half-float block-compressed (BC6H compression) format
cudaChannelFormatKindSignedBlockCompressed6H = 28
3 channel signed half-float block-compressed (BC6H compression) format
cudaChannelFormatKindUnsignedBlockCompressed7 = 29

4 channel unsigned normalized block-compressed (BC7 compression) format

cudaChannelFormatKindUnsignedBlockCompressed7SRGB = 30

4 channel unsigned normalized block-compressed (BC7 compression) format with sRGB encoding

cudaChannelFormatKindUnsignedNormalized1010102 = 31

4 channel unsigned normalized (10-bit, 10-bit, 10-bit, 2-bit) format

enum cudaClusterSchedulingPolicy

Cluster scheduling policies. These may be passed to cudaFuncSetAttribute

Values

cudaClusterSchedulingPolicyDefault = 0

the default policy

cudaClusterSchedulingPolicySpread = 1

spread the blocks within a cluster to the SMs

cudaClusterSchedulingPolicyLoadBalancing = 2

allow the hardware to load-balance the blocks in a cluster to the SMs

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

cudaDevAttrMaxTexture2DLayeredHeight = 28

Maximum 2D layered texture height

cudaDevAttrMaxTexture2DLayeredLayers = 29

Maximum layers in a 2D layered texture

cudaDevAttrSurfaceAlignment = 30

Alignment requirement for surfaces

cudaDevAttrConcurrentKernels = 31

Device can possibly execute multiple kernels concurrently

cudaDevAttrEccEnabled = 32

Device has ECC support enabled

cudaDevAttrPciBusId = 33

PCI bus ID of the device

cudaDevAttrPciDeviceId = 34

PCI device ID of the device

cudaDevAttrTccDriver = 35

Device is using TCC driver model

cudaDevAttrMemoryClockRate = 36

Peak memory clock frequency in kilohertz

cudaDevAttrGlobalMemoryBusWidth = 37

Global memory bus width in bits

cudaDevAttrL2CacheSize = 38

Size of L2 cache in bytes

cudaDevAttrMaxThreadsPerMultiProcessor = 39

Maximum resident threads per multiprocessor

cudaDevAttrAsyncEngineCount = 40

Number of asynchronous engines

cudaDevAttrUnifiedAddressing = 41

Device shares a unified address space with the host

cudaDevAttrMaxTexture1DLayeredWidth = 42

Maximum 1D layered texture width

cudaDevAttrMaxTexture1DLayeredLayers = 43

Maximum layers in a 1D layered texture

cudaDevAttrMaxTexture2DGatherWidth = 45

Maximum 2D texture width if cudaArrayTextureGather is set

cudaDevAttrMaxTexture2DGatherHeight = 46

Maximum 2D texture height if cudaArrayTextureGather is set

cudaDevAttrMaxTexture3DWidthAlt = 47
Alternate maximum 3D texture width
cudaDevAttrMaxTexture3DHeightAlt = 48
Alternate maximum 3D texture height
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

cudaDevAttrMaxTexture2DLinearPitch = 72

Maximum 2D linear texture pitch in bytes

cudaDevAttrMaxTexture2DMipmappedWidth = 73

Maximum mipmapped 2D texture width

cudaDevAttrMaxTexture2DMipmappedHeight = 74

Maximum mipmapped 2D texture height

cudaDevAttrComputeCapabilityMajor = 75

Major compute capability version number

cudaDevAttrComputeCapabilityMinor = 76

Minor compute capability version number

cudaDevAttrMaxTexture1DMipmappedWidth = 77

Maximum mipmapped 1D texture width

cudaDevAttrStreamPrioritiesSupported = 78

Device supports stream priorities

cudaDevAttrGlobalL1CacheSupported = 79

Device supports caching globals in L1

cudaDevAttrLocalL1CacheSupported = 80

Device supports caching locals in L1

cudaDevAttrMaxSharedMemoryPerMultiprocessor = 81

Maximum shared memory available per multiprocessor in bytes

cudaDevAttrMaxRegistersPerMultiprocessor = 82

Maximum number of 32-bit registers available per multiprocessor

cudaDevAttrManagedMemory = 83

Device can allocate managed memory on this system

cudaDevAttrIsMultiGpuBoard = 84

Device is on a multi-GPU board

cudaDevAttrMultiGpuBoardGroupID = 85

Unique identifier for a group of devices on the same multi-GPU board

cudaDevAttrHostNativeAtomicSupported = 86

Link between the device and the host supports native atomic operations

cudaDevAttrSingleToDoublePrecisionPerfRatio = 87

Ratio of single precision performance (in floating-point operations per second) to double precision performance

cudaDevAttrPageableMemoryAccess = 88

Device supports coherently accessing pageable memory without calling cudaHostRegister on it

cudaDevAttrConcurrentManagedAccess = 89

Device can coherently access managed memory concurrently with the CPU

cudaDevAttrComputePreemptionSupported = 90

Device supports Compute Preemption

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

cudaDevAttrCooperativeMultiDeviceLaunch = 96

Deprecated, cudaLaunchCooperativeKernelMultiDevice is deprecated.

cudaDevAttrMaxSharedMemoryPerBlockOptin = 97

The maximum optin shared memory per block. This value may vary by chip. See cudaFuncSetAttribute

cudaDevAttrCanFlushRemoteWrites = 98

Device supports flushing of outstanding remote writes.

cudaDevAttrHostRegisterSupported = 99

Device supports host memory registration via cudaHostRegister.

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.

cudaDevAttrReservedSharedMemoryPerBlock = 111

Shared memory reserved by CUDA driver per block in bytes

cudaDevAttrSparseCudaArraySupported = 112

Device supports sparse CUDA arrays and sparse CUDA mipmapped arrays

cudaDevAttrHostRegisterReadOnlySupported = 113

Device supports using the cudaHostRegister flag cudaHostRegisterReadOnly to register memory that must be mapped as read-only to the GPU

cudaDevAttrTimelineSemaphoreInteropSupported = 114

External timeline semaphore interop is supported on the device

cudaDevAttrMaxTimelineSemaphoreInteropSupported = 114

Deprecated, External timeline semaphore interop is supported on the device

cudaDevAttrMemoryPoolsSupported = 115

Device supports using the cudaMallocAsync and cudaMemPool family of APIs

cudaDevAttrGPUDirectRDMASupported = 116

Device supports GPUDirect RDMA APIs, like nvidia_p2p_get_pages (see 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 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

cudaDevAttrClusterLaunch = 120

Indicates device supports cluster launch

cudaDevAttrDeferredMappingCudaArraySupported = 121

Device supports deferred mapping CUDA arrays and CUDA mipmapped arrays

cudaDevAttrReserved122 = 122

cudaDevAttrReserved123 = 123

cudaDevAttrReserved124 = 124

cudaDevAttrIpcEventSupport = 125

Device supports IPC Events.

cudaDevAttrMemSyncDomainCount = 126

Number of memory synchronization domains the device supports.

cudaDevAttrReserved127 = 127

cudaDevAttrReserved128 = 128

cudaDevAttrReserved129 = 129

cudaDevAttrNumaConfig = 130

NUMA configuration of a device: value is of type cudaDeviceNumaConfig enum

cudaDevAttrNumaId = 131

NUMA node ID of the GPU memory

cudaDevAttrReserved132 = 132

cudaDevAttrMpsEnabled = 133

Contexts created on this device will be shared via MPS

cudaDevAttrHostNumaId = 134

NUMA ID of the host node closest to the device or -1 when system does not support NUMA

cudaDevAttrD3D12CigSupported = 135

Device supports CIG with D3D12.

cudaDevAttrVulkanCigSupported = 138

Device supports CIG with Vulkan.

cudaDevAttrGpuPciDeviceId = 139

The combined 16-bit PCI device ID and 16-bit PCI vendor ID.

cudaDevAttrGpuPciSubsystemId = 140

The combined 16-bit PCI subsystem ID and 16-bit PCI subsystem vendor ID.

cudaDevAttrReserved141 = 141

cudaDevAttrHostNumaMemoryPoolsSupported = 142
Device supports HOST_NUMA location with the cudaMallocAsync and cudaMemPool family of APIs
cudaDevAttrHostNumaMultinodeIpcSupported = 143
Device supports HostNuma location IPC between nodes in a multi-node system.
cudaDevAttrMax

enum cudaDeviceNumaConfig

CUDA device NUMA config

Values

cudaDeviceNumaConfigNone = 0
The GPU is not a NUMA node
cudaDeviceNumaConfigNumaNode
The GPU is a NUMA node, cudaDevAttrNumaId contains its NUMA ID

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 cudaDriverEntryPointQueryResult

Enum for status from obtaining driver entry points, used with cudaApiGetDriverEntryPoint

Values

cudaDriverEntryPointSuccess = 0
Search for symbol found a match
cudaDriverEntryPointSymbolNotFound = 1
Search for symbol was not found
cudaDriverEntryPointVersionNotSufficent = 2
Search for symbol was found but version wasn't great enough

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.

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.

cudaEglColorFormatYUYV422 = 12
Y, U, V in one surface, interleaved as UYVY in one channel.

cudaEglColorFormatUYVY422 = 13
Y, U, V in one surface, interleaved as YUYV in one channel.

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.

cudaEglColorFormatVYUY_ER = 26

Extended Range Y, U, V in one surface, interleaved as YVYU in one channel.

cudaEglColorFormatUYVY_ER = 27

Extended Range Y, U, V in one surface, interleaved as YUYV in one channel.

cudaEglColorFormatYUYV_ER = 28

Extended Range Y, U, V in one surface, interleaved as UYVY in one channel.

cudaEglColorFormatYVYU_ER = 29

Extended Range Y, U, V in one surface, interleaved as VYUY in one channel.

cudaEglColorFormatYUVA_ER = 31

Extended Range Y, U, V, A four channels in one surface, interleaved as AVUY.

cudaEglColorFormatAYUV_ER = 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.
cudaEglColorFormatYVU444Planar_ER = 39
Extended Range Y, V, U in three surfaces, U/V width = Y width, U/V height = Y height.
cudaEglColorFormatYVU422Planar_ER = 40
Extended Range Y, V, U in three surfaces, U/V width = 1/2 Y width, U/V height = Y height.
cudaEglColorFormatYVU420Planar_ER = 41
Extended Range Y, V, U in three surfaces, U/V width = 1/2 Y width, U/V height = 1/2 Y height.
cudaEglColorFormatYVU444SemiPlanar_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.
cudaEglColorFormatBayerRCCB = 73
Bayer format - one channel in one surface with interleaved RCCB ordering.
cudaEglColorFormatBayerCRBC = 74
Bayer format - one channel in one surface with interleaved CRBC ordering.
cudaEglColorFormatBayerCBRC = 75
Bayer format - one channel in one surface with interleaved CBRC ordering.
cudaEglColorFormatBayer10CCCC = 76
Bayer10 format - one channel in one surface with interleaved CCCC ordering. Out of 16 bits, 10 bits used 6 bits No-op.
cudaEglColorFormatBayer12BCCR = 77
Bayer12 format - one channel in one surface with interleaved BCCR ordering. Out of 16 bits, 12 bits used 4 bits No-op.
cudaEglColorFormatBayer12RCCB = 78
Bayer12 format - one channel in one surface with interleaved RCCB ordering. Out of 16 bits, 12 bits used 4 bits No-op.
cudaEglColorFormatBayer12CRBC = 79
Bayer12 format - one channel in one surface with interleaved CRBC ordering. Out of 16 bits, 12 bits used 4 bits No-op.
cudaEglColorFormatBayer12CBRC = 80
Bayer12 format - one channel in one surface with interleaved CBRC ordering. Out of 16 bits, 12 bits used 4 bits No-op.
cudaEglColorFormatBayer12CCCC = 81
Bayer12 format - one channel in one surface with interleaved CCCC ordering. Out of 16 bits, 12 bits used 4 bits No-op.
cudaEglColorFormatY = 82
Color format for single Y plane.
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.
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.

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.

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.

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.

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.

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.

cudaEglColorFormatYVYU = 104

Y, U, V in one surface, interleaved as YVYU in one channel.

cudaEglColorFormatVYUY = 105

Y, U, V in one surface, interleaved as VYUY in one channel.

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.

cudaEglColorFormatUYVY709 = 114

Y, U, V in one surface, interleaved as UYVY in one channel.

cudaEglColorFormatUYVY709_ER = 115

Extended Range Y, U, V in one surface, interleaved as UYVY in one channel.

cudaEglColorFormatUYVY2020 = 116

Y, U, V in one surface, interleaved as UYVY in one channel.

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 or other resources 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 an unsupported data type 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 cudaComputeModeProhibited, cudaComputeModeExclusiveProcess, 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. Note that cudaDeviceSynchronize made from device runtime is only supported on devices of compute capability < 9.0.

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.

cudaErrorUnsupportedDevSideSync = 225

This indicates that the code to be compiled by the PTX JIT contains unsupported call to cudaDeviceSynchronize.

cudaErrorContained = 226

This indicates that an exception occurred on the device that is now contained by the GPU's error containment capability. Common causes are - a. Certain types of invalid accesses of peer GPU memory over nvlink b. Certain classes of hardware errors 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.

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.

cudaErrorLossyQuery = 402

This indicates an attempt was made to introspect an object in a way that would discard semantically important information. This is either due to the object using funtionality newer than the API version used to introspect it or omission of optional return arguments.

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.

cudaErrorTensorMemoryLeak = 721

An exception occurred on the device while exiting a kernel using tensor memory: the tensor memory was not completely deallocated. 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.

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.

cudaErrorMpsRpcFailure = 806

This error indicates that the remote procedural call between the MPS server and the MPS client failed.

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.

cudaErrorMpsMaxClientsReached = 808

This error indicates that the hardware resources required to create MPS client have been exhausted.

cudaErrorMpsMaxConnectionsReached = 809

This error indicates the the hardware resources required to device connections have been exhausted.

cudaErrorMpsClientTerminated = 810

This error indicates that the MPS client has been terminated by the server. To continue using CUDA, the process must be terminated and relaunched.

cudaErrorCdpNotSupported = 811

This error indicates, that the program is using CUDA Dynamic Parallelism, but the current configuration, like MPS, does not support it.

cudaErrorCdpVersionMismatch = 812

This error indicates, that the program contains an unsupported interaction between different versions of CUDA Dynamic Parallelism.

cudaErrorStreamCaptureUnsupported = 900

The operation is not permitted when the stream is capturing.

cudaErrorStreamCaptureInvalidated = 901

The current capture sequence on the stream has been invalidated due to a previous error.

cudaErrorStreamCaptureMerge = 902

The operation would have resulted in a merge of two independent capture sequences.

cudaErrorStreamCaptureUnmatched = 903

The capture was not initiated in this stream.

cudaErrorStreamCaptureUnjoined = 904

The capture sequence contains a fork that was not joined to the primary stream.

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.

cudaErrorStreamCaptureImplicit = 906

The operation would have resulted in a disallowed implicit dependency on a current capture sequence from cudaStreamLegacy.

cudaErrorCapturedEvent = 907

The operation is not permitted on an event which was last recorded in a capturing stream.

cudaErrorStreamCaptureWrongThread = 908

A stream capture sequence not initiated with the cudaStreamCaptureModeRelaxed argument to cudaStreamBeginCapture was passed to cudaStreamEndCapture in a different thread.

cudaErrorTimeout = 909

This indicates that the wait operation has timed out.

cudaErrorGraphExecUpdateFailure = 910

This error indicates that the graph update was not performed because it included changes which violated constraints specific to instantiated graph update.

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.

cudaErrorInvalidClusterSize = 912

This indicates that a kernel launch error has occurred due to cluster misconfiguration.

cudaErrorFunctionNotLoaded = 913

Indiciates a function handle is not loaded when calling an API that requires a loaded function.

cudaErrorInvalidResourceType = 914

This error indicates one or more resources passed in are not valid resource types for the operation.

cudaErrorInvalidResourceConfiguration = 915

This error indicates one or more resources are insufficient or non-applicable for the operation.

cudaErrorUnknown = 999

This indicates that an unknown internal error has occurred.

cudaErrorApiFailureBase = 10000

enum cudaExternalMemoryHandleType

External memory handle types

Values

cudaExternalMemoryHandleTypeOpaqueFd = 1
Handle is an opaque file descriptor
cudaExternalMemoryHandleTypeOpaqueWin32 = 2
Handle is an opaque shared NT handle
cudaExternalMemoryHandleTypeOpaqueWin32Kmt = 3
Handle is an opaque, globally shared handle
cudaExternalMemoryHandleTypeD3D12Heap = 4
Handle is a D3D12 heap object
cudaExternalMemoryHandleTypeD3D12Resource = 5
Handle is a D3D12 committed resource
cudaExternalMemoryHandleTypeD3D11Resource = 6
Handle is a shared NT handle to a D3D11 resource
cudaExternalMemoryHandleTypeD3D11ResourceKmt = 7
Handle is a globally shared handle to a D3D11 resource
cudaExternalMemoryHandleTypeNvSciBuf = 8
Handle is an NvSciBuf object

enum cudaExternalSemaphoreHandleType

External semaphore handle types

Values

cudaExternalSemaphoreHandleTypeOpaqueFd = 1
Handle is an opaque file descriptor
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() to the currently active CUDA device context.

enum cudaFuncAttribute

CUDA function attributes that can be set using cudaFuncSetAttribute

Values

cudaFuncAttributeMaxDynamicSharedMemorySize = 8
Maximum dynamic shared memory size
cudaFuncAttributePreferredSharedMemoryCarveout = 9
Preferred shared memory-L1 cache split

cudaFuncAttributeClusterDimMustBeSet = 10
Indicator to enforce valid cluster dimension specification on kernel launch
cudaFuncAttributeRequiredClusterWidth = 11
Required cluster width
cudaFuncAttributeRequiredClusterHeight = 12
Required cluster height
cudaFuncAttributeRequiredClusterDepth = 13
Required cluster depth
cudaFuncAttributeNonPortableClusterSizeAllowed = 14
Whether non-portable cluster scheduling policy is supported
cudaFuncAttributeClusterSchedulingPolicyPreference = 15
Required cluster scheduling policy preference
cudaFuncAttributeMax

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 For more details see cuGetProcAddress

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 cudaGraphChildGraphNodeOwnership

Child graph node ownership

Values

cudaGraphChildGraphOwnershipClone = 0
: Default behavior for a child graph node. Child graph is cloned into the parent and memory allocation/free nodes can't be present in the child graph.

cudaGraphChildGraphOwnershipMove = 1
: The child graph is moved to the parent. The handle to the child graph is owned by the parent and will be destroyed when the parent is destroyed.The following restrictions apply to child graphs after they have been moved: Cannot be independently instantiated or destroyed; Cannot be added as a child graph of a separate parent graph; Cannot be used as an argument to cudaGraphExecUpdate; Cannot have additional memory allocation or free nodes added.

enum cudaGraphConditionalNodeType

CUDA conditional node types

Values

cudaGraphCondTypeIf = 0
: Conditional 'if/else' Node. Body[0] executed if condition is non-zero. If `size` == 2, an optional ELSE graph is created and this is executed if the condition is zero.

cudaGraphCondTypeWhile = 1
: Conditional 'while' Node. Body executed repeatedly while condition value is non-zero.

cudaGraphCondTypeSwitch = 2
: Conditional 'switch' Node. Body[n] is executed once, where 'n' is the value of the condition. If the condition does not match a body index, no body is launched.

enum cudaGraphDebugDotFlags

CUDA Graph debug write options

Values

cudaGraphDebugDotFlagsVerbose = 1<<0
Output all debug data as if every debug flag is enabled
cudaGraphDebugDotFlagsKernelNodeParams = 1<<2
Adds cudaKernelNodeParams to output
cudaGraphDebugDotFlagsMemcpyNodeParams = 1<<3
Adds cudaMemcpy3DParms to output
cudaGraphDebugDotFlagsMemsetNodeParams = 1<<4
Adds cudaMemsetParams to output
cudaGraphDebugDotFlagsHostNodeParams = 1<<5
Adds cudaHostNodeParams to output
cudaGraphDebugDotFlagsEventNodeParams = 1<<6
Adds cudaEvent_t handle from record and wait nodes to output
cudaGraphDebugDotFlagsExtSemasSignalNodeParams = 1<<7
Adds cudaExternalSemaphoreSignalNodeParams values to output
cudaGraphDebugDotFlagsExtSemasWaitNodeParams = 1<<8
Adds cudaExternalSemaphoreWaitNodeParams to output
cudaGraphDebugDotFlagsKernelNodeAttributes = 1<<9
Adds cudaKernelNodeAttrID values to output
cudaGraphDebugDotFlagsHandles = 1<<10
Adds node handles and every kernel function handle to output
cudaGraphDebugDotFlagsConditionalNodeParams = 1<<15
Adds cudaConditionalNodeParams to output

enum cudaGraphDependencyType

Type annotations that can be applied to graph edges as part of cudaGraphEdgeData.

Values

cudaGraphDependencyTypeDefault = 0
This is an ordinary dependency.
cudaGraphDependencyTypeProgrammatic = 1
This dependency type allows the downstream node to use `cudaGridDependencySynchronize()` . It may only be used between kernel nodes, and must be used with either the cudaGraphKernelNodePortProgrammatic or cudaGraphKernelNodePortLaunchCompletion outgoing port.

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
cudaGraphExecUpdateErrorAttributesChanged = 0x8
The update failed because the node attributes changed in a way that is not supported

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.
cudaGraphInstantiateFlagUpload = 2
Automatically upload the graph after instantiation. Only supported by cudaGraphInstantiateWithParams. The upload will be performed using the stream provided in `instantiateParams`.
cudaGraphInstantiateFlagDeviceLaunch = 4

Instantiate the graph to be launchable from the device. This flag can only be used on platforms which support unified addressing. This flag cannot be used in conjunction with cudaGraphInstantiateFlagAutoFreeOnLaunch.

cudaGraphInstantiateFlagUseNodePriority = 8

Run the graph using the per-node priority attributes rather than the priority of the stream it is launched into.

enum cudaGraphInstantiateResult

Graph instantiation results

Values

cudaGraphInstantiateSuccess = 0

Instantiation succeeded

cudaGraphInstantiateError = 1

Instantiation failed for an unexpected reason which is described in the return value of the function

cudaGraphInstantiateInvalidStructure = 2

Instantiation failed due to invalid structure, such as cycles

cudaGraphInstantiateNodeOperationNotSupported = 3

Instantiation for device launch failed because the graph contained an unsupported operation

cudaGraphInstantiateMultipleDevicesNotSupported = 4

Instantiation for device launch failed due to the nodes belonging to different contexts

cudaGraphInstantiateConditionalHandleUnused = 5

One or more conditional handles are not associated with conditional nodes

enum cudaGraphKernelNodeField

Specifies the field to update when performing multiple node updates from the device

Values

cudaGraphKernelNodeFieldInvalid = 0

Invalid field

cudaGraphKernelNodeFieldGridDim

Grid dimension update

cudaGraphKernelNodeFieldParam

Kernel parameter update

cudaGraphKernelNodeFieldEnabled

Node enable/disable

enum cudaGraphMemAttributeType

Graph memory attributes

Values

cudaGraphMemAttrUsedMemCurrent = 0x0
(value type = cuuint64_t) Amount of memory, in bytes, currently associated with graphs.

cudaGraphMemAttrUsedMemHigh = 0x1
(value type = cuuint64_t) 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 = 0x2
(value type = cuuint64_t) Amount of memory, in bytes, currently allocated for use by the CUDA graphs asynchronous allocator.

cudaGraphMemAttrReservedMemHigh = 0x3
(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

cudaGraphNodeTypeConditional = 0x0d

Conditional nodeMay be used to implement a conditional execution path or loop inside of a graph. The graph(s) contained within the body of the conditional node can be selectively executed or iterated upon based on the value of a conditional variable.Handles must be created in advance of creating the node using cudaGraphConditionalHandleCreate.The following restrictions apply to graphs which contain conditional nodes: 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.To set the control value, supply a default value when creating the handle and/or call cudaGraphSetConditional from device code.

cudaGraphNodeTypeCount

enum cudaJit_CacheMode

Caching modes for dlcm

Values

cudaJitCacheOptionNone = 0

Compile with no -dlcm flag specified

cudaJitCacheOptionCG

Compile with L1 cache disabled

cudaJitCacheOptionCA

Compile with L1 cache enabled

enum cudaJit_Fallback

Cubin matching fallback strategies

Values

cudaPreferPtx = 0

Prefer to compile ptx if exact binary match not found

cudaPreferBinary

Prefer to fall back to compatible binary code if exact match not found

enum cudaJitOption

Online compiler and linker options

Values

cudaJitMaxRegisters = 0

Max number of registers that a thread may use. Option type: unsigned int Applies to: compiler only

cudaJitThreadsPerBlock = 1

IN: Specifies minimum number of threads per block to target compilation for OUT: Returns the number of threads the compiler actually targeted. This restricts the resource utilization of the compiler (e.g. max registers) such that a block with the given number of threads should be able to launch based on register limitations. Note, this option does not currently take into account any other resource limitations, such as shared memory utilization. Option type: unsigned int Applies to: compiler only

cudaJitWallTime = 2

Overwrites the option value with the total wall clock time, in milliseconds, spent in the compiler and linker Option type: float Applies to: compiler and linker

cudaJitInfoLogBuffer = 3

Pointer to a buffer in which to print any log messages that are informational in nature (the buffer size is specified via option cudaJitInfoLogBufferSizeBytes) Option type: char * Applies to: compiler and linker

cudaJitInfoLogBufferSizeBytes = 4

IN: Log buffer size in bytes. Log messages will be capped at this size (including null terminator) OUT: Amount of log buffer filled with messages Option type: unsigned int Applies to: compiler and linker

cudaJitErrorLogBuffer = 5

Pointer to a buffer in which to print any log messages that reflect errors (the buffer size is specified via option cudaJitErrorLogBufferSizeBytes) Option type: char * Applies to: compiler and linker

cudaJitErrorLogBufferSizeBytes = 6

IN: Log buffer size in bytes. Log messages will be capped at this size (including null terminator) OUT: Amount of log buffer filled with messages Option type: unsigned int Applies to: compiler and linker

cudaJitOptimizationLevel = 7

Level of optimizations to apply to generated code (0 - 4), with 4 being the default and highest level of optimizations. Option type: unsigned int Applies to: compiler only

cudaJitFallbackStrategy = 10

Specifies choice of fallback strategy if matching cubin is not found. Choice is based on supplied cudaJit_Fallback. Option type: unsigned int for enumerated type cudaJit_Fallback Applies to: compiler only

cudaJitGenerateDebugInfo = 11

Specifies whether to create debug information in output (-g) (0: false, default) Option type: int Applies to: compiler and linker

cudaJitLogVerbose = 12

Generate verbose log messages (0: false, default) Option type: int Applies to: compiler and linker

cudaJitGenerateLineInfo = 13

Generate line number information (-lineinfo) (0: false, default) Option type: int Applies to: compiler only

cudaJitCacheMode = 14

Specifies whether to enable caching explicitly (-dlcm) Choice is based on supplied cudaJit_CacheMode. Option type: unsigned int for enumerated type cudaJit_CacheMode Applies to: compiler only

cudaJitPositionIndependentCode = 30

Generate position independent code (0: false) Option type: int Applies to: compiler only

cudaJitMinCtaPerSm = 31

This option hints to the JIT compiler the minimum number of CTAs from the kernel's grid to be mapped to a SM. This option is ignored when used together with cudaJitMaxRegisters or cudaJitThreadsPerBlock. Optimizations based on this option need cudaJitMaxThreadsPerBlock to be specified as well. For kernels already using PTX directive .minnctapersm, this option will be ignored by default. Use cudaJitOverrideDirectiveValues to let this option take precedence over the PTX directive. Option type: unsigned int Applies to: compiler only

cudaJitMaxThreadsPerBlock = 32

Maximum number threads in a thread block, computed as the product of the maximum extent specifed for each dimension of the block. This limit is guaranteed not to be exeeded in any invocation of the kernel. Exceeding the the maximum number of threads results in runtime error or kernel launch failure. For kernels already using PTX directive .maxntid, this option will be ignored by default. Use cudaJitOverrideDirectiveValues to let this option take precedence over the PTX directive. Option type: int Applies to: compiler only

cudaJitOverrideDirectiveValues = 33

This option lets the values specified using cudaJitMaxRegisters, cudaJitThreadsPerBlock, cudaJitMaxThreadsPerBlock and cudaJitMinCtaPerSm take precedence over any PTX directives. (0: Disable, default; 1: Enable) Option type: int Applies to: compiler only

enum cudaLaunchAttributeID

Launch attributes enum; used as id field of cudaLaunchAttribute

Values

cudaLaunchAttributeIgnore = 0

Ignored entry, for convenient composition

cudaLaunchAttributeAccessPolicyWindow = 1

Valid for streams, graph nodes, launches. See cudaLaunchAttributeValue::accessPolicyWindow.

cudaLaunchAttributeCooperative = 2

Valid for graph nodes, launches. See cudaLaunchAttributeValue::cooperative.

cudaLaunchAttributeSynchronizationPolicy = 3

Valid for streams. See cudaLaunchAttributeValue::syncPolicy.

cudaLaunchAttributeClusterDimension = 4

Valid for graph nodes, launches. See cudaLaunchAttributeValue::clusterDim.

cudaLaunchAttributeClusterSchedulingPolicyPreference = 5

Valid for graph nodes, launches. See cudaLaunchAttributeValue::clusterSchedulingPolicyPreference.

cudaLaunchAttributeProgrammaticStreamSerialization = 6

Valid for launches. Setting cudaLaunchAttributeValue::programmaticStreamSerializationAllowed to non-0 signals that the kernel will use programmatic means to resolve its stream dependency, so that the CUDA runtime should opportunistically allow the grid's execution to overlap with the previous kernel in the stream, if that kernel requests the overlap. The dependent launches can choose to wait on the dependency using the programmatic sync (cudaGridDependencySynchronize() or equivalent PTX instructions).

cudaLaunchAttributeProgrammaticEvent = 7

Valid for launches. Set cudaLaunchAttributeValue::programmaticEvent to record the event. Event recorded through this launch attribute is guaranteed to only trigger after all block in the associated kernel trigger the event. A block can trigger the event programmatically in a future CUDA release. A trigger can also be inserted at the beginning of each block's execution if triggerAtBlockStart is set to non-0. The dependent launches can choose to wait on the dependency using the programmatic sync (cudaGridDependencySynchronize() or equivalent PTX instructions). Note that dependents (including the CPU thread calling cudaEventSynchronize()) are not guaranteed to observe the release precisely when it is released. For example, cudaEventSynchronize() may only observe the event trigger long after the associated kernel has completed. This recording type is primarily meant for establishing programmatic dependency between device tasks. Note also this type of dependency allows, but does not guarantee, concurrent execution of tasks. The event supplied must not be an interprocess or interop event. The event must disable timing (i.e. must be created with the cudaEventDisableTiming flag set).

cudaLaunchAttributePriority = 8

Valid for streams, graph nodes, launches. See cudaLaunchAttributeValue::priority.

cudaLaunchAttributeMemSyncDomainMap = 9

Valid for streams, graph nodes, launches. See cudaLaunchAttributeValue::memSyncDomainMap.

cudaLaunchAttributeMemSyncDomain = 10

Valid for streams, graph nodes, launches. See cudaLaunchAttributeValue::memSyncDomain.

cudaLaunchAttributePreferredClusterDimension = 11

Valid for graph nodes and launches. Set cudaLaunchAttributeValue::preferredClusterDim to allow the kernel launch to specify a preferred substitute cluster dimension. Blocks may be grouped according to either the dimensions specified with this attribute (grouped into a "preferred substitute cluster"), or the one specified with cudaLaunchAttributeClusterDimension attribute (grouped into a "regular cluster"). The cluster dimensions of a "preferred substitute cluster" shall be an integer multiple greater than zero of the regular cluster dimensions. The device will attempt - on a best-effort basis - to group thread blocks into preferred clusters over grouping them into regular clusters. When it deems necessary (primarily when the device temporarily runs out of physical resources to launch the larger preferred clusters), the device may switch to launch the regular clusters instead to attempt to utilize as much of the physical device resources as possible. Each type of cluster will have its enumeration / coordinate setup as if the grid consists solely of its type of cluster. For example, if the preferred substitute cluster dimensions double the regular cluster dimensions, there might be simultaneously a regular cluster indexed at (1,0,0), and a preferred cluster indexed at (1,0,0). In this example, the preferred substitute cluster (1,0,0) replaces regular clusters (2,0,0) and (3,0,0) and groups their blocks. This attribute will only take effect when a regular cluster dimension has been specified. The preferred substitute cluster dimension must be an integer multiple greater than zero of the regular cluster dimension and must divide the grid. It must also be no more than `maxBlocksPerCluster`, if it is set in the kernel's `__launch_bounds__`. Otherwise it must be less than the maximum value the driver can support. Otherwise, setting this attribute to a value physically unable to fit on any particular device is permitted.

cudaLaunchAttributeLaunchCompletionEvent = 12

Valid for launches. Set cudaLaunchAttributeValue::launchCompletionEvent to record the event. Nominally, the event is triggered once all blocks of the kernel have begun execution. Currently this is a best effort. If a kernel B has a launch completion dependency on a kernel A, B may wait until A is complete. Alternatively, blocks of B may begin before all blocks of A have begun, for example if B can claim execution resources unavailable to A (e.g. they run on different GPUs) or if B is a higher priority than A. Exercise caution if such an ordering inversion could lead to deadlock. A launch completion event is nominally similar to a programmatic event with `triggerAtBlockStart` set except that it is not visible to `cudaGridDependencySynchronize()` and can be used with compute capability less than 9.0. The event supplied must not be an interprocess or interop event. The event must disable timing (i.e. must be created with the cudaEventDisableTiming flag set).

cudaLaunchAttributeDeviceUpdatableKernelNode = 13

Valid for graph nodes, launches. This attribute is graphs-only, and passing it to a launch in a non-capturing stream will result in an error. :cudaLaunchAttributeValue::deviceUpdatableKernelNode::deviceUpdatable can only be set to 0 or 1. Setting the field to 1 indicates that the corresponding kernel node should be device-updatable. On success, a handle will be returned via cudaLaunchAttributeValue::deviceUpdatableKernelNode::devNode which can be passed to the various device-side update functions to update the node's kernel parameters from within another kernel. For more information on the types of device updates that can be made, as well as the relevant limitations thereof, see cudaGraphKernelNodeUpdatesApply. Nodes which are device-updatable have additional restrictions compared to regular kernel nodes. Firstly, device-updatable nodes cannot be removed from their graph via cudaGraphDestroyNode. Additionally, once opted-in to this functionality, a node cannot opt out, and any attempt to set the deviceUpdatable attribute to 0 will result in an error. Device-updatable kernel nodes also cannot have their attributes copied to/from another kernel node via cudaGraphKernelNodeCopyAttributes. Graphs containing one or more device-updatable nodes also do not allow multiple instantiation, and neither the graph nor its instantiated version can be passed to cudaGraphExecUpdate. If a graph contains device-updatable nodes and updates those nodes from the device from within the graph, the graph must be uploaded with cuGraphUpload before it is launched. For such a graph, if host-side executable graph updates are made to the device-updatable nodes, the graph must be uploaded before it is launched again.

cudaLaunchAttributePreferredSharedMemoryCarveout = 14

Valid for launches. On devices where the L1 cache and shared memory use the same hardware resources, setting cudaLaunchAttributeValue::sharedMemCarveout to a percentage between 0-100 signals sets the shared memory carveout preference in percent of the total shared memory for that kernel launch. This attribute takes precedence over cudaFuncAttributePreferredSharedMemoryCarveout. This is only a hint, and the driver can choose a different configuration if required for the launch.

enum cudaLaunchMemSyncDomain

Memory Synchronization Domain

A kernel can be launched in a specified memory synchronization domain that affects all memory operations issued by that kernel. A memory barrier issued in one domain will only order memory operations in that domain, thus eliminating latency increase from memory barriers ordering unrelated traffic.

By default, kernels are launched in domain 0. Kernel launched with cudaLaunchMemSyncDomainRemote will have a different domain ID. User may also alter the domain ID with cudaLaunchMemSyncDomainMap for a specific stream / graph node / kernel launch. See cudaLaunchAttributeMemSyncDomain, cudaStreamSetAttribute, cudaLaunchKernelEx, cudaGraphKernelNodeSetAttribute.

Memory operations done in kernels launched in different domains are considered system-scope distanced. In other words, a GPU scoped memory synchronization is not sufficient for memory order to be observed by kernels in another memory synchronization domain even if they are on the same GPU.

Values

cudaLaunchMemSyncDomainDefault = 0
Launch kernels in the default domain
cudaLaunchMemSyncDomainRemote = 1
Launch kernels in the remote domain

enum cudaLibraryOption

Library options to be specified with cudaLibraryLoadData() or cudaLibraryLoadFromFile()

Values

cudaLibraryHostUniversalFunctionAndDataTable = 0
cudaLibraryBinaryIsPreserved = 1
Specifes that the argument `code` passed to cudaLibraryLoadData() will be preserved. Specifying this option will let the driver know that `code` can be accessed at any point until cudaLibraryUnload(). The default behavior is for the driver to allocate and maintain its own copy of `code`. Note that this is only a memory usage optimization hint and the driver can choose to ignore it if required. Specifying this option with cudaLibraryLoadFromFile() is invalid and will return cudaErrorInvalidValue.

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)
cudaMemHandleTypeFabric = 0x8
Allows a fabric handle to be used for exporting. (cudaMemFabricHandle_t)

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 cudaMemcpy3DOperandType

These flags allow applications to convey the operand type for individual copies specified in cudaMemcpy3DBatchAsync.

Values

cudaMemcpyOperandTypePointer = 0x1
Memcpy operand is a valid pointer.
cudaMemcpyOperandTypeArray = 0x2
Memcpy operand is a CUarray.
cudaMemcpyOperandTypeMax = 0x7FFFFFFF

enum cudaMemcpyFlags

Flags to specify for copies within a batch. For more details see cudaMemcpyBatchAsync.

Values

cudaMemcpyFlagDefault = 0x0
cudaMemcpyFlagPreferOverlapWithCompute = 0x1
Hint to the driver to try and overlap the copy with compute work on the SMs.

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
cudaMemLocationTypeHost = 2
Location is host, id is ignored
cudaMemLocationTypeHostNuma = 3
Location is a host NUMA node, thus id is a host NUMA node id
cudaMemLocationTypeHostNumaCurrent = 4
Location is the host NUMA node closest to the current thread's CPU, id is ignored

enum cudaMemoryAdvise

CUDA Memory Advise values

Values

cudaMemAdviseSetReadMostly = 1
Data will mostly be read and only occassionally 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 occassionally 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
cudaMemRangeAttributePreferredLocationType = 5
The preferred location type of the range
cudaMemRangeAttributePreferredLocationId = 6
The preferred location id of the range
cudaMemRangeAttributeLastPrefetchLocationType = 7
The last location type to which the range was prefetched
cudaMemRangeAttributeLastPrefetchLocationId = 8
The last location id to which the range was prefetched

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

Deprecated

CUDA shared memory configuration

Values

cudaSharedMemBankSizeDefault = 0
cudaSharedMemBankSizeFourByte = 1

cudaSharedMemBankSizeEightByte = 2

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
cudaFilterModeLinear = 1
Linear filter mode

enum cudaTextureReadMode

CUDA texture read modes

Values

cudaReadModeElementType = 0

Read texture as specified element type

cudaReadModeNormalizedFloat = 1

Read texture as normalized float

enum cudaUserObjectFlags

Flags for user objects for graphs

Values

cudaUserObjectNoDestructorSync = 0x1

Indicates the destructor execution is not synchronized by any CUDA handle.

enum cudaUserObjectRetainFlags

Flags for retaining user object references for graphs

Values

cudaGraphUserObjectMove = 0x1

Transfer references from the caller rather than creating new references.

typedef cudaArray *cudaArray_const_t

CUDA array (as source copy argument)

typedef cudaArray *cudaArray_t

CUDA array

typedef cudaAsyncCallbackEntry *cudaAsyncCallbackHandle_t

CUDA async callback handle

typedef struct CUeglStreamConnection_st *cudaEglStreamConnection

CUDA EGLSream 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 unsigned long long cudaGraphConditionalHandle

CUDA handle for conditional graph nodes

typedef struct CUgraphDeviceUpdatableNode_st *cudaGraphDeviceNode_t

CUDA device node handle for device-side node update

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 CUkern_st *cudaKernel_t

CUDA kernel

typedef struct CUlib_st *cudaLibrary_t

CUDA library

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 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 cudaExternalMemoryGetMappedMipmappedArray if the mipmapped array is used as a color target in a graphics API

#define cudaArrayCubemap 0x04

Must be set in cudaMalloc3DArray to create a cubemap CUDA array

#define cudaArrayDefault 0x00

Default CUDA array allocation flag

#define cudaArrayDeferredMapping 0x80

Must be set in cudaMallocArray, cudaMalloc3DArray or cudaMallocMipmappedArray in order to create a deferred mapping CUDA array or CUDA mipmapped array

#define cudaArrayLayered 0x01

Must be set in cudaMalloc3DArray to create a layered CUDA array

#define cudaArraySparse 0x40

Must be set in cudaMallocArray, cudaMalloc3DArray or cudaMallocMipmappedArray 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 cudaMallocArray or cudaMalloc3DArray in order to bind surfaces to the CUDA array

#define cudaArrayTextureGather 0x08

Must be set in cudaMallocArray or cudaMalloc3DArray 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 0xff

Device flags mask

#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 cudaDeviceSyncMemops 0x80

Device flag - Ensure synchronous memory operations on this context will synchronize

#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 cudaGraphKernelNodePortDefault 0

This port activates when the kernel has finished executing.

#define cudaGraphKernelNodePortLaunchCompletion 2

This port activates when all blocks of the kernel have begun execution. See also cudaLaunchAttributeLaunchCompletionEvent.

#define cudaGraphKernelNodePortProgrammatic 1

This port activates when all blocks of the kernel have performed cudaTriggerProgrammaticLaunchCompletion() or have terminated. It must be used with edge type cudaGraphDependencyTypeProgrammatic. See also cudaLaunchAttributeProgrammaticEvent.

#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 cudaInitDeviceFlagsAreValid 0x01

Tell the CUDA runtime that DeviceFlags is being set in cudaInitDevice call

#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 cudaMemPoolCreateUsageHwDecompress 0x2

This flag, if set, indicates that the memory will be used as a buffer for hardware accelerated decompression.

#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.

6.1. 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 7. Data Structures

Here are the data structures with brief descriptions:

__cudaOccupancyB2DHelper
cudaAccessPolicyWindow
cudaArrayMemoryRequirements
cudaArraySparseProperties
cudaAsyncNotificationInfo_t
cudaChannelFormatDesc
cudaChildGraphNodeParams
cudaConditionalNodeParams
cudaDeviceProp
cudaEglFrame
cudaEglPlaneDesc
cudaEventRecordNodeParams
cudaEventWaitNodeParams
cudaExtent
cudaExternalMemoryBufferDesc
cudaExternalMemoryHandleDesc
cudaExternalMemoryMipmappedArrayDesc
cudaExternalSemaphoreHandleDesc
cudaExternalSemaphoreSignalNodeParams
cudaExternalSemaphoreSignalNodeParamsV2
cudaExternalSemaphoreSignalParams
cudaExternalSemaphoreSignalParams_v1
cudaExternalSemaphoreWaitNodeParams
cudaExternalSemaphoreWaitNodeParamsV2
cudaExternalSemaphoreWaitParams
cudaExternalSemaphoreWaitParams_v1
cudaFuncAttributes
cudaGraphEdgeData
cudaGraphExecUpdateResultInfo
cudaGraphInstantiateParams
cudaGraphKernelNodeUpdate
cudaGraphNodeParams

cudaHostNodeParams
cudaHostNodeParamsV2
cudaIpcEventHandle_t
cudaIpcMemHandle_t
cudaKernelNodeParams
cudaKernelNodeParamsV2
cudaLaunchAttribute
cudaLaunchAttributeValue
cudaLaunchConfig_t
cudaLaunchMemSyncDomainMap
cudaLaunchParams
cudaMemAccessDesc
cudaMemAllocNodeParams
cudaMemAllocNodeParamsV2
cudaMemcpy3DOperand
cudaMemcpy3DParms
cudaMemcpy3DPeerParms
cudaMemcpyAttributes
cudaMemcpyNodeParams
cudaMemFreeNodeParams
cudaMemLocation
cudaMemPoolProps
cudaMemPoolPtrExportData
cudaMemsetParams
cudaMemsetParamsV2
cudaOffset3D
cudaPitchedPtr
cudaPointerAttributes
cudaPos
cudaResourceDesc
cudaResourceViewDesc
cudaTextureDesc
cudaUUID_t

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.

enumcudaAccessProperty cudaAccessPolicyWindow::hitProp

CUaccessProperty set for hit.

float cudaAccessPolicyWindow::hitRatio

hitRatio specifies percentage of lines assigned hitProp, rest are assigned missProp.

enumcudaAccessProperty 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. cudaArrayMemoryRequirements Struct Reference

CUDA array and CUDA mipmapped array memory requirements

size_t cudaArrayMemoryRequirements::alignment

Alignment necessary for mapping the array.

size_t cudaArrayMemoryRequirements::size

Total size of the array.

7.4. 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

7.5. cudaAsyncNotificationInfo_t Struct Reference

Information describing an async notification event

unsigned long long cudaAsyncNotificationInfo_t::bytesOverBudget

The number of bytes that the process has allocated above its device memory budget

cudaAsyncNotificationInfo_t::@39 cudaAsyncNotificationInfo_t::info

Information about the notification. `type` must be checked in order to interpret this field.

cudaAsyncNotificationInfo_t::@39::@40 cudaAsyncNotificationInfo_t::overBudget

Information about notifications of type `cudaAsyncNotificationTypeOverBudget`

cudaAsyncNotificationType cudaAsyncNotificationInfo_t::type

The type of notification being sent

7.6. cudaChannelFormatDesc Struct Reference

CUDA Channel format descriptor

enumcudaChannelFormatKind cudaChannelFormatDesc::f

Channel format kind

int cudaChannelFormatDesc::w

w

int cudaChannelFormatDesc::x

x

int cudaChannelFormatDesc::y

y

int cudaChannelFormatDesc::z

z

7.7. cudaChildGraphNodeParams Struct Reference

Child graph node parameters

cudaGraph_t cudaChildGraphNodeParams::graph

The child graph to clone into the node for node creation, or a handle to the graph owned by the node for node query. The graph must not contain conditional nodes. Graphs containing memory allocation or memory free nodes must set the ownership to be moved to the parent.

enumcudaGraphChildGraphNodeOwnership cudaChildGraphNodeParams::ownership

The ownership relationship of the child graph node.

7.8. cudaConditionalNodeParams Struct Reference

CUDA conditional node parameters

cudaGraphConditionalHandle cudaConditionalNodeParams::handle

Conditional node handle. Handles must be created in advance of creating the node using cudaGraphConditionalHandleCreate.

cudaGraph_t *cudaConditionalNodeParams::phGraph_out

CUDA-owned array populated with conditional node child graphs during creation of the node. Valid for the lifetime of the conditional node. The contents of the graph(s) are subject to the following constraints:

- Allowed node types are kernel nodes, empty nodes, child graphs, memsets, memcopies, and conditionals. This applies recursively to child graphs and conditional bodies.
- All kernels, including kernels in nested conditionals or child graphs at any level, must belong to the same CUDA context.

These graphs may be populated using graph node creation APIs or cudaStreamBeginCaptureToGraph. cudaGraphCondTypeIf: phGraph_out[0] is executed when the condition is non-zero. If `size` == 2, phGraph_out[1] will be executed when the condition is zero. cudaGraphCondTypeWhile: phGraph_out[0] is executed as long as the condition is non-zero. cudaGraphCondTypeSwitch: phGraph_out[n] is executed when the condition is equal to n. If the condition >= `size`, no body graph is executed.

unsigned int cudaConditionalNodeParams::size

Size of graph output array. Allowed values are 1 for cudaGraphCondTypeWhile, 1 or 2 for cudaGraphCondTypeWhile, or any value greater than zero for cudaGraphCondTypeSwitch.

enumcudaGraphConditionalNodeType cudaConditionalNodeParams::type

Type of conditional node.

7.9. cudaDeviceProp Struct Reference

CUDA device properties

int cudaDeviceProp::accessPolicyMaxWindowSize

The maximum value of cudaAccessPolicyWindow::num_bytes.

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

Deprecated, Clock frequency in kilohertz

int cudaDeviceProp::clusterLaunch

Indicates device supports cluster launch

int cudaDeviceProp::computeMode

Deprecated, 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::deferredMappingCudaArraySupported

1 if the device supports deferred mapping CUDA arrays and CUDA mipmapped arrays

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

unsigned int cudaDeviceProp::gpuDirectRDMAFlushWritesOptions

Bitmask to be interpreted according to the cudaFlushGPUDirectRDMAWritesOptions enum

int cudaDeviceProp::gpuDirectRDMASupported

1 if the device supports GPUDirect RDMA APIs, 0 otherwise

int cudaDeviceProp::gpuDirectRDMAWritesOrdering

See the cudaGPUDirectRDMAWritesOrdering enum for numerical values

int cudaDeviceProp::hostNativeAtomicSupported

Link between the device and the host supports native atomic operations

int cudaDeviceProp::hostRegisterReadOnlySupported

Device supports using the cudaHostRegister flag cudaHostRegisterReadOnly to register memory that must be mapped as read-only to the GPU

int cudaDeviceProp::hostRegisterSupported

Device supports host memory registration via cudaHostRegister.

int cudaDeviceProp::integrated

Device is integrated as opposed to discrete

int cudaDeviceProp::ipcEventSupported

Device supports IPC Events.

int cudaDeviceProp::isMultiGpuBoard

Device is on a multi-GPU board

int cudaDeviceProp::kernelExecTimeoutEnabled

Deprecated, 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

Deprecated, Peak memory clock frequency in kilohertz

int cudaDeviceProp::memoryPoolsSupported

1 if the device supports using the cudaMallocAsync and cudaMemPool family of APIs, 0 otherwise

unsigned int cudaDeviceProp::memoryPoolSupportedHandleTypes

Bitmask of handle types supported with mempool-based IPC

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

int cudaDeviceProp::reserved

Reserved for future use

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

Deprecated, Ratio of single precision performance (in floating-point operations per second) to double precision performance

int cudaDeviceProp::sparseCudaArraySupported

1 if the device supports sparse CUDA arrays and sparse CUDA mipmapped arrays, 0 otherwise

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

int cudaDeviceProp::timelineSemaphoreInteropSupported

External timeline semaphore interop is supported on the device

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

int cudaDeviceProp::unifiedFunctionPointers

Indicates device supports unified pointers

cudaUUID_t cudaDeviceProp::uuid

16-byte unique identifier

int cudaDeviceProp::warpSize

Warp size in threads

7.10. 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:

```
typedef struct cudaEglPlaneDesc_st {
          unsigned int width;
          unsigned int height;
          unsigned int depth;
          unsigned int pitch;
          unsigned int numChannels;
          struct cudaChannelFormatDesc channelDesc;
          unsigned int reserved[4];
      } 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 cudaEglPlaneDesc

struct cudaPitchedPtr cudaEglFrame::pPitch

Array of Pointers corresponding to each plane

7.11. 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

7.12. cudaEventRecordNodeParams Struct Reference

Event record node parameters

cudaEvent_t cudaEventRecordNodeParams::event

The event to record when the node executes

7.13. cudaEventWaitNodeParams Struct Reference

Event wait node parameters

cudaEvent_t cudaEventWaitNodeParams::event

The event to wait on from the node

7.14. 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

7.15. 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

7.16. 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

enumcudaExternalMemoryHandleType cudaExternalMemoryHandleDesc::type

Type of the handle

cudaExternalMemoryHandleDesc::@10::@11 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.

7.17. 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.

7.18. 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

enumcudaExternalSemaphoreHandleType cudaExternalSemaphoreHandleDesc::type

Type of the handle

cudaExternalSemaphoreHandleDesc::@12::@13 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.

7.19. 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.

cudaExternalSemaphoreSignalParams *cudaExternalSemaphoreSignalNodeParams::paramsArray

Array of external semaphore signal parameters.

7.20. cudaExternalSemaphoreSignalNodeParamsV2 Struct Reference

External semaphore signal node parameters

cudaExternalSemaphore_t *cudaExternalSemaphoreSignalNodeParamsV2::extSemArray

Array of external semaphore handles.

unsigned int cudaExternalSemaphoreSignalNodeParamsV2::numExtSems

Number of handles and parameters supplied in extSemArray and paramsArray.

cudaExternalSemaphoreSignalParams *cudaExternalSemaphoreSignalNodeParamsV2::paramsArray

Array of external semaphore signal parameters.

7.21. cudaExternalSemaphoreSignalParams Struct Reference

External semaphore signal parameters, compatible with driver type

void *cudaExternalSemaphoreSignalParams::fence

Pointer to NvSciSyncFence. Valid if cudaExternalSemaphoreHandleType is of type cudaExternalSemaphoreHandleTypeNvSciSync.

cudaExternalSemaphoreSignalParams::@22::@23 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::@22::@25 cudaExternalSemaphoreSignalParams::keyedMutex

Parameters for keyed mutex objects

unsigned long long cudaExternalSemaphoreSignalParams::value

Value of fence to be signaled

7.22. cudaExternalSemaphoreSignalParams_v1 Struct Reference

External semaphore signal parameters(deprecated)

void *cudaExternalSemaphoreSignalParams_v1::fence

Pointer to NvSciSyncFence. Valid if cudaExternalSemaphoreHandleType is of type cudaExternalSemaphoreHandleTypeNvSciSync.

cudaExternalSemaphoreSignalParams_v1::@14::@15 cudaExternalSemaphoreSignalParams_v1::fence

Parameters for fence objects

unsigned int cudaExternalSemaphoreSignalParams_v1::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_v1::@14::@17 cudaExternalSemaphoreSignalParams_v1::keyedMutex

Parameters for keyed mutex objects

unsigned long long cudaExternalSemaphoreSignalParams_v1::value

Value of fence to be signaled

7.23. 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.

7.24. cudaExternalSemaphoreWaitNodeParamsV2 Struct Reference

External semaphore wait node parameters

cudaExternalSemaphore_t *cudaExternalSemaphoreWaitNodeParamsV2::extSemArray

Array of external semaphore handles.

unsigned int cudaExternalSemaphoreWaitNodeParamsV2::numExtSems

Number of handles and parameters supplied in extSemArray and paramsArray.

cudaExternalSemaphoreWaitParams *cudaExternalSemaphoreWaitNodeParamsV2::paramsArray

Array of external semaphore wait parameters.

7.25. 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::@26::@27 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::@26::@29 cudaExternalSemaphoreWaitParams::keyedMutex

Parameters for keyed mutex objects

unsigned int cudaExternalSemaphoreWaitParams::timeoutMs

Timeout in milliseconds to wait to acquire the mutex

unsigned long long cudaExternalSemaphoreWaitParams::value

Value of fence to be waited on

7.26. cudaExternalSemaphoreWaitParams_v1 Struct Reference

External semaphore wait parameters(deprecated)

void *cudaExternalSemaphoreWaitParams_v1::fence

Pointer to NvSciSyncFence. Valid if cudaExternalSemaphoreHandleType is of type cudaExternalSemaphoreHandleTypeNvSciSync.

cudaExternalSemaphoreWaitParams_v1::@18::@19 cudaExternalSemaphoreWaitParams_v1::fence

Parameters for fence objects

unsigned int cudaExternalSemaphoreWaitParams_v1::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_v1::key

Value of key to acquire the mutex with

cudaExternalSemaphoreWaitParams_v1::@18::@21 cudaExternalSemaphoreWaitParams_v1::keyedMutex

Parameters for keyed mutex objects

unsigned int cudaExternalSemaphoreWaitParams_v1::timeoutMs

Timeout in milliseconds to wait to acquire the mutex

unsigned long long cudaExternalSemaphoreWaitParams_v1::value

Value of fence to be waited on

7.27. 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.

int cudaFuncAttributes::clusterDimMustBeSet

If this attribute is set, the kernel must launch with a valid cluster dimension specified.

int cudaFuncAttributes::clusterSchedulingPolicyPreference

The block scheduling policy of a function. See cudaFuncSetAttribute

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::nonPortableClusterSizeAllowed

Whether the function can be launched with non-portable cluster size. 1 is allowed, 0 is disallowed. A non-portable cluster size may only function on the specific SKUs the program is tested on. The launch might fail if the program is run on a different hardware platform.

CUDA API provides cudaOccupancyMaxActiveClusters to assist with checking whether the desired size can be launched on the current device.

Portable Cluster Size

A portable cluster size is guaranteed to be functional on all compute capabilities higher than the target compute capability. The portable cluster size for sm_90 is 8 blocks per cluster. This value may increase for future compute capabilities.

The specific hardware unit may support higher cluster sizes that's not guaranteed to be portable. See cudaFuncSetAttribute

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.

int cudaFuncAttributes::requiredClusterWidth

The required cluster width/height/depth in blocks. The values must either all be 0 or all be positive. The validity of the cluster dimensions is otherwise checked at launch time.

If the value is set during compile time, it cannot be set at runtime. Setting it at runtime should return cudaErrorNotPermitted. See cudaFuncSetAttribute

int cudaFuncAttributes::reserved

Reserved for future use.

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.

7.28. cudaGraphEdgeData Struct Reference

Optional annotation for edges in a CUDA graph. Note, all edges implicitly have annotations and default to a zero-initialized value if not specified. A zero-initialized struct indicates a standard full serialization of two nodes with memory visibility.

unsigned char cudaGraphEdgeData::from_port

This indicates when the dependency is triggered from the upstream node on the edge. The meaning is specfic to the node type. A value of 0 in all cases means full completion of the upstream node, with memory visibility to the downstream node or portion thereof (indicated by `to_port`). Only kernel nodes define non-zero ports. A kernel node can use the following output port types: cudaGraphKernelNodePortDefault, cudaGraphKernelNodePortProgrammatic, or cudaGraphKernelNodePortLaunchCompletion.

unsigned char cudaGraphEdgeData::reserved

These bytes are unused and must be zeroed. This ensures compatibility if additional fields are added in the future.

unsigned char cudaGraphEdgeData::to_port

This indicates what portion of the downstream node is dependent on the upstream node or portion thereof (indicated by `from_port`). The meaning is specific to the node type. A value of 0 in all cases means the entirety of the downstream node is dependent on the upstream work. Currently no node types define non-zero ports. Accordingly, this field must be set to zero.

unsigned char cudaGraphEdgeData::type

This should be populated with a value from cudaGraphDependencyType. (It is typed as char due to compiler-specific layout of bitfields.) See cudaGraphDependencyType.

7.29. cudaGraphExecUpdateResultInfo Struct Reference

Result information returned by cudaGraphExecUpdate

cudaGraphNode_t cudaGraphExecUpdateResultInfo::errorFromNode

The from node of error edge when the topologies do not match. Otherwise NULL.

cudaGraphNode_t cudaGraphExecUpdateResultInfo::errorNode

The "to node" of the error edge when the topologies do not match. The error node when the error is associated with a specific node. NULL when the error is generic.

enumcudaGraphExecUpdateResult cudaGraphExecUpdateResultInfo::result

Gives more specific detail when a cuda graph update fails.

7.30. cudaGraphInstantiateParams Struct Reference

Graph instantiation parameters

cudaGraphNode_t cudaGraphInstantiateParams::errNode_out

The node which caused instantiation to fail, if any

unsigned long long cudaGraphInstantiateParams::flags

Instantiation flags

cudaGraphInstantiateResult cudaGraphInstantiateParams::result_out

Whether instantiation was successful. If it failed, the reason why

cudaStream_t cudaGraphInstantiateParams::uploadStream

Upload stream

7.31. cudaGraphKernelNodeUpdate Struct Reference

Struct to specify a single node update to pass as part of a larger array to cudaGraphKernelNodeUpdatesApply

enumcudaGraphKernelNodeField cudaGraphKernelNodeUpdate::field

Which type of update to apply. Determines how updateData is interpreted

uint3 cudaGraphKernelNodeUpdate::gridDim

Grid dimensions

unsigned int cudaGraphKernelNodeUpdate::isEnabled

Node enable/disable data. Nonzero if the node should be enabled, 0 if it should be disabled

cudaGraphDeviceNode_t cudaGraphKernelNodeUpdate::node

Node to update

size_t cudaGraphKernelNodeUpdate::offset

Offset into the parameter buffer at which to apply the update

cudaGraphKernelNodeUpdate::@32::@33 cudaGraphKernelNodeUpdate::param

Kernel parameter data

const void *cudaGraphKernelNodeUpdate::pValue

Kernel parameter data to write in

size_t cudaGraphKernelNodeUpdate::size

Number of bytes to update

cudaGraphKernelNodeUpdate::@32 cudaGraphKernelNodeUpdate::updateData

Update data to apply. Which field is used depends on field's value

7.32. cudaGraphNodeParams Struct Reference

Graph node parameters. See cudaGraphAddNode.

struct cudaMemAllocNodeParamsV2 cudaGraphNodeParams::alloc

Memory allocation node parameters.

struct cudaConditionalNodeParams cudaGraphNodeParams::conditional

Conditional node parameters.

struct cudaEventRecordNodeParams cudaGraphNodeParams::eventRecord

Event record node parameters.

struct cudaEventWaitNodeParams cudaGraphNodeParams::eventWait

Event wait node parameters.

struct cudaExternalSemaphoreSignalNodeParamsV2 cudaGraphNodeParams::extSemSignal

External semaphore signal node parameters.

struct cudaExternalSemaphoreWaitNodeParamsV2 cudaGraphNodeParams::extSemWait

External semaphore wait node parameters.

struct cudaMemFreeNodeParams cudaGraphNodeParams::free

Memory free node parameters.

struct cudaChildGraphNodeParams cudaGraphNodeParams::graph

Child graph node parameters.

struct cudaHostNodeParamsV2 cudaGraphNodeParams::host

Host node parameters.

struct cudaKernelNodeParamsV2 cudaGraphNodeParams::kernel

Kernel node parameters.

struct cudaMemcpyNodeParams cudaGraphNodeParams::memcpy

Memcpy node parameters.

struct cudaMemsetParamsV2 cudaGraphNodeParams::memset

Memset node parameters.

int cudaGraphNodeParams::reserved0

Reserved. Must be zero.

long long cudaGraphNodeParams::reserved1

Padding. Unused bytes must be zero.

long long cudaGraphNodeParams::reserved2

Reserved bytes. Must be zero.

enumcudaGraphNodeType cudaGraphNodeParams::type

Type of the node

7.33. 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

7.34. cudaHostNodeParamsV2 Struct Reference

CUDA host node parameters

cudaHostFn_t cudaHostNodeParamsV2::fn

The function to call when the node executes

void *cudaHostNodeParamsV2::userData

Argument to pass to the function

7.35. cudaIpcEventHandle_t Struct Reference

CUDA IPC event handle

7.36. cudaIpcMemHandle_t Struct Reference

CUDA IPC memory handle

7.37. cudaKernelNodeParams Struct Reference

CUDA GPU kernel node parameters

dim3 cudaKernelNodeParams::blockDim

Block dimensions

**cudaKernelNodeParams::extra

Pointer to kernel arguments in the "extra" format

void *cudaKernelNodeParams::func

Kernel to launch

dim3 cudaKernelNodeParams::gridDim

Grid dimensions

**cudaKernelNodeParams::kernelParams

Array of pointers to individual kernel arguments

unsigned int cudaKernelNodeParams::sharedMemBytes

Dynamic shared-memory size per thread block in bytes

7.38. cudaKernelNodeParamsV2 Struct Reference

CUDA GPU kernel node parameters

uint3 cudaKernelNodeParamsV2::blockDim

Block dimensions

**cudaKernelNodeParamsV2::extra

Pointer to kernel arguments in the "extra" format

void *cudaKernelNodeParamsV2::func

Kernel to launch

uint3 cudaKernelNodeParamsV2::gridDim

Grid dimensions

**cudaKernelNodeParamsV2::kernelParams

Array of pointers to individual kernel arguments

unsigned int cudaKernelNodeParamsV2::sharedMemBytes

Dynamic shared-memory size per thread block in bytes

7.39. cudaLaunchAttribute Struct Reference

Launch attribute

cudaLaunchAttributeID cudaLaunchAttribute::id

Attribute to set

cudaLaunchAttribute::val

Value of the attribute

7.40. cudaLaunchAttributeValue Union Reference

Launch attributes union; used as value field of cudaLaunchAttribute

struct cudaAccessPolicyWindow cudaLaunchAttributeValue::accessPolicyWindow

Value of launch attribute cudaLaunchAttributeAccessPolicyWindow.

cudaLaunchAttributeValue::@34 cudaLaunchAttributeValue::clusterDim

Value of launch attribute cudaLaunchAttributeClusterDimension that represents the desired cluster dimensions for the kernel. Opaque type with the following fields:

- `x` - The X dimension of the cluster, in blocks. Must be a divisor of the grid X dimension.
- `y` - The Y dimension of the cluster, in blocks. Must be a divisor of the grid Y dimension.
- `z` - The Z dimension of the cluster, in blocks. Must be a divisor of the grid Z dimension.

enumcudaClusterSchedulingPolicy cudaLaunchAttributeValue::clusterSchedulingPolicyPreference

Value of launch attribute cudaLaunchAttributeClusterSchedulingPolicyPreference. Cluster scheduling policy preference for the kernel.

int cudaLaunchAttributeValue::cooperative

Value of launch attribute cudaLaunchAttributeCooperative. Nonzero indicates a cooperative kernel (see cudaLaunchCooperativeKernel).

cudaLaunchAttributeValue::@38 cudaLaunchAttributeValue::deviceUpdatableKernelNode

Value of launch attribute cudaLaunchAttributeDeviceUpdatableKernelNode with the following fields:

- `int` deviceUpdatable - Whether or not the resulting kernel node should be device-updatable.
- `cudaGraphDeviceNode_t` devNode - Returns a handle to pass to the various device-side update functions.

cudaLaunchAttributeValue::@37 cudaLaunchAttributeValue::launchCompletionEvent

Value of launch attribute cudaLaunchAttributeLaunchCompletionEvent with the following fields:

- `cudaEvent_t` event - Event to fire when the last block launches.
- `int` flags - Event record flags, see cudaEventRecordWithFlags. Does not accept cudaEventRecordExternal.

cudaLaunchMemSyncDomain cudaLaunchAttributeValue::memSyncDomain

Value of launch attribute cudaLaunchAttributeMemSyncDomain. See cudaLaunchMemSyncDomain.

struct cudaLaunchMemSyncDomainMap cudaLaunchAttributeValue::memSyncDomainMap

Value of launch attribute cudaLaunchAttributeMemSyncDomainMap. See cudaLaunchMemSyncDomainMap.

cudaLaunchAttributeValue::@36 cudaLaunchAttributeValue::preferredClusterDim

Value of launch attribute cudaLaunchAttributePreferredClusterDimension that represents the desired preferred cluster dimensions for the kernel. Opaque type with the following fields:

- `x` - The X dimension of the preferred cluster, in blocks. Must be a divisor of the grid X dimension, and must be a multiple of the `x` field of cudaLaunchAttributeValue::clusterDim.
- `y` - The Y dimension of the preferred cluster, in blocks. Must be a divisor of the grid Y dimension, and must be a multiple of the `y` field of cudaLaunchAttributeValue::clusterDim.
- `z` - The Z dimension of the preferred cluster, in blocks. Must be equal to the `z` field of cudaLaunchAttributeValue::clusterDim.

int cudaLaunchAttributeValue::priority

Value of launch attribute cudaLaunchAttributePriority. Execution priority of the kernel.

cudaLaunchAttributeValue::@35 cudaLaunchAttributeValue::programmaticEvent

Value of launch attribute cudaLaunchAttributeProgrammaticEvent with the following fields:

- `cudaEvent_t` event - Event to fire when all blocks trigger it.
- `int` flags; - Event record flags, see cudaEventRecordWithFlags. Does not accept cudaEventRecordExternal.
- `int` triggerAtBlockStart - If this is set to non-0, each block launch will automatically trigger the event.

int cudaLaunchAttributeValue::programmaticStreamSerializationAllowe

Value of launch attribute cudaLaunchAttributeProgrammaticStreamSerialization.

unsigned int cudaLaunchAttributeValue::sharedMemCarveout

Value of launch attribute cudaLaunchAttributePreferredSharedMemoryCarveout.

enum cudaSynchronizationPolicy cudaLaunchAttributeValue::syncPolicy

Value of launch attribute cudaLaunchAttributeSynchronizationPolicy. cudaSynchronizationPolicy for work queued up in this stream.

7.41. cudaLaunchConfig_t Struct Reference

CUDA extensible launch configuration

cudaLaunchAttribute *cudaLaunchConfig_t::attrs

List of attributes; nullable if cudaLaunchConfig_t::numAttrs == 0

dim3 cudaLaunchConfig_t::blockDim

Block dimensions

size_t cudaLaunchConfig_t::dynamicSmemBytes

Dynamic shared-memory size per thread block in bytes

dim3 cudaLaunchConfig_t::gridDim

Grid dimensions

unsigned int cudaLaunchConfig_t::numAttrs

Number of attributes populated in cudaLaunchConfig_t::attrs

cudaStream_t cudaLaunchConfig_t::stream

Stream identifier

7.42. cudaLaunchMemSyncDomainMap Struct Reference

Memory Synchronization Domain map

See cudaLaunchMemSyncDomain.

By default, kernels are launched in domain 0. Kernel launched with cudaLaunchMemSyncDomainRemote will have a different domain ID. User may also alter the domain ID with cudaLaunchMemSyncDomainMap for a specific stream / graph node / kernel launch. See cudaLaunchAttributeMemSyncDomainMap.

Domain ID range is available through cudaDevAttrMemSyncDomainCount.

unsigned char cudaLaunchMemSyncDomainMap::default_

The default domain ID to use for designated kernels

unsigned char cudaLaunchMemSyncDomainMap::remote

The remote domain ID to use for designated kernels

7.43. cudaLaunchParams Struct Reference

CUDA launch parameters

**cudaLaunchParams::args

Arguments

dim3 cudaLaunchParams::blockDim

Block dimentions

void *cudaLaunchParams::func

Device function symbol

dim3 cudaLaunchParams::gridDim

Grid dimentions

size_t cudaLaunchParams::sharedMem

Shared memory

cudaStream_t cudaLaunchParams::stream

Stream identifier

7.44. cudaMemAccessDesc Struct Reference

Memory access descriptor

enumcudaMemAccessFlags cudaMemAccessDesc::flags

CUmemProt accessibility flags to set on the request

struct cudaMemLocation cudaMemAccessDesc::location

Location on which the request is to change it's accessibility

7.45. 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

7.46. cudaMemAllocNodeParamsV2 Struct Reference

Memory allocation node parameters

size_t cudaMemAllocNodeParamsV2::accessDescCount

in: Number of `accessDescs`s

cudaMemAccessDesc *cudaMemAllocNodeParamsV2::accessDescs

in: number of memory access descriptors. Must not exceed the number of GPUs.

size_t cudaMemAllocNodeParamsV2::bytesize

in: size in bytes of the requested allocation

void *cudaMemAllocNodeParamsV2::dptr

out: address of the allocation returned by CUDA

struct cudaMemPoolProps cudaMemAllocNodeParamsV2::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

7.47. cudaMemcpy3DOperand Struct Reference

Struct representing an operand for copy with cudaMemcpy3DBatchAsync

cudaMemcpy3DOperand::@7::@9 cudaMemcpy3DOperand::array

Struct representing an operand when cudaMemcpy3DOperand::type is cudaMemcpyOperandTypeArray

size_t cudaMemcpy3DOperand::layerHeight

Height of each layer in elements.

struct cudaMemLocation cudaMemcpy3DOperand::locHint

Hint location for the operand. Ignored when the pointers are not managed memory or memory allocated outside CUDA.

cudaMemcpy3DOperand::@7::@8 cudaMemcpy3DOperand::ptr

Struct representing an operand when cudaMemcpy3DOperand::type is cudaMemcpyOperandTypePointer

size_t cudaMemcpy3DOperand::rowLength

Length of each row in elements.

7.48. 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

enumcudaMemcpyKind 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

7.49. 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 cudaPitchedPtr cudaMemcpy3DPeerParms::srcPtr

Pitched source memory address

7.50. cudaMemcpyAttributes Struct Reference

Attributes specific to copies within a batch. For more details on usage see cudaMemcpyBatchAsync.

struct cudaMemLocation cudaMemcpyAttributes::dstLocHint

Hint location for the destination operand. Ignored when the pointers are not managed memory or memory allocated outside CUDA.

unsigned int cudaMemcpyAttributes::flags

Additional flags for copies with this attribute. See cudaMemcpyFlags.

enumcudaMemcpySrcAccessOrder cudaMemcpyAttributes::srcAccessOrder

Source access ordering to be observed for copies with this attribute.

struct cudaMemLocation cudaMemcpyAttributes::srcLocHint

Hint location for the source operand. Ignored when the pointers are not managed memory or memory allocated outside CUDA.

7.51. cudaMemcpyNodeParams Struct Reference

Memcpy node parameters

struct cudaMemcpy3DParms cudaMemcpyNodeParams::copyParams

Parameters for the memory copy

int cudaMemcpyNodeParams::flags

Must be zero

int cudaMemcpyNodeParams::reserved

Must be zero

7.52. cudaMemFreeNodeParams Struct Reference

Memory free node parameters

void *cudaMemFreeNodeParams::dptr

in: the pointer to free

7.53. cudaMemLocation Struct Reference

Specifies a memory location.

To specify a gpu, set type = cudaMemLocationTypeDevice and set id = the gpu's device ordinal. To specify a cpu NUMA node, set type = cudaMemLocationTypeHostNuma and set id = host NUMA node id.

int cudaMemLocation::id

identifier for a given this location's CUmemLocationType.

enumcudaMemLocationType cudaMemLocation::type

Specifies the location type, which modifies the meaning of id.

7.54. cudaMemPoolProps Struct Reference

Specifies the properties of allocations made from the pool.

enumcudaMemAllocationType cudaMemPoolProps::allocType

Allocation type. Currently must be specified as cudaMemAllocationTypePinned

enumcudaMemAllocationHandleType cudaMemPoolProps::handleTypes

Handle types that will be supported by allocations from the pool.

struct cudaMemLocation cudaMemPoolProps::location

Location allocations should reside.

size_t cudaMemPoolProps::maxSize

Maximum pool size. When set to 0, defaults to a system dependent value.

unsigned char cudaMemPoolProps::reserved

reserved for future use, must be 0

unsigned short cudaMemPoolProps::usage

Bitmask indicating intended usage for the pool.

void *cudaMemPoolProps::win32SecurityAttributes

Windows-specific LPSECURITYATTRIBUTES required when cudaMemHandleTypeWin32 is specified. This security attribute defines the scope of which exported allocations may be tranferred to other processes. In all other cases, this field is required to be zero.

7.55. cudaMemPoolPtrExportData Struct Reference

Opaque data for exporting a pool allocation

7.56. 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

7.57. cudaMemsetParamsV2 Struct Reference

CUDA Memset node parameters

void *cudaMemsetParamsV2::dst

Destination device pointer

unsigned int cudaMemsetParamsV2::elementSize

Size of each element in bytes. Must be 1, 2, or 4.

size_t cudaMemsetParamsV2::height

Number of rows

size_t cudaMemsetParamsV2::pitch

Pitch of destination device pointer. Unused if height is 1

unsigned int cudaMemsetParamsV2::value

Value to be set

size_t cudaMemsetParamsV2::width

Width of the row in elements

7.58. cudaOffset3D Struct Reference

Struct representing offset into a cudaArray_t in elements

7.59. 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

7.60. 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.

enumcudaMemoryType cudaPointerAttributes::type

The type of memory - cudaMemoryTypeUnregistered, cudaMemoryTypeHost, cudaMemoryTypeDevice or cudaMemoryTypeManaged.

7.61. 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

7.62. 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

enumcudaResourceType cudaResourceDesc::resType

Resource type

size_t cudaResourceDesc::sizeInBytes

Size in bytes

size_t cudaResourceDesc::width

Width of the array in elements

7.63. 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

enumcudaResourceViewFormat cudaResourceViewDesc::format

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

7.64. cudaTextureDesc Struct Reference

CUDA texture descriptor

enumcudaTextureAddressMode cudaTextureDesc::addressMode

Texture address mode for up to 3 dimensions

float cudaTextureDesc::borderColor

Texture Border Color

int cudaTextureDesc::disableTrilinearOptimization

Disable any trilinear filtering optimizations.

enumcudaTextureFilterMode 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

enumcudaTextureFilterMode 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

enumcudaTextureReadMode cudaTextureDesc::readMode

Texture read mode

int cudaTextureDesc::seamlessCubemap

Enable seamless cube map filtering.

int cudaTextureDesc::sRGB

Perform sRGB->linear conversion during texture read

7.65. CUuuid_st Struct Reference

CUDA UUID types

char CUuuid_st::bytes

< CUDA definition of UUID

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
  cudaMemAllocNodeParamsV2
accessDescs
  cudaMemAllocNodeParamsV2
  cudaMemAllocNodeParams
accessPolicyMaxWindowSize
  cudaDeviceProp
accessPolicyWindow
  cudaLaunchAttributeValue
addressMode
  cudaTextureDesc
alignment
  cudaArrayMemoryRequirements
alloc
  cudaGraphNodeParams
allocType
  cudaMemPoolProps
args
  cudaLaunchParams
array
  cudaResourceDesc
  cudaMemcpy3DOperand
asyncEngineCount
  cudaDeviceProp
attrs
  cudaLaunchConfig_t

B

base_ptr
cudaAccessPolicyWindow
binaryVersion
cudaFuncAttributes
blockDim
cudaKernelNodeParams
cudaKernelNodeParamsV2
cudaLaunchParams
cudaLaunchConfig_t
borderColor
cudaTextureDesc
bytes
cudaUUID_t
bytesize
cudaMemAllocNodeParams
cudaMemAllocNodeParamsV2
bytesOverBudget
cudaAsyncNotificationInfo_t

C

cacheModeCA
cudaFuncAttributes
canMapHostMemory
cudaDeviceProp
canUseHostPointerForRegisteredMem
cudaDeviceProp
channelDesc
cudaEglPlaneDesc
clockRate
cudaDeviceProp
clusterDim
cudaLaunchAttributeValue
clusterDimMustBeSet
cudaFuncAttributes
clusterLaunch
cudaDeviceProp
clusterSchedulingPolicyPreference
cudaFuncAttributes
cudaLaunchAttributeValue
computeMode
cudaDeviceProp

computePreemptionSupported
cudaDeviceProp
concurrentKernels
cudaDeviceProp
concurrentManagedAccess
cudaDeviceProp
conditional
cudaGraphNodeParams
constSizeBytes
cudaFuncAttributes
cooperative
cudaLaunchAttributeValue
cooperativeLaunch
cudaDeviceProp
cooperativeMultiDeviceLaunch
cudaDeviceProp
copyParams
cudaMemcpyNodeParams

D

default_
cudaLaunchMemSyncDomainMap
deferredMappingCudaArraySupported
cudaDeviceProp
depth
cudaArraySparseProperties
cudaExtent
cudaEglPlaneDesc
cudaResourceViewDesc
desc
cudaResourceDesc
device
cudaPointerAttributes
deviceOverlap
cudaDeviceProp
devicePointer
cudaPointerAttributes
deviceUpdatableKernelNode
cudaLaunchAttributeValue
devPtr
cudaResourceDesc
directManagedMemAccessFromHost
cudaDeviceProp

disableTrilinearOptimization
 cudaTextureDesc
dptr
 cudaMemAllocNodeParamsV2
 cudaMemFreeNodeParams
 cudaMemAllocNodeParams
dst
 cudaMemsetParamsV2
 cudaMemsetParams
dstArray
 cudaMemcpy3DPeerParms
 cudaMemcpy3DParms
dstDevice
 cudaMemcpy3DPeerParms
dstLocHint
 cudaMemcpyAttributes
dstPos
 cudaMemcpy3DParms
 cudaMemcpy3DPeerParms
dstPtr
 cudaMemcpy3DPeerParms
 cudaMemcpy3DParms
dynamicSmemBytes
 cudaLaunchConfig_t

E

ECCEnabled
 cudaDeviceProp
eglColorFormat
 cudaEglFrame
elementSize
 cudaMemsetParamsV2
 cudaMemsetParams
errNode_out
 cudaGraphInstantiateParams
errorFromNode
 cudaGraphExecUpdateResultInfo
errorNode
 cudaGraphExecUpdateResultInfo
event
 cudaEventRecordNodeParams
 cudaEventWaitNodeParams

eventRecord
cudaGraphNodeParams
eventWait
cudaGraphNodeParams
extent
cudaMemcpy3DParms
cudaMemcpy3DPeerParms
cudaExternalMemoryMipmappedArrayDesc
extra
cudaKernelNodeParamsV2
cudaKernelNodeParams
extSemArray
cudaExternalSemaphoreSignalNodeParamsV2
cudaExternalSemaphoreSignalNodeParams
cudaExternalSemaphoreWaitNodeParams
cudaExternalSemaphoreWaitNodeParamsV2
extSemSignal
cudaGraphNodeParams
extSemWait
cudaGraphNodeParams

F

f
cudaChannelFormatDesc
fd
cudaExternalMemoryHandleDesc
cudaExternalSemaphoreHandleDesc
fence
cudaExternalSemaphoreSignalParams_v1
cudaExternalSemaphoreSignalParams
cudaExternalSemaphoreSignalParams_v1
cudaExternalSemaphoreWaitParams
cudaExternalSemaphoreWaitParams_v1
field
cudaGraphKernelNodeUpdate
filterMode
cudaTextureDesc
firstLayer
cudaResourceViewDesc
firstMipmapLevel
cudaResourceViewDesc
flags
cudaArraySparseProperties

cudaExternalSemaphoreHandleDesc
cudaMemcpyNodeParams
cudaMemAccessDesc
cudaMemcpyAttributes
cudaExternalMemoryHandleDesc
cudaExternalMemoryBufferDesc
cudaExternalMemoryMipmappedArrayDesc
cudaExternalSemaphoreSignalParams_v1
cudaExternalSemaphoreWaitParams
cudaExternalSemaphoreWaitParams_v1
cudaExternalSemaphoreSignalParams
cudaGraphInstantiateParams

fn

cudaHostNodeParamsV2
cudaHostNodeParams

format

cudaResourceViewDesc

formatDesc

cudaExternalMemoryMipmappedArrayDesc

frameType

cudaEglFrame

free

cudaGraphNodeParams

from_port

cudaGraphEdgeData

func

cudaKernelNodeParamsV2
cudaLaunchParams
cudaKernelNodeParams

G

globalL1CacheSupported

cudaDeviceProp

gpuDirectRDMAFlushWritesOptions

cudaDeviceProp

gpuDirectRDMASupported

cudaDeviceProp

gpuDirectRDMAWritesOrdering

cudaDeviceProp

graph

cudaChildGraphNodeParams
cudaGraphNodeParams

gridDim
cudaGraphKernelNodeUpdate
cudaKernelNodeParamsV2
cudaLaunchConfig_t
cudaLaunchParams
cudaKernelNodeParams

H

handle
cudaExternalMemoryHandleDesc
cudaExternalSemaphoreHandleDesc
cudaConditionalNodeParams
handleTypes
cudaMemPoolProps
height
cudaMemsetParams
cudaMemsetParamsV2
cudaEglPlaneDesc
cudaResourceDesc
cudaResourceViewDesc
cudaArraySparseProperties
cudaExtent
hitProp
cudaAccessPolicyWindow
hitRatio
cudaAccessPolicyWindow
host
cudaGraphNodeParams
hostNativeAtomicSupported
cudaDeviceProp
hostPointer
cudaPointerAttributes
hostRegisterReadOnlySupported
cudaDeviceProp
hostRegisterSupported
cudaDeviceProp

I

id
cudaMemLocation
cudaLaunchAttribute
info
cudaAsyncNotificationInfo_t

integrated
cudaDeviceProp
ipcEventSupported
cudaDeviceProp
isEnabled
cudaGraphKernelNodeUpdate
isMultiGpuBoard
cudaDeviceProp

K

kernel
cudaGraphNodeParams
kernelExecTimeoutEnabled
cudaDeviceProp
kernelParams
cudaKernelNodeParamsV2
cudaKernelNodeParams
key
cudaExternalSemaphoreWaitParams_v1
cudaExternalSemaphoreWaitParams
keyedMutex
cudaExternalSemaphoreSignalParams
cudaExternalSemaphoreWaitParams
cudaExternalSemaphoreWaitParams_v1
cudaExternalSemaphoreSignalParams_v1
kind
cudaMemcpy3DParms

L

l2CacheSize
cudaDeviceProp
lastLayer
cudaResourceViewDesc
lastMipmapLevel
cudaResourceViewDesc
launchCompletionEvent
cudaLaunchAttributeValue
layerHeight
cudaMemcpy3DOperand
localL1CacheSupported
cudaDeviceProp
localSizeBytes
cudaFuncAttributes

location
[cudaMemPoolProps](#)
[cudaMemAccessDesc](#)
locHint
[cudaMemcpy3DOperand](#)
luid
[cudaDeviceProp](#)
luidDeviceNodeMask
[cudaDeviceProp](#)

M

major
[cudaDeviceProp](#)
managedMemory
[cudaDeviceProp](#)
maxAnisotropy
[cudaTextureDesc](#)
maxBlocksPerMultiProcessor
[cudaDeviceProp](#)
maxDynamicSharedSizeBytes
[cudaFuncAttributes](#)
maxGridSize
[cudaDeviceProp](#)
maxMipmapLevelClamp
[cudaTextureDesc](#)
maxSize
[cudaMemPoolProps](#)
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
maxThreadsPerBlock
cudaDeviceProp
cudaFuncAttributes
maxThreadsPerMultiProcessor
cudaDeviceProp
memcpy
cudaGraphNodeParams
memoryBusWidth
cudaDeviceProp
memoryClockRate
cudaDeviceProp
memoryPoolsSupported
cudaDeviceProp
memoryPoolSupportedHandleTypes
cudaDeviceProp
memPitch
cudaDeviceProp

memset
cudaGraphNodeParams
memSyncDomain
cudaLaunchAttributeValue
memSyncDomainMap
cudaLaunchAttributeValue
minMipmapLevelClamp
cudaTextureDesc
minor
cudaDeviceProp
mipmap
cudaResourceDesc
mipmapFilterMode
cudaTextureDesc
mipmapLevelBias
cudaTextureDesc
miptailFirstLevel
cudaArraySparseProperties
miptailSize
cudaArraySparseProperties
missProp
cudaAccessPolicyWindow
multiGpuBoardGroupID
cudaDeviceProp
multiProcessorCount
cudaDeviceProp

N

name
cudaDeviceProp
cudaExternalMemoryHandleDesc
cudaExternalSemaphoreHandleDesc
node
cudaGraphKernelNodeUpdate
nonPortableClusterSizeAllowed
cudaFuncAttributes
normalizedCoords
cudaTextureDesc
num_bytes
cudaAccessPolicyWindow
numAttrs
cudaLaunchConfig_t

numChannels
cudaEglPlaneDesc
numExtSems
cudaExternalSemaphoreSignalNodeParamsV2
cudaExternalSemaphoreSignalNodeParams
cudaExternalSemaphoreWaitNodeParamsV2
cudaExternalSemaphoreWaitNodeParams
numLevels
cudaExternalMemoryMipmappedArrayDesc
numRegs
cudaFuncAttributes
nvSciBufObject
cudaExternalMemoryHandleDesc
nvSciSyncObj
cudaExternalSemaphoreHandleDesc

O

offset
cudaExternalMemoryBufferDesc
cudaExternalMemoryMipmappedArrayDesc
cudaGraphKernelNodeUpdate
overBudget
cudaAsyncNotificationInfo_t
ownership
cudaChildGraphNodeParams

P

pageableMemoryAccess
cudaDeviceProp
pageableMemoryAccessUsesHostPageTables
cudaDeviceProp
param
cudaGraphKernelNodeUpdate
paramsArray
cudaExternalSemaphoreSignalNodeParams
cudaExternalSemaphoreSignalNodeParamsV2
cudaExternalSemaphoreWaitNodeParams
cudaExternalSemaphoreWaitNodeParamsV2
pArray
cudaEglFrame
pciBusID
cudaDeviceProp

pciDeviceID
cudaDeviceProp
pciDomainID
cudaDeviceProp
persistingL2CacheMaxSize
cudaDeviceProp
phGraph_out
cudaConditionalNodeParams
pitch
cudaMemsetParams
cudaMemsetParamsV2
cudaEglPlaneDesc
cudaPitchedPtr
pitchInBytes
cudaResourceDesc
planeCount
cudaEglFrame
planeDesc
cudaEglFrame
poolProps
cudaMemAllocNodeParams
cudaMemAllocNodeParamsV2
pPitch
cudaEglFrame
preferredClusterDim
cudaLaunchAttributeValue
preferredShmemCarveout
cudaFuncAttributes
priority
cudaLaunchAttributeValue
programmaticEvent
cudaLaunchAttributeValue
programmaticStreamSerializationAllowed
cudaLaunchAttributeValue
ptr
cudaPitchedPtr
cudaMemcpy3DOperand
ptxVersion
cudaFuncAttributes
pValue
cudaGraphKernelNodeUpdate

R

readMode
cudaTextureDesc
regsPerBlock
cudaDeviceProp
regsPerMultiprocessor
cudaDeviceProp
remote
cudaLaunchMemSyncDomainMap
requiredClusterWidth
cudaFuncAttributes
reserved
cudaFuncAttributes
cudaMemPoolProps
cudaDeviceProp
cudaGraphEdgeData
cudaEglPlaneDesc
cudaMemcpyNodeParams
reserved0
cudaGraphNodeParams
reserved1
cudaGraphNodeParams
reserved2
cudaGraphNodeParams
reservedSharedMemPerBlock
cudaDeviceProp
resType
cudaResourceDesc
result
cudaGraphExecUpdateResultInfo
result_out
cudaGraphInstantiateParams
rowLength
cudaMemcpy3DOperand

S

seamlessCubemap
cudaTextureDesc
sharedMem
cudaLaunchParams
sharedMemBytes
cudaKernelNodeParamsV2

cudaKernelNodeParams
sharedMemCarveout
cudaLaunchAttributeValue
sharedMemPerBlock
cudaDeviceProp
sharedMemPerBlockOptin
cudaDeviceProp
sharedMemPerMultiprocessor
cudaDeviceProp
sharedSizeBytes
cudaFuncAttributes
singleToDoublePrecisionPerfRatio
cudaDeviceProp
size
cudaArrayMemoryRequirements
cudaExternalMemoryHandleDesc
cudaExternalMemoryBufferDesc
cudaConditionalNodeParams
cudaGraphKernelNodeUpdate
sizeInBytes
cudaResourceDesc
sparseCudaArraySupported
cudaDeviceProp
srcAccessOrder
cudaMemcpyAttributes
srcArray
cudaMemcpy3DPeerParms
cudaMemcpy3DParms
srcDevice
cudaMemcpy3DPeerParms
srcLocHint
cudaMemcpyAttributes
srcPos
cudaMemcpy3DParms
cudaMemcpy3DPeerParms
srcPtr
cudaMemcpy3DPeerParms
cudaMemcpy3DParms
sRGB
cudaTextureDesc
stream
cudaLaunchConfig_t
cudaLaunchParams

streamPrioritiesSupported
cudaDeviceProp
surfaceAlignment
cudaDeviceProp
syncPolicy
cudaLaunchAttributeValue

T

tccDriver
cudaDeviceProp
textureAlignment
cudaDeviceProp
texturePitchAlignment
cudaDeviceProp
timelineSemaphoreInteropSupported
cudaDeviceProp
timeoutMs
cudaExternalSemaphoreWaitParams_v1
cudaExternalSemaphoreWaitParams
to_port
cudaGraphEdgeData
totalConstMem
cudaDeviceProp
totalGlobalMem
cudaDeviceProp
type
cudaConditionalNodeParams
cudaAsyncNotificationInfo_t
cudaExternalSemaphoreHandleDesc
cudaGraphEdgeData
cudaGraphNodeParams
cudaMemLocation
cudaExternalMemoryHandleDesc
cudaPointerAttributes

U

unifiedAddressing
cudaDeviceProp
unifiedFunctionPointers
cudaDeviceProp
updateData
cudaGraphKernelNodeUpdate

uploadStream
cudaGraphInstantiateParams
usage
cudaMemPoolProps
userData
cudaHostNodeParams
cudaHostNodeParamsV2
uuid
cudaDeviceProp

V

val
cudaLaunchAttribute
value
cudaExternalSemaphoreWaitParams
cudaExternalSemaphoreSignalParams
cudaExternalSemaphoreWaitParams_v1
cudaExternalSemaphoreSignalParams_v1
cudaMemsetParamsV2
cudaMemsetParams

W

w
cudaChannelFormatDesc
warpSize
cudaDeviceProp
width
cudaArraySparseProperties
cudaResourceDesc
cudaResourceViewDesc
cudaExtent
cudaMemsetParamsV2
cudaMemsetParams
cudaEglPlaneDesc
win32
cudaExternalMemoryHandleDesc
cudaExternalSemaphoreHandleDesc
win32SecurityAttributes
cudaMemPoolProps

X

x
cudaChannelFormatDesc

cudaPos

xsize

cudaPitchedPtr

Y

y

cudaChannelFormatDesc

cudaPos

ysize

cudaPitchedPtr

Z

z

cudaChannelFormatDesc

cudaPos

Chapter 9. Deprecated List

Global cudaDeviceGetSharedMemConfig

Global cudaDeviceSetSharedMemConfig

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 cudaFuncSetSharedMemConfig

Global cudaMemcpyArrayToArray

Global cudaMemcpyFromArray

Global cudaMemcpyFromArrayAsync

Global cudaMemcpyToArray

Global cudaMemcpyToArrayAsync

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 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 cudaSharedMemConfig

Global cudaDeviceBlockingSync

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

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
https://www.nvidia.com