



CUDA Driver API

API Reference Manual

Table of Contents

Chapter 1. Difference between the driver and runtime APIs.....	1
Chapter 2. API synchronization behavior.....	3
Chapter 3. Stream synchronization behavior.....	5
Chapter 4. Graph object thread safety.....	7
Chapter 5. Rules for version mixing.....	8
Chapter 6. Modules.....	9
6.1. Data types used by CUDA driver.....	10
CUaccessPolicyWindow_v1.....	11
CUarrayMapInfo_v1.....	11
CUasyncNotificationInfo.....	11
CUcheckpointCheckpointArgs.....	11
CUcheckpointLockArgs.....	11
CUcheckpointRestoreArgs.....	11
CUcheckpointUnlockArgs.....	11
CUctxCigParam.....	11
CUctxCreateParams.....	11
CUDA_ARRAY3D_DESCRIPTOR_v2.....	11
CUDA_ARRAY_DESCRIPTOR_v2.....	11
CUDA_ARRAY_MEMORY_REQUIREMENTS_v1.....	11
CUDA_ARRAY_SPARSE_PROPERTIES_v1.....	11
CUDA_BATCH_MEM_OP_NODE_PARAMS_v2.....	11
CUDA_CHILD_GRAPH_NODE_PARAMS.....	11
CUDA_CONDITIONAL_NODE_PARAMS.....	11
CUDA_EVENT_RECORD_NODE_PARAMS.....	11
CUDA_EVENT_WAIT_NODE_PARAMS.....	12
CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v1.....	12
CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v2.....	12
CUDA_EXT_SEM_WAIT_NODE_PARAMS_v1.....	12
CUDA_EXT_SEM_WAIT_NODE_PARAMS_v2.....	12
CUDA_EXTERNAL_MEMORY_BUFFER_DESC_v1.....	12
CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1.....	12
CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC_v1.....	12
CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_v1.....	12
CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1.....	12

CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1.....	12
CUDA_GRAPH_INSTANTIATE_PARAMS.....	12
CUDA_HOST_NODE_PARAMS_v1.....	12
CUDA_HOST_NODE_PARAMS_v2.....	13
CUDA_KERNEL_NODE_PARAMS_v1.....	13
CUDA_KERNEL_NODE_PARAMS_v2.....	13
CUDA_KERNEL_NODE_PARAMS_v3.....	13
CUDA_LAUNCH_PARAMS_v1.....	13
CUDA_MEM_ALLOC_NODE_PARAMS_v1.....	13
CUDA_MEM_ALLOC_NODE_PARAMS_v2.....	13
CUDA_MEM_FREE_NODE_PARAMS.....	13
CUDA_MEMCPY2D_v2.....	13
CUDA_MEMCPY3D_PEER_v1.....	13
CUDA_MEMCPY3D_v2.....	13
CUDA_MEMCPY_NODE_PARAMS.....	13
CUDA_MEMSET_NODE_PARAMS_v1.....	13
CUDA_MEMSET_NODE_PARAMS_v2.....	13
CUDA_POINTER_ATTRIBUTE_P2P_TOKENS_v1.....	13
CUDA_RESOURCE_DESC_v1.....	13
CUDA_RESOURCE_VIEW_DESC_v1.....	14
CUDA_TEXTURE_DESC_v1.....	14
CUdevprop_v1.....	14
CUeglFrame_v1.....	14
CUexecAffinityParam_v1.....	14
CUexecAffinitySmCount_v1.....	14
CUextent3D_v1.....	14
CUgraphEdgeData.....	14
CUgraphExecUpdateResultInfo_v1.....	14
CUgraphNodeParams.....	14
CUipcEventHandle_v1.....	14
CUipcMemHandle_v1.....	14
CUlaunchAttribute.....	14
CUlaunchAttributeValue.....	14
CUlaunchConfig.....	14
CUlaunchMemSyncDomainMap.....	14
CUmemAccessDesc_v1.....	14
CUmemAllocationProp_v1.....	15
CUmemcpy3DOperand_v1.....	15

CUmecpyAttributes_v1.....	15
CUMemFabricHandle_v1.....	15
CUMemLocation_v1.....	15
CUMemPoolProps_v1.....	15
CUMemPoolPtrExportData_v1.....	15
CUMulticastObjectProp_v1.....	15
CUOffset3D_v1.....	15
CUstreamBatchMemOpParams_v1.....	15
CUtensorMap.....	15
cl_context_flags.....	15
cl_event_flags.....	16
CUaccessProperty.....	16
CUaddress_mode.....	16
CUarray_cubemap_face.....	17
CUarray_format.....	17
CUarraySparseSubresourceType.....	20
CUasyncNotificationType.....	20
CUclusterSchedulingPolicy.....	20
CUcomputemode.....	20
CUctx_flags.....	21
CUDA_POINTER_ATTRIBUTE_ACCESS_FLAGS.....	21
CUdevice_attribute.....	22
CUdevice_P2PAttribute.....	29
CUdeviceNumaConfig.....	29
CUdriverProcAddress_flags.....	30
CUdriverProcAddressQueryResult.....	30
CUeglColorFormat.....	30
CUeglFrameType.....	37
CUeglResourceLocationFlags.....	38
CUEvent_flags.....	38
CUEvent_record_flags.....	38
CUEvent_sched_flags.....	39
CUEvent_wait_flags.....	39
CUexecAffinityType.....	39
CUexternalMemoryHandleType.....	39
CUexternalSemaphoreHandleType.....	40
CUfilter_mode.....	41
CUflushGPUDirectRDMAWritesOptions.....	41

CUflushGPUDirectRDMAWritesScope.....	41
CUflushGPUDirectRDMAWritesTarget.....	41
CUfunc_cache.....	42
CUfunction_attribute.....	42
CUGPUDirectRDMAWritesOrdering.....	44
CUgraphConditionalNodeType.....	44
CUgraphDebugDot_flags.....	44
CUgraphDependencyType.....	45
CUgraphExecUpdateResult.....	45
CUgraphicsMapResourceFlags.....	46
CUgraphicsRegisterFlags.....	46
CUgraphInstantiate_flags.....	47
CUgraphInstantiateResult.....	47
CUgraphNodeType.....	47
CUipcMem_flags.....	48
CUjit_cacheMode.....	49
CUjit_fallback.....	49
CUjit_option.....	49
CUjit_target.....	53
CUjitInputType.....	54
CUlaunchAttributeID.....	55
CUlaunchMemSyncDomain.....	57
CULibraryOption.....	58
CULimit.....	58
CUmем_advise.....	59
CUmемAccess_flags.....	60
CUmемAllocationCompType.....	60
CUmемAllocationGranularity_flags.....	60
CUmемAllocationHandleType.....	60
CUmемAllocationType.....	61
CUmемAttach_flags.....	61
CUmемcpy3DOperandType.....	61
CUmемcpyFlags.....	62
CUmемcpySrcAccessOrder.....	62
CUmемHandleType.....	62
CUmемLocationType.....	63
CUmемOperationType.....	63
CUmemorytype.....	63

CUMemPool_attribute.....	63
CUMemRangeFlags.....	64
CUMemRangeHandleType.....	64
CUMulticastGranularity_flags.....	65
CUoccupancy_flags.....	65
CUpointer_attribute.....	65
CUprocessState.....	66
CUresourcetype.....	67
CUresourceViewFormat.....	67
CUresult.....	69
CUshared_carveout.....	76
CUsharedconfig.....	76
CUstream_flags.....	77
CUstreamBatchMemOpType.....	77
CUstreamCaptureMode.....	77
CUstreamCaptureStatus.....	77
CUstreamMemoryBarrier_flags.....	78
CUstreamUpdateCaptureDependencies_flags.....	78
CUstreamWaitValue_flags.....	78
CUstreamWriteValue_flags.....	79
CUtensorMapDataType.....	79
CUtensorMapFloatOOBfill.....	80
CUtensorMapIm2ColWideMode.....	80
CUtensorMapInterleave.....	80
CUtensorMapL2promotion.....	80
CUtensorMapSwizzle.....	80
CUuserObject_flags.....	81
CUuserObjectRetain_flags.....	81
CUaccessPolicyWindow.....	81
CUarray.....	81
CUasyncCallback.....	81
CUasyncCallbackHandle.....	82
CUcontext.....	82
CUdevice.....	82
CUdevice_v1.....	82
CUdeviceptr.....	82
CUdeviceptr_v2.....	82
CUeglStreamConnection.....	82

CUevent.....	82
CUexecAffinityParam.....	82
CUexternalMemory.....	82
CUexternalSemaphore.....	83
CUfunction.....	83
CUgraph.....	83
CUgraphConditionalHandle.....	83
CUgraphDeviceNode.....	83
CUgraphExec.....	83
CUgraphicsResource.....	83
CUgraphNode.....	83
CUgreenCtx.....	83
CUhostFn.....	83
CUkernel.....	84
CULibrary.....	84
CUmemoryPool.....	84
CUmipmappedArray.....	84
CUmodule.....	84
CUoccupancyB2DSize.....	84
CUstream.....	84
CUstreamCallback.....	84
CUsurfObject.....	84
CUsurfObject_v1.....	85
CUsurfref.....	85
CUtexObject.....	85
CUtexObject_v1.....	85
CUtexref.....	85
CUuserObject.....	85
CU_ARRAY_SPARSE_PROPERTIES_SINGLE_MIPTAIL.....	85
CU_DEVICE_CPU.....	85
CU_DEVICE_INVALID.....	85
CU_GRAPH_COND_ASSIGN_DEFAULT.....	85
CU_GRAPH_KERNEL_NODE_PORT_DEFAULT.....	86
CU_GRAPH_KERNEL_NODE_PORT_LAUNCH_ORDER.....	86
CU_GRAPH_KERNEL_NODE_PORT_PROGRAMMATIC.....	86
CU_IPC_HANDLE_SIZE.....	86
CU_LAUNCH_PARAM_BUFFER_POINTER.....	86
CU_LAUNCH_PARAM_BUFFER_POINTER_AS_INT.....	87

CU_LAUNCH_PARAM_BUFFER_SIZE.....	87
CU_LAUNCH_PARAM_BUFFER_SIZE_AS_INT.....	87
CU_LAUNCH_PARAM_END.....	87
CU_LAUNCH_PARAM_END_AS_INT.....	87
CU_MEM_CREATE_USAGE_HW_DECOMPRESS.....	87
CU_MEM_CREATE_USAGE_TILE_POOL.....	87
CU_MEM_POOL_CREATE_USAGE_HW_DECOMPRESS.....	88
CU_MEMHOSTALLOC_DEVICEMAP.....	88
CU_MEMHOSTALLOC_PORTABLE.....	88
CU_MEMHOSTALLOC_WRITECOMBINED.....	88
CU_MEMHOSTREGISTER_DEVICEMAP.....	88
CU_MEMHOSTREGISTER_IOMEMORY.....	88
CU_MEMHOSTREGISTER_PORTABLE.....	88
CU_MEMHOSTREGISTER_READ_ONLY.....	89
CU_PARAM_TR_DEFAULT.....	89
CU_STREAM_LEGACY.....	89
CU_STREAM_PER_THREAD.....	89
CU_TENSOR_MAP_NUM_QWORDS.....	89
CU_TRSA_OVERRIDE_FORMAT.....	89
CU_TRSF_DISABLE_TRILINEAR_OPTIMIZATION.....	90
CU_TRSF_NORMALIZED_COORDINATES.....	90
CU_TRSF_READ_AS_INTEGER.....	90
CU_TRSF_SEAMLESS_CUBEMAP.....	90
CU_TRSF_SRGB.....	90
CUDA_ARRAY3D_2DARRAY.....	90
CUDA_ARRAY3D_COLOR_ATTACHMENT.....	90
CUDA_ARRAY3D_CUBEMAP.....	90
CUDA_ARRAY3D_DEFERRED_MAPPING.....	91
CUDA_ARRAY3D_DEPTH_TEXTURE.....	91
CUDA_ARRAY3D_LAYERED.....	91
CUDA_ARRAY3D_SPARSE.....	91
CUDA_ARRAY3D_SURFACE_LDST.....	91
CUDA_ARRAY3D_TEXTURE_GATHER.....	91
CUDA_ARRAY3D_VIDEO_ENCODE_DECODE.....	91
CUDA_COOPERATIVE_LAUNCH_MULTI_DEVICE_NO_POST_LAUNCH_SYNC.....	92
CUDA_COOPERATIVE_LAUNCH_MULTI_DEVICE_NO_PRE_LAUNCH_SYNC.....	92
CUDA_EGL_INFINITE_TIMEOUT.....	92
CUDA_EXTERNAL_MEMORY_DEDICATED.....	92

CUDA_EXTERNAL_SEMAPHORE_SIGNAL_SKIP_NVSCIBUF_MEMSYNC.....	92
CUDA_EXTERNAL_SEMAPHORE_WAIT_SKIP_NVSCIBUF_MEMSYNC.....	93
CUDA_NVSCISYNC_ATTR_SIGNAL.....	93
CUDA_NVSCISYNC_ATTR_WAIT.....	93
CUDA_VERSION.....	93
MAX_PLANES.....	93
6.2. Error Handling.....	93
cuGetErrorName.....	94
cuGetErrorString.....	94
6.3. Initialization.....	95
cuInit.....	95
6.4. Version Management.....	95
cuDriverGetVersion.....	96
6.5. Device Management.....	96
cuDeviceGet.....	96
cuDeviceGetAttribute.....	97
cuDeviceGetCount.....	104
cuDeviceGetDefaultMemPool.....	104
cuDeviceGetExecAffinitySupport.....	105
cuDeviceGetLuid.....	106
cuDeviceGetMemPool.....	106
cuDeviceGetName.....	107
cuDeviceGetNvSciSyncAttributes.....	108
cuDeviceGetTexture1DLinearMaxWidth.....	109
cuDeviceGetUuid.....	110
cuDeviceGetUuid_v2.....	111
cuDeviceSetMemPool.....	111
cuDeviceTotalMem.....	112
cuFlushGPUDirectRDMAWrites.....	113
6.6. Device Management [DEPRECATED].....	114
cuDeviceComputeCapability.....	114
cuDeviceGetProperties.....	115
6.7. Primary Context Management.....	116
cuDevicePrimaryCtxGetState.....	116
cuDevicePrimaryCtxRelease.....	117
cuDevicePrimaryCtxReset.....	118
cuDevicePrimaryCtxRetain.....	118
cuDevicePrimaryCtxSetFlags.....	119

6.8. Context Management.....	121
cuCtxCreate.....	122
cuCtxCreate_v3.....	124
cuCtxCreate_v4.....	127
cuCtxDestroy.....	130
cuCtxGetApiVersion.....	131
cuCtxGetCacheConfig.....	132
cuCtxGetCurrent.....	133
cuCtxGetDevice.....	133
cuCtxGetExecAffinity.....	134
cuCtxGetFlags.....	134
cuCtxGetId.....	135
cuCtxGetLimit.....	136
cuCtxGetStreamPriorityRange.....	137
cuCtxPopCurrent.....	138
cuCtxPushCurrent.....	138
cuCtxRecordEvent.....	139
cuCtxResetPersistingL2Cache.....	140
cuCtxSetCacheConfig.....	140
cuCtxSetCurrent.....	141
cuCtxSetFlags.....	142
cuCtxSetLimit.....	143
cuCtxSynchronize.....	144
cuCtxWaitEvent.....	145
6.9. Context Management [DEPRECATED].....	146
cuCtxAttach.....	146
cuCtxDetach.....	147
cuCtxGetSharedMemConfig.....	148
cuCtxSetSharedMemConfig.....	149
6.10. Module Management.....	150
CUmoduleLoadingMode.....	150
cuLinkAddData.....	150
cuLinkAddFile.....	151
cuLinkComplete.....	152
cuLinkCreate.....	153
cuLinkDestroy.....	154
cuModuleEnumerateFunctions.....	154
cuModuleGetFunction.....	155

cuModuleGetFunctionCount.....	156
cuModuleGetGlobal.....	156
cuModuleGetLoadingMode.....	157
cuModuleLoad.....	158
cuModuleLoadData.....	159
cuModuleLoadDataEx.....	160
cuModuleLoadFatBinary.....	161
cuModuleUnload.....	162
6.11. Module Management [DEPRECATED].....	162
cuModuleGetSurfRef.....	163
cuModuleGetTexRef.....	163
6.12. Library Management.....	164
cuKernelGetAttribute.....	164
cuKernelGetFunction.....	166
cuKernelGetLibrary.....	167
cuKernelGetName.....	168
cuKernelGetParamInfo.....	168
cuKernelSetAttribute.....	169
cuKernelSetCacheConfig.....	171
cuLibraryEnumerateKernels.....	172
cuLibraryGetGlobal.....	173
cuLibraryGetKernel.....	173
cuLibraryGetKernelCount.....	174
cuLibraryGetManaged.....	175
cuLibraryGetModule.....	175
cuLibraryGetUnifiedFunction.....	176
cuLibraryLoadData.....	177
cuLibraryLoadFromFile.....	178
cuLibraryUnload.....	180
6.13. Memory Management.....	180
CUmemDecompressParams.....	181
CUmemDecompressAlgorithm.....	181
cuArray3DCreate.....	181
cuArray3DGetDescriptor.....	185
cuArrayCreate.....	186
cuArrayDestroy.....	188
cuArrayGetDescriptor.....	189
cuArrayGetMemoryRequirements.....	190

cuArrayGetPlane.....	191
cuArrayGetSparseProperties.....	192
cuDeviceGetByPCIBusId.....	193
cuDeviceGetPCIBusId.....	193
cuDeviceRegisterAsyncNotification.....	194
cuDeviceUnregisterAsyncNotification.....	195
cuIpcCloseMemHandle.....	196
cuIpcGetEventHandle.....	197
cuIpcGetMemHandle.....	198
cuIpcOpenEventHandle.....	198
cuIpcOpenMemHandle.....	199
cuMemAlloc.....	201
cuMemAllocHost.....	201
cuMemAllocManaged.....	203
cuMemAllocPitch.....	205
cuMemBatchDecompressAsync.....	207
cuMemcpy.....	208
cuMemcpy2D.....	209
cuMemcpy2DAsync.....	212
cuMemcpy2DUnaligned.....	214
cuMemcpy3D.....	217
cuMemcpy3DAsync.....	220
cuMemcpy3DBatchAsync.....	223
cuMemcpy3DPeer.....	225
cuMemcpy3DPeerAsync.....	225
cuMemcpyAsync.....	226
cuMemcpyAtoA.....	227
cuMemcpyAtoD.....	228
cuMemcpyAtoH.....	229
cuMemcpyAtoHAsync.....	231
cuMemcpyBatchAsync.....	232
cuMemcpyDtoA.....	234
cuMemcpyDtoD.....	235
cuMemcpyDtoDAsync.....	236
cuMemcpyDtoH.....	237
cuMemcpyDtoHAsync.....	238
cuMemcpyHtoA.....	239
cuMemcpyHtoAAsync.....	240

cuMemcpyHtoD.....	242
cuMemcpyHtoDAsync.....	243
cuMemcpyPeer.....	244
cuMemcpyPeerAsync.....	245
cuMemFree.....	246
cuMemFreeHost.....	247
cuMemGetAddressRange.....	247
cuMemGetHandleForAddressRange.....	248
cuMemGetInfo.....	249
cuMemHostAlloc.....	250
cuMemHostGetDevicePointer.....	252
cuMemHostGetFlags.....	254
cuMemHostRegister.....	254
cuMemHostUnregister.....	256
cuMemsetD16.....	257
cuMemsetD16Async.....	258
cuMemsetD2D16.....	259
cuMemsetD2D16Async.....	260
cuMemsetD2D32.....	261
cuMemsetD2D32Async.....	262
cuMemsetD2D8.....	263
cuMemsetD2D8Async.....	264
cuMemsetD32.....	266
cuMemsetD32Async.....	267
cuMemsetD8.....	268
cuMemsetD8Async.....	269
cuMipmappedArrayCreate.....	270
cuMipmappedArrayDestroy.....	273
cuMipmappedArrayGetLevel.....	274
cuMipmappedArrayGetMemoryRequirements.....	275
cuMipmappedArrayGetSparseProperties.....	276
6.14. Virtual Memory Management.....	276
cuMemAddressFree.....	277
cuMemAddressReserve.....	277
cuMemCreate.....	278
cuMemExportToShareableHandle.....	280
cuMemGetAccess.....	281
cuMemGetAllocationGranularity.....	281

cuMemGetAllocationPropertiesFromHandle.....	282
cuMemImportFromShareableHandle.....	283
cuMemMap.....	284
cuMemMapArrayAsync.....	285
cuMemRelease.....	288
cuMemRetainAllocationHandle.....	289
cuMemSetAccess.....	289
cuMemUnmap.....	290
6.15. Stream Ordered Memory Allocator.....	291
cuMemAllocAsync.....	292
cuMemAllocFromPoolAsync.....	293
cuMemFreeAsync.....	294
cuMemPoolCreate.....	294
cuMemPoolDestroy.....	296
cuMemPoolExportPointer.....	296
cuMemPoolExportToShareableHandle.....	297
cuMemPoolGetAccess.....	298
cuMemPoolGetAttribute.....	298
cuMemPoolImportFromShareableHandle.....	300
cuMemPoolImportPointer.....	301
cuMemPoolSetAccess.....	302
cuMemPoolSetAttribute.....	302
cuMemPoolTrimTo.....	303
6.16. Multicast Object Management.....	304
cuMulticastAddDevice.....	305
cuMulticastBindAddr.....	306
cuMulticastBindMem.....	307
cuMulticastCreate.....	308
cuMulticastGetGranularity.....	309
cuMulticastUnbind.....	310
6.17. Unified Addressing.....	311
cuMemAdvise.....	312
cuMemAdvise_v2.....	315
cuMemPrefetchAsync.....	319
cuMemPrefetchAsync_v2.....	320
cuMemRangeGetAttribute.....	322
cuMemRangeGetAttributes.....	325
cuPointerGetAttribute.....	326

cuPointerGetAttributes.....	329
cuPointerSetAttribute.....	331
6.18. Stream Management.....	332
cuStreamAddCallback.....	332
cuStreamAttachMemAsync.....	333
cuStreamBeginCapture.....	335
cuStreamBeginCaptureToGraph.....	336
cuStreamCopyAttributes.....	338
cuStreamCreate.....	338
cuStreamCreateWithPriority.....	339
cuStreamDestroy.....	340
cuStreamEndCapture.....	341
cuStreamGetAttribute.....	342
cuStreamGetCaptureInfo.....	342
cuStreamGetCaptureInfo_v3.....	344
cuStreamGetCtx.....	345
cuStreamGetCtx_v2.....	346
cuStreamGetDevice.....	348
cuStreamGetFlags.....	348
cuStreamGetId.....	349
cuStreamGetPriority.....	350
cuStreamIsCapturing.....	351
cuStreamQuery.....	352
cuStreamSetAttribute.....	352
cuStreamSynchronize.....	353
cuStreamUpdateCaptureDependencies.....	354
cuStreamUpdateCaptureDependencies_v2.....	355
cuStreamWaitEvent.....	356
cuThreadExchangeStreamCaptureMode.....	357
6.19. Event Management.....	358
cuEventCreate.....	358
cuEventDestroy.....	359
cuEventElapsedTime.....	360
cuEventElapsedTime_v2.....	361
cuEventQuery.....	362
cuEventRecord.....	363
cuEventRecordWithFlags.....	364
cuEventSynchronize.....	365

6.20. External Resource Interoperability.....	365
cuDestroyExternalMemory.....	366
cuDestroyExternalSemaphore.....	366
cuExternalMemoryGetMappedBuffer.....	367
cuExternalMemoryGetMappedMipmappedArray.....	368
cuImportExternalMemory.....	369
cuImportExternalSemaphore.....	372
cuSignalExternalSemaphoresAsync.....	375
cuWaitExternalSemaphoresAsync.....	377
6.21. Stream Memory Operations.....	379
cuStreamBatchMemOp.....	380
cuStreamWaitValue32.....	381
cuStreamWaitValue64.....	382
cuStreamWriteValue32.....	383
cuStreamWriteValue64.....	384
6.22. Execution Control.....	385
cuFuncGetAttribute.....	385
cuFuncGetModule.....	387
cuFuncGetName.....	388
cuFuncGetParamInfo.....	388
cuFuncIsLoaded.....	389
cuFuncLoad.....	390
cuFuncSetAttribute.....	390
cuFuncSetCacheConfig.....	392
cuLaunchCooperativeKernel.....	393
cuLaunchCooperativeKernelMultiDevice.....	395
cuLaunchHostFunc.....	398
cuLaunchKernel.....	399
cuLaunchKernelEx.....	402
6.23. Execution Control [DEPRECATED].....	406
cuFuncSetBlockShape.....	407
cuFuncSetSharedMemConfig.....	408
cuFuncSetSharedSize.....	409
cuLaunch.....	410
cuLaunchGrid.....	411
cuLaunchGridAsync.....	412
cuParamSetf.....	413
cuParamSeti.....	414

cuParamSetSize.....	414
cuParamSetTexRef.....	415
cuParamSetv.....	416
6.24. Graph Management.....	417
cuDeviceGetGraphMemAttribute.....	417
cuDeviceGraphMemTrim.....	418
cuDeviceSetGraphMemAttribute.....	418
cuGraphAddBatchMemOpNode.....	419
cuGraphAddChildGraphNode.....	420
cuGraphAddDependencies.....	421
cuGraphAddDependencies_v2.....	422
cuGraphAddEmptyNode.....	423
cuGraphAddEventRecordNode.....	424
cuGraphAddEventWaitNode.....	425
cuGraphAddExternalSemaphoresSignalNode.....	426
cuGraphAddExternalSemaphoresWaitNode.....	428
cuGraphAddHostNode.....	429
cuGraphAddKernelNode.....	430
cuGraphAddMemAllocNode.....	432
cuGraphAddMemcpyNode.....	434
cuGraphAddMemFreeNode.....	435
cuGraphAddMemsetNode.....	437
cuGraphAddNode.....	438
cuGraphAddNode_v2.....	439
cuGraphBatchMemOpNodeGetParams.....	440
cuGraphBatchMemOpNodeSetParams.....	441
cuGraphChildGraphNodeGetGraph.....	442
cuGraphClone.....	442
cuGraphConditionalHandleCreate.....	443
cuGraphCreate.....	444
cuGraphDebugDotPrint.....	445
cuGraphDestroy.....	445
cuGraphDestroyNode.....	446
cuGraphEventRecordNodeGetEvent.....	447
cuGraphEventRecordNodeSetEvent.....	447
cuGraphEventWaitNodeGetEvent.....	448
cuGraphEventWaitNodeSetEvent.....	449
cuGraphExecBatchMemOpNodeSetParams.....	450

cuGraphExecChildGraphNodeSetParams.....	451
cuGraphExecDestroy.....	452
cuGraphExecEventRecordNodeSetEvent.....	453
cuGraphExecEventWaitNodeSetEvent.....	454
cuGraphExecExternalSemaphoresSignalNodeSetParams.....	455
cuGraphExecExternalSemaphoresWaitNodeSetParams.....	456
cuGraphExecGetFlags.....	457
cuGraphExecHostNodeSetParams.....	458
cuGraphExecKernelNodeSetParams.....	459
cuGraphExecMemcpyNodeSetParams.....	460
cuGraphExecMemsetNodeSetParams.....	461
cuGraphExecNodeSetParams.....	462
cuGraphExecUpdate.....	464
cuGraphExternalSemaphoresSignalNodeGetParams.....	467
cuGraphExternalSemaphoresSignalNodeSetParams.....	468
cuGraphExternalSemaphoresWaitNodeGetParams.....	469
cuGraphExternalSemaphoresWaitNodeSetParams.....	470
cuGraphGetEdges.....	471
cuGraphGetEdges_v2.....	472
cuGraphGetNodes.....	473
cuGraphGetRootNodes.....	474
cuGraphHostNodeGetParams.....	475
cuGraphHostNodeSetParams.....	475
cuGraphInstantiate.....	476
cuGraphInstantiateWithParams.....	478
cuGraphKernelNodeCopyAttributes.....	480
cuGraphKernelNodeGetAttribute.....	481
cuGraphKernelNodeGetParams.....	482
cuGraphKernelNodeSetAttribute.....	483
cuGraphKernelNodeSetParams.....	483
cuGraphLaunch.....	484
cuGraphMemAllocNodeGetParams.....	485
cuGraphMemcpyNodeGetParams.....	486
cuGraphMemcpyNodeSetParams.....	486
cuGraphMemFreeNodeGetParams.....	487
cuGraphMemsetNodeGetParams.....	488
cuGraphMemsetNodeSetParams.....	488
cuGraphNodeFindInClone.....	489

cuGraphNodeGetDependencies.....	490
cuGraphNodeGetDependencies_v2.....	491
cuGraphNodeGetDependentNodes.....	492
cuGraphNodeGetDependentNodes_v2.....	493
cuGraphNodeGetEnabled.....	494
cuGraphNodeGetType.....	495
cuGraphNodeSetEnabled.....	495
cuGraphNodeSetParams.....	497
cuGraphReleaseUserObject.....	497
cuGraphRemoveDependencies.....	498
cuGraphRemoveDependencies_v2.....	499
cuGraphRetainUserObject.....	500
cuGraphUpload.....	501
cuUserObjectCreate.....	502
cuUserObjectRelease.....	503
cuUserObjectRetain.....	503
6.25. Occupancy.....	504
cuOccupancyAvailableDynamicSMemPerBlock.....	504
cuOccupancyMaxActiveBlocksPerMultiprocessor.....	505
cuOccupancyMaxActiveBlocksPerMultiprocessorWithFlags.....	506
cuOccupancyMaxActiveClusters.....	507
cuOccupancyMaxPotentialBlockSize.....	508
cuOccupancyMaxPotentialBlockSizeWithFlags.....	510
cuOccupancyMaxPotentialClusterSize.....	511
6.26. Texture Reference Management [DEPRECATED].....	512
cuTexRefCreate.....	512
cuTexRefDestroy.....	513
cuTexRefGetAddress.....	513
cuTexRefGetAddressMode.....	514
cuTexRefGetArray.....	515
cuTexRefGetBorderColor.....	515
cuTexRefGetFilterMode.....	516
cuTexRefGetFlags.....	517
cuTexRefGetFormat.....	517
cuTexRefGetMaxAnisotropy.....	518
cuTexRefGetMipmapFilterMode.....	519
cuTexRefGetMipmapLevelBias.....	519
cuTexRefGetMipmapLevelClamp.....	520

cuTexRefGetMipmappedArray.....	521
cuTexRefSetAddress.....	521
cuTexRefSetAddress2D.....	522
cuTexRefSetAddressMode.....	524
cuTexRefSetArray.....	525
cuTexRefSetBorderColor.....	525
cuTexRefSetFilterMode.....	526
cuTexRefSetFlags.....	527
cuTexRefSetFormat.....	528
cuTexRefSetMaxAnisotropy.....	528
cuTexRefSetMipmapFilterMode.....	529
cuTexRefSetMipmapLevelBias.....	530
cuTexRefSetMipmapLevelClamp.....	530
cuTexRefSetMipmappedArray.....	531
6.27. Surface Reference Management [DEPRECATED].....	532
cuSurfRefGetArray.....	532
cuSurfRefSetArray.....	533
6.28. Texture Object Management.....	533
cuTexObjectCreate.....	534
cuTexObjectDestroy.....	538
cuTexObjectGetResourceDesc.....	539
cuTexObjectGetResourceViewDesc.....	539
cuTexObjectGetTextureDesc.....	540
6.29. Surface Object Management.....	540
cuSurfObjectCreate.....	541
cuSurfObjectDestroy.....	541
cuSurfObjectGetResourceDesc.....	542
6.30. Tensor Map Object Management.....	542
cuTensorMapEncodeIm2col.....	543
cuTensorMapEncodeIm2colWide.....	548
cuTensorMapEncodeTiled.....	553
cuTensorMapReplaceAddress.....	557
6.31. Peer Context Memory Access.....	558
cuCtxDisablePeerAccess.....	558
cuCtxEnablePeerAccess.....	559
cuDeviceCanAccessPeer.....	560
cuDeviceGetP2PAttribute.....	561
6.32. Graphics Interoperability.....	562

cuGraphicsMapResources.....	562
cuGraphicsResourceGetMappedMipmappedArray.....	563
cuGraphicsResourceGetMappedPointer.....	564
cuGraphicsResourceSetMapFlags.....	565
cuGraphicsSubResourceGetMappedArray.....	566
cuGraphicsUnmapResources.....	567
cuGraphicsUnregisterResource.....	568
6.33. Driver Entry Point Access.....	568
cuGetProcAddress.....	569
6.34. Coredump Attributes Control API.....	570
CUCoredumpGenerationFlags.....	570
CUCoredumpSettings.....	571
cuCoredumpGetAttribute.....	571
cuCoredumpGetAttributeGlobal.....	573
cuCoredumpSetAttribute.....	574
cuCoredumpSetAttributeGlobal.....	576
6.35. Green Contexts.....	578
CUdevResource.....	579
CUdevSmResource.....	579
CUdevResourceType.....	579
CUdevResourceDesc.....	579
cuCtxFromGreenCtx.....	579
cuCtxGetDevResource.....	580
cuDeviceGetDevResource.....	581
cuDevResourceGenerateDesc.....	581
cuDevSmResourceSplitByCount.....	582
cuGreenCtxCreate.....	584
cuGreenCtxDestroy.....	585
cuGreenCtxGetDevResource.....	586
cuGreenCtxRecordEvent.....	586
cuGreenCtxStreamCreate.....	587
cuGreenCtxWaitEvent.....	588
cuStreamGetGreenCtx.....	589
6.36. CUDA Checkpointing.....	590
cuCheckpointProcessCheckpoint.....	590
cuCheckpointProcessGetRestoreThreadId.....	591
cuCheckpointProcessGetState.....	591
cuCheckpointProcessLock.....	592

cuCheckpointProcessRestore.....	592
cuCheckpointProcessUnlock.....	593
6.37. Profiler Control [DEPRECATED].....	594
cuProfilerInitialize.....	594
6.38. Profiler Control.....	595
cuProfilerStart.....	595
cuProfilerStop.....	595
6.39. OpenGL Interoperability.....	596
OpenGL Interoperability [DEPRECATED].....	596
CUGLDeviceList.....	596
cuGLGetDevices.....	597
cuGraphicsGLRegisterBuffer.....	598
cuGraphicsGLRegisterImage.....	599
cuWGLGetDevice.....	600
6.39.1. OpenGL Interoperability [DEPRECATED].....	601
CUGLmap_flags.....	601
cuGLCtxCreate.....	601
cuGLInit.....	602
cuGLMapBufferObject.....	603
cuGLMapBufferObjectAsync.....	604
cuGLRegisterBufferObject.....	605
cuGLSetBufferObjectMapFlags.....	605
cuGLUnmapBufferObject.....	606
cuGLUnmapBufferObjectAsync.....	607
cuGLUnregisterBufferObject.....	608
6.40. Direct3D 9 Interoperability.....	608
Direct3D 9 Interoperability [DEPRECATED].....	609
CUd3d9DeviceList.....	609
cuD3D9CtxCreate.....	609
cuD3D9CtxCreateOnDevice.....	610
cuD3D9GetDevice.....	611
cuD3D9GetDevices.....	612
cuD3D9GetDirect3DDevice.....	613
cuGraphicsD3D9RegisterResource.....	614
6.40.1. Direct3D 9 Interoperability [DEPRECATED].....	616
CUd3d9map_flags.....	616
CUd3d9register_flags.....	616
cuD3D9MapResources.....	616

cuD3D9RegisterResource.....	617
cuD3D9ResourceGetMappedArray.....	619
cuD3D9ResourceGetMappedPitch.....	620
cuD3D9ResourceGetMappedPointer.....	621
cuD3D9ResourceGetMappedSize.....	622
cuD3D9ResourceGetSurfaceDimensions.....	623
cuD3D9ResourceSetMapFlags.....	625
cuD3D9UnmapResources.....	626
cuD3D9UnregisterResource.....	627
6.41. Direct3D 10 Interoperability.....	627
Direct3D 10 Interoperability [DEPRECATED].....	627
CUd3d10DeviceList.....	627
cuD3D10GetDevice.....	628
cuD3D10GetDevices.....	629
cuGraphicsD3D10RegisterResource.....	630
6.41.1. Direct3D 10 Interoperability [DEPRECATED].....	632
CUD3D10map_flags.....	632
CUD3D10register_flags.....	632
cuD3D10CtxCreate.....	632
cuD3D10CtxCreateOnDevice.....	633
cuD3D10GetDirect3DDevice.....	634
cuD3D10MapResources.....	635
cuD3D10RegisterResource.....	636
cuD3D10ResourceGetMappedArray.....	637
cuD3D10ResourceGetMappedPitch.....	638
cuD3D10ResourceGetMappedPointer.....	639
cuD3D10ResourceGetMappedSize.....	640
cuD3D10ResourceGetSurfaceDimensions.....	641
cuD3D10ResourceSetMapFlags.....	642
cuD3D10UnmapResources.....	643
cuD3D10UnregisterResource.....	644
6.42. Direct3D 11 Interoperability.....	645
Direct3D 11 Interoperability [DEPRECATED].....	645
CUd3d11DeviceList.....	645
cuD3D11GetDevice.....	646
cuD3D11GetDevices.....	646
cuGraphicsD3D11RegisterResource.....	648
6.42.1. Direct3D 11 Interoperability [DEPRECATED].....	650

cuD3D11CtxCreate.....	650
cuD3D11CtxCreateOnDevice.....	651
cuD3D11GetDirect3DDevice.....	652
6.43. VDPAU Interoperability.....	652
cuGraphicsVDPAURegisterOutputSurface.....	652
cuGraphicsVDPAURegisterVideoSurface.....	654
cuVDPAUCtxCreate.....	655
cuVDPAUGetDevice.....	656
6.44. EGL Interoperability.....	656
cuEGLStreamConsumerAcquireFrame.....	657
cuEGLStreamConsumerConnect.....	658
cuEGLStreamConsumerConnectWithFlags.....	658
cuEGLStreamConsumerDisconnect.....	659
cuEGLStreamConsumerReleaseFrame.....	660
cuEGLStreamProducerConnect.....	660
cuEGLStreamProducerDisconnect.....	661
cuEGLStreamProducerPresentFrame.....	662
cuEGLStreamProducerReturnFrame.....	663
cuEventCreateFromEGLSync.....	663
cuGraphicsEGLRegisterImage.....	664
cuGraphicsResourceGetMappedEglFrame.....	665
Chapter 7. Data Structures.....	667
CUaccessPolicyWindow_v1.....	669
base_ptr.....	669
hitProp.....	669
hitRatio.....	669
missProp.....	669
num_bytes.....	669
CUarrayMapInfo_v1.....	669
deviceBitMask.....	670
extentDepth.....	670
extentHeight.....	670
extentWidth.....	670
flags.....	670
layer.....	670
level.....	670
memHandleType.....	670
memOperationType.....	670

offset.....	670
offsetX.....	671
offsetY.....	671
offsetZ.....	671
reserved.....	671
resourceType.....	671
size.....	671
subresourceType.....	671
CUasyncNotificationInfo.....	671
CUcheckpointCheckpointArgs.....	671
reserved.....	672
CUcheckpointLockArgs.....	672
reserved0.....	672
reserved1.....	672
timeoutMs.....	672
CUcheckpointRestoreArgs.....	672
reserved.....	672
CUcheckpointUnlockArgs.....	672
reserved.....	672
CUctxCigParam.....	673
CUctxCreateParams.....	673
CUDA_ARRAY3D_DESCRIPTOR_v2.....	673
Depth.....	673
Flags.....	673
Format.....	673
Height.....	673
NumChannels.....	673
Width.....	674
CUDA_ARRAY_DESCRIPTOR_v2.....	674
Format.....	674
Height.....	674
NumChannels.....	674
Width.....	674
CUDA_ARRAY_MEMORY_REQUIREMENTS_v1.....	674
alignment.....	674
size.....	675
CUDA_ARRAY_SPARSE_PROPERTIES_v1.....	675
depth.....	675

flags.....	675
height.....	675
miptailFirstLevel.....	675
miptailSize.....	675
width.....	676
CUDA_CHILD_GRAPH_NODE_PARAMS.....	676
graph.....	676
CUDA_CONDITIONAL_NODE_PARAMS.....	676
ctx.....	676
handle.....	676
phGraph_out.....	677
size.....	677
type.....	677
CUDA_EVENT_RECORD_NODE_PARAMS.....	677
event.....	678
CUDA_EVENT_WAIT_NODE_PARAMS.....	678
event.....	678
CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v1.....	678
extSemArray.....	678
numExtSems.....	678
paramsArray.....	678
CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v2.....	679
extSemArray.....	679
numExtSems.....	679
paramsArray.....	679
CUDA_EXT_SEM_WAIT_NODE_PARAMS_v1.....	679
extSemArray.....	679
numExtSems.....	679
paramsArray.....	680
CUDA_EXT_SEM_WAIT_NODE_PARAMS_v2.....	680
extSemArray.....	680
numExtSems.....	680
paramsArray.....	680
CUDA_EXTERNAL_MEMORY_BUFFER_DESC_v1.....	680
flags.....	681
offset.....	681
size.....	681
CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1.....	681

fd.....	681
flags.....	681
handle.....	681
name.....	682
nvSciBufObject.....	682
size.....	682
type.....	682
win32.....	682
CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC_v1.....	683
arrayDesc.....	683
numLevels.....	683
offset.....	683
CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_v1.....	683
fd.....	683
flags.....	683
handle.....	684
name.....	684
nvSciSyncObj.....	684
type.....	684
win32.....	684
CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1.....	685
fence.....	685
fence.....	685
flags.....	685
key.....	685
keyedMutex.....	685
value.....	686
CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1.....	686
fence.....	686
flags.....	686
key.....	686
keyedMutex.....	686
nvSciSync.....	687
timeoutMs.....	687
value.....	687
CUDA_GRAPH_INSTANTIATE_PARAMS.....	687
flags.....	687
hErrNode_out.....	687

hUploadStream.....	687
result_out.....	688
CUDA_HOST_NODE_PARAMS_v1.....	688
fn.....	688
userData.....	688
CUDA_HOST_NODE_PARAMS_v2.....	688
fn.....	688
userData.....	688
CUDA_KERNEL_NODE_PARAMS_v1.....	688
blockDimX.....	689
blockDimY.....	689
blockDimZ.....	689
extra.....	689
func.....	689
gridDimX.....	689
gridDimY.....	689
gridDimZ.....	689
kernelParams.....	690
sharedMemBytes.....	690
CUDA_KERNEL_NODE_PARAMS_v2.....	690
blockDimX.....	690
blockDimY.....	690
blockDimZ.....	690
ctx.....	690
extra.....	690
func.....	691
gridDimX.....	691
gridDimY.....	691
gridDimZ.....	691
kern.....	691
kernelParams.....	691
sharedMemBytes.....	691
CUDA_KERNEL_NODE_PARAMS_v3.....	691
blockDimX.....	692
blockDimY.....	692
blockDimZ.....	692
ctx.....	692
extra.....	692

func.....	692
gridDimX.....	692
gridDimY.....	692
gridDimZ.....	693
kern.....	693
kernelParams.....	693
sharedMemBytes.....	693
CUDA_LAUNCH_PARAMS_v1.....	693
blockDimX.....	693
blockDimY.....	693
blockDimZ.....	693
function.....	693
gridDimX.....	694
gridDimY.....	694
gridDimZ.....	694
hStream.....	694
kernelParams.....	694
sharedMemBytes.....	694
CUDA_MEM_ALLOC_NODE_PARAMS_v1.....	694
accessDescCount.....	694
accessDescs.....	694
bytesize.....	695
dptr.....	695
poolProps.....	695
CUDA_MEM_ALLOC_NODE_PARAMS_v2.....	695
accessDescCount.....	695
accessDescs.....	695
bytesize.....	695
dptr.....	696
poolProps.....	696
CUDA_MEM_FREE_NODE_PARAMS.....	696
dptr.....	696
CUDA_MEMCPY2D_v2.....	696
dstArray.....	696
dstDevice.....	696
dstHost.....	696
dstMemoryType.....	697
dstPitch.....	697

dstXInBytes.....	697
dstY.....	697
Height.....	697
srcArray.....	697
srcDevice.....	697
srcHost.....	697
srcMemoryType.....	697
srcPitch.....	697
srcXInBytes.....	698
srcY.....	698
WidthInBytes.....	698
CUDA_MEMCPY3D_PEER_v1.....	698
Depth.....	698
dstArray.....	698
dstContext.....	698
dstDevice.....	698
dstHeight.....	698
dstHost.....	698
dstLOD.....	699
dstMemoryType.....	699
dstPitch.....	699
dstXInBytes.....	699
dstY.....	699
dstZ.....	699
Height.....	699
srcArray.....	699
srcContext.....	699
srcDevice.....	699
srcHeight.....	700
srcHost.....	700
srcLOD.....	700
srcMemoryType.....	700
srcPitch.....	700
srcXInBytes.....	700
srcY.....	700
srcZ.....	700
WidthInBytes.....	700
CUDA_MEMCPY3D_v2.....	700

Depth.....	701
dstArray.....	701
dstDevice.....	701
dstHeight.....	701
dstHost.....	701
dstLOD.....	701
dstMemoryType.....	701
dstPitch.....	701
dstXInBytes.....	701
dstY.....	701
dstZ.....	702
Height.....	702
reserved0.....	702
reserved1.....	702
srcArray.....	702
srcDevice.....	702
srcHeight.....	702
srcHost.....	702
srcLOD.....	702
srcMemoryType.....	702
srcPitch.....	703
srcXInBytes.....	703
srcY.....	703
srcZ.....	703
WidthInBytes.....	703
CUDA_MEMCPY_NODE_PARAMS.....	703
copyCtx.....	703
copyParams.....	703
flags.....	703
reserved.....	704
CUDA_MEMSET_NODE_PARAMS_v1.....	704
dst.....	704
elementSize.....	704
height.....	704
pitch.....	704
value.....	704
width.....	704
CUDA_MEMSET_NODE_PARAMS_v2.....	705

ctx.....	705
dst.....	705
elementSize.....	705
height.....	705
pitch.....	705
value.....	705
width.....	705
CUDA_POINTER_ATTRIBUTE_P2P_TOKENS_v1.....	706
CUDA_RESOURCE_DESC_v1.....	706
devPtr.....	706
flags.....	706
format.....	706
hArray.....	706
height.....	706
hMipmappedArray.....	706
numChannels.....	707
pitchInBytes.....	707
resType.....	707
sizeInBytes.....	707
width.....	707
CUDA_RESOURCE_VIEW_DESC_v1.....	707
depth.....	707
firstLayer.....	707
firstMipmapLevel.....	707
format.....	708
height.....	708
lastLayer.....	708
lastMipmapLevel.....	708
width.....	708
CUDA_TEXTURE_DESC_v1.....	708
addressMode.....	708
borderColor.....	708
filterMode.....	709
flags.....	709
maxAnisotropy.....	709
maxMipmapLevelClamp.....	709
minMipmapLevelClamp.....	709
mipmapFilterMode.....	709

mipmapLevelBias.....	709
CUdevprop_v1.....	709
clockRate.....	709
maxGridSize.....	710
maxThreadsDim.....	710
maxThreadsPerBlock.....	710
memPitch.....	710
regsPerBlock.....	710
sharedMemPerBlock.....	710
SIMDWidth.....	710
textureAlign.....	710
totalConstantMemory.....	710
CUdevResource.....	710
CUdevSmResource.....	711
smCount.....	711
CUeglFrame_v1.....	711
cuFormat.....	711
depth.....	711
eglColorFormat.....	711
frameType.....	711
height.....	711
numChannels.....	712
pArray.....	712
pitch.....	712
planeCount.....	712
pPitch.....	712
width.....	712
CUexecAffinityParam_v1.....	712
CUexecAffinitySmCount_v1.....	712
val.....	712
CUextent3D_v1.....	713
CUgraphEdgeData.....	713
from_port.....	713
reserved.....	713
to_port.....	713
type.....	713
CUgraphExecUpdateResultInfo_v1.....	714
errorFromNode.....	714

errorNode.....	714
result.....	714
CUGraphNodeParams.....	714
alloc.....	714
conditional.....	714
eventRecord.....	715
eventWait.....	715
extSemSignal.....	715
extSemWait.....	715
free.....	715
graph.....	715
host.....	715
kernel.....	715
memcpy.....	716
memOp.....	716
memset.....	716
reserved0.....	716
reserved1.....	716
reserved2.....	716
type.....	716
CUIpcEventHandle_v1.....	716
CUIpcMemHandle_v1.....	716
CUlaunchAttribute.....	717
id.....	717
value.....	717
CUlaunchAttributeValue.....	717
accessPolicyWindow.....	717
clusterDim.....	717
clusterSchedulingPolicyPreference.....	717
cooperative.....	718
deviceUpdatableKernelNode.....	718
launchCompletionEvent.....	718
memSyncDomain.....	718
memSyncDomainMap.....	718
preferredClusterDim.....	719
priority.....	719
programmaticEvent.....	719
programmaticStreamSerializationAllowed.....	719

sharedMemCarveout.....	719
syncPolicy.....	720
CUlaunchConfig.....	720
attrs.....	720
blockDimX.....	720
blockDimY.....	720
blockDimZ.....	720
gridDimX.....	720
gridDimY.....	720
gridDimZ.....	720
hStream.....	720
numAttrs.....	721
sharedMemBytes.....	721
CUlaunchMemSyncDomainMap.....	721
default_.....	721
remote.....	721
CUmemAccessDesc_v1.....	721
flags.....	721
location.....	721
CUmemAllocationProp_v1.....	722
compressionType.....	722
location.....	722
requestedHandleTypes.....	722
type.....	722
usage.....	722
win32HandleMetaData.....	722
CUmemcpy3DOperand_v1.....	723
array.....	723
layerHeight.....	723
locHint.....	723
ptr.....	723
rowLength.....	723
CUmemcpyAttributes_v1.....	723
dstLocHint.....	724
flags.....	724
srcAccessOrder.....	724
srcLocHint.....	724
CUmemDecompressParams.....	724

algo.....	724
dst.....	724
dstActBytes.....	725
dstNumBytes.....	725
src.....	725
srcNumBytes.....	725
CUmemFabricHandle_v1.....	725
CUmemLocation_v1.....	725
id.....	725
type.....	725
CUmemPoolProps_v1.....	726
allocType.....	726
handleTypes.....	726
location.....	726
maxSize.....	726
reserved.....	726
usage.....	726
win32SecurityAttributes.....	726
CUmemPoolPtrExportData_v1.....	726
CUmulticastObjectProp_v1.....	727
flags.....	727
handleTypes.....	727
numDevices.....	727
size.....	727
CUoffset3D_v1.....	727
CUstreamBatchMemOpParams_v1.....	727
CUtensorMap.....	727
7.14. Difference between the driver and runtime APIs.....	728
Chapter 8. Data Fields.....	729
Chapter 9. Deprecated List.....	745

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:

- ▶ [Data types used by CUDA driver](#)
- ▶ [Error Handling](#)
- ▶ [Initialization](#)
- ▶ [Version Management](#)
- ▶ [Device Management](#)
- ▶ [Device Management \[DEPRECATED\]](#)
- ▶ [Primary Context Management](#)
- ▶ [Context Management](#)
- ▶ [Context Management \[DEPRECATED\]](#)
- ▶ [Module Management](#)
- ▶ [Module Management \[DEPRECATED\]](#)
- ▶ [Library Management](#)
- ▶ [Memory Management](#)
- ▶ [Virtual Memory Management](#)
- ▶ [Stream Ordered Memory Allocator](#)
- ▶ [Multicast Object Management](#)
- ▶ [Unified Addressing](#)
- ▶ [Stream Management](#)
- ▶ [Event Management](#)
- ▶ [External Resource Interoperability](#)
- ▶ [Stream Memory Operations](#)
- ▶ [Execution Control](#)
- ▶ [Execution Control \[DEPRECATED\]](#)
- ▶ [Graph Management](#)
- ▶ [Occupancy](#)
- ▶ [Texture Reference Management \[DEPRECATED\]](#)

- ▶ [Surface Reference Management \[DEPRECATED\]](#)
- ▶ [Texture Object Management](#)
- ▶ [Surface Object Management](#)
- ▶ [Tensor Map Object Management](#)
- ▶ [Peer Context Memory Access](#)
- ▶ [Graphics Interoperability](#)
- ▶ [Driver Entry Point Access](#)
- ▶ [Coredump Attributes Control API](#)
- ▶ [Green Contexts](#)
- ▶ [CUDA Checkpointing](#)
- ▶ [Profiler Control \[DEPRECATED\]](#)
- ▶ [Profiler Control](#)
- ▶ [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](#)

6.1. Data types used by CUDA driver

struct CUaccessPolicyWindow_v1

struct CUarrayMapInfo_v1

struct CUasyncNotificationInfo

struct CUcheckpointCheckpointArgs

struct CUcheckpointLockArgs

struct CUcheckpointRestoreArgs

struct CUcheckpointUnlockArgs

struct CUctxCigParam

struct CUctxCreateParams

struct CUDA_ARRAY3D_DESCRIPTOR_v2

struct CUDA_ARRAY_DESCRIPTOR_v2

struct CUDA_ARRAY_MEMORY_REQUIREMENTS_v1

struct CUDA_ARRAY_SPARSE_PROPERTIES_v1

struct CUDA_BATCH_MEM_OP_NODE_PARAMS_v2

struct CUDA_CHILD_GRAPH_NODE_PARAMS

struct CUDA_CONDITIONAL_NODE_PARAMS

struct CUDA_EVENT_RECORD_NODE_PARAMS

struct CUDA_EVENT_WAIT_NODE_PARAMS

struct CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v1

struct CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v2

struct CUDA_EXT_SEM_WAIT_NODE_PARAMS_v1

struct CUDA_EXT_SEM_WAIT_NODE_PARAMS_v2

struct
CUDA_EXTERNAL_MEMORY_BUFFER_DESC_v1

struct
CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1

struct
CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC_v1

struct
CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_v1

struct
CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1

struct
CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1

struct CUDA_GRAPH_INSTANTIATE_PARAMS

struct CUDA_HOST_NODE_PARAMS_v1

struct CUDA_HOST_NODE_PARAMS_v2

struct CUDA_KERNEL_NODE_PARAMS_v1

struct CUDA_KERNEL_NODE_PARAMS_v2

struct CUDA_KERNEL_NODE_PARAMS_v3

struct CUDA_LAUNCH_PARAMS_v1

struct CUDA_MEM_ALLOC_NODE_PARAMS_v1

struct CUDA_MEM_ALLOC_NODE_PARAMS_v2

struct CUDA_MEM_FREE_NODE_PARAMS

struct CUDA_MEMCPY2D_v2

struct CUDA_MEMCPY3D_PEER_v1

struct CUDA_MEMCPY3D_v2

struct CUDA_MEMCPY_NODE_PARAMS

struct CUDA_MEMSET_NODE_PARAMS_v1

struct CUDA_MEMSET_NODE_PARAMS_v2

struct
CUDA_POINTER_ATTRIBUTE_P2P_TOKENS_v1

struct CUDA_RESOURCE_DESC_v1

struct CUDA_RESOURCE_VIEW_DESC_v1

struct CUDA_TEXTURE_DESC_v1

struct CUdevprop_v1

struct CUeglFrame_v1

struct CUexecAffinityParam_v1

struct CUexecAffinitySmCount_v1

struct CUextent3D_v1

struct CUgraphEdgeData

struct CUgraphExecUpdateResultInfo_v1

struct CUgraphNodeParams

struct CUipcEventHandle_v1

struct CUipcMemHandle_v1

struct CUlaunchAttribute

union CUlaunchAttributeValue

struct CUlaunchConfig

struct CUlaunchMemSyncDomainMap

struct CUmemAccessDesc_v1

struct CUmemAllocationProp_v1

struct CUmemcpy3DOperand_v1

struct CUmemcpyAttributes_v1

struct CUmemFabricHandle_v1

struct CUmemLocation_v1

struct CUmemPoolProps_v1

struct CUmemPoolPtrExportData_v1

struct CUmulticastObjectProp_v1

struct CUoffset3D_v1

union CUstreamBatchMemOpParams_v1

struct CUtensorMap

enum cl_context_flags

NVCL context scheduling flags

Values

NVCL_CTX_SCHED_AUTO = 0x00

Automatic scheduling

NVCL_CTX_SCHED_SPIN = 0x01

Set spin as default scheduling

NVCL_CTX_SCHED_YIELD = 0x02

Set yield as default scheduling

NVCL_CTX_SCHED_BLOCKING_SYNC = 0x04

Set blocking synchronization as default scheduling

enum cl_event_flags

NVCL event scheduling flags

Values

NVCL_EVENT_SCHED_AUTO = 0x00

Automatic scheduling

NVCL_EVENT_SCHED_SPIN = 0x01

Set spin as default scheduling

NVCL_EVENT_SCHED_YIELD = 0x02

Set yield as default scheduling

NVCL_EVENT_SCHED_BLOCKING_SYNC = 0x04

Set blocking synchronization as default scheduling

enum CUaccessProperty

Specifies performance hint with [CUaccessPolicyWindow](#) for hitProp and missProp members.

Values

CU_ACCESS_PROPERTY_NORMAL = 0

Normal cache persistence.

CU_ACCESS_PROPERTY_STREAMING = 1

Streaming access is less likely to persist from cache.

CU_ACCESS_PROPERTY_PERSISTING = 2

Persisting access is more likely to persist in cache.

enum CUaddress_mode

Texture reference addressing modes

Values

CU_TR_ADDRESS_MODE_WRAP = 0

Wrapping address mode

CU_TR_ADDRESS_MODE_CLAMP = 1

Clamp to edge address mode

CU_TR_ADDRESS_MODE_MIRROR = 2

Mirror address mode

CU_TR_ADDRESS_MODE_BORDER = 3

Border address mode

enum CUarray_cubemap_face

Array indices for cube faces

Values

CU_CUBEMAP_FACE_POSITIVE_X = 0x00
Positive X face of cubemap

CU_CUBEMAP_FACE_NEGATIVE_X = 0x01
Negative X face of cubemap

CU_CUBEMAP_FACE_POSITIVE_Y = 0x02
Positive Y face of cubemap

CU_CUBEMAP_FACE_NEGATIVE_Y = 0x03
Negative Y face of cubemap

CU_CUBEMAP_FACE_POSITIVE_Z = 0x04
Positive Z face of cubemap

CU_CUBEMAP_FACE_NEGATIVE_Z = 0x05
Negative Z face of cubemap

enum CUarray_format

Array formats

Values

CU_AD_FORMAT_UNSIGNED_INT8 = 0x01
Unsigned 8-bit integers

CU_AD_FORMAT_UNSIGNED_INT16 = 0x02
Unsigned 16-bit integers

CU_AD_FORMAT_UNSIGNED_INT32 = 0x03
Unsigned 32-bit integers

CU_AD_FORMAT_SIGNED_INT8 = 0x08
Signed 8-bit integers

CU_AD_FORMAT_SIGNED_INT16 = 0x09
Signed 16-bit integers

CU_AD_FORMAT_SIGNED_INT32 = 0x0a
Signed 32-bit integers

CU_AD_FORMAT_HALF = 0x10
16-bit floating point

CU_AD_FORMAT_FLOAT = 0x20
32-bit floating point

CU_AD_FORMAT_NV12 = 0xb0
8-bit YUV planar format, with 4:2:0 sampling

CU_AD_FORMAT_UNORM_INT8X1 = 0xc0

1 channel unsigned 8-bit normalized integer
CU_AD_FORMAT_UNORM_INT8X2 = 0xc1

2 channel unsigned 8-bit normalized integer
CU_AD_FORMAT_UNORM_INT8X4 = 0xc2

4 channel unsigned 8-bit normalized integer
CU_AD_FORMAT_UNORM_INT16X1 = 0xc3

1 channel unsigned 16-bit normalized integer
CU_AD_FORMAT_UNORM_INT16X2 = 0xc4

2 channel unsigned 16-bit normalized integer
CU_AD_FORMAT_UNORM_INT16X4 = 0xc5

4 channel unsigned 16-bit normalized integer
CU_AD_FORMAT_SNORM_INT8X1 = 0xc6

1 channel signed 8-bit normalized integer
CU_AD_FORMAT_SNORM_INT8X2 = 0xc7

2 channel signed 8-bit normalized integer
CU_AD_FORMAT_SNORM_INT8X4 = 0xc8

4 channel signed 8-bit normalized integer
CU_AD_FORMAT_SNORM_INT16X1 = 0xc9

1 channel signed 16-bit normalized integer
CU_AD_FORMAT_SNORM_INT16X2 = 0xca

2 channel signed 16-bit normalized integer
CU_AD_FORMAT_SNORM_INT16X4 = 0xcb

4 channel signed 16-bit normalized integer
CU_AD_FORMAT_BC1_UNORM = 0x91

4 channel unsigned normalized block-compressed (BC1 compression) format
CU_AD_FORMAT_BC1_UNORM_SRGB = 0x92

4 channel unsigned normalized block-compressed (BC1 compression) format with sRGB encoding
CU_AD_FORMAT_BC2_UNORM = 0x93

4 channel unsigned normalized block-compressed (BC2 compression) format
CU_AD_FORMAT_BC2_UNORM_SRGB = 0x94

4 channel unsigned normalized block-compressed (BC2 compression) format with sRGB encoding
CU_AD_FORMAT_BC3_UNORM = 0x95

4 channel unsigned normalized block-compressed (BC3 compression) format
CU_AD_FORMAT_BC3_UNORM_SRGB = 0x96

4 channel unsigned normalized block-compressed (BC3 compression) format with sRGB encoding
CU_AD_FORMAT_BC4_UNORM = 0x97

1 channel unsigned normalized block-compressed (BC4 compression) format
CU_AD_FORMAT_BC4_SNORM = 0x98

1 channel signed normalized block-compressed (BC4 compression) format
CU_AD_FORMAT_BC5_UNORM = 0x99

2 channel unsigned normalized block-compressed (BC5 compression) format
CU_AD_FORMAT_BC5_SNORM = 0x9a

2 channel signed normalized block-compressed (BC5 compression) format

CU_AD_FORMAT_BC6H_UF16 = 0x9b

3 channel unsigned half-float block-compressed (BC6H compression) format

CU_AD_FORMAT_BC6H_SF16 = 0x9c

3 channel signed half-float block-compressed (BC6H compression) format

CU_AD_FORMAT_BC7_UNORM = 0x9d

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

CU_AD_FORMAT_BC7_UNORM_SRGB = 0x9e

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

CU_AD_FORMAT_P010 = 0x9f

10-bit YUV planar format, with 4:2:0 sampling

CU_AD_FORMAT_P016 = 0xa1

16-bit YUV planar format, with 4:2:0 sampling

CU_AD_FORMAT_NV16 = 0xa2

8-bit YUV planar format, with 4:2:2 sampling

CU_AD_FORMAT_P210 = 0xa3

10-bit YUV planar format, with 4:2:2 sampling

CU_AD_FORMAT_P216 = 0xa4

16-bit YUV planar format, with 4:2:2 sampling

CU_AD_FORMAT_YUY2 = 0xa5

2 channel, 8-bit YUV packed planar format, with 4:2:2 sampling

CU_AD_FORMAT_Y210 = 0xa6

2 channel, 10-bit YUV packed planar format, with 4:2:2 sampling

CU_AD_FORMAT_Y216 = 0xa7

2 channel, 16-bit YUV packed planar format, with 4:2:2 sampling

CU_AD_FORMAT_AYUV = 0xa8

4 channel, 8-bit YUV packed planar format, with 4:4:4 sampling

CU_AD_FORMAT_Y410 = 0xa9

10-bit YUV packed planar format, with 4:4:4 sampling

CU_AD_FORMAT_Y416 = 0xb1

4 channel, 12-bit YUV packed planar format, with 4:4:4 sampling

CU_AD_FORMAT_Y444_PLANAR8 = 0xb2

3 channel 8-bit YUV planar format, with 4:4:4 sampling

CU_AD_FORMAT_Y444_PLANAR10 = 0xb3

3 channel 10-bit YUV planar format, with 4:4:4 sampling

CU_AD_FORMAT_YUV444_8bit_SemiPlanar = 0xb4

3 channel 8-bit YUV semi-planar format, with 4:4:4 sampling

CU_AD_FORMAT_YUV444_16bit_SemiPlanar = 0xb5

3 channel 16-bit YUV semi-planar format, with 4:4:4 sampling

CU_AD_FORMAT_UNORM_INT_101010_2 = 0x50

4 channel unorm R10G10B10A2 RGB format

CU_AD_FORMAT_MAX = 0x7FFFFFFF

enum CUarraySparseSubresourceType

Sparse subresource types

Values

CU_ARRAY_SPARSE_SUBRESOURCE_TYPE_SPARSE_LEVEL = 0

CU_ARRAY_SPARSE_SUBRESOURCE_TYPE_MIPTAIL = 1

enum CUasyncNotificationType

Types of async notification that can be sent

Values

CU_ASYNC_NOTIFICATION_TYPE_OVER_BUDGET = 0x1

enum CUclusterSchedulingPolicy

Cluster scheduling policies. These may be passed to [cuFuncSetAttribute](#) or [cuKernelSetAttribute](#)

Values

CU_CLUSTER_SCHEDULING_POLICY_DEFAULT = 0

the default policy

CU_CLUSTER_SCHEDULING_POLICY_SPREAD = 1

spread the blocks within a cluster to the SMs

CU_CLUSTER_SCHEDULING_POLICY_LOAD_BALANCING = 2

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

enum CUcomputemode

Compute Modes

Values

CU_COMPUTEMODE_DEFAULT = 0

Default compute mode (Multiple contexts allowed per device)

CU_COMPUTEMODE_PROHIBITED = 2

Compute-prohibited mode (No contexts can be created on this device at this time)

CU_COMPUTEMODE_EXCLUSIVE_PROCESS = 3

Compute-exclusive-process mode (Only one context used by a single process can be present on this device at a time)

enum CUctx_flags

Context creation flags

Values

CU_CTX_SCHED_AUTO = 0x00

Automatic scheduling

CU_CTX_SCHED_SPIN = 0x01

Set spin as default scheduling

CU_CTX_SCHED_YIELD = 0x02

Set yield as default scheduling

CU_CTX_SCHED_BLOCKING_SYNC = 0x04

Set blocking synchronization as default scheduling

CU_CTX_BLOCKING_SYNC = 0x04

Set blocking synchronization as default scheduling [Deprecated](#) This flag was deprecated as of CUDA 4.0 and was replaced with [CU_CTX_SCHED_BLOCKING_SYNC](#).

CU_CTX_SCHED_MASK = 0x07

CU_CTX_MAP_HOST = 0x08

[Deprecated](#) This flag was deprecated as of CUDA 11.0 and it no longer has any effect. All contexts as of CUDA 3.2 behave as though the flag is enabled.

CU_CTX_LMEM_RESIZE_TO_MAX = 0x10

Keep local memory allocation after launch

CU_CTX_COREDUMP_ENABLE = 0x20

Trigger coredumps from exceptions in this context

CU_CTX_USER_COREDUMP_ENABLE = 0x40

Enable user pipe to trigger coredumps in this context

CU_CTX_SYNC_MEMOPS = 0x80

Ensure synchronous memory operations on this context will synchronize

CU_CTX_FLAGS_MASK = 0xFF

enum CUDA_POINTER_ATTRIBUTE_ACCESS_FLAGS

Access flags that specify the level of access the current context's device has on the memory referenced.

Values

CU_POINTER_ATTRIBUTE_ACCESS_FLAG_NONE = 0x0

No access, meaning the device cannot access this memory at all, thus must be staged through accessible memory in order to complete certain operations

CU_POINTER_ATTRIBUTE_ACCESS_FLAG_READ = 0x1

Read-only access, meaning writes to this memory are considered invalid accesses and thus return error in that case.

CU_POINTER_ATTRIBUTE_ACCESS_FLAG_READWRITE = 0x3

Read-write access, the device has full read-write access to the memory

enum CUdevice_attribute

Device properties

Values

CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK = 1

Maximum number of threads per block

CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X = 2

Maximum block dimension X

CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y = 3

Maximum block dimension Y

CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z = 4

Maximum block dimension Z

CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X = 5

Maximum grid dimension X

CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y = 6

Maximum grid dimension Y

CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z = 7

Maximum grid dimension Z

CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK = 8

Maximum shared memory available per block in bytes

CU_DEVICE_ATTRIBUTE_SHARED_MEMORY_PER_BLOCK = 8

Deprecated, use CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK

CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY = 9

Memory available on device for __constant__ variables in a CUDA C kernel in bytes

CU_DEVICE_ATTRIBUTE_WARP_SIZE = 10

Warp size in threads

CU_DEVICE_ATTRIBUTE_MAX_PITCH = 11

Maximum pitch in bytes allowed by memory copies

CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK = 12

Maximum number of 32-bit registers available per block

CU_DEVICE_ATTRIBUTE_REGISTERS_PER_BLOCK = 12

Deprecated, use CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK

CU_DEVICE_ATTRIBUTE_CLOCK_RATE = 13

Typical clock frequency in kilohertz

CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT = 14

Alignment requirement for textures

CU_DEVICE_ATTRIBUTE_GPU_OVERLAP = 15

Device can possibly copy memory and execute a kernel concurrently. Deprecated. Use instead

CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT.

CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT = 16

Number of multiprocessors on device

CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT = 17

Specifies whether there is a run time limit on kernels

CU_DEVICE_ATTRIBUTE_INTEGRATED = 18

Device is integrated with host memory

CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY = 19

Device can map host memory into CUDA address space

CU_DEVICE_ATTRIBUTE_COMPUTE_MODE = 20

Compute mode (See [CUcomputemode](#) for details)

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH = 21

Maximum 1D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_WIDTH = 22

Maximum 2D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_HEIGHT = 23

Maximum 2D texture height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH = 24

Maximum 3D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT = 25

Maximum 3D texture height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH = 26

Maximum 3D texture depth

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH = 27

Maximum 2D layered texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT = 28

Maximum 2D layered texture height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS = 29

Maximum layers in a 2D layered texture

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_WIDTH = 27

Deprecated, use CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_HEIGHT = 28

Deprecated, use CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_NUMSLICES = 29

Deprecated, use CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS

CU_DEVICE_ATTRIBUTE_SURFACE_ALIGNMENT = 30

Alignment requirement for surfaces

CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS = 31

Device can possibly execute multiple kernels concurrently

CU_DEVICE_ATTRIBUTE_ECC_ENABLED = 32

Device has ECC support enabled

CU_DEVICE_ATTRIBUTE_PCI_BUS_ID = 33

PCI bus ID of the device

CU_DEVICE_ATTRIBUTE_PCI_DEVICE_ID = 34

PCI device ID of the device

CU_DEVICE_ATTRIBUTE_TCC_DRIVER = 35

Device is using TCC driver model

CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE = 36

Peak memory clock frequency in kilohertz

CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH = 37

Global memory bus width in bits

CU_DEVICE_ATTRIBUTE_L2_CACHE_SIZE = 38

Size of L2 cache in bytes

CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR = 39

Maximum resident threads per multiprocessor

CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT = 40

Number of asynchronous engines

CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING = 41

Device shares a unified address space with the host

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH = 42

Maximum 1D layered texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS = 43

Maximum layers in a 1D layered texture

CU_DEVICE_ATTRIBUTE_CAN_TEX2D_GATHER = 44

Deprecated, do not use.

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_GATHER_WIDTH = 45

Maximum 2D texture width if CUDA_ARRAY3D_TEXTURE_GATHER is set

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_GATHER_HEIGHT = 46

Maximum 2D texture height if CUDA_ARRAY3D_TEXTURE_GATHER is set

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH_ALTERNATE = 47

Alternate maximum 3D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT_ALTERNATE = 48

Alternate maximum 3D texture height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH_ALTERNATE = 49

Alternate maximum 3D texture depth

CU_DEVICE_ATTRIBUTE_PCI_DOMAIN_ID = 50

PCI domain ID of the device

CU_DEVICE_ATTRIBUTE_TEXTURE_PITCH_ALIGNMENT = 51

Pitch alignment requirement for textures

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_WIDTH = 52

Maximum cubemap texture width/height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_LAYERED_WIDTH = 53

Maximum cubemap layered texture width/height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_LAYERED_LAYERS = 54

Maximum layers in a cubemap layered texture

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_WIDTH = 55

Maximum 1D surface width

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_WIDTH = 56

Maximum 2D surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_HEIGHT = 57
 Maximum 2D surface height
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_WIDTH = 58
 Maximum 3D surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_HEIGHT = 59
 Maximum 3D surface height
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_DEPTH = 60
 Maximum 3D surface depth
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_WIDTH = 61
 Maximum 1D layered surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_LAYERS = 62
 Maximum layers in a 1D layered surface
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_WIDTH = 63
 Maximum 2D layered surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_HEIGHT = 64
 Maximum 2D layered surface height
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_LAYERS = 65
 Maximum layers in a 2D layered surface
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_WIDTH = 66
 Maximum cubemap surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED_WIDTH = 67
 Maximum cubemap layered surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED_LAYERS = 68
 Maximum layers in a cubemap layered surface
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LINEAR_WIDTH = 69
 Deprecated, do not use. Use [cudaDeviceGetTexture1DLinearMaxWidth\(\)](#) or [cuDeviceGetTexture1DLinearMaxWidth\(\)](#) instead.
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_WIDTH = 70
 Maximum 2D linear texture width
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_HEIGHT = 71
 Maximum 2D linear texture height
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_PITCH = 72
 Maximum 2D linear texture pitch in bytes
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_WIDTH = 73
 Maximum mipmapped 2D texture width
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_HEIGHT = 74
 Maximum mipmapped 2D texture height
CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MAJOR = 75
 Major compute capability version number
CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MINOR = 76
 Minor compute capability version number
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_MIPMAPPED_WIDTH = 77

Maximum mipmapped 1D texture width

CU_DEVICE_ATTRIBUTE_STREAM_PRIORITIES_SUPPORTED = 78

Device supports stream priorities

CU_DEVICE_ATTRIBUTE_GLOBAL_L1_CACHE_SUPPORTED = 79

Device supports caching globals in L1

CU_DEVICE_ATTRIBUTE_LOCAL_L1_CACHE_SUPPORTED = 80

Device supports caching locals in L1

CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_MULTIPROCESSOR = 81

Maximum shared memory available per multiprocessor in bytes

CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_MULTIPROCESSOR = 82

Maximum number of 32-bit registers available per multiprocessor

CU_DEVICE_ATTRIBUTE_MANAGED_MEMORY = 83

Device can allocate managed memory on this system

CU_DEVICE_ATTRIBUTE_MULTI_GPU_BOARD = 84

Device is on a multi-GPU board

CU_DEVICE_ATTRIBUTE_MULTI_GPU_BOARD_GROUP_ID = 85

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

CU_DEVICE_ATTRIBUTE_HOST_NATIVE_ATOMIC_SUPPORTED = 86

Link between the device and the host supports native atomic operations (this is a placeholder attribute, and is not supported on any current hardware)

CU_DEVICE_ATTRIBUTE_SINGLE_TO_DOUBLE_PRECISION_PERF_RATIO = 87

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

CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS = 88

Device supports coherently accessing pageable memory without calling `cudaHostRegister` on it

CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS = 89

Device can coherently access managed memory concurrently with the CPU

CU_DEVICE_ATTRIBUTE_COMPUTE_PREEMPTION_SUPPORTED = 90

Device supports compute preemption.

CU_DEVICE_ATTRIBUTE_CAN_USE_HOST_POINTER_FOR_REGISTERED_MEM = 91

Device can access host registered memory at the same virtual address as the CPU

CU_DEVICE_ATTRIBUTE_CAN_USE_STREAM_MEM_OPS_V1 = 92

Deprecated, along with v1 MemOps API, [cuStreamBatchMemOp](#) and related APIs are supported.

CU_DEVICE_ATTRIBUTE_CAN_USE_64_BIT_STREAM_MEM_OPS_V1 = 93

Deprecated, along with v1 MemOps API, 64-bit operations are supported in [cuStreamBatchMemOp](#) and related APIs.

CU_DEVICE_ATTRIBUTE_CAN_USE_STREAM_WAIT_VALUE_NOR_V1 = 94

Deprecated, along with v1 MemOps API, [CU_STREAM_WAIT_VALUE_NOR](#) is supported.

CU_DEVICE_ATTRIBUTE_COOPERATIVE_LAUNCH = 95

Device supports launching cooperative kernels via [cuLaunchCooperativeKernel](#)

CU_DEVICE_ATTRIBUTE_COOPERATIVE_MULTI_DEVICE_LAUNCH = 96

Deprecated, [cuLaunchCooperativeKernelMultiDevice](#) is deprecated.

CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK_OPTIN = 97

Maximum optin shared memory per block

CU_DEVICE_ATTRIBUTE_CAN_FLUSH_REMOTE_WRITES = 98

The [CU_STREAM_WAIT_VALUE_FLUSH](#) flag and the [CU_STREAM_MEM_OP_FLUSH_REMOTE_WRITES](#) MemOp are supported on the device. See [Stream Memory Operations](#) for additional details.

CU_DEVICE_ATTRIBUTE_HOST_REGISTER_SUPPORTED = 99

Device supports host memory registration via [cudaHostRegister](#).

CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES = 100

Device accesses pageable memory via the host's page tables.

CU_DEVICE_ATTRIBUTE_DIRECT_MANAGED_MEM_ACCESS_FROM_HOST = 101

The host can directly access managed memory on the device without migration.

CU_DEVICE_ATTRIBUTE_VIRTUAL_ADDRESS_MANAGEMENT_SUPPORTED = 102

Deprecated, Use

[CU_DEVICE_ATTRIBUTE_VIRTUAL_MEMORY_MANAGEMENT_SUPPORTED](#)

CU_DEVICE_ATTRIBUTE_VIRTUAL_MEMORY_MANAGEMENT_SUPPORTED = 102

Device supports virtual memory management APIs like [cuMemAddressReserve](#), [cuMemCreate](#), [cuMemMap](#) and related APIs

CU_DEVICE_ATTRIBUTE_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR_SUPPORTED = 103

Device supports exporting memory to a posix file descriptor with [cuMemExportToShareableHandle](#), if requested via [cuMemCreate](#)

CU_DEVICE_ATTRIBUTE_HANDLE_TYPE_WIN32_HANDLE_SUPPORTED = 104

Device supports exporting memory to a Win32 NT handle with [cuMemExportToShareableHandle](#), if requested via [cuMemCreate](#)

CU_DEVICE_ATTRIBUTE_HANDLE_TYPE_WIN32_KMT_HANDLE_SUPPORTED = 105

Device supports exporting memory to a Win32 KMT handle with [cuMemExportToShareableHandle](#), if requested via [cuMemCreate](#)

CU_DEVICE_ATTRIBUTE_MAX_BLOCKS_PER_MULTIPROCESSOR = 106

Maximum number of blocks per multiprocessor

CU_DEVICE_ATTRIBUTE_GENERIC_COMPRESSION_SUPPORTED = 107

Device supports compression of memory

CU_DEVICE_ATTRIBUTE_MAX_PERSISTING_L2_CACHE_SIZE = 108

Maximum L2 persisting lines capacity setting in bytes.

CU_DEVICE_ATTRIBUTE_MAX_ACCESS_POLICY_WINDOW_SIZE = 109

Maximum value of [CUaccessPolicyWindow::num](#) bytes.

CU_DEVICE_ATTRIBUTE_GPU_DIRECT_RDMA_WITH_CUDA_VMM_SUPPORTED = 110

Device supports specifying the GPUDirect RDMA flag with [cuMemCreate](#)

CU_DEVICE_ATTRIBUTE_RESERVED_SHARED_MEMORY_PER_BLOCK = 111

Shared memory reserved by CUDA driver per block in bytes

CU_DEVICE_ATTRIBUTE_SPARSE_CUDA_ARRAY_SUPPORTED = 112

Device supports sparse CUDA arrays and sparse CUDA mipmapped arrays

CU_DEVICE_ATTRIBUTE_READ_ONLY_HOST_REGISTER_SUPPORTED = 113

Device supports using the [cuMemHostRegister](#) flag CU_MEMHOSTREGISTER_READ_ONLY to register memory that must be mapped as read-only to the GPU

CU_DEVICE_ATTRIBUTE_TIMELINE_SEMAPHORE_INTEROP_SUPPORTED = 114

External timeline semaphore interop is supported on the device

CU_DEVICE_ATTRIBUTE_MEMORY_POOLS_SUPPORTED = 115

Device supports using the [cuMemAllocAsync](#) and cuMemPool family of APIs

CU_DEVICE_ATTRIBUTE_GPU_DIRECT_RDMA_SUPPORTED = 116

Device supports GPUDirect RDMA APIs, like `nvidia_p2p_get_pages` (see <https://docs.nvidia.com/cuda/gpudirect-rdma> for more information)

CU_DEVICE_ATTRIBUTE_GPU_DIRECT_RDMA_FLUSH_WRITES_OPTIONS = 117

The returned attribute shall be interpreted as a bitmask, where the individual bits are described by the [CUflushGPUDirectRDMAWritesOptions](#) enum

CU_DEVICE_ATTRIBUTE_GPU_DIRECT_RDMA_WRITES_ORDERING = 118

GPUDirect RDMA writes to the device do not need to be flushed for consumers within the scope indicated by the returned attribute. See [CUGPUDirectRDMAWritesOrdering](#) for the numerical values returned here.

CU_DEVICE_ATTRIBUTE_MEMPOOL_SUPPORTED_HANDLE_TYPES = 119

Handle types supported with mempool based IPC

CU_DEVICE_ATTRIBUTE_CLUSTER_LAUNCH = 120

Indicates device supports cluster launch

CU_DEVICE_ATTRIBUTE_DEFERRED_MAPPING_CUDA_ARRAY_SUPPORTED = 121

Device supports deferred mapping CUDA arrays and CUDA mipmapped arrays

CU_DEVICE_ATTRIBUTE_CAN_USE_64_BIT_STREAM_MEM_OPS = 122

64-bit operations are supported in [cuStreamBatchMemOp](#) and related MemOp APIs.

CU_DEVICE_ATTRIBUTE_CAN_USE_STREAM_WAIT_VALUE_NOR = 123

[CU_STREAM_WAIT_VALUE_NOR](#) is supported by MemOp APIs.

CU_DEVICE_ATTRIBUTE_DMA_BUF_SUPPORTED = 124

Device supports buffer sharing with `dma_buf` mechanism.

CU_DEVICE_ATTRIBUTE_IPC_EVENT_SUPPORTED = 125

Device supports IPC Events.

CU_DEVICE_ATTRIBUTE_MEM_SYNC_DOMAIN_COUNT = 126

Number of memory domains the device supports.

CU_DEVICE_ATTRIBUTE_TENSOR_MAP_ACCESS_SUPPORTED = 127

Device supports accessing memory using Tensor Map.

CU_DEVICE_ATTRIBUTE_HANDLE_TYPE_FABRIC_SUPPORTED = 128

Device supports exporting memory to a fabric handle with [cuMemExportToShareableHandle\(\)](#) or requested with [cuMemCreate\(\)](#)

CU_DEVICE_ATTRIBUTE_UNIFIED_FUNCTION_POINTERS = 129

Device supports unified function pointers.

CU_DEVICE_ATTRIBUTE_NUMA_CONFIG = 130

NUMA configuration of a device: value is of type [CUdeviceNumaConfig](#) enum

CU_DEVICE_ATTRIBUTE_NUMA_ID = 131

NUMA node ID of the GPU memory

CU_DEVICE_ATTRIBUTE_MULTICAST_SUPPORTED = 132

Device supports switch multicast and reduction operations.

CU_DEVICE_ATTRIBUTE_MPS_ENABLED = 133

Indicates if contexts created on this device will be shared via MPS

CU_DEVICE_ATTRIBUTE_HOST_NUMA_ID = 134

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

CU_DEVICE_ATTRIBUTE_D3D12_CIG_SUPPORTED = 135

Device supports CIG with D3D12.

CU_DEVICE_ATTRIBUTE_MEM_DECOMPRESS_ALGORITHM_MASK = 136

The returned value shall be interpreted as a bitmask, where the individual bits are described by the

[CUmemDecompressAlgorithm](#) enum.

CU_DEVICE_ATTRIBUTE_MEM_DECOMPRESS_MAXIMUM_LENGTH = 137

The returned value is the maximum length in bytes of a single decompress operation that is allowed.

CU_DEVICE_ATTRIBUTE_GPU_PCI_DEVICE_ID = 139

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

CU_DEVICE_ATTRIBUTE_GPU_PCI_SUBSYSTEM_ID = 140

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

CU_DEVICE_ATTRIBUTE_HOST_NUMA_MULTINODE_IPC_SUPPORTED = 143

Device supports HOST_NUMA location IPC between nodes in a multi-node system.

CU_DEVICE_ATTRIBUTE_MAX

enum CUdevice_P2PAttribute

P2P Attributes

Values

CU_DEVICE_P2P_ATTRIBUTE_PERFORMANCE_RANK = 0x01

A relative value indicating the performance of the link between two devices

CU_DEVICE_P2P_ATTRIBUTE_ACCESS_SUPPORTED = 0x02

P2P Access is enable

CU_DEVICE_P2P_ATTRIBUTE_NATIVE_ATOMIC_SUPPORTED = 0x03

Atomic operation over the link supported

CU_DEVICE_P2P_ATTRIBUTE_ACCESS_ACCESS_SUPPORTED = 0x04

[Deprecated](#) use CU_DEVICE_P2P_ATTRIBUTE_CUDA_ARRAY_ACCESS_SUPPORTED instead

CU_DEVICE_P2P_ATTRIBUTE_CUDA_ARRAY_ACCESS_SUPPORTED = 0x04

Accessing CUDA arrays over the link supported

enum CUdeviceNumaConfig

CUDA device NUMA configuration

Values

CU_DEVICE_NUMA_CONFIG_NONE = 0

The GPU is not a NUMA node

CU_DEVICE_NUMA_CONFIG_NUMA_NODE

The GPU is a NUMA node, CU_DEVICE_ATTRIBUTE_NUMA_ID contains its NUMA ID

enum CUdriverProcAddressFlags

Flags to specify search options. For more details see [cuGetProcAddress](#)

Values

CU_GET_PROC_ADDRESS_DEFAULT = 0

Default search mode for driver symbols.

CU_GET_PROC_ADDRESS_LEGACY_STREAM = 1<<0

Search for legacy versions of driver symbols.

CU_GET_PROC_ADDRESS_PER_THREAD_DEFAULT_STREAM = 1<<1

Search for per-thread versions of driver symbols.

enum CUdriverProcAddressQueryResult

Flags to indicate search status. For more details see [cuGetProcAddress](#)

Values

CU_GET_PROC_ADDRESS_SUCCESS = 0

Symbol was successfully found

CU_GET_PROC_ADDRESS_SYMBOL_NOT_FOUND = 1

Symbol was not found in search

CU_GET_PROC_ADDRESS_VERSION_NOT_SUFFICIENT = 2

Symbol was found but version supplied was not sufficient

enum CUeglColorFormat

CUDA EGL Color Format - The different planar and multiplanar formats currently supported for CUDA_EGL interops. Three channel formats are currently not supported for [CU_EGL_FRAME_TYPE_ARRAY](#)

Values

CU_EGL_COLOR_FORMAT_YUV420_PLANAR = 0x00

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.

CU_EGL_COLOR_FORMAT_YUV420_SEMIPLANAR = 0x01

Y, UV in two surfaces (UV as one surface) with VU byte ordering, width, height ratio same as YUV420Planar.

CU_EGL_COLOR_FORMAT_YUV422_PLANAR = 0x02

Y, U, V each in a separate surface, U/V width = 1/2 Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YUV422_SEMIPLANAR = 0x03

Y, UV in two surfaces with VU byte ordering, width, height ratio same as YUV422Planar.

CU_EGL_COLOR_FORMAT_RGB = 0x04

R/G/B three channels in one surface with BGR byte ordering. Only pitch linear format supported.

CU_EGL_COLOR_FORMAT_BGR = 0x05

R/G/B three channels in one surface with RGB byte ordering. Only pitch linear format supported.

CU_EGL_COLOR_FORMAT_ARGB = 0x06

R/G/B/A four channels in one surface with BGRA byte ordering.

CU_EGL_COLOR_FORMAT_RGBA = 0x07

R/G/B/A four channels in one surface with ABGR byte ordering.

CU_EGL_COLOR_FORMAT_L = 0x08

single luminance channel in one surface.

CU_EGL_COLOR_FORMAT_R = 0x09

single color channel in one surface.

CU_EGL_COLOR_FORMAT_YUV444_PLANAR = 0x0A

Y, U, V in three surfaces, each in a separate surface, U/V width = Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YUV444_SEMIPLANAR = 0x0B

Y, UV in two surfaces (UV as one surface) with VU byte ordering, width, height ratio same as YUV444Planar.

CU_EGL_COLOR_FORMAT_YUYV_422 = 0x0C

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

CU_EGL_COLOR_FORMAT_UYVY_422 = 0x0D

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

CU_EGL_COLOR_FORMAT_ABGR = 0x0E

R/G/B/A four channels in one surface with RGBA byte ordering.

CU_EGL_COLOR_FORMAT_BGRA = 0x0F

R/G/B/A four channels in one surface with ARGB byte ordering.

CU_EGL_COLOR_FORMAT_A = 0x10

Alpha color format - one channel in one surface.

CU_EGL_COLOR_FORMAT_RG = 0x11

R/G color format - two channels in one surface with GR byte ordering

CU_EGL_COLOR_FORMAT_AYUV = 0x12

Y, U, V, A four channels in one surface, interleaved as VUYA.

CU_EGL_COLOR_FORMAT_YVU444_SEMIPLANAR = 0x13

Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YVU422_SEMIPLANAR = 0x14

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.

CU_EGL_COLOR_FORMAT_YVU420_SEMIPLANAR = 0x15

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.

CU_EGL_COLOR_FORMAT_Y10V10U10_444_SEMIPLANAR = 0x16

Y10, V10U10 in two surfaces (VU as one surface) with UV byte ordering, U/V width = Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_Y10V10U10_420_SEMIPLANAR = 0x17

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.

CU_EGL_COLOR_FORMAT_Y12V12U12_444_SEMIPLANAR = 0x18

Y12, V12U12 in two surfaces (VU as one surface) with UV byte ordering, U/V width = Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_Y12V12U12_420_SEMIPLANAR = 0x19

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.

CU_EGL_COLOR_FORMAT_VYUY_ER = 0x1A

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

CU_EGL_COLOR_FORMAT_UYVY_ER = 0x1B

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

CU_EGL_COLOR_FORMAT_YUYV_ER = 0x1C

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

CU_EGL_COLOR_FORMAT_YVYU_ER = 0x1D

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

CU_EGL_COLOR_FORMAT_YUV_ER = 0x1E

Extended Range Y, U, V three channels in one surface, interleaved as VUY. Only pitch linear format supported.

CU_EGL_COLOR_FORMAT_YUVA_ER = 0x1F

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

CU_EGL_COLOR_FORMAT_AYUV_ER = 0x20

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

CU_EGL_COLOR_FORMAT_YUV444_PLANAR_ER = 0x21

Extended Range Y, U, V in three surfaces, U/V width = Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YUV422_PLANAR_ER = 0x22

Extended Range Y, U, V in three surfaces, U/V width = 1/2 Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YUV420_PLANAR_ER = 0x23

Extended Range Y, U, V in three surfaces, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_YUV444_SEMIPLANAR_ER = 0x24

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.

CU_EGL_COLOR_FORMAT_YUV422_SEMIPLANAR_ER = 0x25

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.

CU_EGL_COLOR_FORMAT_YUV420_SEMIPLANAR_ER = 0x26

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.

CU_EGL_COLOR_FORMAT_YVU444_PLANAR_ER = 0x27

Extended Range Y, V, U in three surfaces, U/V width = Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YVU422_PLANAR_ER = 0x28

Extended Range Y, V, U in three surfaces, U/V width = 1/2 Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YVU420_PLANAR_ER = 0x29

Extended Range Y, V, U in three surfaces, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_YVU444_SEMIPLANAR_ER = 0x2A

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.

CU_EGL_COLOR_FORMAT_YVU422_SEMIPLANAR_ER = 0x2B

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.

CU_EGL_COLOR_FORMAT_YVU420_SEMIPLANAR_ER = 0x2C

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.

CU_EGL_COLOR_FORMAT_BAYER_RGGB = 0x2D

Bayer format - one channel in one surface with interleaved RGGB ordering.

CU_EGL_COLOR_FORMAT_BAYER_BGGR = 0x2E

Bayer format - one channel in one surface with interleaved BGGR ordering.

CU_EGL_COLOR_FORMAT_BAYER_GRGB = 0x2F

Bayer format - one channel in one surface with interleaved GRGB ordering.

CU_EGL_COLOR_FORMAT_BAYER_GBRG = 0x30

Bayer format - one channel in one surface with interleaved GBRG ordering.

CU_EGL_COLOR_FORMAT_BAYER10_RGGB = 0x31

Bayer10 format - one channel in one surface with interleaved RGGB ordering. Out of 16 bits, 10 bits used 6 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER10_BGGR = 0x32

Bayer10 format - one channel in one surface with interleaved BGGR ordering. Out of 16 bits, 10 bits used 6 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER10_GRGB = 0x33

Bayer10 format - one channel in one surface with interleaved GRGB ordering. Out of 16 bits, 10 bits used 6 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER10_GBRG = 0x34

Bayer10 format - one channel in one surface with interleaved GBRG ordering. Out of 16 bits, 10 bits used 6 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_RGGB = 0x35

Bayer12 format - one channel in one surface with interleaved RGGB ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_BGGR = 0x36

Bayer12 format - one channel in one surface with interleaved BGGR ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_GRBG = 0x37

Bayer12 format - one channel in one surface with interleaved GRBG ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_GBRG = 0x38

Bayer12 format - one channel in one surface with interleaved GBRG ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER14_RGGB = 0x39

Bayer14 format - one channel in one surface with interleaved RGGB ordering. Out of 16 bits, 14 bits used 2 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER14_BGGR = 0x3A

Bayer14 format - one channel in one surface with interleaved BGGR ordering. Out of 16 bits, 14 bits used 2 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER14_GRBG = 0x3B

Bayer14 format - one channel in one surface with interleaved GRBG ordering. Out of 16 bits, 14 bits used 2 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER14_GBRG = 0x3C

Bayer14 format - one channel in one surface with interleaved GBRG ordering. Out of 16 bits, 14 bits used 2 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER20_RGGB = 0x3D

Bayer20 format - one channel in one surface with interleaved RGGB ordering. Out of 32 bits, 20 bits used 12 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER20_BGGR = 0x3E

Bayer20 format - one channel in one surface with interleaved BGGR ordering. Out of 32 bits, 20 bits used 12 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER20_GRBG = 0x3F

Bayer20 format - one channel in one surface with interleaved GRBG ordering. Out of 32 bits, 20 bits used 12 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER20_GBRG = 0x40

Bayer20 format - one channel in one surface with interleaved GBRG ordering. Out of 32 bits, 20 bits used 12 bits No-op.

CU_EGL_COLOR_FORMAT_YVU444_PLANAR = 0x41

Y, V, U in three surfaces, each in a separate surface, U/V width = Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YVU422_PLANAR = 0x42

Y, V, U in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YVU420_PLANAR = 0x43

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.

CU_EGL_COLOR_FORMAT_BAYER_ISP_RGGB = 0x44

Nvidia proprietary Bayer ISP format - one channel in one surface with interleaved RGGB ordering and mapped to opaque integer datatype.

CU_EGL_COLOR_FORMAT_BAYER_ISP_BGGR = 0x45

Nvidia proprietary Bayer ISP format - one channel in one surface with interleaved BGGR ordering and mapped to opaque integer datatype.

CU_EGL_COLOR_FORMAT_BAYER_ISP_GRBG = 0x46

Nvidia proprietary Bayer ISP format - one channel in one surface with interleaved GRBG ordering and mapped to opaque integer datatype.

CU_EGL_COLOR_FORMAT_BAYER_ISP_GBRG = 0x47

Nvidia proprietary Bayer ISP format - one channel in one surface with interleaved GBRG ordering and mapped to opaque integer datatype.

CU_EGL_COLOR_FORMAT_BAYER_BCCR = 0x48

Bayer format - one channel in one surface with interleaved BCCR ordering.

CU_EGL_COLOR_FORMAT_BAYER_RCCB = 0x49

Bayer format - one channel in one surface with interleaved RCCB ordering.

CU_EGL_COLOR_FORMAT_BAYER_CRBC = 0x4A

Bayer format - one channel in one surface with interleaved CRBC ordering.

CU_EGL_COLOR_FORMAT_BAYER_CBRC = 0x4B

Bayer format - one channel in one surface with interleaved CBRC ordering.

CU_EGL_COLOR_FORMAT_BAYER10_CCCC = 0x4C

Bayer10 format - one channel in one surface with interleaved CCCC ordering. Out of 16 bits, 10 bits used 6 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_BCCR = 0x4D

Bayer12 format - one channel in one surface with interleaved BCCR ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_RCCB = 0x4E

Bayer12 format - one channel in one surface with interleaved RCCB ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_CRBC = 0x4F

Bayer12 format - one channel in one surface with interleaved CRBC ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_CBRC = 0x50

Bayer12 format - one channel in one surface with interleaved CBRC ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_CCCC = 0x51

Bayer12 format - one channel in one surface with interleaved CCCC ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_Y = 0x52

Color format for single Y plane.

CU_EGL_COLOR_FORMAT_YUV420_SEMIPLANAR_2020 = 0x53

Y, UV in two surfaces (UV as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_YVU420_SEMIPLANAR_2020 = 0x54

Y, VU in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_YUV420_PLANAR_2020 = 0x55

Y, U, V each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_YVU420_PLANAR_2020 = 0x56

Y, V, U each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_YUV420_SEMIPLANAR_709 = 0x57

Y, UV in two surfaces (UV as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_YVU420_SEMIPLANAR_709 = 0x58

Y, VU in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_YUV420_PLANAR_709 = 0x59

Y, U, V each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_YVU420_PLANAR_709 = 0x5A

Y, V, U each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_Y10V10U10_420_SEMIPLANAR_709 = 0x5B

Y10, V10U10 in two surfaces (VU as one surface), U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_Y10V10U10_420_SEMIPLANAR_2020 = 0x5C

Y10, V10U10 in two surfaces (VU as one surface), U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_Y10V10U10_422_SEMIPLANAR_2020 = 0x5D

Y10, V10U10 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_Y10V10U10_422_SEMIPLANAR = 0x5E

Y10, V10U10 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_Y10V10U10_422_SEMIPLANAR_709 = 0x5F

Y10, V10U10 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_Y_ER = 0x60

Extended Range Color format for single Y plane.

CU_EGL_COLOR_FORMAT_Y_709_ER = 0x61

Extended Range Color format for single Y plane.

CU_EGL_COLOR_FORMAT_Y10_ER = 0x62

Extended Range Color format for single Y10 plane.

CU_EGL_COLOR_FORMAT_Y10_709_ER = 0x63

Extended Range Color format for single Y10 plane.

CU_EGL_COLOR_FORMAT_Y12_ER = 0x64

Extended Range Color format for single Y12 plane.

CU_EGL_COLOR_FORMAT_Y12_709_ER = 0x65

Extended Range Color format for single Y12 plane.

CU_EGL_COLOR_FORMAT_YUVA = 0x66

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

CU_EGL_COLOR_FORMAT_YUV = 0x67

Y, U, V three channels in one surface, interleaved as VUY. Only pitch linear format supported.

CU_EGL_COLOR_FORMAT_YVYU = 0x68

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

CU_EGL_COLOR_FORMAT_VYUY = 0x69

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

CU_EGL_COLOR_FORMAT_Y10V10U10_420_SEMIPLANAR_ER = 0x6A

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.

CU_EGL_COLOR_FORMAT_Y10V10U10_420_SEMIPLANAR_709_ER = 0x6B

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.

CU_EGL_COLOR_FORMAT_Y10V10U10_444_SEMIPLANAR_ER = 0x6C

Extended Range Y10, V10U10 in two surfaces (VU as one surface) U/V width = Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_Y10V10U10_444_SEMIPLANAR_709_ER = 0x6D

Extended Range Y10, V10U10 in two surfaces (VU as one surface) U/V width = Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_Y12V12U12_420_SEMIPLANAR_ER = 0x6E

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.

CU_EGL_COLOR_FORMAT_Y12V12U12_420_SEMIPLANAR_709_ER = 0x6F

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.

CU_EGL_COLOR_FORMAT_Y12V12U12_444_SEMIPLANAR_ER = 0x70

Extended Range Y12, V12U12 in two surfaces (VU as one surface) U/V width = Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_Y12V12U12_444_SEMIPLANAR_709_ER = 0x71

Extended Range Y12, V12U12 in two surfaces (VU as one surface) U/V width = Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_UYVY_709 = 0x72

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

CU_EGL_COLOR_FORMAT_UYVY_709_ER = 0x73

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

CU_EGL_COLOR_FORMAT_UYVY_2020 = 0x74

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

CU_EGL_COLOR_FORMAT_MAX

enum CUeglFrameType

CUDA EglFrame type - array or pointer

Values

CU_EGL_FRAME_TYPE_ARRAY = 0

Frame type CUDA array

CU_EGL_FRAME_TYPE_POINTER = 1

Frame type pointer

enum CUeglResourceLocationFlags

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 [CUeglResourceLocationFlags](#) to give a hint about the desired location.

[CU_EGL_RESOURCE_LOCATION_SYSMEM](#) - the frame data is made resident on the system memory to be accessed by CUDA.

[CU_EGL_RESOURCE_LOCATION_VIDMEM](#) - 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

CU_EGL_RESOURCE_LOCATION_SYSMEM = 0x00

Resource location sysmem

CU_EGL_RESOURCE_LOCATION_VIDMEM = 0x01

Resource location vidmem

enum CUevent_flags

Event creation flags

Values

CU_EVENT_DEFAULT = 0x0

Default event flag

CU_EVENT_BLOCKING_SYNC = 0x1

Event uses blocking synchronization

CU_EVENT_DISABLE_TIMING = 0x2

Event will not record timing data

CU_EVENT_INTERPROCESS = 0x4

Event is suitable for interprocess use. CU_EVENT_DISABLE_TIMING must be set

enum CUevent_record_flags

Event record flags

Values

CU_EVENT_RECORD_DEFAULT = 0x0

Default event record flag

CU_EVENT_RECORD_EXTERNAL = 0x1

When using stream capture, create an event record node instead of the default behavior. This flag is invalid when used outside of capture.

enum CUevent_sched_flags

Event sched flags

Values

CU_EVENT_SCHED_AUTO = 0x00

Automatic scheduling

CU_EVENT_SCHED_SPIN = 0x01

Set spin as default scheduling

CU_EVENT_SCHED_YIELD = 0x02

Set yield as default scheduling

CU_EVENT_SCHED_BLOCKING_SYNC = 0x04

Set blocking synchronization as default scheduling

enum CUevent_wait_flags

Event wait flags

Values

CU_EVENT_WAIT_DEFAULT = 0x0

Default event wait flag

CU_EVENT_WAIT_EXTERNAL = 0x1

When using stream capture, create an event wait node instead of the default behavior. This flag is invalid when used outside of capture.

enum CUexecAffinityType

Execution Affinity Types

Values

CU_EXEC_AFFINITY_TYPE_SM_COUNT = 0

Create a context with limited SMs.

CU_EXEC_AFFINITY_TYPE_MAX

enum CUexternalMemoryHandleType

External memory handle types

Values

CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD = 1

Handle is an opaque file descriptor

CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32 = 2

Handle is an opaque shared NT handle

CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT = 3

Handle is an opaque, globally shared handle

CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP = 4

Handle is a D3D12 heap object

CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE = 5

Handle is a D3D12 committed resource

CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE = 6

Handle is a shared NT handle to a D3D11 resource

CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE_KMT = 7

Handle is a globally shared handle to a D3D11 resource

CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF = 8

Handle is an NvSciBuf object

enum CUexternalSemaphoreHandleType

External semaphore handle types

Values

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD = 1

Handle is an opaque file descriptor

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32 = 2

Handle is an opaque shared NT handle

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT = 3

Handle is an opaque, globally shared handle

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE = 4

Handle is a shared NT handle referencing a D3D12 fence object

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE = 5

Handle is a shared NT handle referencing a D3D11 fence object

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC = 6

Opaque handle to NvSciSync Object

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX = 7

Handle is a shared NT handle referencing a D3D11 keyed mutex object

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX_KMT = 8

Handle is a globally shared handle referencing a D3D11 keyed mutex object

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_TIMELINE_SEMAPHORE_FD = 9

Handle is an opaque file descriptor referencing a timeline semaphore

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_TIMELINE_SEMAPHORE_WIN32 = 10

Handle is an opaque shared NT handle referencing a timeline semaphore

enum CUfilter_mode

Texture reference filtering modes

Values

CU_TR_FILTER_MODE_POINT = 0

Point filter mode

CU_TR_FILTER_MODE_LINEAR = 1

Linear filter mode

enum CUflushGPUDirectRDMAWritesOptions

Bitmasks for [CU_DEVICE_ATTRIBUTE_GPU_DIRECT_RDMA_FLUSH_WRITES_OPTIONS](#)

Values

CU_FLUSH_GPU_DIRECT_RDMA_WRITES_OPTION_HOST = 1<<0

[cuFlushGPUDirectRDMAWrites\(\)](#) and its CUDA Runtime API counterpart are supported on the device.

CU_FLUSH_GPU_DIRECT_RDMA_WRITES_OPTION_MEMOPS = 1<<1

The [CU_STREAM_WAIT_VALUE_FLUSH](#) flag and the [CU_STREAM_MEM_OP_FLUSH_REMOTE_WRITES](#) MemOp are supported on the device.

enum CUflushGPUDirectRDMAWritesScope

The scopes for [cuFlushGPUDirectRDMAWrites](#)

Values

CU_FLUSH_GPU_DIRECT_RDMA_WRITES_TO_OWNER = 100

Blocks until remote writes are visible to the CUDA device context owning the data.

CU_FLUSH_GPU_DIRECT_RDMA_WRITES_TO_ALL_DEVICES = 200

Blocks until remote writes are visible to all CUDA device contexts.

enum CUflushGPUDirectRDMAWritesTarget

The targets for [cuFlushGPUDirectRDMAWrites](#)

Values

CU_FLUSH_GPU_DIRECT_RDMA_WRITES_TARGET_CURRENT_CTX = 0

Sets the target for [cuFlushGPUDirectRDMAWrites\(\)](#) to the currently active CUDA device context.

enum CUfunc_cache

Function cache configurations

Values

CU_FUNC_CACHE_PREFER_NONE = 0x00

no preference for shared memory or L1 (default)

CU_FUNC_CACHE_PREFER_SHARED = 0x01

prefer larger shared memory and smaller L1 cache

CU_FUNC_CACHE_PREFER_L1 = 0x02

prefer larger L1 cache and smaller shared memory

CU_FUNC_CACHE_PREFER_EQUAL = 0x03

prefer equal sized L1 cache and shared memory

enum CUfunction_attribute

Function properties

Values

CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK = 0

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.

CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES = 1

The size in bytes of statically-allocated shared memory required by this function. This does not include dynamically-allocated shared memory requested by the user at runtime.

CU_FUNC_ATTRIBUTE_CONST_SIZE_BYTES = 2

The size in bytes of user-allocated constant memory required by this function.

CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES = 3

The size in bytes of local memory used by each thread of this function.

CU_FUNC_ATTRIBUTE_NUM_REGS = 4

The number of registers used by each thread of this function.

CU_FUNC_ATTRIBUTE_PTX_VERSION = 5

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. Note that this may return the undefined value of 0 for cubins compiled prior to CUDA 3.0.

CU_FUNC_ATTRIBUTE_BINARY_VERSION = 6

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. Note that this will return a value of 10 for legacy cubins that do not have a properly-encoded binary architecture version.

CU_FUNC_ATTRIBUTE_CACHE_MODE_CA = 7

The attribute to indicate whether the function has been compiled with user specified option "-Xptxas --dlcm=ca" set .

CU_FUNC_ATTRIBUTE_MAX_DYNAMIC_SHARED_SIZE_BYTES = 8

The maximum size in bytes of dynamically-allocated shared memory that can be used by this function. If the user-specified dynamic shared memory size is larger than this value, the launch will fail. See [cuFuncSetAttribute](#), [cuKernelSetAttribute](#)

CU_FUNC_ATTRIBUTE_PREFERRED_SHARED_MEMORY_CARVEOUT = 9

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. Refer to [CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_MULTIPROCESSOR](#). This is only a hint, and the driver can choose a different ratio if required to execute the function. See [cuFuncSetAttribute](#), [cuKernelSetAttribute](#)

CU_FUNC_ATTRIBUTE_CLUSTER_SIZE_MUST_BE_SET = 10

If this attribute is set, the kernel must launch with a valid cluster size specified. See [cuFuncSetAttribute](#), [cuKernelSetAttribute](#)

CU_FUNC_ATTRIBUTE_REQUIRED_CLUSTER_WIDTH = 11

The required cluster width 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 will return CUDA_ERROR_NOT_PERMITTED. See [cuFuncSetAttribute](#), [cuKernelSetAttribute](#)

CU_FUNC_ATTRIBUTE_REQUIRED_CLUSTER_HEIGHT = 12

The required cluster height 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 CUDA_ERROR_NOT_PERMITTED. See [cuFuncSetAttribute](#), [cuKernelSetAttribute](#)

CU_FUNC_ATTRIBUTE_REQUIRED_CLUSTER_DEPTH = 13

The required cluster 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 CUDA_ERROR_NOT_PERMITTED. See [cuFuncSetAttribute](#), [cuKernelSetAttribute](#)

CU_FUNC_ATTRIBUTE_NON_PORTABLE_CLUSTER_SIZE_ALLOWED = 14

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 [cuFuncSetAttribute](#), [cuKernelSetAttribute](#)

CU_FUNC_ATTRIBUTE_CLUSTER_SCHEDULING_POLICY_PREFERENCE = 15

The block scheduling policy of a function. The value type is `CUclusterSchedulingPolicy` / `cudaClusterSchedulingPolicy`. See [cuFuncSetAttribute](#), [cuKernelSetAttribute](#)

CU_FUNC_ATTRIBUTE_MAX

enum CUGPUDirectRDMAWritesOrdering

Platform native ordering for GPUDirect RDMA writes

Values

CU_GPU_DIRECT_RDMA_WRITES_ORDERING_NONE = 0

The device does not natively support ordering of remote writes. [cuFlushGPUDirectRDMAWrites\(\)](#) can be leveraged if supported.

CU_GPU_DIRECT_RDMA_WRITES_ORDERING_OWNER = 100

Natively, the device can consistently consume remote writes, although other CUDA devices may not.

CU_GPU_DIRECT_RDMA_WRITES_ORDERING_ALL_DEVICES = 200

Any CUDA device in the system can consistently consume remote writes to this device.

enum CUgraphConditionalNodeType

Conditional node types

Values

CU_GRAPH_COND_TYPE_IF = 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.

CU_GRAPH_COND_TYPE_WHILE = 1

Conditional 'while' Node. Body executed repeatedly while condition value is non-zero.

CU_GRAPH_COND_TYPE_SWITCH = 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 CUgraphDebugDot_flags

The additional write options for [cuGraphDebugDotPrint](#)

Values

CU_GRAPH_DEBUG_DOT_FLAGS_VERBOSE = 1<<0

Output all debug data as if every debug flag is enabled

CU_GRAPH_DEBUG_DOT_FLAGS_RUNTIME_TYPES = 1<<1

Use CUDA Runtime structures for output

CU_GRAPH_DEBUG_DOT_FLAGS_KERNEL_NODE_PARAMS = 1<<2

Adds CUDA_KERNEL_NODE_PARAMS values to output

CU_GRAPH_DEBUG_DOT_FLAGS_MEMCPY_NODE_PARAMS = 1<<3

Adds CUDA_MEMCPY3D values to output

CU_GRAPH_DEBUG_DOT_FLAGS_MEMSET_NODE_PARAMS = 1<<4

Adds CUDA_MEMSET_NODE_PARAMS values to output

CU_GRAPH_DEBUG_DOT_FLAGS_HOST_NODE_PARAMS = 1<<5

Adds CUDA_HOST_NODE_PARAMS values to output

CU_GRAPH_DEBUG_DOT_FLAGS_EVENT_NODE_PARAMS = 1<<6

Adds CUevent handle from record and wait nodes to output

CU_GRAPH_DEBUG_DOT_FLAGS_EXT_SEMAS_SIGNAL_NODE_PARAMS = 1<<7

Adds CUDA_EXT_SEM_SIGNAL_NODE_PARAMS values to output

CU_GRAPH_DEBUG_DOT_FLAGS_EXT_SEMAS_WAIT_NODE_PARAMS = 1<<8

Adds CUDA_EXT_SEM_WAIT_NODE_PARAMS values to output

CU_GRAPH_DEBUG_DOT_FLAGS_KERNEL_NODE_ATTRIBUTES = 1<<9

Adds CUkernelNodeAttrValue values to output

CU_GRAPH_DEBUG_DOT_FLAGS_HANDLES = 1<<10

Adds node handles and every kernel function handle to output

CU_GRAPH_DEBUG_DOT_FLAGS_MEM_ALLOC_NODE_PARAMS = 1<<11

Adds memory alloc node parameters to output

CU_GRAPH_DEBUG_DOT_FLAGS_MEM_FREE_NODE_PARAMS = 1<<12

Adds memory free node parameters to output

CU_GRAPH_DEBUG_DOT_FLAGS_BATCH_MEM_OP_NODE_PARAMS = 1<<13

Adds batch mem op node parameters to output

CU_GRAPH_DEBUG_DOT_FLAGS_EXTRA_TOPO_INFO = 1<<14

Adds edge numbering information

CU_GRAPH_DEBUG_DOT_FLAGS_CONDITIONAL_NODE_PARAMS = 1<<15

Adds conditional node parameters to output

enum CUgraphDependencyType

Type annotations that can be applied to graph edges as part of [CUgraphEdgeData](#).

Values

CU_GRAPH_DEPENDENCY_TYPE_DEFAULT = 0

This is an ordinary dependency.

CU_GRAPH_DEPENDENCY_TYPE_PROGRAMMATIC = 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 [CU_GRAPH_KERNEL_NODE_PORT_PROGRAMMATIC](#) or [CU_GRAPH_KERNEL_NODE_PORT_LAUNCH_ORDER](#) outgoing port.

enum CUgraphExecUpdateResult

CUDA Graph Update error types

Values

CU_GRAPH_EXEC_UPDATE_SUCCESS = 0x0

The update succeeded

CU_GRAPH_EXEC_UPDATE_ERROR = 0x1

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

CU_GRAPH_EXEC_UPDATE_ERROR_TOPOLOGY_CHANGED = 0x2

The update failed because the topology changed

CU_GRAPH_EXEC_UPDATE_ERROR_NODE_TYPE_CHANGED = 0x3

The update failed because a node type changed

CU_GRAPH_EXEC_UPDATE_ERROR_FUNCTION_CHANGED = 0x4

The update failed because the function of a kernel node changed (CUDA driver < 11.2)

CU_GRAPH_EXEC_UPDATE_ERROR_PARAMETERS_CHANGED = 0x5

The update failed because the parameters changed in a way that is not supported

CU_GRAPH_EXEC_UPDATE_ERROR_NOT_SUPPORTED = 0x6

The update failed because something about the node is not supported

CU_GRAPH_EXEC_UPDATE_ERROR_UNSUPPORTED_FUNCTION_CHANGE = 0x7

The update failed because the function of a kernel node changed in an unsupported way

CU_GRAPH_EXEC_UPDATE_ERROR_ATTRIBUTES_CHANGED = 0x8

The update failed because the node attributes changed in a way that is not supported

enum CUgraphicsMapResourceFlags

Flags for mapping and unmapping interop resources

Values

CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE = 0x00

CU_GRAPHICS_MAP_RESOURCE_FLAGS_READ_ONLY = 0x01

CU_GRAPHICS_MAP_RESOURCE_FLAGS_WRITE_DISCARD = 0x02

enum CUgraphicsRegisterFlags

Flags to register a graphics resource

Values

CU_GRAPHICS_REGISTER_FLAGS_NONE = 0x00

CU_GRAPHICS_REGISTER_FLAGS_READ_ONLY = 0x01

CU_GRAPHICS_REGISTER_FLAGS_WRITE_DISCARD = 0x02

CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST = 0x04

CU_GRAPHICS_REGISTER_FLAGS_TEXTURE_GATHER = 0x08

enum CUgraphInstantiate_flags

Flags for instantiating a graph

Values

CUDA_GRAPH_INSTANTIATE_FLAG_AUTO_FREE_ON_LAUNCH = 1

Automatically free memory allocated in a graph before relaunching.

CUDA_GRAPH_INSTANTIATE_FLAG_UPLOAD = 2

Automatically upload the graph after instantiation. Only supported by [cuGraphInstantiateWithParams](#). The upload will be performed using the stream provided in `instantiateParams`.

CUDA_GRAPH_INSTANTIATE_FLAG_DEVICE_LAUNCH = 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 `CUDA_GRAPH_INSTANTIATE_FLAG_AUTO_FREE_ON_LAUNCH`.

CUDA_GRAPH_INSTANTIATE_FLAG_USE_NODE_PRIORITY = 8

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

enum CUgraphInstantiateResult

Graph instantiation results

Values

CUDA_GRAPH_INSTANTIATE_SUCCESS = 0

Instantiation succeeded

CUDA_GRAPH_INSTANTIATE_ERROR = 1

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

CUDA_GRAPH_INSTANTIATE_INVALID_STRUCTURE = 2

Instantiation failed due to invalid structure, such as cycles

CUDA_GRAPH_INSTANTIATE_NODE_OPERATION_NOT_SUPPORTED = 3

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

CUDA_GRAPH_INSTANTIATE_MULTIPLE_CTXS_NOT_SUPPORTED = 4

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

CUDA_GRAPH_INSTANTIATE_CONDITIONAL_HANDLE_UNUSED = 5

One or more conditional handles are not associated with conditional nodes

enum CUgraphNodeType

Graph node types

Values

CU_GRAPH_NODE_TYPE_KERNEL = 0

GPU kernel node

CU_GRAPH_NODE_TYPE_MEMCPY = 1

Memcpy node

CU_GRAPH_NODE_TYPE_MEMSET = 2

Memset node

CU_GRAPH_NODE_TYPE_HOST = 3

Host (executable) node

CU_GRAPH_NODE_TYPE_GRAPH = 4

Node which executes an embedded graph

CU_GRAPH_NODE_TYPE_EMPTY = 5

Empty (no-op) node

CU_GRAPH_NODE_TYPE_WAIT_EVENT = 6

External event wait node

CU_GRAPH_NODE_TYPE_EVENT_RECORD = 7

External event record node

CU_GRAPH_NODE_TYPE_EXT_SEMAS_SIGNAL = 8

External semaphore signal node

CU_GRAPH_NODE_TYPE_EXT_SEMAS_WAIT = 9

External semaphore wait node

CU_GRAPH_NODE_TYPE_MEM_ALLOC = 10

Memory Allocation Node

CU_GRAPH_NODE_TYPE_MEM_FREE = 11

Memory Free Node

CU_GRAPH_NODE_TYPE_BATCH_MEM_OP = 12

Batch MemOp Node

CU_GRAPH_NODE_TYPE_CONDITIONAL = 13

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 [cuGraphConditionalHandleCreate](#).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.

enum CUipcMem_flags

CUDA Ipc Mem Flags

Values

CU_IPC_MEM_LAZY_ENABLE_PEER_ACCESS = 0x1

Automatically enable peer access between remote devices as needed

enum CUjit_cacheMode

Caching modes for dlcm

Values

CU_JIT_CACHE_OPTION_NONE = 0

Compile with no -dlcm flag specified

CU_JIT_CACHE_OPTION_CG

Compile with L1 cache disabled

CU_JIT_CACHE_OPTION_CA

Compile with L1 cache enabled

enum CUjit_fallback

Cubin matching fallback strategies

Values

CU_PREFER_PTX = 0

Prefer to compile ptx if exact binary match not found

CU_PREFER_BINARY

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

enum CUjit_option

Online compiler and linker options

Values

CU_JIT_MAX_REGISTERS = 0

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

CU_JIT_THREADS_PER_BLOCK = 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. Cannot be combined with [CU_JIT_TARGET](#). Option type: unsigned int Applies to: compiler only

CU_JIT_WALL_TIME = 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

CU_JIT_INFO_LOG_BUFFER = 3

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

CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES = 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

CU_JIT_ERROR_LOG_BUFFER = 5

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

CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES = 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

CU_JIT_OPTIMIZATION_LEVEL = 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

CU_JIT_TARGET_FROM_CUCONTEXT = 8

No option value required. Determines the target based on the current attached context (default)
Option type: No option value needed Applies to: compiler and linker

CU_JIT_TARGET = 9

Target is chosen based on supplied [CUjit_target](#). Cannot be combined with [CU_JIT_THREADS_PER_BLOCK](#). Option type: unsigned int for enumerated type [CUjit_target](#)
Applies to: compiler and linker

CU_JIT_FALLBACK_STRATEGY = 10

Specifies choice of fallback strategy if matching cubin is not found. Choice is based on supplied [CUjit_fallback](#). This option cannot be used with cuLink* APIs as the linker requires exact matches.
Option type: unsigned int for enumerated type [CUjit_fallback](#) Applies to: compiler only

CU_JIT_GENERATE_DEBUG_INFO = 11

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

CU_JIT_LOG_VERBOSE = 12

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

CU_JIT_GENERATE_LINE_INFO = 13

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

CU_JIT_CACHE_MODE = 14

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

CU_JIT_NEW_SM3X_OPT = 15

Deprecated This jit option is deprecated and should not be used.

CU_JIT_FAST_COMPILE = 16

This jit option is used for internal purpose only.

CU_JIT_GLOBAL_SYMBOL_NAMES = 17

Array of device symbol names that will be relocated to the corresponding host addresses stored in [CU_JIT_GLOBAL_SYMBOL_ADDRESSES](#). Must contain [CU_JIT_GLOBAL_SYMBOL_COUNT](#) entries. When loading a device module, driver will relocate all encountered unresolved symbols to the host addresses. It is only allowed to register symbols that correspond to unresolved global variables. It is illegal to register the same device symbol at multiple addresses. Option type: const char ** Applies to: dynamic linker only

CU_JIT_GLOBAL_SYMBOL_ADDRESSES = 18

Array of host addresses that will be used to relocate corresponding device symbols stored in [CU_JIT_GLOBAL_SYMBOL_NAMES](#). Must contain [CU_JIT_GLOBAL_SYMBOL_COUNT](#) entries. Option type: void ** Applies to: dynamic linker only

CU_JIT_GLOBAL_SYMBOL_COUNT = 19

Number of entries in [CU_JIT_GLOBAL_SYMBOL_NAMES](#) and [CU_JIT_GLOBAL_SYMBOL_ADDRESSES](#) arrays. Option type: unsigned int Applies to: dynamic linker only

CU_JIT_LTO = 20

Deprecated Enable link-time optimization (-dlto) for device code (Disabled by default). This option is not supported on 32-bit platforms. Option type: int Applies to: compiler and linker Only valid with LTO-IR compiled with toolkits prior to CUDA 12.0

CU_JIT_FTZ = 21

Deprecated Control single-precision denormals (-ftz) support (0: false, default). 1 : flushes denormal values to zero 0 : preserves denormal values Option type: int Applies to: link-time optimization specified with CU_JIT_LTO Only valid with LTO-IR compiled with toolkits prior to CUDA 12.0

CU_JIT_PREC_DIV = 22

Deprecated Control single-precision floating-point division and reciprocals (-prec-div) support (1: true, default). 1 : Enables the IEEE round-to-nearest mode 0 : Enables the fast approximation mode Option type: int Applies to: link-time optimization specified with CU_JIT_LTO Only valid with LTO-IR compiled with toolkits prior to CUDA 12.0

CU_JIT_PREC_SQRT = 23

Deprecated Control single-precision floating-point square root (-prec-sqrt) support (1: true, default). 1 : Enables the IEEE round-to-nearest mode 0 : Enables the fast approximation mode Option type: int Applies to: link-time optimization specified with CU_JIT_LTO Only valid with LTO-IR compiled with toolkits prior to CUDA 12.0

CU_JIT_FMA = 24

Deprecated Enable/Disable the contraction of floating-point multiplies and adds/subtracts into floating-point multiply-add (-fma) operations (1: Enable, default; 0: Disable). Option type: int

Applies to: link-time optimization specified with CU_JIT_LTO Only valid with LTO-IR compiled with toolkits prior to CUDA 12.0

CU_JIT_REFERENCED_KERNEL_NAMES = 25

Deprecated Array of kernel names that should be preserved at link time while others can be removed. Must contain [CU_JIT_REFERENCED_KERNEL_COUNT](#) entries. Note that kernel names can be mangled by the compiler in which case the mangled name needs to be specified. Wildcard "*" can be used to represent zero or more characters instead of specifying the full or mangled name. It is important to note that the wildcard "*" is also added implicitly. For example, specifying "foo" will match "foobaz", "barfoo", "barfoobaz" and thus preserve all kernels with those names. This can be avoided by providing a more specific name like "barfoobaz". Option type: const char ** Applies to: dynamic linker only Only valid with LTO-IR compiled with toolkits prior to CUDA 12.0

CU_JIT_REFERENCED_KERNEL_COUNT = 26

Deprecated Number of entries in [CU_JIT_REFERENCED_KERNEL_NAMES](#) array. Option type: unsigned int Applies to: dynamic linker only Only valid with LTO-IR compiled with toolkits prior to CUDA 12.0

CU_JIT_REFERENCED_VARIABLE_NAMES = 27

Deprecated Array of variable names (__device__ and/or __constant__) that should be preserved at link time while others can be removed. Must contain [CU_JIT_REFERENCED_VARIABLE_COUNT](#) entries. Note that variable names can be mangled by the compiler in which case the mangled name needs to be specified. Wildcard "*" can be used to represent zero or more characters instead of specifying the full or mangled name. It is important to note that the wildcard "*" is also added implicitly. For example, specifying "foo" will match "foobaz", "barfoo", "barfoobaz" and thus preserve all variables with those names. This can be avoided by providing a more specific name like "barfoobaz". Option type: const char ** Applies to: link-time optimization specified with CU_JIT_LTO Only valid with LTO-IR compiled with toolkits prior to CUDA 12.0

CU_JIT_REFERENCED_VARIABLE_COUNT = 28

Deprecated Number of entries in [CU_JIT_REFERENCED_VARIABLE_NAMES](#) array. Option type: unsigned int Applies to: link-time optimization specified with CU_JIT_LTO Only valid with LTO-IR compiled with toolkits prior to CUDA 12.0

CU_JIT_OPTIMIZE_UNUSED_DEVICE_VARIABLES = 29

Deprecated This option serves as a hint to enable the JIT compiler/linker to remove constant (__constant__) and device (__device__) variables unreferenced in device code (Disabled by default). Note that host references to constant and device variables using APIs like [cuModuleGetGlobal\(\)](#) with this option specified may result in undefined behavior unless the variables are explicitly specified using [CU_JIT_REFERENCED_VARIABLE_NAMES](#). Option type: int Applies to: link-time optimization specified with CU_JIT_LTO Only valid with LTO-IR compiled with toolkits prior to CUDA 12.0

CU_JIT_POSITION_INDEPENDENT_CODE = 30

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

CU_JIT_MIN_CTA_PER_SM = 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 [CU_JIT_MAX_REGISTERS](#) or [CU_JIT_THREADS_PER_BLOCK](#). Optimizations based on this option need [CU_JIT_MAX_THREADS_PER_BLOCK](#) to be specified as well. For kernels already using PTX directive `.minnctapersm`, this option will be ignored by default. Use [CU_JIT_OVERRIDE_DIRECTIVE_VALUES](#) to let this option take precedence over the PTX directive. Option type: unsigned int Applies to: compiler only

CU_JIT_MAX_THREADS_PER_BLOCK = 32

Maximum number threads in a thread block, computed as the product of the maximum extent specified for each dimension of the block. This limit is guaranteed not to be exceeded 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 [CU_JIT_OVERRIDE_DIRECTIVE_VALUES](#) to let this option take precedence over the PTX directive. Option type: int Applies to: compiler only

CU_JIT_OVERRIDE_DIRECTIVE_VALUES = 33

This option lets the values specified using [CU_JIT_MAX_REGISTERS](#), [CU_JIT_THREADS_PER_BLOCK](#), [CU_JIT_MAX_THREADS_PER_BLOCK](#) and [CU_JIT_MIN_CTA_PER_SM](#) take precedence over any PTX directives. (0: Disable, default; 1: Enable) Option type: int Applies to: compiler only

CU_JIT_NUM_OPTIONS

enum CUjit_target

Online compilation targets

Values

CU_TARGET_COMPUTE_30 = 30

Compute device class 3.0

CU_TARGET_COMPUTE_32 = 32

Compute device class 3.2

CU_TARGET_COMPUTE_35 = 35

Compute device class 3.5

CU_TARGET_COMPUTE_37 = 37

Compute device class 3.7

CU_TARGET_COMPUTE_50 = 50

Compute device class 5.0

CU_TARGET_COMPUTE_52 = 52

Compute device class 5.2

CU_TARGET_COMPUTE_53 = 53

Compute device class 5.3

CU_TARGET_COMPUTE_60 = 60

Compute device class 6.0.

CU_TARGET_COMPUTE_61 = 61

Compute device class 6.1.
CU_TARGET_COMPUTE_62 = 62
 Compute device class 6.2.
CU_TARGET_COMPUTE_70 = 70
 Compute device class 7.0.
CU_TARGET_COMPUTE_72 = 72
 Compute device class 7.2.
CU_TARGET_COMPUTE_75 = 75
 Compute device class 7.5.
CU_TARGET_COMPUTE_80 = 80
 Compute device class 8.0.
CU_TARGET_COMPUTE_86 = 86
 Compute device class 8.6.
CU_TARGET_COMPUTE_87 = 87
 Compute device class 8.7.
CU_TARGET_COMPUTE_89 = 89
 Compute device class 8.9.
CU_TARGET_COMPUTE_90 = 90
 Compute device class 9.0.
CU_TARGET_COMPUTE_100 = 100
 Compute device class 10.0.
CU_TARGET_COMPUTE_101 = 101
 Compute device class 10.1.
CU_TARGET_COMPUTE_120 = 120
 Compute device class 12.0. Compute device class 9.0. with accelerated features.
**CU_TARGET_COMPUTE_90A = CU_COMPUTE_ACCELERATED_TARGET_BASE
 +CU_TARGET_COMPUTE_90**
 Compute device class 10.0. with accelerated features.
**CU_TARGET_COMPUTE_100A = CU_COMPUTE_ACCELERATED_TARGET_BASE
 +CU_TARGET_COMPUTE_100**
 Compute device class 10.1 with accelerated features.
**CU_TARGET_COMPUTE_101A = CU_COMPUTE_ACCELERATED_TARGET_BASE
 +CU_TARGET_COMPUTE_101**
 Compute device class 12.0. with accelerated features.
**CU_TARGET_COMPUTE_120A = CU_COMPUTE_ACCELERATED_TARGET_BASE
 +CU_TARGET_COMPUTE_120**

enum CUjitInputType

Device code formats

Values

CU_JIT_INPUT_CUBIN = 0

Compiled device-class-specific device code Applicable options: none

CU_JIT_INPUT_PTX = 1

PTX source code Applicable options: PTX compiler options

CU_JIT_INPUT_FATBINARY = 2

Bundle of multiple cubins and/or PTX of some device code Applicable options: PTX compiler options, [CU_JIT_FALLBACK_STRATEGY](#)

CU_JIT_INPUT_OBJECT = 3

Host object with embedded device code Applicable options: PTX compiler options, [CU_JIT_FALLBACK_STRATEGY](#)

CU_JIT_INPUT_LIBRARY = 4

Archive of host objects with embedded device code Applicable options: PTX compiler options, [CU_JIT_FALLBACK_STRATEGY](#)

CU_JIT_INPUT_NVVM = 5

[Deprecated](#) High-level intermediate code for link-time optimization Applicable options: NVVM compiler options, PTX compiler options Only valid with LTO-IR compiled with toolkits prior to CUDA 12.0

CU_JIT_NUM_INPUT_TYPES = 6

enum CUlaunchAttributeID

Launch attributes enum; used as id field of [CUlaunchAttribute](#)

Values

CU_LAUNCH_ATTRIBUTE_IGNORE = 0

Ignored entry, for convenient composition

CU_LAUNCH_ATTRIBUTE_ACCESS_POLICY_WINDOW = 1

Valid for streams, graph nodes, launches. See [CUlaunchAttributeValue::accessPolicyWindow](#).

CU_LAUNCH_ATTRIBUTE_COOPERATIVE = 2

Valid for graph nodes, launches. See [CUlaunchAttributeValue::cooperative](#).

CU_LAUNCH_ATTRIBUTE_SYNCHRONIZATION_POLICY = 3

Valid for streams. See [CUlaunchAttributeValue::syncPolicy](#).

CU_LAUNCH_ATTRIBUTE_CLUSTER_DIMENSION = 4

Valid for graph nodes, launches. See [CUlaunchAttributeValue::clusterDim](#).

CU_LAUNCH_ATTRIBUTE_CLUSTER_SCHEDULING_POLICY_PREFERENCE = 5

Valid for graph nodes, launches. See [CUlaunchAttributeValue::clusterSchedulingPolicyPreference](#).

CU_LAUNCH_ATTRIBUTE_PROGRAMMATIC_STREAM_SERIALIZATION = 6

Valid for launches. Setting [CUlaunchAttributeValue::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).

CU_LAUNCH_ATTRIBUTE_PROGRAMMATIC_EVENT = 7

Valid for launches. Set [CUlaunchAttributeValue::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 through PTX [launchdep.release](#) or CUDA builtin function [cudaTriggerProgrammaticLaunchCompletion\(\)](#). 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 [cuEventSynchronize\(\)](#)) are not guaranteed to observe the release precisely when it is released. For example, [cuEventSynchronize\(\)](#) 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 [CU_EVENT_DISABLE_TIMING](#) flag set).

CU_LAUNCH_ATTRIBUTE_PRIORITY = 8

Valid for streams, graph nodes, launches. See [CUlaunchAttributeValue::priority](#).

CU_LAUNCH_ATTRIBUTE_MEM_SYNC_DOMAIN_MAP = 9

Valid for streams, graph nodes, launches. See [CUlaunchAttributeValue::memSyncDomainMap](#).

CU_LAUNCH_ATTRIBUTE_MEM_SYNC_DOMAIN = 10

Valid for streams, graph nodes, launches. See [CUlaunchAttributeValue::memSyncDomain](#).

CU_LAUNCH_ATTRIBUTE_PREFERRED_CLUSTER_DIMENSION = 11

Valid for graph nodes, launches. Set [CUlaunchAttributeValue::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 [CU_LAUNCH_ATTRIBUTE_CLUSTER_DIMENSION](#) 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.

CU_LAUNCH_ATTRIBUTE_LAUNCH_COMPLETION_EVENT = 12

Valid for launches. Set [CUlaunchAttributeValue::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 [CU_EVENT_DISABLE_TIMING](#) flag set).

CU_LAUNCH_ATTRIBUTE_DEVICE_UPDATABLE_KERNEL_NODE = 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. [CUlaunchAttributeValue::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 [CUlaunchAttributeValue::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 [cuGraphDestroyNode](#). 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 [cuGraphKernelNodeCopyAttributes](#). 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 [cuGraphExecUpdate](#). 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.

CU_LAUNCH_ATTRIBUTE_PREFERRED_SHARED_MEMORY_CARVEOUT = 14

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

enum CUlaunchMemSyncDomain

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 [CU_LAUNCH_MEM_SYNC_DOMAIN_REMOTE](#) will have a different domain ID. User may also alter the domain ID with [CUlaunchMemSyncDomainMap](#) for a specific stream / graph node / kernel launch. See [CU_LAUNCH_ATTRIBUTE_MEM_SYNC_DOMAIN](#), [cuStreamSetAttribute](#), [cuLaunchKernelEx](#), [cuGraphKernelNodeSetAttribute](#).

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

CU_LAUNCH_MEM_SYNC_DOMAIN_DEFAULT = 0

Launch kernels in the default domain

CU_LAUNCH_MEM_SYNC_DOMAIN_REMOTE = 1

Launch kernels in the remote domain

enum CULibraryOption

Library options to be specified with [cuLibraryLoadData\(\)](#) or [cuLibraryLoadFromFile\(\)](#)

Values

CU_LIBRARY_HOST_UNIVERSAL_FUNCTION_AND_DATA_TABLE = 0

CU_LIBRARY_BINARY_IS_PRESERVED = 1

Specifies that the argument `code` passed to [cuLibraryLoadData\(\)](#) will be preserved. Specifying this option will let the driver know that `code` can be accessed at any point until [cuLibraryUnload\(\)](#).

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 [cuLibraryLoadFromFile\(\)](#) is invalid and will return [CUDA_ERROR_INVALID_VALUE](#).

CU_LIBRARY_NUM_OPTIONS

enum CULimit

Limits

Values

CU_LIMIT_STACK_SIZE = 0x00

GPU thread stack size

CU_LIMIT_PRINTF_FIFO_SIZE = 0x01

GPU printf FIFO size

CU_LIMIT_MALLOC_HEAP_SIZE = 0x02

GPU malloc heap size

CU_LIMIT_DEV_RUNTIME_SYNC_DEPTH = 0x03

GPU device runtime launch synchronize depth

CU_LIMIT_DEV_RUNTIME_PENDING_LAUNCH_COUNT = 0x04

GPU device runtime pending launch count

CU_LIMIT_MAX_L2_FETCH_GRANULARITY = 0x05

A value between 0 and 128 that indicates the maximum fetch granularity of L2 (in Bytes). This is a hint

CU_LIMIT_PERSISTING_L2_CACHE_SIZE = 0x06

A size in bytes for L2 persisting lines cache size

CU_LIMIT_SHMEM_SIZE = 0x07

A maximum size in bytes of shared memory available to CUDA kernels on a CIG context. Can only be queried, cannot be set

CU_LIMIT_CIG_ENABLED = 0x08

A non-zero value indicates this CUDA context is a CIG-enabled context. Can only be queried, cannot be set

CU_LIMIT_CIG_SHMEM_FALLBACK_ENABLED = 0x09

When set to zero, CUDA will fail to launch a kernel on a CIG context, instead of using the fallback path, if the kernel uses more shared memory than available

CU_LIMIT_MAX

enum CUmem_advise

Memory advise values

Values

CU_MEM_ADVISE_SET_READ_MOSTLY = 1

Data will mostly be read and only occasionally be written to

CU_MEM_ADVISE_UNSET_READ_MOSTLY = 2

Undo the effect of [CU_MEM_ADVISE_SET_READ_MOSTLY](#)

CU_MEM_ADVISE_SET_PREFERRED_LOCATION = 3

Set the preferred location for the data as the specified device

CU_MEM_ADVISE_UNSET_PREFERRED_LOCATION = 4

Clear the preferred location for the data

CU_MEM_ADVISE_SET_ACCESSED_BY = 5

Data will be accessed by the specified device, so prevent page faults as much as possible

CU_MEM_ADVISE_UNSET_ACCESSED_BY = 6

Let the Unified Memory subsystem decide on the page faulting policy for the specified device

enum CUmemAccess_flags

Specifies the memory protection flags for mapping.

Values

CU_MEM_ACCESS_FLAGS_PROT_NONE = 0x0

Default, make the address range not accessible

CU_MEM_ACCESS_FLAGS_PROT_READ = 0x1

Make the address range read accessible

CU_MEM_ACCESS_FLAGS_PROT_READWRITE = 0x3

Make the address range read-write accessible

CU_MEM_ACCESS_FLAGS_PROT_MAX = 0x7FFFFFFF

enum CUmemAllocationCompType

Specifies compression attribute for an allocation.

Values

CU_MEM_ALLOCATION_COMP_NONE = 0x0

Allocating non-compressible memory

CU_MEM_ALLOCATION_COMP_GENERIC = 0x1

Allocating compressible memory

enum CUmemAllocationGranularity_flags

Flag for requesting different optimal and required granularities for an allocation.

Values

CU_MEM_ALLOC_GRANULARITY_MINIMUM = 0x0

Minimum required granularity for allocation

CU_MEM_ALLOC_GRANULARITY_RECOMMENDED = 0x1

Recommended granularity for allocation for best performance

enum CUmemAllocationHandleType

Flags for specifying particular handle types

Values

CU_MEM_HANDLE_TYPE_NONE = 0x0

Does not allow any export mechanism. >

CU_MEM_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR = 0x1

Allows a file descriptor to be used for exporting. Permitted only on POSIX systems. (int)

CU_MEM_HANDLE_TYPE_WIN32 = 0x2

Allows a Win32 NT handle to be used for exporting. (HANDLE)

CU_MEM_HANDLE_TYPE_WIN32_KMT = 0x4

Allows a Win32 KMT handle to be used for exporting. (D3DKMT_HANDLE)

CU_MEM_HANDLE_TYPE_FABRIC = 0x8

Allows a fabric handle to be used for exporting. (CUmemFabricHandle)

CU_MEM_HANDLE_TYPE_MAX = 0x7FFFFFFF

enum CUmemAllocationType

Defines the allocation types available

Values

CU_MEM_ALLOCATION_TYPE_INVALID = 0x0

CU_MEM_ALLOCATION_TYPE_PINNED = 0x1

This allocation type is 'pinned', i.e. cannot migrate from its current location while the application is actively using it

CU_MEM_ALLOCATION_TYPE_MAX = 0x7FFFFFFF

enum CUmemAttach_flags

CUDA Mem Attach Flags

Values

CU_MEM_ATTACH_GLOBAL = 0x1

Memory can be accessed by any stream on any device

CU_MEM_ATTACH_HOST = 0x2

Memory cannot be accessed by any stream on any device

CU_MEM_ATTACH_SINGLE = 0x4

Memory can only be accessed by a single stream on the associated device

enum CUmemcpy3DOperandType

These flags allow applications to convey the operand type for individual copies specified in [cuMemcpy3DBatchAsync](#).

Values

CU_MEMCPY_OPERAND_TYPE_POINTER = 0x1

Memcpy operand is a valid pointer.

CU_MEMCPY_OPERAND_TYPE_ARRAY = 0x2

Memcpy operand is a CUarray.

CU_MEMCPY_OPERAND_TYPE_MAX = 0x7FFFFFFF

enum CUmemcopyFlags

Flags to specify for copies within a batch. For more details see [cuMemcopyBatchAsync](#).

Values

CU_MEMCPY_FLAG_DEFAULT = 0x0

CU_MEMCPY_FLAG_PREFER_OVERLAP_WITH_COMPUTE = 0x1

Hint to the driver to try and overlap the copy with compute work on the SMs.

enum CUmemcopySrcAccessOrder

These flags allow applications to convey the source access ordering CUDA must maintain. The destination will always be accessed in stream order.

Values

CU_MEMCPY_SRC_ACCESS_ORDER_INVALID = 0x0

Default invalid.

CU_MEMCPY_SRC_ACCESS_ORDER_STREAM = 0x1

Indicates that access to the source pointer must be in stream order.

CU_MEMCPY_SRC_ACCESS_ORDER_DURING_API_CALL = 0x2

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.

CU_MEMCPY_SRC_ACCESS_ORDER_ANY = 0x3

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.

CU_MEMCPY_SRC_ACCESS_ORDER_MAX = 0x7FFFFFFF

enum CUmemHandleType

Memory handle types

Values

CU_MEM_HANDLE_TYPE_GENERIC = 0

enum CUmemLocationType

Specifies the type of location

Values

CU_MEM_LOCATION_TYPE_INVALID = 0x0

CU_MEM_LOCATION_TYPE_DEVICE = 0x1

Location is a device location, thus id is a device ordinal

CU_MEM_LOCATION_TYPE_HOST = 0x2

Location is host, id is ignored

CU_MEM_LOCATION_TYPE_HOST_NUMA = 0x3

Location is a host NUMA node, thus id is a host NUMA node id

CU_MEM_LOCATION_TYPE_HOST_NUMA_CURRENT = 0x4

Location is a host NUMA node of the current thread, id is ignored

CU_MEM_LOCATION_TYPE_MAX = 0x7FFFFFFF

enum CUmemOperationType

Memory operation types

Values

CU_MEM_OPERATION_TYPE_MAP = 1

CU_MEM_OPERATION_TYPE_UNMAP = 2

enum CUmemorytype

Memory types

Values

CU_MEMORYTYPE_HOST = 0x01

Host memory

CU_MEMORYTYPE_DEVICE = 0x02

Device memory

CU_MEMORYTYPE_ARRAY = 0x03

Array memory

CU_MEMORYTYPE_UNIFIED = 0x04

Unified device or host memory

enum CUmemPool_attribute

CUDA memory pool attributes

Values

CU_MEMPOOL_ATTR_REUSE_FOLLOW_EVENT_DEPENDENCIES = 1

(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)

CU_MEMPOOL_ATTR_REUSE_ALLOW_OPPORTUNISTIC

(value type = int) Allow reuse of already completed frees when there is no dependency between the free and allocation. (default enabled)

CU_MEMPOOL_ATTR_REUSE_ALLOW_INTERNAL_DEPENDENCIES

(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).

CU_MEMPOOL_ATTR_RELEASE_THRESHOLD

(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)

CU_MEMPOOL_ATTR_RESERVED_MEM_CURRENT

(value type = cuuint64_t) Amount of backing memory currently allocated for the mempool.

CU_MEMPOOL_ATTR_RESERVED_MEM_HIGH

(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.

CU_MEMPOOL_ATTR_USED_MEM_CURRENT

(value type = cuuint64_t) Amount of memory from the pool that is currently in use by the application.

CU_MEMPOOL_ATTR_USED_MEM_HIGH

(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 CUmemRangeFlags

Flag for requesting handle type for address range.

Values

CU_MEM_RANGE_FLAG_DMA_BUF_MAPPING_TYPE_PCIE = 0x1

Indicates that DMA_BUF handle should be mapped via PCIe BAR1

enum CUmemRangeHandleType

Specifies the handle type for address range

Values

CU_MEM_RANGE_HANDLE_TYPE_DMA_BUF_FD = 0x1

CU_MEM_RANGE_HANDLE_TYPE_MAX = 0x7FFFFFFF

enum CUmulticastGranularity_flags

Flags for querying different granularities for a multicast object

Values

CU_MULTICAST_GRANULARITY_MINIMUM = 0x0

Minimum required granularity

CU_MULTICAST_GRANULARITY_RECOMMENDED = 0x1

Recommended granularity for best performance

enum CUoccupancy_flags

Occupancy calculator flag

Values

CU_OCCUPANCY_DEFAULT = 0x0

Default behavior

CU_OCCUPANCY_DISABLE_CACHING_OVERRIDE = 0x1

Assume global caching is enabled and cannot be automatically turned off

enum CUpointer_attribute

Pointer information

Values

CU_POINTER_ATTRIBUTE_CONTEXT = 1

The [CUcontext](#) on which a pointer was allocated or registered

CU_POINTER_ATTRIBUTE_MEMORY_TYPE = 2

The [CUmemorytype](#) describing the physical location of a pointer

CU_POINTER_ATTRIBUTE_DEVICE_POINTER = 3

The address at which a pointer's memory may be accessed on the device

CU_POINTER_ATTRIBUTE_HOST_POINTER = 4

The address at which a pointer's memory may be accessed on the host

CU_POINTER_ATTRIBUTE_P2P_TOKENS = 5

A pair of tokens for use with the nv-p2p.h Linux kernel interface

CU_POINTER_ATTRIBUTE_SYNC_MEMOPS = 6

Synchronize every synchronous memory operation initiated on this region

CU_POINTER_ATTRIBUTE_BUFFER_ID = 7

A process-wide unique ID for an allocated memory region

CU_POINTER_ATTRIBUTE_IS_MANAGED = 8

Indicates if the pointer points to managed memory

CU_POINTER_ATTRIBUTE_DEVICE_ORDINAL = 9

A device ordinal of a device on which a pointer was allocated or registered

CU_POINTER_ATTRIBUTE_IS_LEGACY_CUDA_IPC_CAPABLE = 10

1 if this pointer maps to an allocation that is suitable for [cudaIpcGetMemHandle](#), 0 otherwise

CU_POINTER_ATTRIBUTE_RANGE_START_ADDR = 11

Starting address for this requested pointer

CU_POINTER_ATTRIBUTE_RANGE_SIZE = 12

Size of the address range for this requested pointer

CU_POINTER_ATTRIBUTE_MAPPED = 13

1 if this pointer is in a valid address range that is mapped to a backing allocation, 0 otherwise

CU_POINTER_ATTRIBUTE_ALLOWED_HANDLE_TYPES = 14

Bitmask of allowed [CUmemAllocationHandleType](#) for this allocation

CU_POINTER_ATTRIBUTE_IS_GPU_DIRECT_RDMA_CAPABLE = 15

1 if the memory this pointer is referencing can be used with the GPUDirect RDMA API

CU_POINTER_ATTRIBUTE_ACCESS_FLAGS = 16

Returns the access flags the device associated with the current context has on the corresponding memory referenced by the pointer given

CU_POINTER_ATTRIBUTE_MEMPOOL_HANDLE = 17

Returns the mempool handle for the allocation if it was allocated from a mempool. Otherwise returns NULL.

CU_POINTER_ATTRIBUTE_MAPPING_SIZE = 18

Size of the actual underlying mapping that the pointer belongs to

CU_POINTER_ATTRIBUTE_MAPPING_BASE_ADDR = 19

The start address of the mapping that the pointer belongs to

CU_POINTER_ATTRIBUTE_MEMORY_BLOCK_ID = 20

A process-wide unique id corresponding to the physical allocation the pointer belongs to

CU_POINTER_ATTRIBUTE_IS_HW_DECOMPRESS_CAPABLE = 21

Returns in `*data` a boolean that indicates whether the pointer points to memory that is capable to be used for hardware accelerated decompression.

enum CUprocessState

CUDA Process States

Values

CU_PROCESS_STATE_RUNNING = 0

Default process state

CU_PROCESS_STATE_LOCKED

CUDA API locks are taken so further CUDA API calls will block

CU_PROCESS_STATE_CHECKPOINTED

Application memory contents have been checkpointed and underlying allocations and device handles have been released

CU_PROCESS_STATE_FAILED

Application entered an uncorrectable error during the checkpoint/restore process

enum CUresource_type

Resource types

Values

CU_RESOURCE_TYPE_ARRAY = 0x00

Array resource

CU_RESOURCE_TYPE_MIPMAPPED_ARRAY = 0x01

Mipmapped array resource

CU_RESOURCE_TYPE_LINEAR = 0x02

Linear resource

CU_RESOURCE_TYPE_PITCH2D = 0x03

Pitch 2D resource

enum CUresource_view_format

Resource view format

Values

CU_RES_VIEW_FORMAT_NONE = 0x00

No resource view format (use underlying resource format)

CU_RES_VIEW_FORMAT_UINT_1X8 = 0x01

1 channel unsigned 8-bit integers

CU_RES_VIEW_FORMAT_UINT_2X8 = 0x02

2 channel unsigned 8-bit integers

CU_RES_VIEW_FORMAT_UINT_4X8 = 0x03

4 channel unsigned 8-bit integers

CU_RES_VIEW_FORMAT_SINT_1X8 = 0x04

1 channel signed 8-bit integers

CU_RES_VIEW_FORMAT_SINT_2X8 = 0x05

2 channel signed 8-bit integers

CU_RES_VIEW_FORMAT_SINT_4X8 = 0x06

4 channel signed 8-bit integers

CU_RES_VIEW_FORMAT_UINT_1X16 = 0x07

1 channel unsigned 16-bit integers

CU_RES_VIEW_FORMAT_UINT_2X16 = 0x08

2 channel unsigned 16-bit integers

CU_RES_VIEW_FORMAT_UINT_4X16 = 0x09

4 channel unsigned 16-bit integers
CU_RES_VIEW_FORMAT_SINT_1X16 = 0x0a
 1 channel signed 16-bit integers
CU_RES_VIEW_FORMAT_SINT_2X16 = 0x0b
 2 channel signed 16-bit integers
CU_RES_VIEW_FORMAT_SINT_4X16 = 0x0c
 4 channel signed 16-bit integers
CU_RES_VIEW_FORMAT_UINT_1X32 = 0x0d
 1 channel unsigned 32-bit integers
CU_RES_VIEW_FORMAT_UINT_2X32 = 0x0e
 2 channel unsigned 32-bit integers
CU_RES_VIEW_FORMAT_UINT_4X32 = 0x0f
 4 channel unsigned 32-bit integers
CU_RES_VIEW_FORMAT_SINT_1X32 = 0x10
 1 channel signed 32-bit integers
CU_RES_VIEW_FORMAT_SINT_2X32 = 0x11
 2 channel signed 32-bit integers
CU_RES_VIEW_FORMAT_SINT_4X32 = 0x12
 4 channel signed 32-bit integers
CU_RES_VIEW_FORMAT_FLOAT_1X16 = 0x13
 1 channel 16-bit floating point
CU_RES_VIEW_FORMAT_FLOAT_2X16 = 0x14
 2 channel 16-bit floating point
CU_RES_VIEW_FORMAT_FLOAT_4X16 = 0x15
 4 channel 16-bit floating point
CU_RES_VIEW_FORMAT_FLOAT_1X32 = 0x16
 1 channel 32-bit floating point
CU_RES_VIEW_FORMAT_FLOAT_2X32 = 0x17
 2 channel 32-bit floating point
CU_RES_VIEW_FORMAT_FLOAT_4X32 = 0x18
 4 channel 32-bit floating point
CU_RES_VIEW_FORMAT_UNSIGNED_BC1 = 0x19
 Block compressed 1
CU_RES_VIEW_FORMAT_UNSIGNED_BC2 = 0x1a
 Block compressed 2
CU_RES_VIEW_FORMAT_UNSIGNED_BC3 = 0x1b
 Block compressed 3
CU_RES_VIEW_FORMAT_UNSIGNED_BC4 = 0x1c
 Block compressed 4 unsigned
CU_RES_VIEW_FORMAT_SIGNED_BC4 = 0x1d
 Block compressed 4 signed
CU_RES_VIEW_FORMAT_UNSIGNED_BC5 = 0x1e
 Block compressed 5 unsigned

CU_RES_VIEW_FORMAT_SIGNED_BC5 = 0x1f

Block compressed 5 signed

CU_RES_VIEW_FORMAT_UNSIGNED_BC6H = 0x20

Block compressed 6 unsigned half-float

CU_RES_VIEW_FORMAT_SIGNED_BC6H = 0x21

Block compressed 6 signed half-float

CU_RES_VIEW_FORMAT_UNSIGNED_BC7 = 0x22

Block compressed 7

enum CUresult

Error codes

Values

CUDA_SUCCESS = 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 [cuEventQuery\(\)](#) and [cuStreamQuery\(\)](#)).

CUDA_ERROR_INVALID_VALUE = 1

This indicates that one or more of the parameters passed to the API call is not within an acceptable range of values.

CUDA_ERROR_OUT_OF_MEMORY = 2

The API call failed because it was unable to allocate enough memory or other resources to perform the requested operation.

CUDA_ERROR_NOT_INITIALIZED = 3

This indicates that the CUDA driver has not been initialized with [cuInit\(\)](#) or that initialization has failed.

CUDA_ERROR_DEINITIALIZED = 4

This indicates that the CUDA driver is in the process of shutting down.

CUDA_ERROR_PROFILER_DISABLED = 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.

CUDA_ERROR_PROFILER_NOT_INITIALIZED = 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 [cuProfilerStart](#) or [cuProfilerStop](#) without initialization.

CUDA_ERROR_PROFILER_ALREADY_STARTED = 7

[Deprecated](#) This error return is deprecated as of CUDA 5.0. It is no longer an error to call [cuProfilerStart\(\)](#) when profiling is already enabled.

CUDA_ERROR_PROFILER_ALREADY_STOPPED = 8

[Deprecated](#) This error return is deprecated as of CUDA 5.0. It is no longer an error to call [cuProfilerStop\(\)](#) when profiling is already disabled.

CUDA_ERROR_STUB_LIBRARY = 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.

CUDA_ERROR_DEVICE_UNAVAILABLE = 46

This indicates that requested CUDA device is unavailable at the current time. Devices are often unavailable due to use of [CU_COMPUTEMODE_EXCLUSIVE_PROCESS](#) or [CU_COMPUTEMODE_PROHIBITED](#).

CUDA_ERROR_NO_DEVICE = 100

This indicates that no CUDA-capable devices were detected by the installed CUDA driver.

CUDA_ERROR_INVALID_DEVICE = 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.

CUDA_ERROR_DEVICE_NOT_LICENSED = 102

This error indicates that the Grid license is not applied.

CUDA_ERROR_INVALID_IMAGE = 200

This indicates that the device kernel image is invalid. This can also indicate an invalid CUDA module.

CUDA_ERROR_INVALID_CONTEXT = 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. This can also be returned if the green context passed to an API call was not converted to a [CUcontext](#) using [cuCtxFromGreenCtx](#) API.

CUDA_ERROR_CONTEXT_ALREADY_CURRENT = 202

This indicated that the context being supplied as a parameter to the API call was already the active context. [Deprecated](#) This error return is deprecated as of CUDA 3.2. It is no longer an error to attempt to push the active context via [cuCtxPushCurrent\(\)](#).

CUDA_ERROR_MAP_FAILED = 205

This indicates that a map or register operation has failed.

CUDA_ERROR_UNMAP_FAILED = 206

This indicates that an unmap or unregister operation has failed.

CUDA_ERROR_ARRAY_IS_MAPPED = 207

This indicates that the specified array is currently mapped and thus cannot be destroyed.

CUDA_ERROR_ALREADY_MAPPED = 208

This indicates that the resource is already mapped.

CUDA_ERROR_NO_BINARY_FOR_GPU = 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.

CUDA_ERROR_ALREADY_ACQUIRED = 210

This indicates that a resource has already been acquired.

CUDA_ERROR_NOT_MAPPED = 211

This indicates that a resource is not mapped.

CUDA_ERROR_NOT_MAPPED_AS_ARRAY = 212

This indicates that a mapped resource is not available for access as an array.

CUDA_ERROR_NOT_MAPPED_AS_POINTER = 213

This indicates that a mapped resource is not available for access as a pointer.

CUDA_ERROR_ECC_UNCORRECTABLE = 214

This indicates that an uncorrectable ECC error was detected during execution.

CUDA_ERROR_UNSUPPORTED_LIMIT = 215

This indicates that the [CUlimit](#) passed to the API call is not supported by the active device.

CUDA_ERROR_CONTEXT_ALREADY_IN_USE = 216

This indicates that the [CUcontext](#) passed to the API call can only be bound to a single CPU thread at a time but is already bound to a CPU thread.

CUDA_ERROR_PEER_ACCESS_UNSUPPORTED = 217

This indicates that peer access is not supported across the given devices.

CUDA_ERROR_INVALID_PTX = 218

This indicates that a PTX JIT compilation failed.

CUDA_ERROR_INVALID_GRAPHICS_CONTEXT = 219

This indicates an error with OpenGL or DirectX context.

CUDA_ERROR_NVLINK_UNCORRECTABLE = 220

This indicates that an uncorrectable NVLink error was detected during the execution.

CUDA_ERROR_JIT_COMPILER_NOT_FOUND = 221

This indicates that the PTX JIT compiler library was not found.

CUDA_ERROR_UNSUPPORTED_PTX_VERSION = 222

This indicates that the provided PTX was compiled with an unsupported toolchain.

CUDA_ERROR_JIT_COMPILATION_DISABLED = 223

This indicates that the PTX JIT compilation was disabled.

CUDA_ERROR_UNSUPPORTED_EXEC_AFFINITY = 224

This indicates that the [CUexecAffinityType](#) passed to the API call is not supported by the active device.

CUDA_ERROR_UNSUPPORTED_DEVSIDE_SYNC = 225

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

CUDA_ERROR_CONTAINED = 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.

CUDA_ERROR_INVALID_SOURCE = 300

This indicates that the device kernel source is invalid. This includes compilation/linker errors encountered in device code or user error.

CUDA_ERROR_FILE_NOT_FOUND = 301

This indicates that the file specified was not found.

CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND = 302

This indicates that a link to a shared object failed to resolve.

CUDA_ERROR_SHARED_OBJECT_INIT_FAILED = 303

This indicates that initialization of a shared object failed.

CUDA_ERROR_OPERATING_SYSTEM = 304

This indicates that an OS call failed.

CUDA_ERROR_INVALID_HANDLE = 400

This indicates that a resource handle passed to the API call was not valid. Resource handles are opaque types like [CUstream](#) and [CUEvent](#).

CUDA_ERROR_ILLEGAL_STATE = 401

This indicates that a resource required by the API call is not in a valid state to perform the requested operation.

CUDA_ERROR_LOSSY_QUERY = 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 functionality newer than the API version used to introspect it or omission of optional return arguments.

CUDA_ERROR_NOT_FOUND = 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.

CUDA_ERROR_NOT_READY = 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 [CUDA_SUCCESS](#) (which indicates completion). Calls that may return this value include [cuEventQuery\(\)](#) and [cuStreamQuery\(\)](#).

CUDA_ERROR_ILLEGAL_ADDRESS = 700

While executing a kernel, 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.

CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES = 701

This indicates that a launch did not occur because it did not have appropriate resources. 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. Passing arguments of the wrong size (i.e. a 64-bit pointer when a 32-bit int is expected) is equivalent to passing too many arguments and can also result in this error.

CUDA_ERROR_LAUNCH_TIMEOUT = 702

This indicates that the device kernel took too long to execute. This can only occur if timeouts are enabled - see the device attribute [CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT](#) 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.

CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING = 703

This error indicates a kernel launch that uses an incompatible texturing mode.

CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED = 704

This error indicates that a call to [cuCtxEnablePeerAccess\(\)](#) is trying to re-enable peer access to a context which has already had peer access to it enabled.

CUDA_ERROR_PEER_ACCESS_NOT_ENABLED = 705

This error indicates that [cuCtxDisablePeerAccess\(\)](#) is trying to disable peer access which has not been enabled yet via [cuCtxEnablePeerAccess\(\)](#).

CUDA_ERROR_PRIMARY_CONTEXT_ACTIVE = 708

This error indicates that the primary context for the specified device has already been initialized.

CUDA_ERROR_CONTEXT_IS_DESTROYED = 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.

CUDA_ERROR_ASSERT = 710

A device-side assert triggered during kernel execution. The context cannot be used anymore, and must be destroyed. All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.

CUDA_ERROR_TOO_MANY_PEERS = 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 [cuCtxEnablePeerAccess\(\)](#).

CUDA_ERROR_HOST_MEMORY_ALREADY_REGISTERED = 712

This error indicates that the memory range passed to [cuMemHostRegister\(\)](#) has already been registered.

CUDA_ERROR_HOST_MEMORY_NOT_REGISTERED = 713

This error indicates that the pointer passed to [cuMemHostUnregister\(\)](#) does not correspond to any currently registered memory region.

CUDA_ERROR_HARDWARE_STACK_ERROR = 714

While executing a kernel, the device encountered a stack error. This can be 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.

CUDA_ERROR_ILLEGAL_INSTRUCTION = 715

While executing a kernel, the device encountered an illegal instruction. 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.

CUDA_ERROR_MISALIGNED_ADDRESS = 716

While executing a kernel, 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.

CUDA_ERROR_INVALID_ADDRESS_SPACE = 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.

CUDA_ERROR_INVALID_PC = 718

While executing a kernel, the device program counter wrapped its 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.

CUDA_ERROR_LAUNCH_FAILED = 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.

CUDA_ERROR_COOPERATIVE_LAUNCH_TOO_LARGE = 720

This error indicates that the number of blocks launched per grid for a kernel that was launched via either [cuLaunchCooperativeKernel](#) or [cuLaunchCooperativeKernelMultiDevice](#) exceeds the maximum number of blocks as allowed by [cuOccupancyMaxActiveBlocksPerMultiprocessor](#) or [cuOccupancyMaxActiveBlocksPerMultiprocessorWithFlags](#) times the number of multiprocessors as specified by the device attribute [CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT](#).

CUDA_ERROR_TENSOR_MEMORY_LEAK = 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.

CUDA_ERROR_NOT_PERMITTED = 800

This error indicates that the attempted operation is not permitted.

CUDA_ERROR_NOT_SUPPORTED = 801

This error indicates that the attempted operation is not supported on the current system or device.

CUDA_ERROR_SYSTEM_NOT_READY = 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.

CUDA_ERROR_SYSTEM_DRIVER_MISMATCH = 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.

CUDA_ERROR_COMPAT_NOT_SUPPORTED_ON_DEVICE = 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.

CUDA_ERROR_MPS_CONNECTION_FAILED = 805

This error indicates that the MPS client failed to connect to the MPS control daemon or the MPS server.

CUDA_ERROR_MPS_RPC_FAILURE = 806

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

CUDA_ERROR_MPS_SERVER_NOT_READY = 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.

CUDA_ERROR_MPS_MAX_CLIENTS_REACHED = 808

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

CUDA_ERROR_MPS_MAX_CONNECTIONS_REACHED = 809

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

CUDA_ERROR_MPS_CLIENT_TERMINATED = 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.

CUDA_ERROR_CDP_NOT_SUPPORTED = 811

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

CUDA_ERROR_CDP_VERSION_MISMATCH = 812

This error indicates that a module contains an unsupported interaction between different versions of CUDA Dynamic Parallelism.

CUDA_ERROR_STREAM_CAPTURE_UNSUPPORTED = 900

This error indicates that the operation is not permitted when the stream is capturing.

CUDA_ERROR_STREAM_CAPTURE_INVALIDATED = 901

This error indicates that the current capture sequence on the stream has been invalidated due to a previous error.

CUDA_ERROR_STREAM_CAPTURE_MERGE = 902

This error indicates that the operation would have resulted in a merge of two independent capture sequences.

CUDA_ERROR_STREAM_CAPTURE_UNMATCHED = 903

This error indicates that the capture was not initiated in this stream.

CUDA_ERROR_STREAM_CAPTURE_UNJOINED = 904

This error indicates that the capture sequence contains a fork that was not joined to the primary stream.

CUDA_ERROR_STREAM_CAPTURE_ISOLATION = 905

This error indicates that a dependency would have been created which crosses the capture sequence boundary. Only implicit in-stream ordering dependencies are allowed to cross the boundary.

CUDA_ERROR_STREAM_CAPTURE_IMPLICIT = 906

This error indicates a disallowed implicit dependency on a current capture sequence from `cudaStreamLegacy`.

CUDA_ERROR_CAPTURED_EVENT = 907

This error indicates that the operation is not permitted on an event which was last recorded in a capturing stream.

CUDA_ERROR_STREAM_CAPTURE_WRONG_THREAD = 908

A stream capture sequence not initiated with the `CU_STREAM_CAPTURE_MODE_RELAXED` argument to [cuStreamBeginCapture](#) was passed to [cuStreamEndCapture](#) in a different thread.

CUDA_ERROR_TIMEOUT = 909

This error indicates that the timeout specified for the wait operation has lapsed.

CUDA_ERROR_GRAPH_EXEC_UPDATE_FAILURE = 910

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

CUDA_ERROR_EXTERNAL_DEVICE = 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.

CUDA_ERROR_INVALID_CLUSTER_SIZE = 912

Indicates a kernel launch error due to cluster misconfiguration.

CUDA_ERROR_FUNCTION_NOT_LOADED = 913

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

CUDA_ERROR_INVALID_RESOURCE_TYPE = 914

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

CUDA_ERROR_INVALID_RESOURCE_CONFIGURATION = 915

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

CUDA_ERROR_KEY_ROTATION = 916

This error indicates that an error happened during the key rotation sequence.

CUDA_ERROR_UNKNOWN = 999

This indicates that an unknown internal error has occurred.

enum CUshared_carveout

Shared memory carveout configurations. These may be passed to [cuFuncSetAttribute](#) or [cuKernelSetAttribute](#)

Values

CU_SHARED_MEM_CARVEOUT_DEFAULT = -1

No preference for shared memory or L1 (default)

CU_SHARED_MEM_CARVEOUT_MAX_SHARED = 100

Prefer maximum available shared memory, minimum L1 cache

CU_SHARED_MEM_CARVEOUT_MAX_L1 = 0

Prefer maximum available L1 cache, minimum shared memory

enum CUsharedconfig

Deprecated

Shared memory configurations

Values

CU_SHARED_MEM_CONFIG_DEFAULT_BANK_SIZE = 0x00

set default shared memory bank size

CU_SHARED_MEM_CONFIG_FOUR_BYTE_BANK_SIZE = 0x01

set shared memory bank width to four bytes

CU_SHARED_MEM_CONFIG_EIGHT_BYTE_BANK_SIZE = 0x02

set shared memory bank width to eight bytes

enum CUstream_flags

Stream creation flags

Values

CU_STREAM_DEFAULT = 0x0

Default stream flag

CU_STREAM_NON_BLOCKING = 0x1

Stream does not synchronize with stream 0 (the NULL stream)

enum CUstreamBatchMemOpType

Operations for [cuStreamBatchMemOp](#)

Values

CU_STREAM_MEM_OP_WAIT_VALUE_32 = 1

Represents a [cuStreamWaitValue32](#) operation

CU_STREAM_MEM_OP_WRITE_VALUE_32 = 2

Represents a [cuStreamWriteValue32](#) operation

CU_STREAM_MEM_OP_WAIT_VALUE_64 = 4

Represents a [cuStreamWaitValue64](#) operation

CU_STREAM_MEM_OP_WRITE_VALUE_64 = 5

Represents a [cuStreamWriteValue64](#) operation

CU_STREAM_MEM_OP_BARRIER = 6

Insert a memory barrier of the specified type

CU_STREAM_MEM_OP_FLUSH_REMOTE_WRITES = 3

This has the same effect as [CU_STREAM_WAIT_VALUE_FLUSH](#), but as a standalone operation.

enum CUstreamCaptureMode

Possible modes for stream capture thread interactions. For more details see [cuStreamBeginCapture](#) and [cuThreadExchangeStreamCaptureMode](#)

Values

CU_STREAM_CAPTURE_MODE_GLOBAL = 0

CU_STREAM_CAPTURE_MODE_THREAD_LOCAL = 1

CU_STREAM_CAPTURE_MODE_RELAXED = 2

enum CUstreamCaptureStatus

Possible stream capture statuses returned by [cuStreamIsCapturing](#)

Values

CU_STREAM_CAPTURE_STATUS_NONE = 0

Stream is not capturing

CU_STREAM_CAPTURE_STATUS_ACTIVE = 1

Stream is actively capturing

CU_STREAM_CAPTURE_STATUS_INVALIDATED = 2

Stream is part of a capture sequence that has been invalidated, but not terminated

enum CUstreamMemoryBarrier_flags

Flags for `cuStreamMemoryBarrier`

Values

CU_STREAM_MEMORY_BARRIER_TYPE_SYS = 0x0

System-wide memory barrier.

CU_STREAM_MEMORY_BARRIER_TYPE_GPU = 0x1

Limit memory barrier scope to the GPU.

enum CUstreamUpdateCaptureDependencies_flags

Flags for `cuStreamUpdateCaptureDependencies`

Values

CU_STREAM_ADD_CAPTURE_DEPENDENCIES = 0x0

Add new nodes to the dependency set

CU_STREAM_SET_CAPTURE_DEPENDENCIES = 0x1

Replace the dependency set with the new nodes

enum CUstreamWaitValue_flags

Flags for `cuStreamWaitValue32` and `cuStreamWaitValue64`

Values

CU_STREAM_WAIT_VALUE_GEQ = 0x0

Wait until $(\text{int32_t})(*\text{addr} - \text{value}) \geq 0$ (or int64_t for 64 bit values). Note this is a cyclic comparison which ignores wraparound. (Default behavior.)

CU_STREAM_WAIT_VALUE_EQ = 0x1

Wait until $*\text{addr} == \text{value}$.

CU_STREAM_WAIT_VALUE_AND = 0x2

Wait until $(*\text{addr} \& \text{value}) \neq 0$.

CU_STREAM_WAIT_VALUE_NOR = 0x3

Wait until $\sim(*\text{addr} \mid \text{value}) \neq 0$. Support for this operation can be queried with [cuDeviceGetAttribute\(\)](#) and [CU_DEVICE_ATTRIBUTE_CAN_USE_STREAM_WAIT_VALUE_NOR](#).

CU_STREAM_WAIT_VALUE_FLUSH = 1<<30

Follow the wait operation with a flush of outstanding remote writes. This means that, if a remote write operation is guaranteed to have reached the device before the wait can be satisfied, that write is guaranteed to be visible to downstream device work. The device is permitted to reorder remote writes internally. For example, this flag would be required if two remote writes arrive in a defined order, the wait is satisfied by the second write, and downstream work needs to observe the first write. Support for this operation is restricted to selected platforms and can be queried with [CU_DEVICE_ATTRIBUTE_CAN_FLUSH_REMOTE_WRITES](#).

enum CUstreamWriteValue_flags

Flags for [cuStreamWriteValue32](#)

Values

CU_STREAM_WRITE_VALUE_DEFAULT = 0x0

Default behavior

CU_STREAM_WRITE_VALUE_NO_MEMORY_BARRIER = 0x1

Permits the write to be reordered with writes which were issued before it, as a performance optimization. Normally, [cuStreamWriteValue32](#) will provide a memory fence before the write, which has similar semantics to `__threadfence_system()` but is scoped to the stream rather than a CUDA thread. This flag is not supported in the v2 API.

enum CUtensorMapDataType

Tensor map data type

Values

CU_TENSOR_MAP_DATA_TYPE_UINT8 = 0

CU_TENSOR_MAP_DATA_TYPE_UINT16

CU_TENSOR_MAP_DATA_TYPE_UINT32

CU_TENSOR_MAP_DATA_TYPE_INT32

CU_TENSOR_MAP_DATA_TYPE_UINT64

CU_TENSOR_MAP_DATA_TYPE_INT64

CU_TENSOR_MAP_DATA_TYPE_FLOAT16

CU_TENSOR_MAP_DATA_TYPE_FLOAT32

CU_TENSOR_MAP_DATA_TYPE_FLOAT64

CU_TENSOR_MAP_DATA_TYPE_BFLOAT16

CU_TENSOR_MAP_DATA_TYPE_FLOAT32_FTZ

CU_TENSOR_MAP_DATA_TYPE_TFLOAT32

CU_TENSOR_MAP_DATA_TYPE_TFLOAT32_FTZ

CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN8B
CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B
CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B

enum CUtensorMapFloatOOBfill

Tensor map out-of-bounds fill type

Values

CU_TENSOR_MAP_FLOAT_OOB_FILL_NONE = 0
CU_TENSOR_MAP_FLOAT_OOB_FILL_NAN_REQUEST_ZERO_FMA

enum CUtensorMapIm2ColWideMode

Tensor map Im2Col wide mode

Values

CU_TENSOR_MAP_IM2COL_WIDE_MODE_W = 0
CU_TENSOR_MAP_IM2COL_WIDE_MODE_W128

enum CUtensorMapInterleave

Tensor map interleave layout type

Values

CU_TENSOR_MAP_INTERLEAVE_NONE = 0
CU_TENSOR_MAP_INTERLEAVE_16B
CU_TENSOR_MAP_INTERLEAVE_32B

enum CUtensorMapL2promotion

Tensor map L2 promotion type

Values

CU_TENSOR_MAP_L2_PROMOTION_NONE = 0
CU_TENSOR_MAP_L2_PROMOTION_L2_64B
CU_TENSOR_MAP_L2_PROMOTION_L2_128B
CU_TENSOR_MAP_L2_PROMOTION_L2_256B

enum CUtensorMapSwizzle

Tensor map swizzling mode of shared memory banks

Values

CU_TENSOR_MAP_SWIZZLE_NONE = 0
CU_TENSOR_MAP_SWIZZLE_32B
CU_TENSOR_MAP_SWIZZLE_64B
CU_TENSOR_MAP_SWIZZLE_128B
CU_TENSOR_MAP_SWIZZLE_128B_ATOM_32B
CU_TENSOR_MAP_SWIZZLE_128B_ATOM_32B_FLIP_8B
CU_TENSOR_MAP_SWIZZLE_128B_ATOM_64B

enum CUUserObject_flags

Flags for user objects for graphs

Values

CU_USER_OBJECT_NO_DESTRUCTOR_SYNC = 1
 Indicates the destructor execution is not synchronized by any CUDA handle.

enum CUUserObjectRetain_flags

Flags for retaining user object references for graphs

Values

CU_GRAPH_USER_OBJECT_MOVE = 1
 Transfer references from the caller rather than creating new references.

typedef CUaccessPolicyWindow

Access policy window

typedef struct CUarray_st *CUarray

CUDA array

typedef (*CUasyncCallback) (CUasyncNotificationInfo* info, void* userData, CUasyncCallbackHandle callback)

CUDA async notification callback

```
typedef struct CUasyncCallbackEntry_st  
*CUasyncCallbackHandle
```

CUDA async notification callback handle

```
typedef struct CUctx_st *CUcontext
```

A regular context handle

```
typedef CUdevice
```

CUDA device

```
typedef int CUdevice_v1
```

CUDA device

```
typedef CUdeviceptr
```

CUDA device pointer

```
typedef unsigned int CUdeviceptr_v2
```

CUDA device pointer CUdeviceptr is defined as an unsigned integer type whose size matches the size of a pointer on the target platform.

```
typedef struct CUeglStreamConnection_st  
*CUeglStreamConnection
```

CUDA EGLSream Connection

```
typedef struct CUevent_st *CUevent
```

CUDA event

```
typedef CUexecAffinityParam
```

Execution Affinity Parameters

```
typedef struct CUextMemory_st *CUexternalMemory
```

CUDA external memory

```
typedef struct CUextSemaphore_st *CUexternalSemaphore
```

CUDA external semaphore

```
typedef struct CUfunc_st *CUfunction
```

CUDA function

```
typedef struct CUgraph_st *CUgraph
```

CUDA graph

```
typedef cuuint64_t CUgraphConditionalHandle
```

CUDA graph conditional handle

```
typedef struct CUgraphDeviceUpdatableNode_st  
*CUgraphDeviceNode
```

CUDA graph device node handle

```
typedef struct CUgraphExec_st *CUgraphExec
```

CUDA executable graph

```
typedef struct CUgraphicsResource_st  
*CUgraphicsResource
```

CUDA graphics interop resource

```
typedef struct CUgraphNode_st *CUgraphNode
```

CUDA graph node

```
typedef struct CUgreenCtx_st *CUgreenCtx
```

A green context handle. This handle can be used safely from only one CPU thread at a time. Created via [cuGreenCtxCreate](#)

```
typedef void (CUDA_CB *CUhostFn) (void* userData)
```

CUDA host function

```
typedef struct CUkern_st *CUkernel
```

CUDA kernel

```
typedef struct CULib_st *CULibrary
```

CUDA library

```
typedef struct CUmempoolHandle_st *CUMemoryPool
```

CUDA memory pool

```
typedef struct CUmipmappedArray_st  
*CUmipmappedArray
```

CUDA mipmapped array

```
typedef struct CUmod_st *CUmodule
```

CUDA module

```
typedef size_t (CUDA_CB *CUoccupancyB2DSize) (int  
blockSize)
```

Block size to per-block dynamic shared memory mapping for a certain kernel

```
typedef struct CUstream_st *CUstream
```

CUDA stream

```
typedef void (CUDA_CB *CUstreamCallback) (CUstream  
hStream, CUresult status, void* userData)
```

CUDA stream callback

```
typedef CUsurfObject
```

An opaque value that represents a CUDA surface object

```
typedef unsigned long long CUsurfObject_v1
```

An opaque value that represents a CUDA surface object

```
typedef struct CUsurfref_st *CUsurfref
```

CUDA surface reference

```
typedef CUtexObject
```

An opaque value that represents a CUDA texture object

```
typedef unsigned long long CUtexObject_v1
```

An opaque value that represents a CUDA texture object

```
typedef struct CUtexref_st *CUtexref
```

CUDA texture reference

```
typedef struct CUuserObject_st *CUuserObject
```

CUDA user object for graphs

```
#define
```

```
CU_ARRAY_SPARSE_PROPERTIES_SINGLE_MIPTAIL  
0x1
```

Indicates that the layered sparse CUDA array or CUDA mipmapped array has a single mip tail region for all layers

```
#define CU_DEVICE_CPU ((CUdevice)-1)
```

Device that represents the CPU

```
#define CU_DEVICE_INVALID ((CUdevice)-2)
```

Device that represents an invalid device

```
#define CU_GRAPH_COND_ASSIGN_DEFAULT 0x1
```

Conditional node handle flags Default value is applied when graph is launched.

```
#define
CU_GRAPH_KERNEL_NODE_PORT_DEFAULT 0
```

This port activates when the kernel has finished executing.

```
#define
CU_GRAPH_KERNEL_NODE_PORT_LAUNCH_ORDER
2
```

This port activates when all blocks of the kernel have begun execution. See also [CU_LAUNCH_ATTRIBUTE_LAUNCH_COMPLETION_EVENT](#).

```
#define
CU_GRAPH_KERNEL_NODE_PORT_PROGRAMMATIC
1
```

This port activates when all blocks of the kernel have performed [cudaTriggerProgrammaticLaunchCompletion\(\)](#) or have terminated. It must be used with edge type [CU_GRAPH_DEPENDENCY_TYPE_PROGRAMMATIC](#). See also [CU_LAUNCH_ATTRIBUTE_PROGRAMMATIC_EVENT](#).

```
#define CU_IPC_HANDLE_SIZE 64
```

CUDA IPC handle size

```
#define CU_LAUNCH_PARAM_BUFFER_POINTER
((void*)CU_LAUNCH_PARAM_BUFFER_POINTER_AS_INT)
```

Indicator that the next value in the `extra` parameter to [cuLaunchKernel](#) will be a pointer to a buffer containing all kernel parameters used for launching kernel `f`. This buffer needs to honor all alignment/padding requirements of the individual parameters. If [CU_LAUNCH_PARAM_BUFFER_SIZE](#) is not also specified in the `extra` array, then [CU_LAUNCH_PARAM_BUFFER_POINTER](#) will have no effect.

```
#define
CU_LAUNCH_PARAM_BUFFER_POINTER_AS_INT
0x01
```

C++ compile time constant for CU_LAUNCH_PARAM_BUFFER_POINTER

```
#define CU_LAUNCH_PARAM_BUFFER_SIZE
((void*)CU_LAUNCH_PARAM_BUFFER_SIZE_AS_INT)
```

Indicator that the next value in the `extra` parameter to `cuLaunchKernel` will be a pointer to a `size_t` which contains the size of the buffer specified with `CU_LAUNCH_PARAM_BUFFER_POINTER`. It is required that `CU_LAUNCH_PARAM_BUFFER_POINTER` also be specified in the `extra` array if the value associated with `CU_LAUNCH_PARAM_BUFFER_SIZE` is not zero.

```
#define
CU_LAUNCH_PARAM_BUFFER_SIZE_AS_INT 0x02
```

C++ compile time constant for CU_LAUNCH_PARAM_BUFFER_SIZE

```
#define CU_LAUNCH_PARAM_END
((void*)CU_LAUNCH_PARAM_END_AS_INT)
```

End of array terminator for the `extra` parameter to `cuLaunchKernel`

```
#define CU_LAUNCH_PARAM_END_AS_INT 0x00
```

C++ compile time constant for CU_LAUNCH_PARAM_END

```
#define
CU_MEM_CREATE_USAGE_HW_DECOMPRESS 0x2
```

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

```
#define CU_MEM_CREATE_USAGE_TILE_POOL 0x1
```

This flag if set indicates that the memory will be used as a tile pool.

```
#define
```

```
CU_MEM_POOL_CREATE_USAGE_HW_DECOMPRESS  
0x2
```

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

```
#define CU_MEMHOSTALLOC_DEVICEMAP 0x02
```

If set, host memory is mapped into CUDA address space and [cuMemHostGetDevicePointer\(\)](#) may be called on the host pointer. Flag for [cuMemHostAlloc\(\)](#)

```
#define CU_MEMHOSTALLOC_PORTABLE 0x01
```

If set, host memory is portable between CUDA contexts. Flag for [cuMemHostAlloc\(\)](#)

```
#define CU_MEMHOSTALLOC_WRITECOMBINED  
0x04
```

If set, host memory is allocated as write-combined - fast to write, faster to DMA, slow to read except via SSE4 streaming load instruction (MOVNTDQA). Flag for [cuMemHostAlloc\(\)](#)

```
#define CU_MEMHOSTREGISTER_DEVICEMAP 0x02
```

If set, host memory is mapped into CUDA address space and [cuMemHostGetDevicePointer\(\)](#) may be called on the host pointer. Flag for [cuMemHostRegister\(\)](#)

```
#define CU_MEMHOSTREGISTER_IOMEMORY 0x04
```

If set, the passed memory pointer is treated as pointing to some memory-mapped I/O space, e.g. belonging to a third-party PCIe device. On Windows the flag is a no-op. On Linux that memory is marked as non cache-coherent for the GPU and is expected to be physically contiguous. It may return [CUDA_ERROR_NOT_PERMITTED](#) if run as an unprivileged user, [CUDA_ERROR_NOT_SUPPORTED](#) on older Linux kernel versions. On all other platforms, it is not supported and [CUDA_ERROR_NOT_SUPPORTED](#) is returned. Flag for [cuMemHostRegister\(\)](#)

```
#define CU_MEMHOSTREGISTER_PORTABLE 0x01
```

If set, host memory is portable between CUDA contexts. Flag for [cuMemHostRegister\(\)](#)

#define CU_MEMHOSTREGISTER_READ_ONLY 0x08

If set, the passed memory pointer is treated as pointing to memory that is considered read-only by the device. On platforms without [CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES](#), 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 [CU_DEVICE_ATTRIBUTE_READ_ONLY_HOST_REGISTER_SUPPORTED](#). Using this flag with a current context associated with a device that does not have this attribute set will cause [cuMemHostRegister](#) to error with [CUDA_ERROR_NOT_SUPPORTED](#).

#define CU_PARAM_TR_DEFAULT -1

For texture references loaded into the module, use default texunit from texture reference.

#define CU_STREAM_LEGACY ((CUstream)0x1)

Legacy stream handle

Stream handle that can be passed as a CUstream to use an implicit stream with legacy synchronization behavior.

See details of the [synchronization behavior](#).

#define CU_STREAM_PER_THREAD ((CUstream)0x2)

Per-thread stream handle

Stream handle that can be passed as a CUstream to use an implicit stream with per-thread synchronization behavior.

See details of the [synchronization behavior](#).

#define CU_TENSOR_MAP_NUM_QWORDS 16

Size of tensor map descriptor

#define CU_TRSA_OVERRIDE_FORMAT 0x01

Override the texref format with a format inferred from the array. Flag for [cuTexRefSetArray\(\)](#)

```
#define
CU_TRSF_DISABLE_TRILINEAR_OPTIMIZATION
0x20
```

Disable any trilinear filtering optimizations. Flag for [cuTexRefSetFlags\(\)](#) and [cuTexObjectCreate\(\)](#)

```
#define CU_TRSF_NORMALIZED_COORDINATES
0x02
```

Use normalized texture coordinates in the range [0,1) instead of [0,dim). Flag for [cuTexRefSetFlags\(\)](#) and [cuTexObjectCreate\(\)](#)

```
#define CU_TRSF_READ_AS_INTEGER 0x01
```

Read the texture as integers rather than promoting the values to floats in the range [0,1]. Flag for [cuTexRefSetFlags\(\)](#) and [cuTexObjectCreate\(\)](#)

```
#define CU_TRSF_SEAMLESS_CUBEMAP 0x40
```

Enable seamless cube map filtering. Flag for [cuTexObjectCreate\(\)](#)

```
#define CU_TRSF_SRGB 0x10
```

Perform sRGB->linear conversion during texture read. Flag for [cuTexRefSetFlags\(\)](#) and [cuTexObjectCreate\(\)](#)

```
#define CUDA_ARRAY3D_2DARRAY 0x01
```

Deprecated, use `CUDA_ARRAY3D_LAYERED`

```
#define CUDA_ARRAY3D_COLOR_ATTACHMENT
0x20
```

This flag indicates that the CUDA array may be bound as a color target in an external graphics API

```
#define CUDA_ARRAY3D_CUBEMAP 0x04
```

If set, the CUDA array is a collection of six 2D arrays, representing faces of a cube. The width of such a CUDA array must be equal to its height, and Depth must be six. If [CUDA_ARRAY3D_LAYERED](#) flag is also set, then the CUDA array is a collection of cubemaps and Depth must be a multiple of six.

```
#define CUDA_ARRAY3D_DEFERRED_MAPPING  
0x80
```

This flag if set indicates that the CUDA array or CUDA mipmapped array will allow deferred memory mapping

```
#define CUDA_ARRAY3D_DEPTH_TEXTURE 0x10
```

This flag if set indicates that the CUDA array is a DEPTH_TEXTURE.

```
#define CUDA_ARRAY3D_LAYERED 0x01
```

If set, the CUDA array is a collection of layers, where each layer is either a 1D or a 2D array and the Depth member of CUDA_ARRAY3D_DESCRIPTOR specifies the number of layers, not the depth of a 3D array.

```
#define CUDA_ARRAY3D_SPARSE 0x40
```

This flag if set indicates that the CUDA array or CUDA mipmapped array is a sparse CUDA array or CUDA mipmapped array respectively

```
#define CUDA_ARRAY3D_SURFACE_LDST 0x02
```

This flag must be set in order to bind a surface reference to the CUDA array

```
#define CUDA_ARRAY3D_TEXTURE_GATHER 0x08
```

This flag must be set in order to perform texture gather operations on a CUDA array.

```
#define CUDA_ARRAY3D_VIDEO_ENCODE_DECODE  
0x100
```

This flag indicates that the CUDA array will be used for hardware accelerated video encode/decode operations.

```
#define
```

```
CUDA_COOPERATIVE_LAUNCH_MULTI_DEVICE_NO_POST_0x02
```

If set, any subsequent work pushed in a stream that participated in a call to [cuLaunchCooperativeKernelMultiDevice](#) will only wait for the kernel launched on the GPU corresponding to that stream to complete before it begins execution.

```
#define
```

```
CUDA_COOPERATIVE_LAUNCH_MULTI_DEVICE_NO_PRE_0x01
```

If set, each kernel launched as part of [cuLaunchCooperativeKernelMultiDevice](#) only waits for prior work in the stream corresponding to that GPU to complete before the kernel begins execution.

```
#define CUDA_EGL_INFINITE_TIMEOUT 0xFFFFFFFF
```

Indicates that timeout for [cuEGLStreamConsumerAcquireFrame](#) is infinite.

```
#define CUDA_EXTERNAL_MEMORY_DEDICATED_0x1
```

Indicates that the external memory object is a dedicated resource

```
#define
```

```
CUDA_EXTERNAL_SEMAPHORE_SIGNAL_SKIP_NVSCIBUF_0x01
```

When the `flags` parameter of `CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS` 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 [CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF](#), which otherwise are performed by default to ensure data coherency with other importers of the same `NvSciBuf` memory objects.

#define

CUDA_EXTERNAL_SEMAPHORE_WAIT_SKIP_NVSCIBUF_M 0x02

When the `flags` parameter of `CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS` contains this flag, it indicates that waiting on an external semaphore object should skip performing appropriate memory synchronization operations over all the external memory objects that are imported as [CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF](#), which otherwise are performed by default to ensure data coherency with other importers of the same `NvSciBuf` memory objects.

#define CUDA_NVSCISYNC_ATTR_SIGNAL 0x1

When `flags` of [cuDeviceGetNvSciSyncAttributes](#) is set to this, it indicates that application needs signaler specific `NvSciSyncAttr` to be filled by [cuDeviceGetNvSciSyncAttributes](#).

#define CUDA_NVSCISYNC_ATTR_WAIT 0x2

When `flags` of [cuDeviceGetNvSciSyncAttributes](#) is set to this, it indicates that application needs waiter specific `NvSciSyncAttr` to be filled by [cuDeviceGetNvSciSyncAttributes](#).

#define CUDA_VERSION 12080

CUDA API version number

#define MAX_PLANES 3

Maximum number of planes per frame

6.2. Error Handling

This section describes the error handling functions of the low-level CUDA driver application programming interface.

CUresult cuGetErrorName (CUresult error, const char **pStr)

Gets the string representation of an error code enum name.

Parameters

error

- Error code to convert to string

pStr

- Address of the string pointer.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Sets *pStr to the address of a NULL-terminated string representation of the name of the enum error code error. If the error code is not recognized, [CUDA_ERROR_INVALID_VALUE](#) will be returned and *pStr will be set to the NULL address.

See also:

[CUresult](#), [cudaGetErrorName](#)

CUresult cuGetErrorString (CUresult error, const char **pStr)

Gets the string description of an error code.

Parameters

error

- Error code to convert to string

pStr

- Address of the string pointer.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Sets *pStr to the address of a NULL-terminated string description of the error code error. If the error code is not recognized, [CUDA_ERROR_INVALID_VALUE](#) will be returned and *pStr will be set to the NULL address.

See also:

[CUresult](#), [cudaGetErrorString](#)

6.3. Initialization

This section describes the initialization functions of the low-level CUDA driver application programming interface.

CUresult cuInit (unsigned int Flags)

Initialize the CUDA driver API. Initializes the driver API and must be called before any other function from the driver API in the current process. Currently, the `Flags` parameter must be 0. If `cuInit()` has not been called, any function from the driver API will return `CUDA_ERROR_NOT_INITIALIZED`.

Parameters

Flags

- Initialization flag for CUDA.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_SYSTEM_DRIVER_MISMATCH](#),
[CUDA_ERROR_COMPAT_NOT_SUPPORTED_ON_DEVICE](#)

Description



Note:

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

6.4. Version Management

This section describes the version management functions of the low-level CUDA driver application programming interface.

CUresult cuDriverGetVersion (int *driverVersion)

Returns the latest CUDA version supported by driver.

Parameters

driverVersion

- Returns the CUDA driver version

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns in *driverVersion the 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.

This function automatically returns [CUDA_ERROR_INVALID_VALUE](#) if driverVersion is NULL.



Note:

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

See also:

[cuDriverGetVersion](#), [cudaRuntimeGetVersion](#)

6.5. Device Management

This section describes the device management functions of the low-level CUDA driver application programming interface.

CUresult cuDeviceGet (CUdevice *device, int ordinal)

Returns a handle to a compute device.

Parameters

device

- Returned device handle

ordinal

- Device number to get handle for

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_DEVICE](#)

Description

Returns in `*device` a device handle given an ordinal in the range `[0, cuDeviceGetCount\(\)-1]`.



Note:

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

See also:

[cuDeviceGetAttribute](#), [cuDeviceGetCount](#), [cuDeviceGetName](#), [cuDeviceGetUuid](#), [cuDeviceGetLuid](#),
[cuDeviceTotalMem](#), [cuDeviceGetExecAffinitySupport](#)

CUresult cuDeviceGetAttribute (int *pi, CUdevice_attribute attrib, CUdevice dev)

Returns information about the device.

Parameters

pi

- Returned device attribute value

attrib

- Device attribute to query

dev

- Device handle

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_DEVICE](#)

Description

Returns in `*pi` the integer value of the attribute `attrib` on device `dev`. The supported attributes are:

- ▶ [CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK](#): Maximum number of threads per block;
- ▶ [CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X](#): Maximum x-dimension of a block

- ▶ CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y: Maximum y-dimension of a block
- ▶ CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z: Maximum z-dimension of a block
- ▶ CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X: Maximum x-dimension of a grid
- ▶ CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y: Maximum y-dimension of a grid
- ▶ CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z: Maximum z-dimension of a grid
- ▶ CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK: Maximum amount of shared memory available to a thread block in bytes
- ▶ CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY: Memory available on device for `__constant__` variables in a CUDA C kernel in bytes
- ▶ CU_DEVICE_ATTRIBUTE_WARP_SIZE: Warp size in threads
- ▶ CU_DEVICE_ATTRIBUTE_MAX_PITCH: Maximum pitch in bytes allowed by the memory copy functions that involve memory regions allocated through `cuMemAllocPitch()`
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH: Maximum 1D texture width
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LINEAR_WIDTH: Maximum width for a 1D texture bound to linear memory
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_MIPMAPPED_WIDTH: Maximum mipmapped 1D texture width
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_WIDTH: Maximum 2D texture width
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_HEIGHT: Maximum 2D texture height
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_WIDTH: Maximum width for a 2D texture bound to linear memory
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_HEIGHT: Maximum height for a 2D texture bound to linear memory
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_PITCH: Maximum pitch in bytes for a 2D texture bound to linear memory
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_WIDTH: Maximum mipmapped 2D texture width
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_HEIGHT: Maximum mipmapped 2D texture height
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH: Maximum 3D texture width
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT: Maximum 3D texture height
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH: Maximum 3D texture depth
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH_ALTERNATE: Alternate maximum 3D texture width, 0 if no alternate maximum 3D texture size is supported
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT_ALTERNATE: Alternate maximum 3D texture height, 0 if no alternate maximum 3D texture size is supported
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH_ALTERNATE: Alternate maximum 3D texture depth, 0 if no alternate maximum 3D texture size is supported

- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_WIDTH: Maximum cubemap texture width or height
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH: Maximum 1D layered texture width
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS: Maximum layers in a 1D layered texture
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH: Maximum 2D layered texture width
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT: Maximum 2D layered texture height
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS: Maximum layers in a 2D layered texture
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_LAYERED_WIDTH: Maximum cubemap layered texture width or height
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_LAYERED_LAYERS: Maximum layers in a cubemap layered texture
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_WIDTH: Maximum 1D surface width
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_WIDTH: Maximum 2D surface width
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_HEIGHT: Maximum 2D surface height
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_WIDTH: Maximum 3D surface width
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_HEIGHT: Maximum 3D surface height
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_DEPTH: Maximum 3D surface depth
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_WIDTH: Maximum 1D layered surface width
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_LAYERS: Maximum layers in a 1D layered surface
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_WIDTH: Maximum 2D layered surface width
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_HEIGHT: Maximum 2D layered surface height
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_LAYERS: Maximum layers in a 2D layered surface
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_WIDTH: Maximum cubemap surface width
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED_WIDTH: Maximum cubemap layered surface width
- ▶ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED_LAYERS: Maximum layers in a cubemap layered surface

- ▶ CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK: Maximum number of 32-bit registers available to a thread block
- ▶ CU_DEVICE_ATTRIBUTE_CLOCK_RATE: The typical clock frequency in kilohertz
- ▶ CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT: Alignment requirement; texture base addresses aligned to textureAlign bytes do not need an offset applied to texture fetches
- ▶ CU_DEVICE_ATTRIBUTE_TEXTURE_PITCH_ALIGNMENT: Pitch alignment requirement for 2D texture references bound to pitched memory
- ▶ CU_DEVICE_ATTRIBUTE_GPU_OVERLAP: 1 if the device can concurrently copy memory between host and device while executing a kernel, or 0 if not
- ▶ CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT: Number of multiprocessors on the device
- ▶ CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT: 1 if there is a run time limit for kernels executed on the device, or 0 if not
- ▶ CU_DEVICE_ATTRIBUTE_INTEGRATED: 1 if the device is integrated with the memory subsystem, or 0 if not
- ▶ CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY: 1 if the device can map host memory into the CUDA address space, or 0 if not
- ▶ CU_DEVICE_ATTRIBUTE_COMPUTE_MODE: Compute mode that device is currently in. Available modes are as follows:
 - ▶ CU_COMPUTEMODE_DEFAULT: Default mode - Device is not restricted and can have multiple CUDA contexts present at a single time.
 - ▶ CU_COMPUTEMODE_PROHIBITED: Compute-prohibited mode - Device is prohibited from creating new CUDA contexts.
 - ▶ CU_COMPUTEMODE_EXCLUSIVE_PROCESS: Compute-exclusive-process mode - Device can have only one context used by a single process at a time.
- ▶ CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS: 1 if the device supports executing multiple kernels within the same context simultaneously, or 0 if not. It is not guaranteed that multiple kernels will be resident on the device concurrently so this feature should not be relied upon for correctness.
- ▶ CU_DEVICE_ATTRIBUTE_ECC_ENABLED: 1 if error correction is enabled on the device, 0 if error correction is disabled or not supported by the device
- ▶ CU_DEVICE_ATTRIBUTE_PCI_BUS_ID: PCI bus identifier of the device
- ▶ CU_DEVICE_ATTRIBUTE_PCI_DEVICE_ID: PCI device (also known as slot) identifier of the device
- ▶ CU_DEVICE_ATTRIBUTE_PCI_DOMAIN_ID: PCI domain identifier of the device
- ▶ CU_DEVICE_ATTRIBUTE_TCC_DRIVER: 1 if the device is using a TCC driver. TCC is only available on Tesla hardware running Windows Vista or later
- ▶ CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE: Peak memory clock frequency in kilohertz

- ▶ CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH: Global memory bus width in bits
- ▶ CU_DEVICE_ATTRIBUTE_L2_CACHE_SIZE: Size of L2 cache in bytes. 0 if the device doesn't have L2 cache
- ▶ CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR: Maximum resident threads per multiprocessor
- ▶ CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING: 1 if the device shares a unified address space with the host, or 0 if not
- ▶ CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MAJOR: Major compute capability version number
- ▶ CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MINOR: Minor compute capability version number
- ▶ CU_DEVICE_ATTRIBUTE_GLOBAL_L1_CACHE_SUPPORTED: 1 if device supports caching globals in L1 cache, 0 if caching globals in L1 cache is not supported by the device
- ▶ CU_DEVICE_ATTRIBUTE_LOCAL_L1_CACHE_SUPPORTED: 1 if device supports caching locals in L1 cache, 0 if caching locals in L1 cache is not supported by the device
- ▶ CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_MULTIPROCESSOR: Maximum amount of shared memory available to a multiprocessor in bytes; this amount is shared by all thread blocks simultaneously resident on a multiprocessor
- ▶ CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_MULTIPROCESSOR: Maximum number of 32-bit registers available to a multiprocessor; this number is shared by all thread blocks simultaneously resident on a multiprocessor
- ▶ CU_DEVICE_ATTRIBUTE_MANAGED_MEMORY: 1 if device supports allocating managed memory on this system, 0 if allocating managed memory is not supported by the device on this system.
- ▶ CU_DEVICE_ATTRIBUTE_MULTI_GPU_BOARD: 1 if device is on a multi-GPU board, 0 if not.
- ▶ CU_DEVICE_ATTRIBUTE_MULTI_GPU_BOARD_GROUP_ID: Unique identifier for a group of devices associated with the same board. Devices on the same multi-GPU board will share the same identifier.
- ▶ CU_DEVICE_ATTRIBUTE_HOST_NATIVE_ATOMIC_SUPPORTED: 1 if Link between the device and the host supports native atomic operations.
- ▶ CU_DEVICE_ATTRIBUTE_SINGLE_TO_DOUBLE_PRECISION_PERF_RATIO: Ratio of single precision performance (in floating-point operations per second) to double precision performance.
- ▶ CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS: Device supports coherently accessing pageable memory without calling `cudaHostRegister` on it.
- ▶ CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS: Device can coherently access managed memory concurrently with the CPU.

- ▶ CU_DEVICE_ATTRIBUTE_COMPUTE_PREEMPTION_SUPPORTED: Device supports Compute Preemption.
- ▶ CU_DEVICE_ATTRIBUTE_CAN_USE_HOST_POINTER_FOR_REGISTERED_MEM: Device can access host registered memory at the same virtual address as the CPU.
- ▶ CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK_OPTIN: The maximum per block shared memory size supported on this device. This is the maximum value that can be opted into when using the [cuFuncSetAttribute\(\)](#) or [cuKernelSetAttribute\(\)](#) call. For more details see [CU_FUNC_ATTRIBUTE_MAX_DYNAMIC_SHARED_SIZE_BYTES](#)
- ▶ CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES: Device accesses pageable memory via the host's page tables.
- ▶ CU_DEVICE_ATTRIBUTE_DIRECT_MANAGED_MEM_ACCESS_FROM_HOST: The host can directly access managed memory on the device without migration.
- ▶ CU_DEVICE_ATTRIBUTE_VIRTUAL_MEMORY_MANAGEMENT_SUPPORTED: Device supports virtual memory management APIs like [cuMemAddressReserve](#), [cuMemCreate](#), [cuMemMap](#) and related APIs
- ▶ CU_DEVICE_ATTRIBUTE_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR_SUPPORTED: Device supports exporting memory to a posix file descriptor with [cuMemExportToShareableHandle](#), if requested via [cuMemCreate](#)
- ▶ CU_DEVICE_ATTRIBUTE_HANDLE_TYPE_WIN32_HANDLE_SUPPORTED: Device supports exporting memory to a Win32 NT handle with [cuMemExportToShareableHandle](#), if requested via [cuMemCreate](#)
- ▶ CU_DEVICE_ATTRIBUTE_HANDLE_TYPE_WIN32_KMT_HANDLE_SUPPORTED: Device supports exporting memory to a Win32 KMT handle with [cuMemExportToShareableHandle](#), if requested via [cuMemCreate](#)
- ▶ CU_DEVICE_ATTRIBUTE_MAX_BLOCKS_PER_MULTIPROCESSOR: Maximum number of thread blocks that can reside on a multiprocessor
- ▶ CU_DEVICE_ATTRIBUTE_GENERIC_COMPRESSION_SUPPORTED: Device supports compressible memory allocation via [cuMemCreate](#)
- ▶ CU_DEVICE_ATTRIBUTE_MAX_PERSISTING_L2_CACHE_SIZE: Maximum L2 persisting lines capacity setting in bytes
- ▶ CU_DEVICE_ATTRIBUTE_MAX_ACCESS_POLICY_WINDOW_SIZE: Maximum value of [CUaccessPolicyWindow::num_bytes](#)
- ▶ CU_DEVICE_ATTRIBUTE_GPU_DIRECT_RDMA_WITH_CUDA_VMM_SUPPORTED: Device supports specifying the GPUDirect RDMA flag with [cuMemCreate](#).
- ▶ CU_DEVICE_ATTRIBUTE_RESERVED_SHARED_MEMORY_PER_BLOCK: Amount of shared memory per block reserved by CUDA driver in bytes
- ▶ CU_DEVICE_ATTRIBUTE_SPARSE_CUDA_ARRAY_SUPPORTED: Device supports sparse CUDA arrays and sparse CUDA mipmapped arrays.

- ▶ CU_DEVICE_ATTRIBUTE_READ_ONLY_HOST_REGISTER_SUPPORTED: Device supports using the cuMemHostRegister flag `CU_MEMHOSTREGISTER_READ_ONLY` to register memory that must be mapped as read-only to the GPU
- ▶ CU_DEVICE_ATTRIBUTE_MEMORY_POOLS_SUPPORTED: Device supports using the cuMemAllocAsync and cuMemPool family of APIs
- ▶ CU_DEVICE_ATTRIBUTE_GPU_DIRECT_RDMA_SUPPORTED: Device supports GPUDirect RDMA APIs, like `nvidia_p2p_get_pages` (see <https://docs.nvidia.com/cuda/gpudirect-rdma> for more information)
- ▶ CU_DEVICE_ATTRIBUTE_GPU_DIRECT_RDMA_FLUSH_WRITES_OPTIONS: The returned attribute shall be interpreted as a bitmask, where the individual bits are described by the CUflushGPUDirectRDMAWritesOptions enum
- ▶ CU_DEVICE_ATTRIBUTE_GPU_DIRECT_RDMA_WRITES_ORDERING: GPUDirect RDMA writes to the device do not need to be flushed for consumers within the scope indicated by the returned attribute. See CUGPUDirectRDMAWritesOrdering for the numerical values returned here.
- ▶ CU_DEVICE_ATTRIBUTE_MEMPOOL_SUPPORTED_HANDLE_TYPES: Bitmask of handle types supported with mempool based IPC
- ▶ CU_DEVICE_ATTRIBUTE_DEFERRED_MAPPING_CUDA_ARRAY_SUPPORTED: Device supports deferred mapping CUDA arrays and CUDA mipmapped arrays.
- ▶ CU_DEVICE_ATTRIBUTE_NUMA_CONFIG: NUMA configuration of a device: value is of type CUdeviceNumaConfig enum
- ▶ CU_DEVICE_ATTRIBUTE_NUMA_ID: NUMA node ID of the GPU memory
- ▶ CU_DEVICE_ATTRIBUTE_MULTICAST_SUPPORTED: Device supports switch multicast and reduction operations.
- ▶ CU_DEVICE_ATTRIBUTE_GPU_PCI_DEVICE_ID: The combined 16-bit PCI device ID and 16-bit PCI vendor ID.
- ▶ CU_DEVICE_ATTRIBUTE_GPU_PCI_SUBSYSTEM_ID: The combined 16-bit PCI subsystem ID and 16-bit PCI subsystem vendor ID. ID.

**Note:**

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

See also:

[cuDeviceGetCount](#), [cuDeviceGetName](#), [cuDeviceGetUuid](#), [cuDeviceGet](#), [cuDeviceTotalMem](#), [cuDeviceGetExecAffinitySupport](#), [cudaDeviceGetAttribute](#), [cudaGetDeviceProperties](#)

CUresult cuDeviceGetCount (int *count)

Returns the number of compute-capable devices.

Parameters

count

- Returned number of compute-capable devices

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns in *count the number of devices with compute capability greater than or equal to 2.0 that are available for execution. If there is no such device, [cuDeviceGetCount\(\)](#) returns 0.



Note:

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

See also:

[cuDeviceGetAttribute](#), [cuDeviceGetName](#), [cuDeviceGetUuid](#), [cuDeviceGetLuid](#), [cuDeviceGet](#), [cuDeviceTotalMem](#), [cuDeviceGetExecAffinitySupport](#), [cudaGetDeviceCount](#)

CUresult cuDeviceGetDefaultMemPool (CUmemoryPool *pool_out, CUdevice dev)

Returns the default mempool of a device.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_NOT_SUPPORTED](#)

Description

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



Note:

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

See also:

[cuMemAllocAsync](#), [cuMemPoolTrimTo](#), [cuMemPoolGetAttribute](#), [cuMemPoolSetAttribute](#), [cuMemPoolSetAccess](#), [cuDeviceGetMemPool](#), [cuMemPoolCreate](#)

CUresult cuDeviceGetExecAffinitySupport (int *pi, CUexecAffinityType type, CUdevice dev)

Returns information about the execution affinity support of the device.

Parameters

pi

- 1 if the execution affinity type `type` is supported by the device, or 0 if not

type

- Execution affinity type to query

dev

- Device handle

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

Returns in `*pi` whether execution affinity type `type` is supported by device `dev`. The supported types are:

- [CU_EXEC_AFFINITY_TYPE_SM_COUNT](#): 1 if context with limited SMs is supported by the device, or 0 if not;



Note:

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

See also:

[cuDeviceGetAttribute](#), [cuDeviceGetCount](#), [cuDeviceGetName](#), [cuDeviceGetUuid](#), [cuDeviceGet](#), [cuDeviceTotalMem](#)

CUresult cuDeviceGetLuid (char *luid, unsigned int *deviceNodeMask, CUdevice dev)

Return an LUID and device node mask for the device.

Parameters

luid

- Returned LUID

deviceNodeMask

- Returned device node mask

dev

- Device to get identifier string for

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

Return identifying information (luid and deviceNodeMask) to allow matching device with graphics APIs.

**Note:**

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

See also:

[cuDeviceGetAttribute](#), [cuDeviceGetCount](#), [cuDeviceGetName](#), [cuDeviceGet](#), [cuDeviceTotalMem](#), [cuDeviceGetExecAffinitySupport](#), [cudaGetDeviceProperties](#)

CUresult cuDeviceGetMemPool (CUmemoryPool *pool, CUdevice dev)

Gets the current mempool for a device.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns the last pool provided to [cuDeviceSetMemPool](#) for this device or the device's default memory pool if [cuDeviceSetMemPool](#) has never been called. By default the current mempool is the default mempool for a device. Otherwise the returned pool must have been set with [cuDeviceSetMemPool](#).

See also:

[cuDeviceGetDefaultMemPool](#), [cuMemPoolCreate](#), [cuDeviceSetMemPool](#)

CUresult cuDeviceGetName (char *name, int len, CUdevice dev)

Returns an identifier string for the device.

Parameters

name

- Returned identifier string for the device

len

- Maximum length of string to store in name

dev

- Device to get identifier string for

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

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



Note:

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

See also:

[cuDeviceGetAttribute](#), [cuDeviceGetUuid](#), [cuDeviceGetLuid](#), [cuDeviceGetCount](#), [cuDeviceGet](#), [cuDeviceTotalMem](#), [cuDeviceGetExecAffinitySupport](#), [cudaGetDeviceProperties](#)

CUresult cuDeviceGetNvSciSyncAttributes (void *nvSciSyncAttrList, CUdevice dev, int flags)

Return NvSciSync attributes that this device can support.

Parameters

nvSciSyncAttrList

- Return NvSciSync attributes supported.

dev

- Valid Cuda Device to get NvSciSync attributes for.

flags

- flags describing NvSciSync usage.

Description

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

If `NvSciSyncAttrKey_RequiredPerm` field in `nvSciSyncAttrList` is already set this API will return [CUDA_ERROR_INVALID_VALUE](#).

The applications should set `nvSciSyncAttrList` to a valid NvSciSyncAttrList failing which this API will return [CUDA_ERROR_INVALID_HANDLE](#).

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

- ▶ [CUDA_NVSCISYNC_ATTR_SIGNAL](#), specifies that the applications intends to signal an NvSciSync on this CUDA device.
- ▶ [CUDA_NVSCISYNC_ATTR_WAIT](#), specifies that the applications intends to wait on an NvSciSync on this CUDA device.

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

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

- ▶ `NvSciSyncAccessPerm_SignalOnly` if [CUDA_NVSCISYNC_ATTR_SIGNAL](#) is set in `flags`.
- ▶ `NvSciSyncAccessPerm_WaitOnly` if [CUDA_NVSCISYNC_ATTR_WAIT](#) is set in `flags`.
- ▶ `NvSciSyncAccessPerm_WaitSignal` if both [CUDA_NVSCISYNC_ATTR_WAIT](#) and [CUDA_NVSCISYNC_ATTR_SIGNAL](#) are set in `flags`. `NvSciSyncAttrKey_PrimitiveInfo` is set to

- ▶ `NvSciSyncAttrValPrimitiveType_SystememSemaphore` on any valid device.
- ▶ `NvSciSyncAttrValPrimitiveType_Syncpoint` if device is a Tegra device.
- ▶ `NvSciSyncAttrValPrimitiveType_SystememSemaphorePayload64b` if device is GA10X+. `NvSciSyncAttrKey_GpuId` is set to the same UUID that is returned for this device from [cuDeviceGetUuid](#).

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_NOT_SUPPORTED](#),
[CUDA_ERROR_OUT_OF_MEMORY](#)

See also:

[cuImportExternalSemaphore](#), [cuDestroyExternalSemaphore](#), [cuSignalExternalSemaphoresAsync](#),
[cuWaitExternalSemaphoresAsync](#)

CUresult cuDeviceGetTexture1DLinearMaxWidth (size_t *maxWidthInElements, CUarray_format format, unsigned numChannels, CUdevice dev)

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

Parameters

maxWidthInElements

- Returned maximum number of texture elements allocatable for given `format` and `numChannels`.

format

- Texture format.

numChannels

- Number of channels per texture element.

dev

- Device handle.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_DEVICE](#)

Description

Returns in `maxWidthInElements` the maximum number of texture elements allocatable in a 1D linear texture for given `format` and `numChannels`.

**Note:**

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

See also:

[cuDeviceGetAttribute](#), [cuDeviceGetCount](#), [cuDeviceGetName](#), [cuDeviceGetUuid](#), [cuDeviceGet](#), [cudaMemGetInfo](#), [cuDeviceTotalMem](#)

CUresult cuDeviceGetUuid (CUuuid *uuid, CUdevice dev)

Return an UUID for the device.

Parameters

uuid

- Returned UUID

dev

- Device to get identifier string for

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

Note there is a later version of this API, [cuDeviceGetUuid_v2](#). It will supplant this version in 12.0, which is retained for minor version compatibility.

Returns 16-octets identifying the device `dev` in the structure pointed by the `uuid`.

**Note:**

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

See also:

[cuDeviceGetUuid_v2](#), [cuDeviceGetAttribute](#), [cuDeviceGetCount](#), [cuDeviceGetName](#), [cuDeviceGetLuid](#), [cuDeviceGet](#), [cuDeviceTotalMem](#), [cuDeviceGetExecAffinitySupport](#), [cudaGetDeviceProperties](#)

CUresult cuDeviceGetUuid_v2 (CUuuid *uuid, CUdevice dev)

Return an UUID for the device (11.4+).

Parameters

uuid

- Returned UUID

dev

- Device to get identifier string for

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

Returns 16-octets identifying the device `dev` in the structure pointed by the `uuid`. If the device is in MIG mode, returns its MIG UUID which uniquely identifies the subscribed MIG compute instance.



Note:

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

See also:

[cuDeviceGetAttribute](#), [cuDeviceGetCount](#), [cuDeviceGetName](#), [cuDeviceGetLuid](#), [cuDeviceGet](#), [cuDeviceTotalMem](#), [cudaGetDeviceProperties](#)

CUresult cuDeviceSetMemPool (CUdevice dev, CUmemoryPool pool)

Sets the current memory pool of a device.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

The memory pool must be local to the specified device. [cuMemAllocAsync](#) allocates from the current mempool of the provided stream's device. By default, a device's current memory pool is its default memory pool.



Note:

Use [cuMemAllocFromPoolAsync](#) to specify asynchronous allocations from a device different than the one the stream runs on.

See also:

[cuDeviceGetDefaultMemPool](#), [cuDeviceGetMemPool](#), [cuMemPoolCreate](#), [cuMemPoolDestroy](#), [cuMemAllocFromPoolAsync](#)

CUresult cuDeviceTotalMem (size_t *bytes, CUdevice dev)

Returns the total amount of memory on the device.

Parameters

bytes

- Returned memory available on device in bytes

dev

- Device handle

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

Returns in **bytes* the total amount of memory available on the device *dev* in bytes.



Note:

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

See also:

[cuDeviceGetAttribute](#), [cuDeviceGetCount](#), [cuDeviceGetName](#), [cuDeviceGetUuid](#), [cuDeviceGet](#), [cuDeviceGetExecAffinitySupport](#), [cudaMemGetInfo](#)

CUresult cuFlushGPUDirectRDMAWrites (CUflushGPUDirectRDMAWritesTarget target, CUflushGPUDirectRDMAWritesScope scope)

Blocks until remote writes are visible to the specified scope.

Parameters

target

- The target of the operation, see [CUflushGPUDirectRDMAWritesTarget](#)

scope

- The scope of the operation, see [CUflushGPUDirectRDMAWritesScope](#)

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

Blocks until GPUDirect RDMA writes to the target context via mappings created through APIs like `nvidia_p2p_get_pages` (see <https://docs.nvidia.com/cuda/gpudirect-rdma> for more information), are visible to the specified scope.

If the scope equals or lies within the scope indicated by

[CU_DEVICE_ATTRIBUTE_GPU_DIRECT_RDMA_WRITES_ORDERING](#), the call will be a no-op and can be safely omitted for performance. This can be determined by comparing the numerical values between the two enums, with smaller scopes having smaller values.

On platforms that support GPUDirect RDMA writes via more than one path in hardware (see [CU_MEM_RANGE_FLAG_DMA_BUF_MAPPING_TYPE_PCIE](#)), the user should consider those paths as belonging to separate ordering domains. Note that in such cases CUDA driver will report both RDMA writes ordering and RDMA write scope as `ALL_DEVICES` and a call to `cuFlushGPUDirectRDMA` will be a no-op, but when these multiple paths are used simultaneously, it is the user's responsibility to ensure ordering by using mechanisms outside the scope of CUDA.

Users may query support for this API via

`CU_DEVICE_ATTRIBUTE_FLUSH_FLUSH_GPU_DIRECT_RDMA_OPTIONS`.



Note:

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

6.6. Device Management [DEPRECATED]

This section describes the device management functions of the low-level CUDA driver application programming interface.

CUresult cuDeviceComputeCapability (int *major, int *minor, CUdevice dev)

Returns the compute capability of the device.

Parameters

major

- Major revision number

minor

- Minor revision number

dev

- Device handle

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

Deprecated

This function was deprecated as of CUDA 5.0 and its functionality superseded by [cuDeviceGetAttribute\(\)](#).

Returns in *major and *minor the major and minor revision numbers that define the compute capability of the device dev.



Note:

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

See also:

[cuDeviceGetAttribute](#), [cuDeviceGetCount](#), [cuDeviceGetName](#), [cuDeviceGetUuid](#), [cuDeviceGet](#), [cuDeviceTotalMem](#)

CUresult cuDeviceGetProperties (CUdevprop *prop, CUdevice dev)

Returns properties for a selected device.

Parameters

prop

- Returned properties of device

dev

- Device to get properties for

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

Deprecated

This function was deprecated as of CUDA 5.0 and replaced by [cuDeviceGetAttribute\(\)](#).

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

```
typedef struct CUdevprop_st {
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    int sharedMemPerBlock;
    int totalConstantMemory;
    int SIMDWidth;
    int memPitch;
    int regsPerBlock;
    int clockRate;
    int textureAlign
} CUdevprop;
```

where:

- ▶ maxThreadsPerBlock is the maximum number of threads per block;
- ▶ maxThreadsDim[3] is the maximum sizes of each dimension of a block;
- ▶ maxGridSize[3] is the maximum sizes of each dimension of a grid;
- ▶ sharedMemPerBlock is the total amount of shared memory available per block in bytes;
- ▶ totalConstantMemory is the total amount of constant memory available on the device in bytes;
- ▶ SIMDWidth is the warp size;
- ▶ memPitch is the maximum pitch allowed by the memory copy functions that involve memory regions allocated through [cuMemAllocPitch\(\)](#);
- ▶ regsPerBlock is the total number of registers available per block;
- ▶ clockRate is the clock frequency in kilohertz;

- ▶ `textureAlign` is the alignment requirement; texture base addresses that are aligned to `textureAlign` bytes do not need an offset applied to texture fetches.



Note:

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

See also:

[cuDeviceGetAttribute](#), [cuDeviceGetCount](#), [cuDeviceGetName](#), [cuDeviceGetUuid](#), [cuDeviceGet](#), [cuDeviceTotalMem](#)

6.7. Primary Context Management

This section describes the primary context management functions of the low-level CUDA driver application programming interface.

The primary context is unique per device and shared with the CUDA runtime API. These functions allow integration with other libraries using CUDA.

CUresult cuDevicePrimaryCtxGetState (CUdevice dev, unsigned int *flags, int *active)

Get the state of the primary context.

Parameters

dev

- Device to get primary context flags for

flags

- Pointer to store flags

active

- Pointer to store context state; 0 = inactive, 1 = active

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

Returns in `*flags` the flags for the primary context of `dev`, and in `*active` whether it is active. See [cuDevicePrimaryCtxSetFlags](#) for flag values.

**Note:**

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

See also:

[cuDevicePrimaryCtxSetFlags](#), [cuCtxGetFlags](#), [cuCtxSetFlags](#), [cudaGetDeviceFlags](#)

CUresult cuDevicePrimaryCtxRelease (CUdevice dev)

Release the primary context on the GPU.

Parameters

dev

- Device which primary context is released

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_INVALID_CONTEXT](#)

Description

Releases the primary context interop on the device. A retained context should always be released once the user is done using it. The context is automatically reset once the last reference to it is released. This behavior is different when the primary context was retained by the CUDA runtime from CUDA 4.0 and earlier. In this case, the primary context remains always active.

Releasing a primary context that has not been previously retained will fail with [CUDA_ERROR_INVALID_CONTEXT](#).

Please note that unlike [cuCtxDestroy\(\)](#) this method does not pop the context from stack in any circumstances.

**Note:**

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

See also:

[cuDevicePrimaryCtxRetain](#), [cuCtxDestroy](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#)

CUresult cuDevicePrimaryCtxReset (CUdevice dev)

Destroy all allocations and reset all state on the primary context.

Parameters

dev

- Device for which primary context is destroyed

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_PRIMARY_CONTEXT_ACTIVE](#)

Description

Explicitly destroys and cleans up all resources associated with the current device in the current process.

Note that it is responsibility of the calling function to ensure that no other module in the process is using the device any more. For that reason it is recommended to use [cuDevicePrimaryCtxRelease\(\)](#) in most cases. However it is safe for other modules to call [cuDevicePrimaryCtxRelease\(\)](#) even after resetting the device. Resetting the primary context does not release it, an application that has retained the primary context should explicitly release its usage.



Note:

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

See also:

[cuDevicePrimaryCtxRetain](#), [cuDevicePrimaryCtxRelease](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#), [cudaDeviceReset](#)

CUresult cuDevicePrimaryCtxRetain (CUcontext *pctx, CUdevice dev)

Retain the primary context on the GPU.

Parameters

pctx

- Returned context handle of the new context

dev

- Device for which primary context is requested

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_UNKNOWN](#)

Description

Retains the primary context on the device. Once the user successfully retains the primary context, the primary context will be active and available to the user until the user releases it with [cuDevicePrimaryCtxRelease\(\)](#) or resets it with [cuDevicePrimaryCtxReset\(\)](#). Unlike [cuCtxCreate\(\)](#) the newly retained context is not pushed onto the stack.

Retaining the primary context for the first time will fail with [CUDA_ERROR_UNKNOWN](#) if the compute mode of the device is [CU_COMPUTEMODE_PROHIBITED](#). The function [cuDeviceGetAttribute\(\)](#) can be used with [CU_DEVICE_ATTRIBUTE_COMPUTE_MODE](#) to determine the compute mode of the device. The nvidia-smi tool can be used to set the compute mode for devices. Documentation for nvidia-smi can be obtained by passing a -h option to it.

Please note that the primary context always supports pinned allocations. Other flags can be specified by [cuDevicePrimaryCtxSetFlags\(\)](#).



Note:

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

See also:

[cuDevicePrimaryCtxRelease](#), [cuDevicePrimaryCtxSetFlags](#), [cuCtxCreate](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#)

CUresult cuDevicePrimaryCtxSetFlags (CUdevice dev, unsigned int flags)

Set flags for the primary context.

Parameters

dev

- Device for which the primary context flags are set

flags

- New flags for the device

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

Sets the flags for the primary context on the device overwriting perviously set ones.

The three LSBs of the `flags` parameter can be used to control how the OS thread, which owns the CUDA context at the time of an API call, interacts with the OS scheduler when waiting for results from the GPU. Only one of the scheduling flags can be set when creating a context.

- ▶ [CU_CTX_SCHED_SPIN](#): Instruct CUDA to actively spin when waiting for results from the GPU. This can decrease latency when waiting for the GPU, but may lower the performance of CPU threads if they are performing work in parallel with the CUDA thread.
- ▶ [CU_CTX_SCHED_YIELD](#): Instruct CUDA to yield its thread when waiting for results from the GPU. This can increase latency when waiting for the GPU, but can increase the performance of CPU threads performing work in parallel with the GPU.
- ▶ [CU_CTX_SCHED_BLOCKING_SYNC](#): Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the GPU to finish work.
- ▶ [CU_CTX_BLOCKING_SYNC](#): Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the GPU to finish work.

Deprecated: This flag was deprecated as of CUDA 4.0 and was replaced with [CU_CTX_SCHED_BLOCKING_SYNC](#).

- ▶ [CU_CTX_SCHED_AUTO](#): The default value if the `flags` parameter is zero, uses a heuristic based on the number of active CUDA contexts in the process `C` and the number of logical processors in the system `P`. If $C > P$, then CUDA will yield to other OS threads when waiting for the GPU ([CU_CTX_SCHED_YIELD](#)), otherwise CUDA will not yield while waiting for results and actively spin on the processor ([CU_CTX_SCHED_SPIN](#)). Additionally, on Tegra devices, [CU_CTX_SCHED_AUTO](#) uses a heuristic based on the power profile of the platform and may choose [CU_CTX_SCHED_BLOCKING_SYNC](#) for low-powered devices.
- ▶ [CU_CTX_LMEM_RESIZE_TO_MAX](#): Instruct CUDA to not reduce local memory after resizing local memory for a kernel. This can prevent thrashing by local memory allocations when launching many kernels with high local memory usage at the cost of potentially increased memory usage.

Deprecated: This flag is deprecated and the behavior enabled by this flag is now the default and cannot be disabled.
- ▶ [CU_CTX_COREDUMP_ENABLE](#): If GPU coredumps have not been enabled globally with [cuCoredumpSetAttributeGlobal](#) or environment variables, this flag can be set during context creation to instruct CUDA to create a coredump if this context raises an exception during execution. These environment variables are described in the CUDA-GDB user guide under the

"GPU core dump support" section. The initial settings will be taken from the global settings at the time of context creation. The other settings that control coredump output can be modified by calling [cuCoredumpSetAttribute](#) from the created context after it becomes current.

- ▶ [CU_CTX_USER_COREDUMP_ENABLE](#): If user-triggered GPU coredumps have not been enabled globally with [cuCoredumpSetAttributeGlobal](#) or environment variables, this flag can be set during context creation to instruct CUDA to create a coredump if data is written to a certain pipe that is present in the OS space. These environment variables are described in the CUDA-GDB user guide under the "GPU core dump support" section. It is important to note that the pipe name **must** be set with [cuCoredumpSetAttributeGlobal](#) before creating the context if this flag is used. Setting this flag implies that [CU_CTX_COREDUMP_ENABLE](#) is set. The initial settings will be taken from the global settings at the time of context creation. The other settings that control coredump output can be modified by calling [cuCoredumpSetAttribute](#) from the created context after it becomes current.
- ▶ [CU_CTX_SYNC_MEMOPS](#): Ensures that synchronous memory operations initiated on this context will always synchronize. See further documentation in the section titled "API Synchronization behavior" to learn more about cases when synchronous memory operations can exhibit asynchronous behavior.



Note:

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

See also:

[cuDevicePrimaryCtxRetain](#), [cuDevicePrimaryCtxGetState](#), [cuCtxCreate](#), [cuCtxGetFlags](#), [cuCtxSetFlags](#), [cudaSetDeviceFlags](#)

6.8. Context Management

This section describes the context management functions of the low-level CUDA driver application programming interface.

Please note that some functions are described in [Primary Context Management](#) section.

CUresult cuCtxCreate (CUcontext *pctx, unsigned int flags, CUdevice dev)

Create a CUDA context.

Parameters

pctx

- Returned context handle of the new context

flags

- Context creation flags

dev

- Device to create context on

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_UNKNOWN](#)

Description



Note:

In most cases it is recommended to use [cuDevicePrimaryCtxRetain](#).

Creates a new CUDA context and associates it with the calling thread. The `flags` parameter is described below. The context is created with a usage count of 1 and the caller of [cuCtxCreate\(\)](#) must call [cuCtxDestroy\(\)](#) when done using the context. If a context is already current to the thread, it is supplanted by the newly created context and may be restored by a subsequent call to [cuCtxPopCurrent\(\)](#).

The three LSBs of the `flags` parameter can be used to control how the OS thread, which owns the CUDA context at the time of an API call, interacts with the OS scheduler when waiting for results from the GPU. Only one of the scheduling flags can be set when creating a context.

- ▶ [CU_CTX_SCHED_SPIN](#): Instruct CUDA to actively spin when waiting for results from the GPU. This can decrease latency when waiting for the GPU, but may lower the performance of CPU threads if they are performing work in parallel with the CUDA thread.
- ▶ [CU_CTX_SCHED_YIELD](#): Instruct CUDA to yield its thread when waiting for results from the GPU. This can increase latency when waiting for the GPU, but can increase the performance of CPU threads performing work in parallel with the GPU.
- ▶ [CU_CTX_SCHED_BLOCKING_SYNC](#): Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the GPU to finish work.

- ▶ [CU_CTX_BLOCKING_SYNC](#): Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the GPU to finish work.

Deprecated: This flag was deprecated as of CUDA 4.0 and was replaced with [CU_CTX_SCHED_BLOCKING_SYNC](#).

- ▶ [CU_CTX_SCHED_AUTO](#): The default value if the `flags` parameter is zero, uses a heuristic based on the number of active CUDA contexts in the process `C` and the number of logical processors in the system `P`. If $C > P$, then CUDA will yield to other OS threads when waiting for the GPU ([CU_CTX_SCHED_YIELD](#)), otherwise CUDA will not yield while waiting for results and actively spin on the processor ([CU_CTX_SCHED_SPIN](#)). Additionally, on Tegra devices, [CU_CTX_SCHED_AUTO](#) uses a heuristic based on the power profile of the platform and may choose [CU_CTX_SCHED_BLOCKING_SYNC](#) for low-powered devices.
- ▶ [CU_CTX_MAP_HOST](#): Instruct CUDA to support mapped pinned allocations. This flag must be set in order to allocate pinned host memory that is accessible to the GPU.
- ▶ [CU_CTX_LMEM_RESIZE_TO_MAX](#): Instruct CUDA to not reduce local memory after resizing local memory for a kernel. This can prevent thrashing by local memory allocations when launching many kernels with high local memory usage at the cost of potentially increased memory usage.

Deprecated: This flag is deprecated and the behavior enabled by this flag is now the default and cannot be disabled. Instead, the per-thread stack size can be controlled with [cuCtxSetLimit\(\)](#).

- ▶ [CU_CTX_COREDUMP_ENABLE](#): If GPU coredumps have not been enabled globally with [cuCoredumpSetAttributeGlobal](#) or environment variables, this flag can be set during context creation to instruct CUDA to create a coredump if this context raises an exception during execution. These environment variables are described in the CUDA-GDB user guide under the "GPU core dump support" section. The initial attributes will be taken from the global attributes at the time of context creation. The other attributes that control coredump output can be modified by calling [cuCoredumpSetAttribute](#) from the created context after it becomes current.
- ▶ [CU_CTX_USER_COREDUMP_ENABLE](#): If user-triggered GPU coredumps have not been enabled globally with [cuCoredumpSetAttributeGlobal](#) or environment variables, this flag can be set during context creation to instruct CUDA to create a coredump if data is written to a certain pipe that is present in the OS space. These environment variables are described in the CUDA-GDB user guide under the "GPU core dump support" section. It is important to note that the pipe name `*must*` be set with [cuCoredumpSetAttributeGlobal](#) before creating the context if this flag is used. Setting this flag implies that [CU_CTX_COREDUMP_ENABLE](#) is set. The initial attributes will be taken from the global attributes at the time of context creation. The other attributes that control coredump output can be modified by calling [cuCoredumpSetAttribute](#) from the created context after it becomes current. Setting this flag on any context creation is equivalent to setting the `CU_COREDUMP_ENABLE_USER_TRIGGER` attribute to `true` globally.
- ▶ [CU_CTX_SYNC_MEMOPS](#): Ensures that synchronous memory operations initiated on this context will always synchronize. See further documentation in the section titled "API Synchronization behavior" to learn more about cases when synchronous memory operations can exhibit asynchronous behavior.

Context creation will fail with [CUDA_ERROR_UNKNOWN](#) if the compute mode of the device is [CU_COMPUTEMODE_PROHIBITED](#). The function [cuDeviceGetAttribute\(\)](#) can be used with [CU_DEVICE_ATTRIBUTE_COMPUTE_MODE](#) to determine the compute mode of the device. The `nvidia-smi` tool can be used to set the compute mode for * devices. Documentation for `nvidia-smi` can be obtained by passing a `-h` option to it.



Note:

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

See also:

[cuCtxDestroy](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCoredumpSetAttributeGlobal](#), [cuCoredumpSetAttribute](#), [cuCtxSynchronize](#)

CUresult cuCtxCreate_v3 (CUcontext *pctx, CUexecAffinityParam *paramsArray, int numParams, unsigned int flags, CUdevice dev)

Create a CUDA context with execution affinity.

Parameters

pctx

- Returned context handle of the new context

paramsArray

- Execution affinity parameters

numParams

- Number of execution affinity parameters

flags

- Context creation flags

dev

- Device to create context on

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_UNSUPPORTED_EXEC_AFFINITY](#), [CUDA_ERROR_UNKNOWN](#)

Description

Creates a new CUDA context with execution affinity and associates it with the calling thread. The `paramsArray` and `flags` parameter are described below. The context is created with a usage count of 1 and the caller of `cuCtxCreate()` must call `cuCtxDestroy()` when done using the context. If a context is already current to the thread, it is supplanted by the newly created context and may be restored by a subsequent call to `cuCtxPopCurrent()`.

The type and the amount of execution resource the context can use is limited by `paramsArray` and `numParams`. The `paramsArray` is an array of `CUexecAffinityParam` and the `numParams` describes the size of the array. If two `CUexecAffinityParam` in the array have the same type, the latter execution affinity parameter overrides the former execution affinity parameter. The supported execution affinity types are:

- ▶ `CU_EXEC_AFFINITY_TYPE_SM_COUNT` limits the portion of SMs that the context can use. The portion of SMs is specified as the number of SMs via `CUexecAffinitySmCount`. This limit will be internally rounded up to the next hardware-supported amount. Hence, it is imperative to query the actual execution affinity of the context via `cuCtxGetExecAffinity` after context creation. Currently, this attribute is only supported under Volta+ MPS.

The three LSBs of the `flags` parameter can be used to control how the OS thread, which owns the CUDA context at the time of an API call, interacts with the OS scheduler when waiting for results from the GPU. Only one of the scheduling flags can be set when creating a context.

- ▶ `CU_CTX_SCHED_SPIN`: Instruct CUDA to actively spin when waiting for results from the GPU. This can decrease latency when waiting for the GPU, but may lower the performance of CPU threads if they are performing work in parallel with the CUDA thread.
- ▶ `CU_CTX_SCHED_YIELD`: Instruct CUDA to yield its thread when waiting for results from the GPU. This can increase latency when waiting for the GPU, but can increase the performance of CPU threads performing work in parallel with the GPU.
- ▶ `CU_CTX_SCHED_BLOCKING_SYNC`: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the GPU to finish work.
- ▶ `CU_CTX_BLOCKING_SYNC`: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the GPU to finish work.

Deprecated: This flag was deprecated as of CUDA 4.0 and was replaced with `CU_CTX_SCHED_BLOCKING_SYNC`.

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

- ▶ [CU_CTX_MAP_HOST](#): Instruct CUDA to support mapped pinned allocations. This flag must be set in order to allocate pinned host memory that is accessible to the GPU.
- ▶ [CU_CTX_LMEM_RESIZE_TO_MAX](#): Instruct CUDA to not reduce local memory after resizing local memory for a kernel. This can prevent thrashing by local memory allocations when launching many kernels with high local memory usage at the cost of potentially increased memory usage.

Deprecated: This flag is deprecated and the behavior enabled by this flag is now the default and cannot be disabled. Instead, the per-thread stack size can be controlled with [cuCtxSetLimit\(\)](#).
- ▶ [CU_CTX_COREDUMP_ENABLE](#): If GPU coredumps have not been enabled globally with [cuCoredumpSetAttributeGlobal](#) or environment variables, this flag can be set during context creation to instruct CUDA to create a coredump if this context raises an exception during execution. These environment variables are described in the CUDA-GDB user guide under the "GPU core dump support" section. The initial attributes will be taken from the global attributes at the time of context creation. The other attributes that control coredump output can be modified by calling [cuCoredumpSetAttribute](#) from the created context after it becomes current.
- ▶ [CU_CTX_USER_COREDUMP_ENABLE](#): If user-triggered GPU coredumps have not been enabled globally with [cuCoredumpSetAttributeGlobal](#) or environment variables, this flag can be set during context creation to instruct CUDA to create a coredump if data is written to a certain pipe that is present in the OS space. These environment variables are described in the CUDA-GDB user guide under the "GPU core dump support" section. It is important to note that the pipe name **must** be set with [cuCoredumpSetAttributeGlobal](#) before creating the context if this flag is used. Setting this flag implies that [CU_CTX_COREDUMP_ENABLE](#) is set. The initial attributes will be taken from the global attributes at the time of context creation. The other attributes that control coredump output can be modified by calling [cuCoredumpSetAttribute](#) from the created context after it becomes current. Setting this flag on any context creation is equivalent to setting the `CU_COREDUMP_ENABLE_USER_TRIGGER` attribute to `true` globally.

Context creation will fail with [CUDA_ERROR_UNKNOWN](#) if the compute mode of the device is [CU_COMPUTEMODE_PROHIBITED](#). The function [cuDeviceGetAttribute\(\)](#) can be used with [CU_DEVICE_ATTRIBUTE_COMPUTE_MODE](#) to determine the compute mode of the device. The `nvidia-smi` tool can be used to set the compute mode for * devices. Documentation for `nvidia-smi` can be obtained by passing a `-h` option to it.



Note:

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

See also:

[cuCtxDestroy](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#), [cuCoredumpSetAttributeGlobal](#), [cuCoredumpSetAttribute](#), [CUexecAffinityParam](#)

CUresult cuCtxCreate_v4 (CUcontext *pctx, CUctxCreateParams *ctxCreateParams, unsigned int flags, CUdevice dev)

Create a CUDA context.

Parameters

pctx

- Returned context handle of the new context

ctxCreateParams

- Context creation parameters

flags

- Context creation flags

dev

- Device to create context on

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_DEVICE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_SUPPORTED, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Description

Creates a new CUDA context and associates it with the calling thread. The `flags` parameter is described below. The context is created with a usage count of 1 and the caller of [cuCtxCreate\(\)](#) must call [cuCtxDestroy\(\)](#) when done using the context. If a context is already current to the thread, it is supplanted by the newly created context and may be restored by a subsequent call to [cuCtxPopCurrent\(\)](#).

CUDA context can be created with execution affinity. The type and the amount of execution resource the context can use is limited by `paramsArray` and `numExecAffinityParams` in `execAffinity`. The `paramsArray` is an array of `CUexecAffinityParam` and the `numExecAffinityParams` describes the size of the `paramsArray`. If two `CUexecAffinityParam` in the array have the same type, the latter execution affinity parameter overrides the former execution affinity parameter. The supported execution affinity types are:

- CU_EXEC_AFFINITY_TYPE_SM_COUNT limits the portion of SMs that the context can use. The portion of SMs is specified as the number of SMs via `CUexecAffinitySmCount`. This limit will be internally rounded up to the next hardware-supported amount. Hence, it is imperative to query the actual execution affinity of the context via `cuCtxGetExecAffinity` after context creation. Currently, this attribute is only supported under Volta+ MPS.

CUDA context can be created in CIG(CUDA in Graphics) mode by setting `cigParams`. Data from graphics client is shared with CUDA via the `sharedData` in `cigParams`. Support for D3D12 graphics client can be determined using `cuDeviceGetAttribute()` with `CU_DEVICE_ATTRIBUTE_D3D12_CIG_SUPPORTED`. `sharedData` is a `ID3D12CommandQueue` handle. Either `execAffinityParams` or `cigParams` can be set to a non-null value. Setting both to a non-null value will result in an undefined behavior.

The three LSBs of the `flags` parameter can be used to control how the OS thread, which owns the CUDA context at the time of an API call, interacts with the OS scheduler when waiting for results from the GPU. Only one of the scheduling flags can be set when creating a context.

- ▶ [CU_CTX_SCHED_SPIN](#): Instruct CUDA to actively spin when waiting for results from the GPU. This can decrease latency when waiting for the GPU, but may lower the performance of CPU threads if they are performing work in parallel with the CUDA thread.
- ▶ [CU_CTX_SCHED_YIELD](#): Instruct CUDA to yield its thread when waiting for results from the GPU. This can increase latency when waiting for the GPU, but can increase the performance of CPU threads performing work in parallel with the GPU.
- ▶ [CU_CTX_SCHED_BLOCKING_SYNC](#): Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the GPU to finish work.
- ▶ [CU_CTX_BLOCKING_SYNC](#): Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the GPU to finish work.

Deprecated: This flag was deprecated as of CUDA 4.0 and was replaced with [CU_CTX_SCHED_BLOCKING_SYNC](#).

- ▶ [CU_CTX_SCHED_AUTO](#): The default value if the `flags` parameter is zero, uses a heuristic based on the number of active CUDA contexts in the process `C` and the number of logical processors in the system `P`. If $C > P$, then CUDA will yield to other OS threads when waiting for the GPU ([CU_CTX_SCHED_YIELD](#)), otherwise CUDA will not yield while waiting for results and actively spin on the processor ([CU_CTX_SCHED_SPIN](#)). Additionally, on Tegra devices, [CU_CTX_SCHED_AUTO](#) uses a heuristic based on the power profile of the platform and may choose [CU_CTX_SCHED_BLOCKING_SYNC](#) for low-powered devices.
 - ▶ [CU_CTX_MAP_HOST](#): Instruct CUDA to support mapped pinned allocations. This flag must be set in order to allocate pinned host memory that is accessible to the GPU.
 - ▶ [CU_CTX_LMEM_RESIZE_TO_MAX](#): Instruct CUDA to not reduce local memory after resizing local memory for a kernel. This can prevent thrashing by local memory allocations when launching many kernels with high local memory usage at the cost of potentially increased memory usage.
- Deprecated: This flag is deprecated and the behavior enabled by this flag is now the default and cannot be disabled. Instead, the per-thread stack size can be controlled with [cuCtxSetLimit\(\)](#).
- ▶ [CU_CTX_COREDUMP_ENABLE](#): If GPU coredumps have not been enabled globally with [cuCoredumpSetAttributeGlobal](#) or environment variables, this flag can be set during context creation to instruct CUDA to create a coredump if this context raises an exception during

execution. These environment variables are described in the CUDA-GDB user guide under the "GPU core dump support" section. The initial attributes will be taken from the global attributes at the time of context creation. The other attributes that control coredump output can be modified by calling [cuCoredumpSetAttribute](#) from the created context after it becomes current. This flag is not supported when CUDA context is created in CIG(CUDA in Graphics) mode.

- ▶ [CU_CTX_USER_COREDUMP_ENABLE](#): If user-triggered GPU coredumps have not been enabled globally with [cuCoredumpSetAttributeGlobal](#) or environment variables, this flag can be set during context creation to instruct CUDA to create a coredump if data is written to a certain pipe that is present in the OS space. These environment variables are described in the CUDA-GDB user guide under the "GPU core dump support" section. It is important to note that the pipe name **must** be set with [cuCoredumpSetAttributeGlobal](#) before creating the context if this flag is used. Setting this flag implies that [CU_CTX_COREDUMP_ENABLE](#) is set. The initial attributes will be taken from the global attributes at the time of context creation. The other attributes that control coredump output can be modified by calling [cuCoredumpSetAttribute](#) from the created context after it becomes current. Setting this flag on any context creation is equivalent to setting the `CU_COREDUMP_ENABLE_USER_TRIGGER` attribute to `true` globally. This flag is not supported when CUDA context is created in CIG(CUDA in Graphics) mode.
- ▶ [CU_CTX_SYNC_MEMOPS](#): Ensures that synchronous memory operations initiated on this context will always synchronize. See further documentation in the section titled "API Synchronization behavior" to learn more about cases when synchronous memory operations can exhibit asynchronous behavior.

Context creation will fail with [CUDA_ERROR_UNKNOWN](#) if the compute mode of the device is [CU_COMPUTEMODE_PROHIBITED](#). The function [cuDeviceGetAttribute\(\)](#) can be used with [CU_DEVICE_ATTRIBUTE_COMPUTE_MODE](#) to determine the compute mode of the device. The `nvidia-smi` tool can be used to set the compute mode for * devices. Documentation for `nvidia-smi` can be obtained by passing a `-h` option to it.

Context creation will fail with `:: CUDA_ERROR_INVALID_VALUE` if invalid parameter was passed by client to create the CUDA context.

Context creation in CIG mode will fail with [CUDA_ERROR_NOT_SUPPORTED](#) if CIG is not supported by the device or the driver.



Note:

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

See also:

[cuCtxDestroy](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCoredumpSetAttributeGlobal](#), [cuCoredumpSetAttribute](#), [cuCtxSynchronize](#)

CUresult cuCtxDestroy (CUcontext ctx)

Destroy a CUDA context.

Parameters

ctx

- Context to destroy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Destroys the CUDA context specified by `ctx`. The context `ctx` will be destroyed regardless of how many threads it is current to. It is the responsibility of the calling function to ensure that no API call issues using `ctx` while [cuCtxDestroy\(\)](#) is executing.

Destroys and cleans up all resources associated with the context. It is the caller's responsibility to ensure that the context or its resources are not accessed or passed in subsequent API calls and doing so will result in undefined behavior. These resources include CUDA types [CUmodule](#), [CUfunction](#), [CUstream](#), [CUevent](#), [CUarray](#), [CUmipmappedArray](#), [CUTexObject](#), [CUSurfObject](#), [CUTexref](#), [CUSurfref](#), [CUgraphicsResource](#), [CULinkState](#), [CUexternalMemory](#) and [CUexternalSemaphore](#). These resources also include memory allocations by [cuMemAlloc\(\)](#), [cuMemAllocHost\(\)](#), [cuMemAllocManaged\(\)](#) and [cuMemAllocPitch\(\)](#).

If `ctx` is current to the calling thread then `ctx` will also be popped from the current thread's context stack (as though [cuCtxPopCurrent\(\)](#) were called). If `ctx` is current to other threads, then `ctx` will remain current to those threads, and attempting to access `ctx` from those threads will result in the error [CUDA_ERROR_CONTEXT_IS_DESTROYED](#).



Note:

[cuCtxDestroy\(\)](#) will not destroy memory allocations by [cuMemCreate\(\)](#), [cuMemAllocAsync\(\)](#) and [cuMemAllocFromPoolAsync\(\)](#). These memory allocations are not associated with any CUDA context and need to be destroyed explicitly.



Note:

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

See also:

[cuCtxCreate](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#)

CUresult cuCtxGetApiVersion (CUcontext ctx, unsigned int *version)

Gets the context's API version.

Parameters

ctx

- Context to check

version

- Pointer to version

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_UNKNOWN](#)

Description

Returns a version number in `version` corresponding to the capabilities of the context (e.g. 3010 or 3020), which library developers can use to direct callers to a specific API version. If `ctx` is NULL, returns the API version used to create the currently bound context.

Note that new API versions are only introduced when context capabilities are changed that break binary compatibility, so the API version and driver version may be different. For example, it is valid for the API version to be 3020 while the driver version is 4020.



Note:

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

See also:

[cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#)

CUresult cuCtxGetCacheConfig (CUfunc_cache *pconfig)

Returns the preferred cache configuration for the current context.

Parameters

pconfig

- Returned cache configuration

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

On devices where the L1 cache and shared memory use the same hardware resources, this function returns through `pconfig` the preferred cache configuration for the current context. This is only a preference. The driver 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 `pconfig` of [CU_FUNC_CACHE_PREFER_NONE](#) on devices where the size of the L1 cache and shared memory are fixed.

The supported cache configurations are:

- ▶ [CU_FUNC_CACHE_PREFER_NONE](#): no preference for shared memory or L1 (default)
- ▶ [CU_FUNC_CACHE_PREFER_SHARED](#): prefer larger shared memory and smaller L1 cache
- ▶ [CU_FUNC_CACHE_PREFER_L1](#): prefer larger L1 cache and smaller shared memory
- ▶ [CU_FUNC_CACHE_PREFER_EQUAL](#): prefer equal sized L1 cache and shared memory



Note:

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

See also:

[cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxGetApiVersion](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#), [cuFuncSetCacheConfig](#), [cudaDeviceGetCacheConfig](#)

CUresult cuCtxGetCurrent (CUcontext *pctx)

Returns the CUDA context bound to the calling CPU thread.

Parameters

pctx

- Returned context handle

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),

Description

Returns in *pctx the CUDA context bound to the calling CPU thread. If no context is bound to the calling CPU thread then *pctx is set to NULL and [CUDA_SUCCESS](#) is returned.



Note:

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

See also:

[cuCtxSetCurrent](#), [cuCtxCreate](#), [cuCtxDestroy](#), [cudaGetDevice](#)

CUresult cuCtxGetDevice (CUdevice *device)

Returns the device handle for the current context.

Parameters

device

- Returned device handle for the current context

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

Returns in *device the handle of the current context's device.



Note:

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

See also:

[cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetFlags](#),
[cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#),
[cuCtxSynchronize](#), [cudaGetDevice](#)

CUresult cuCtxGetExecAffinity (CUexecAffinityParam *pExecAffinity, CUexecAffinityType type)

Returns the execution affinity setting for the current context.

Parameters

pExecAffinity

- Returned execution affinity

type

- Execution affinity type to query

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_UNSUPPORTED_EXEC_AFFINITY](#)

Description

Returns in *pExecAffinity the current value of type. The supported [CUexecAffinityType](#) values are:

- [CU_EXEC_AFFINITY_TYPE_SM_COUNT](#): number of SMs the context is limited to use.



Note:

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

See also:

[CUexecAffinityParam](#)

CUresult cuCtxGetFlags (unsigned int *flags)

Returns the flags for the current context.

Parameters

flags

- Pointer to store flags of current context

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

Returns in `*flags` the flags of the current context. See [cuCtxCreate](#) for flag values.



Note:

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

See also:

[cuCtxCreate](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetCurrent](#), [cuCtxGetDevice](#), [cuCtxGetLimit](#), [cuCtxGetSharedMemConfig](#), [cuCtxGetStreamPriorityRange](#), [cuCtxSetFlags](#), [cudaGetDeviceFlags](#)

CUresult cuCtxGetId (CUcontext ctx, unsigned long long *ctxId)

Returns the unique Id associated with the context supplied.

Parameters

ctx

- Context for which to obtain the Id

ctxId

- Pointer to store the Id of the context

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_CONTEXT_IS_DESTROYED](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns in `ctxId` the unique Id which is associated with a given context. The Id is unique for the life of the program for this instance of CUDA. If context is supplied as NULL and there is one current, the Id of the current context is returned.



Note:

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

See also:

[cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPushCurrent](#)

CUresult cuCtxGetLimit (size_t *pvalue, CUlimit limit)

Returns resource limits.

Parameters

pvalue

- Returned size of limit

limit

- Limit to query

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_UNSUPPORTED_LIMIT](#)

Description

Returns in *pvalue the current size of limit. The supported [CUlimit](#) values are:

- ▶ [CU_LIMIT_STACK_SIZE](#): stack size in bytes of each GPU thread.
- ▶ [CU_LIMIT_PRINTF_FIFO_SIZE](#): size in bytes of the FIFO used by the printf() device system call.
- ▶ [CU_LIMIT_MALLOC_HEAP_SIZE](#): size in bytes of the heap used by the malloc() and free() device system calls.
- ▶ [CU_LIMIT_DEV_RUNTIME_SYNC_DEPTH](#): maximum grid depth at which a thread can issue the device runtime call [cudaDeviceSynchronize\(\)](#) to wait on child grid launches to complete.
- ▶ [CU_LIMIT_DEV_RUNTIME_PENDING_LAUNCH_COUNT](#): maximum number of outstanding device runtime launches that can be made from this context.
- ▶ [CU_LIMIT_MAX_L2_FETCH_GRANULARITY](#): L2 cache fetch granularity.
- ▶ [CU_LIMIT_PERSISTING_L2_CACHE_SIZE](#): Persisting L2 cache size in bytes



Note:

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

See also:

[cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#), [cudaDeviceGetLimit](#)

CUresult cuCtxGetStreamPriorityRange (int *leastPriority, int *greatestPriority)

Returns numerical values that correspond to the least and greatest stream priorities.

Parameters

leastPriority

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

greatestPriority

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

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

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

This function will return '0' in both **leastPriority* and **greatestPriority* if the current context's device does not support stream priorities (see [cuDeviceGetAttribute](#)).



Note:

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

See also:

[cuStreamCreateWithPriority](#), [cuStreamGetPriority](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#), [cudaDeviceGetStreamPriorityRange](#)

CUresult cuCtxPopCurrent (CUcontext *pctx)

Pops the current CUDA context from the current CPU thread.

Parameters

pctx

- Returned popped context handle

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#)

Description

Pops the current CUDA context from the CPU thread and passes back the old context handle in *pctx. That context may then be made current to a different CPU thread by calling [cuCtxPushCurrent\(\)](#).

If a context was current to the CPU thread before [cuCtxCreate\(\)](#) or [cuCtxPushCurrent\(\)](#) was called, this function makes that context current to the CPU thread again.



Note:

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

See also:

[cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#)

CUresult cuCtxPushCurrent (CUcontext ctx)

Pushes a context on the current CPU thread.

Parameters

ctx

- Context to push

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Pushes the given context `ctx` onto the CPU thread's stack of current contexts. The specified context becomes the CPU thread's current context, so all CUDA functions that operate on the current context are affected.

The previous current context may be made current again by calling [cuCtxDestroy\(\)](#) or [cuCtxPopCurrent\(\)](#).



Note:

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

See also:

[cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#)

CUresult cuCtxRecordEvent (CUcontext hCtx, CUevent hEvent)

Records an event.

Parameters

hCtx

- Context to record event for

hEvent

- Event to record

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_STREAM_CAPTURE_UNSUPPORTED](#)

Description

Captures in `hEvent` all the activities of the context `hCtx` at the time of this call. `hEvent` and `hCtx` must be from the same CUDA context, otherwise [CUDA_ERROR_INVALID_HANDLE](#) will be returned. Calls such as [cuEventQuery\(\)](#) or [cuCtxWaitEvent\(\)](#) will then examine or wait for completion of the work that was captured. Uses of `hCtx` after this call do not modify `hEvent`. If the context passed to `hCtx` is the primary context, `hEvent` will capture all the activities of the primary context

and its green contexts. If the context passed to `hCtx` is a context converted from green context via [cuCtxFromGreenCtx\(\)](#), `hEvent` will capture only the activities of the green context.



Note:

The API will return [CUDA_ERROR_STREAM_CAPTURE_UNSUPPORTED](#) if the specified context `hCtx` has a stream in the capture mode. In such a case, the call will invalidate all the conflicting captures.

See also:

[cuCtxWaitEvent](#), [cuGreenCtxRecordEvent](#), [cuGreenCtxWaitEvent](#), [cuEventRecord](#)

CUresult cuCtxResetPersistingL2Cache (void)

Resets all persisting lines in cache to normal status.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_NOT_SUPPORTED](#)

Description

[cuCtxResetPersistingL2Cache](#) 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:

[CUaccessPolicyWindow](#)

CUresult cuCtxSetCacheConfig (CUfunc_cache config)

Sets the preferred cache configuration for the current context.

Parameters

config

- Requested cache configuration

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

On devices where the L1 cache and shared memory use the same hardware resources, this sets through `config` the preferred cache configuration for the current context. This is only a preference. The driver 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 [cuFuncSetCacheConfig\(\)](#) or [cuKernelSetCacheConfig\(\)](#) will be preferred over this context-wide setting. Setting the context-wide cache configuration to `CU_FUNC_CACHE_PREFER_NONE` 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:

- ▶ [CU_FUNC_CACHE_PREFER_NONE](#): no preference for shared memory or L1 (default)
- ▶ [CU_FUNC_CACHE_PREFER_SHARED](#): prefer larger shared memory and smaller L1 cache
- ▶ [CU_FUNC_CACHE_PREFER_L1](#): prefer larger L1 cache and smaller shared memory
- ▶ [CU_FUNC_CACHE_PREFER_EQUAL](#): prefer equal sized L1 cache and shared memory



Note:

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

See also:

[cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#), [cuFuncSetCacheConfig](#), [cudaDeviceSetCacheConfig](#), [cuKernelSetCacheConfig](#)

CUresult cuCtxSetCurrent (CUcontext ctx)

Binds the specified CUDA context to the calling CPU thread.

Parameters

ctx

- Context to bind to the calling CPU thread

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#)

Description

Binds the specified CUDA context to the calling CPU thread. If `ctx` is NULL then the CUDA context previously bound to the calling CPU thread is unbound and [CUDA_SUCCESS](#) is returned.

If there exists a CUDA context stack on the calling CPU thread, this will replace the top of that stack with `ctx`. If `ctx` is NULL then this will be equivalent to popping the top of the calling CPU thread's CUDA context stack (or a no-op if the calling CPU thread's CUDA context stack is empty).



Note:

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

See also:

[cuCtxGetCurrent](#), [cuCtxCreate](#), [cuCtxDestroy](#), [cudaSetDevice](#)

CUresult cuCtxSetFlags (unsigned int flags)

Sets the flags for the current context.

Parameters

flags

- Flags to set on the current context

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

Sets the flags for the current context overwriting previously set ones. See [cuDevicePrimaryCtxSetFlags](#) for flag values.



Note:

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

See also:

[cuCtxCreate](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetCurrent](#), [cuCtxGetDevice](#), [cuCtxGetLimit](#), [cuCtxGetSharedMemConfig](#), [cuCtxGetStreamPriorityRange](#), [cuCtxGetFlags](#), [cudaGetDeviceFlags](#), [cuDevicePrimaryCtxSetFlags](#),

CUresult cuCtxSetLimit (CUlimit limit, size_t value)

Set resource limits.

Parameters

limit

- Limit to set

value

- Size of limit

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_UNSUPPORTED_LIMIT](#), [CUDA_ERROR_OUT_OF_MEMORY](#),
[CUDA_ERROR_INVALID_CONTEXT](#)

Description

Setting `limit` to `value` is a request by the application to update the current limit maintained by the context. 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 [cuCtxGetLimit\(\)](#) to find out exactly what the limit has been set to.

Setting each [CUlimit](#) has its own specific restrictions, so each is discussed here.

- ▶ [CU_LIMIT_STACK_SIZE](#) controls the stack size in bytes of each GPU thread. The driver automatically increases the per-thread stack size for each kernel launch as needed. This size isn't reset back to the original value after each launch. Setting this value will take effect immediately, and if necessary, the device will block until all preceding requested tasks are complete.
- ▶ [CU_LIMIT_PRINTF_FIFO_SIZE](#) controls the size in bytes of the FIFO used by the `printf()` device system call. Setting [CU_LIMIT_PRINTF_FIFO_SIZE](#) must be performed before launching any kernel that uses the `printf()` device system call, otherwise [CUDA_ERROR_INVALID_VALUE](#) will be returned.
- ▶ [CU_LIMIT_MALLOC_HEAP_SIZE](#) controls the size in bytes of the heap used by the `malloc()` and `free()` device system calls. Setting [CU_LIMIT_MALLOC_HEAP_SIZE](#) must be performed before launching any kernel that uses the `malloc()` or `free()` device system calls, otherwise [CUDA_ERROR_INVALID_VALUE](#) will be returned.
- ▶ [CU_LIMIT_DEV_RUNTIME_SYNC_DEPTH](#) controls the maximum nesting depth of a grid at which a thread can safely call [cudaDeviceSynchronize\(\)](#). Setting this limit must be performed before any launch of a kernel that uses the device runtime and calls [cudaDeviceSynchronize\(\)](#) above the default sync depth, two levels of grids. Calls to [cudaDeviceSynchronize\(\)](#) will fail with error code [cudaErrorSyncDepthExceeded](#) if the limitation is violated. This limit can be set smaller than the default or up the maximum launch depth of 24. When setting this limit, keep in mind that additional levels of sync depth require the driver to reserve large amounts of

device memory which can no longer be used for user allocations. If these reservations of device memory fail, `cuCtxSetLimit()` will return `CUDA_ERROR_OUT_OF_MEMORY`, and the limit can be reset to a lower value. This limit is only applicable to devices of compute capability < 9.0. Attempting to set this limit on devices of other compute capability versions will result in the error `CUDA_ERROR_UNSUPPORTED_LIMIT` being returned.

- ▶ `CU_LIMIT_DEV_RUNTIME_PENDING_LAUNCH_COUNT` controls the maximum number of outstanding device runtime launches that can be made from the current context. A grid is outstanding from the point of launch up until the grid is known to have been completed. Device runtime launches which violate this limitation fail and return `cudaErrorLaunchPendingCountExceeded` when `cudaGetLastError()` is called after launch. If more pending launches than the default (2048 launches) are needed for a module using the device runtime, this limit can be increased. Keep in mind that being able to sustain additional pending launches will require the driver to reserve larger amounts of device memory upfront which can no longer be used for allocations. If these reservations fail, `cuCtxSetLimit()` will return `CUDA_ERROR_OUT_OF_MEMORY`, and the limit can be reset to a lower value. This limit is only applicable to devices of compute capability 3.5 and higher. Attempting to set this limit on devices of compute capability less than 3.5 will result in the error `CUDA_ERROR_UNSUPPORTED_LIMIT` being returned.
- ▶ `CU_LIMIT_MAX_L2_FETCH_GRANULARITY` controls the L2 cache fetch granularity. Values can range from 0B to 128B. This is purely a performance hint and it can be ignored or clamped depending on the platform.
- ▶ `CU_LIMIT_PERSISTING_L2_CACHE_SIZE` controls size in bytes available for persisting L2 cache. This is purely a performance hint and it can be ignored or clamped depending on the platform.



Note:

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

See also:

`cuCtxCreate`, `cuCtxDestroy`, `cuCtxGetApiVersion`, `cuCtxGetCacheConfig`, `cuCtxGetDevice`, `cuCtxGetFlags`, `cuCtxGetLimit`, `cuCtxPopCurrent`, `cuCtxPushCurrent`, `cuCtxSetCacheConfig`, `cuCtxSynchronize`, `cudaDeviceSetLimit`

CUresult cuCtxSynchronize (void)

Block for the current context's tasks to complete.

Returns

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`

Description

Blocks until the current context has completed all preceding requested tasks. If the current context is the primary context, green contexts that have been created will also be synchronized. [cuCtxSynchronize\(\)](#) returns an error if one of the preceding tasks failed. If the context was created with the [CU_CTX_SCHED_BLOCKING_SYNC](#) flag, the CPU thread will block until the GPU context has finished its work.



Note:

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

See also:

[cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cudaDeviceSynchronize](#)

CUresult cuCtxWaitEvent (CUcontext hCtx, CUevent hEvent)

Make a context wait on an event.

Parameters

hCtx

- Context to wait

hEvent

- Event to wait on

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_STREAM_CAPTURE_UNSUPPORTED](#)

Description

Makes all future work submitted to context `hCtx` wait for all work captured in `hEvent`. The synchronization will be performed on the device and will not block the calling CPU thread. See [cuCtxRecordEvent\(\)](#) for details on what is captured by an event. If the context passed to `hCtx` is the primary context, the primary context and its green contexts will wait for `hEvent`. If the context passed

to `hCtx` is a context converted from green context via [cuCtxFromGreenCtx\(\)](#), the green context will wait for `hEvent`.



Note:

- ▶ `hEvent` may be from a different context or device than `hCtx`.
- ▶ The API will return [CUDA_ERROR_STREAM_CAPTURE_UNSUPPORTED](#) and invalidate the capture if the specified event `hEvent` is part of an ongoing capture sequence or if the specified context `hCtx` has a stream in the capture mode.

See also:

[cuCtxRecordEvent](#), [cuGreenCtxRecordEvent](#), [cuGreenCtxWaitEvent](#), [cuStreamWaitEvent](#)

6.9. Context Management [DEPRECATED]

This section describes the deprecated context management functions of the low-level CUDA driver application programming interface.

CUresult cuCtxAttach (CUcontext *pctx, unsigned int flags)

Increment a context's usage-count.

Parameters

pctx

- Returned context handle of the current context

flags

- Context attach flags (must be 0)

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Note that this function is deprecated and should not be used.

Increments the usage count of the context and passes back a context handle in `*pctx` that must be passed to [cuCtxDetach\(\)](#) when the application is done with the context. [cuCtxAttach\(\)](#) fails if there is no context current to the thread.

Currently, the `flags` parameter must be 0.



Note:

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

See also:

[cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxDetach](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#)

CUresult cuCtxDetach (CUcontext ctx)

Decrement a context's usage-count.

Parameters

ctx

- Context to destroy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#)

Description

Deprecated

Note that this function is deprecated and should not be used.

Decrements the usage count of the context `ctx`, and destroys the context if the usage count goes to 0. The context must be a handle that was passed back by [cuCtxCreate\(\)](#) or [cuCtxAttach\(\)](#), and must be current to the calling thread.



Note:

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

See also:

[cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#)

CUresult cuCtxGetSharedMemConfig (CUsharedconfig *pConfig)

Returns the current shared memory configuration for the current context.

Parameters

pConfig

- returned shared memory configuration

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

This function will return in `pConfig` the current size of shared memory banks in the current context. On devices with configurable shared memory banks, [cuCtxSetSharedMemConfig](#) can be used to change this setting, so that all subsequent kernel launches will by default use the new bank size. When [cuCtxGetSharedMemConfig](#) 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:

- ▶ [CU_SHARED_MEM_CONFIG_FOUR_BYTE_BANK_SIZE](#): shared memory bank width is four bytes.
- ▶ [CU_SHARED_MEM_CONFIG_EIGHT_BYTE_BANK_SIZE](#): shared memory bank width will be eight bytes.



Note:

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

See also:

[cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#), [cuCtxGetSharedMemConfig](#), [cuFuncSetCacheConfig](#), [cudaDeviceGetSharedMemConfig](#)

CUresult cuCtxSetSharedMemConfig (CUsharedconfig config)

Sets the shared memory configuration for the current context.

Parameters

config

- requested shared memory configuration

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Deprecated

On devices with configurable shared memory banks, this function will set the context's shared memory bank size which is used for subsequent kernel launches.

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

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:

- ▶ CU_SHARED_MEM_CONFIG_DEFAULT_BANK_SIZE: set bank width to the default initial setting (currently, four bytes).
- ▶ CU_SHARED_MEM_CONFIG_FOUR_BYTE_BANK_SIZE: set shared memory bank width to be natively four bytes.
- ▶ CU_SHARED_MEM_CONFIG_EIGHT_BYTE_BANK_SIZE: set shared memory bank width to be natively eight bytes.



Note:

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

See also:

[cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#),
[cuCtxGetDevice](#), [cuCtxGetFlags](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#),
[cuCtxSetLimit](#), [cuCtxSynchronize](#), [cuCtxGetSharedMemConfig](#), [cuFuncSetCacheConfig](#),
[cudaDeviceSetSharedMemConfig](#)

6.10. Module Management

This section describes the module management functions of the low-level CUDA driver application programming interface.

enum CUmoduleLoadingMode

CUDA Lazy Loading status

Values

CU_MODULE_EAGER_LOADING = 0x1

Lazy Kernel Loading is not enabled

CU_MODULE_LAZY_LOADING = 0x2

Lazy Kernel Loading is enabled

CUresult cuLinkAddData (CUlinkState state,
CUjitInputType type, void *data, size_t size, const char
*name, unsigned int numOptions, CUjit_option *options,
void **optionValues)

Add an input to a pending linker invocation.

Parameters

state

A pending linker action.

type

The type of the input data.

data

The input data. PTX must be NULL-terminated.

size

The length of the input data.

name

An optional name for this input in log messages.

numOptions

Size of options.

options

Options to be applied only for this input (overrides options from [cuLinkCreate](#)).

optionValues

Array of option values, each cast to void *.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_IMAGE](#), [CUDA_ERROR_INVALID_PTX](#),
[CUDA_ERROR_UNSUPPORTED_PTX_VERSION](#), [CUDA_ERROR_OUT_OF_MEMORY](#),
[CUDA_ERROR_NO_BINARY_FOR_GPU](#)

Description

Ownership of data is retained by the caller. No reference is retained to any inputs after this call returns.

This method accepts only compiler options, which are used if the data must be compiled from PTX, and does not accept any of [CU_JIT_WALL_TIME](#), [CU_JIT_INFO_LOG_BUFFER](#), [CU_JIT_ERROR_LOG_BUFFER](#), [CU_JIT_TARGET_FROM_CUCONTEXT](#), or [CU_JIT_TARGET](#).



Note:

For LTO-IR input, only LTO-IR compiled with toolkits prior to CUDA 12.0 will be accepted

See also:

[cuLinkCreate](#), [cuLinkAddFile](#), [cuLinkComplete](#), [cuLinkDestroy](#)

**CUresult cuLinkAddFile (CUlinkState state,
 CUjitInputType type, const char *path, unsigned int
 numOptions, CUjit_option *options, void **optionValues)**

Add a file input to a pending linker invocation.

Parameters**state**

A pending linker action

type

The type of the input data

path

Path to the input file

numOptions

Size of options

options

Options to be applied only for this input (overrides options from [cuLinkCreate](#))

optionValues

Array of option values, each cast to void *

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_FILE_NOT_FOUND](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_IMAGE](#),
[CUDA_ERROR_INVALID_PTX](#), [CUDA_ERROR_UNSUPPORTED_PTX_VERSION](#),
[CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_NO_BINARY_FOR_GPU](#)

Description

No reference is retained to any inputs after this call returns.

This method accepts only compiler options, which are used if the input must be compiled from PTX, and does not accept any of [CU_JIT_WALL_TIME](#), [CU_JIT_INFO_LOG_BUFFER](#), [CU_JIT_ERROR_LOG_BUFFER](#), [CU_JIT_TARGET_FROM_CUCONTEXT](#), or [CU_JIT_TARGET](#).

This method is equivalent to invoking [cuLinkAddData](#) on the contents of the file.

**Note:**

For LTO-IR input, only LTO-IR compiled with toolkits prior to CUDA 12.0 will be accepted

See also:

[cuLinkCreate](#), [cuLinkAddData](#), [cuLinkComplete](#), [cuLinkDestroy](#)

CUresult cuLinkComplete (CUlinkState state, void **cubinOut, size_t *sizeOut)

Complete a pending linker invocation.

Parameters**state**

A pending linker invocation

cubinOut

On success, this will point to the output image

sizeOut

Optional parameter to receive the size of the generated image

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Completes the pending linker action and returns the cubin image for the linked device code, which can be used with [cuModuleLoadData](#). The cubin is owned by `state`, so it should be loaded before `state` is destroyed via [cuLinkDestroy](#). This call does not destroy `state`.

See also:

[cuLinkCreate](#), [cuLinkAddData](#), [cuLinkAddFile](#), [cuLinkDestroy](#), [cuModuleLoadData](#)

CUresult cuLinkCreate (unsigned int numOptions, CUjit_option *options, void **optionValues, CULinkState *stateOut)

Creates a pending JIT linker invocation.

Parameters

numOptions

Size of options arrays

options

Array of linker and compiler options

optionValues

Array of option values, each cast to void *

stateOut

On success, this will contain a CULinkState to specify and complete this action

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_JIT_COMPILER_NOT_FOUND](#)

Description

If the call is successful, the caller owns the returned CULinkState, which should eventually be destroyed with [cuLinkDestroy](#). The device code machine size (32 or 64 bit) will match the calling application.

Both linker and compiler options may be specified. Compiler options will be applied to inputs to this linker action which must be compiled from PTX. The options [CU_JIT_WALL_TIME](#), [CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES](#), and [CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES](#) will accumulate data until the CULinkState is destroyed.

The data passed in via [cuLinkAddData](#) and [cuLinkAddFile](#) will be treated as relocatable (`-rdc=true` to `nvcc`) when linking the final cubin during [cuLinkComplete](#) and will have similar consequences as offline relocatable device code linking.

`optionValues` must remain valid for the life of the `CULinkState` if output options are used. No other references to inputs are maintained after this call returns.



Note:

For LTO-IR input, only LTO-IR compiled with toolkits prior to CUDA 12.0 will be accepted



Note:

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

See also:

[cuLinkAddData](#), [cuLinkAddFile](#), [cuLinkComplete](#), [cuLinkDestroy](#)

CUresult cuLinkDestroy (CULinkState state)

Destroys state for a JIT linker invocation.

Parameters

state

State object for the linker invocation

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#)

Description

See also:

[cuLinkCreate](#)

CUresult cuModuleEnumerateFunctions (CUfunction *functions, unsigned int numFunctions, CUmodule mod)

Returns the function handles within a module.

Parameters

functions

- Buffer where the function handles are returned to

numFunctions

- Maximum number of function handles may be returned to the buffer

mod

- Module to query from

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns in `functions` a maximum number of `numFunctions` function handles within `mod`. When function loading mode is set to LAZY the function retrieved may be partially loaded. The loading state of a function can be queried using `cuFunctionIsLoaded`. CUDA APIs may load the function automatically when called with partially loaded function handle which may incur additional latency. Alternatively, `cuFunctionLoad` can be used to explicitly load a function. The returned function handles become invalid when the module is unloaded.

See also:

[cuModuleGetFunction](#), [cuModuleGetFunctionCount](#), [cuFuncIsLoaded](#), [cuFuncLoad](#)

CUresult cuModuleGetFunction (CUfunction *hfunc, CUmodule hmod, const char *name)

Returns a function handle.

Parameters**hfunc**

- Returned function handle

hmod

- Module to retrieve function from

name

- Name of function to retrieve

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_FOUND](#)

Description

Returns in `*hfunc` the handle of the function of name `name` located in module `hmod`. If no function of that name exists, [cuModuleGetFunction\(\)](#) returns [CUDA_ERROR_NOT_FOUND](#).

**Note:**

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

See also:

[cuModuleGetGlobal](#), [cuModuleGetTexRef](#), [cuModuleLoad](#), [cuModuleLoadData](#),
[cuModuleLoadDataEx](#), [cuModuleLoadFatBinary](#), [cuModuleUnload](#)

CUresult cuModuleGetFunctionCount (unsigned int *count, CUmodule mod)

Returns the number of functions within a module.

Parameters

count

- Number of functions found within the module

mod

- Module to query

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns in `count` the number of functions in `mod`.

CUresult cuModuleGetGlobal (CUdeviceptr *dptr, size_t *bytes, CUmodule hmod, const char *name)

Returns a global pointer from a module.

Parameters

dptr

- Returned global device pointer

bytes

- Returned global size in bytes

hmod

- Module to retrieve global from

name

- Name of global to retrieve

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_NOT_FOUND](#)

Description

Returns in `*dptr` and `*bytes` the base pointer and size of the global of name `name` located in module `hmod`. If no variable of that name exists, [cuModuleGetGlobal\(\)](#) returns [CUDA_ERROR_NOT_FOUND](#). One of the parameters `dptr` or `bytes` (not both) can be NULL in which case it is ignored.



Note:

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

See also:

[cuModuleGetFunction](#), [cuModuleGetTexRef](#), [cuModuleLoad](#), [cuModuleLoadData](#),
[cuModuleLoadDataEx](#), [cuModuleLoadFatBinary](#), [cuModuleUnload](#), [cudaGetSymbolAddress](#),
[cudaGetSymbolSize](#)

CUresult cuModuleGetLoadingMode (CUmoduleLoadingMode *mode)

Query lazy loading mode.

Parameters

mode

- Returns the lazy loading mode

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

Returns lazy loading mode Module loading mode is controlled by `CUDA_MODULE_LOADING` env variable



Note:

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

See also:

[cuModuleLoad](#),

CUresult cuModuleLoad (CUmodule *module, const char *fname)

Loads a compute module.

Parameters

module

- Returned module

fname

- Filename of module to load

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_PTX](#), [CUDA_ERROR_UNSUPPORTED_PTX_VERSION](#), [CUDA_ERROR_NOT_FOUND](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_FILE_NOT_FOUND](#), [CUDA_ERROR_NO_BINARY_FOR_GPU](#), [CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND](#), [CUDA_ERROR_SHARED_OBJECT_INIT_FAILED](#), [CUDA_ERROR_JIT_COMPILER_NOT_FOUND](#)

Description

Takes a filename `fname` and loads the corresponding module `module` into the current context. The CUDA driver API does not attempt to lazily allocate the resources needed by a module; if the memory for functions and data (constant and global) needed by the module cannot be allocated, [cuModuleLoad\(\)](#) fails. 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 from toolchain 4.0 or later.



Note:

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

See also:

[cuModuleGetFunction](#), [cuModuleGetGlobal](#), [cuModuleGetTexRef](#), [cuModuleLoadData](#), [cuModuleLoadDataEx](#), [cuModuleLoadFatBinary](#), [cuModuleUnload](#)

CUresult cuModuleLoadData (CUmodule *module, const void *image)

Load a module's data.

Parameters

module

- Returned module

image

- Module data to load

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_PTX](#), [CUDA_ERROR_UNSUPPORTED_PTX_VERSION](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_NO_BINARY_FOR_GPU](#), [CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND](#), [CUDA_ERROR_SHARED_OBJECT_INIT_FAILED](#), [CUDA_ERROR_JIT_COMPILER_NOT_FOUND](#)

Description

Takes a pointer `image` and loads the corresponding module `module` into the current context. The `image` may be a cubin or fatbin as output by `nvcc`, or a NULL-terminated PTX, either as output by `nvcc` or hand-written.



Note:

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

See also:

[cuModuleGetFunction](#), [cuModuleGetGlobal](#), [cuModuleGetTexRef](#), [cuModuleLoad](#), [cuModuleLoadDataEx](#), [cuModuleLoadFatBinary](#), [cuModuleUnload](#)

CUresult cuModuleLoadDataEx (CUmodule *module, const void *image, unsigned int numOptions, CUjit_option *options, void **optionValues)

Load a module's data with options.

Parameters

module

- Returned module

image

- Module data to load

numOptions

- Number of options

options

- Options for JIT

optionValues

- Option values for JIT

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_PTX](#), [CUDA_ERROR_UNSUPPORTED_PTX_VERSION](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_NO_BINARY_FOR_GPU](#), [CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND](#), [CUDA_ERROR_SHARED_OBJECT_INIT_FAILED](#), [CUDA_ERROR_JIT_COMPILER_NOT_FOUND](#)

Description

Takes a pointer `image` and loads the corresponding module `module` into the current context. The `image` may be a cubin or fatbin as output by nvcc, or a NULL-terminated PTX, either as output by nvcc or hand-written.



Note:

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

See also:

[cuModuleGetFunction](#), [cuModuleGetGlobal](#), [cuModuleGetTexRef](#), [cuModuleLoad](#), [cuModuleLoadData](#), [cuModuleLoadFatBinary](#), [cuModuleUnload](#)

CUresult cuModuleLoadFatBinary (CUmodule *module, const void *fatCubin)

Load a module's data.

Parameters

module

- Returned module

fatCubin

- Fat binary to load

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_PTX](#), [CUDA_ERROR_UNSUPPORTED_PTX_VERSION](#), [CUDA_ERROR_NOT_FOUND](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_NO_BINARY_FOR_GPU](#), [CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND](#), [CUDA_ERROR_SHARED_OBJECT_INIT_FAILED](#), [CUDA_ERROR_JIT_COMPILER_NOT_FOUND](#)

Description

Takes a pointer `fatCubin` and loads the corresponding module `module` into the current context. The pointer represents a fat binary object, which is a collection of different cubin and/or PTX files, all representing the same device code, but compiled and optimized for different architectures.

Prior to CUDA 4.0, there was no documented API for constructing and using fat binary objects by programmers. Starting with CUDA 4.0, fat binary objects can be constructed by providing the `-fatbin` option to `nvcc`. More information can be found in the `nvcc` document.



Note:

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

See also:

[cuModuleGetFunction](#), [cuModuleGetGlobal](#), [cuModuleGetTexRef](#), [cuModuleLoad](#), [cuModuleLoadData](#), [cuModuleLoadDataEx](#), [cuModuleUnload](#)

CUresult cuModuleUnload (CUmodule hmod)

Unloads a module.

Parameters

hmod

- Module to unload

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_PERMITTED](#)

Description

Unloads a module `hmod` from the current context. Attempting to unload a module which was obtained from the Library Management API such as [cuLibraryGetModule](#) will return [CUDA_ERROR_NOT_PERMITTED](#).



Note:

- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ Use of the handle after this call is undefined behavior.

See also:

[cuModuleGetFunction](#), [cuModuleGetGlobal](#), [cuModuleGetTexRef](#), [cuModuleLoad](#), [cuModuleLoadData](#), [cuModuleLoadDataEx](#), [cuModuleLoadFatBinary](#)

6.11. Module Management [DEPRECATED]

This section describes the deprecated module management functions of the low-level CUDA driver application programming interface.

CUresult cuModuleGetSurfRef (CUsurfref *pSurfRef, CUmodule hmod, const char *name)

Returns a handle to a surface reference.

Parameters

pSurfRef

- Returned surface reference

hmod

- Module to retrieve surface reference from

name

- Name of surface reference to retrieve

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_FOUND](#)

Description

Deprecated

Returns in *pSurfRef the handle of the surface reference of name name in the module hmod. If no surface reference of that name exists, [cuModuleGetSurfRef\(\)](#) returns [CUDA_ERROR_NOT_FOUND](#).



Note:

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

See also:

[cuModuleGetFunction](#), [cuModuleGetGlobal](#), [cuModuleGetTexRef](#), [cuModuleLoad](#), [cuModuleLoadData](#), [cuModuleLoadDataEx](#), [cuModuleLoadFatBinary](#), [cuModuleUnload](#)

CUresult cuModuleGetTexRef (CUtexref *pTexRef, CUmodule hmod, const char *name)

Returns a handle to a texture reference.

Parameters

pTexRef

- Returned texture reference

hmod

- Module to retrieve texture reference from

name

- Name of texture reference to retrieve

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_NOT_FOUND](#)

DescriptionDeprecated

Returns in *pTexRef the handle of the texture reference of name name in the module hmod. If no texture reference of that name exists, [cuModuleGetTexRef\(\)](#) returns [CUDA_ERROR_NOT_FOUND](#). This texture reference handle should not be destroyed, since it will be destroyed when the module is unloaded.

**Note:**

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

See also:

[cuModuleGetFunction](#), [cuModuleGetGlobal](#), [cuModuleGetSurfRef](#), [cuModuleLoad](#),
[cuModuleLoadData](#), [cuModuleLoadDataEx](#), [cuModuleLoadFatBinary](#), [cuModuleUnload](#)

6.12. Library Management

This section describes the library management functions of the low-level CUDA driver application programming interface.

**CUresult cuKernelGetAttribute (int *pi,
 CUfunction_attribute attrib, CUkernel kernel, CUdevice
 dev)**

Returns information about a kernel.

Parameters**pi**

- Returned attribute value

attrib

- Attribute requested

kernel

- Kernel to query attribute of

dev

- Device to query attribute of

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE,
CUDA_ERROR_INVALID_DEVICE

Description

Returns in *pi the integer value of the attribute `attrib` for the kernel `kernel` for the requested device `dev`. The supported attributes are:

- ▶ CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK: The maximum number of threads per block, beyond which a launch of the kernel would fail. This number depends on both the kernel and the requested device.
- ▶ CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES: The size in bytes of statically-allocated shared memory per block required by this kernel. This does not include dynamically-allocated shared memory requested by the user at runtime.
- ▶ CU_FUNC_ATTRIBUTE_CONST_SIZE_BYTES: The size in bytes of user-allocated constant memory required by this kernel.
- ▶ CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES: The size in bytes of local memory used by each thread of this kernel.
- ▶ CU_FUNC_ATTRIBUTE_NUM_REGS: The number of registers used by each thread of this kernel.
- ▶ CU_FUNC_ATTRIBUTE_PTX_VERSION: The PTX virtual architecture version for which the kernel 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. Note that this may return the undefined value of 0 for cubins compiled prior to CUDA 3.0.
- ▶ CU_FUNC_ATTRIBUTE_BINARY_VERSION: The binary architecture version for which the kernel 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. Note that this will return a value of 10 for legacy cubins that do not have a properly-encoded binary architecture version.
- ▶ CU_FUNC_CACHE_MODE_CA: The attribute to indicate whether the kernel has been compiled with user specified option "-Xptxas --dlcm=ca" set.
- ▶ CU_FUNC_ATTRIBUTE_MAX_DYNAMIC_SHARED_SIZE_BYTES: The maximum size in bytes of dynamically-allocated shared memory.
- ▶ CU_FUNC_ATTRIBUTE_PREFERRED_SHARED_MEMORY_CARVEOUT: Preferred shared memory-L1 cache split ratio in percent of total shared memory.

- ▶ [CU_FUNC_ATTRIBUTE_CLUSTER_SIZE_MUST_BE_SET](#): If this attribute is set, the kernel must launch with a valid cluster size specified.
- ▶ [CU_FUNC_ATTRIBUTE_REQUIRED_CLUSTER_WIDTH](#): The required cluster width in blocks.
- ▶ [CU_FUNC_ATTRIBUTE_REQUIRED_CLUSTER_HEIGHT](#): The required cluster height in blocks.
- ▶ [CU_FUNC_ATTRIBUTE_REQUIRED_CLUSTER_DEPTH](#): The required cluster depth in blocks.
- ▶ [CU_FUNC_ATTRIBUTE_NON_PORTABLE_CLUSTER_SIZE_ALLOWED](#): Indicates 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. 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.
- ▶ [CU_FUNC_ATTRIBUTE_CLUSTER_SCHEDULING_POLICY_PREFERENCE](#): The block scheduling policy of a function. The value type is `CUclusterSchedulingPolicy`.

**Note:**

If another thread is trying to set the same attribute on the same device using [cuKernelSetAttribute\(\)](#) simultaneously, the attribute query will give the old or new value depending on the interleavings chosen by the OS scheduler and memory consistency.

See also:

[cuLibraryLoadData](#), [cuLibraryLoadFromFile](#), [cuLibraryUnload](#), [cuKernelSetAttribute](#), [cuLibraryGetKernel](#), [cuLaunchKernel](#), [cuKernelGetFunction](#), [cuLibraryGetModule](#), [cuModuleGetFunction](#), [cuFuncGetAttribute](#)

CUresult cuKernelGetFunction (CUfunction *pFunc, CUkernel kernel)

Returns a function handle.

Parameters

pFunc

- Returned function handle

kernel

- Kernel to retrieve function for the requested context

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_NOT_FOUND](#), [CUDA_ERROR_INVALID_CONTEXT](#),
[CUDA_ERROR_CONTEXT_IS_DESTROYED](#)

Description

Returns in `pFunc` the handle of the function for the requested kernel `kernel` and the current context. If function handle is not found, the call returns [CUDA_ERROR_NOT_FOUND](#).

See also:

[cuLibraryLoadData](#), [cuLibraryLoadFromFile](#), [cuLibraryUnload](#), [cuLibraryGetKernel](#),
[cuLibraryGetModule](#), [cuModuleGetFunction](#)

CUresult cuKernelGetLibrary (CUlibrary *pLib, CUkernel kernel)

Returns a library handle.

Parameters

pLib

- Returned library handle

kernel

- Kernel to retrieve library handle

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_NOT_FOUND](#)

Description

Returns in `pLib` the handle of the library for the requested kernel `kernel`

See also:

[cuLibraryLoadData](#), [cuLibraryLoadFromFile](#), [cuLibraryUnload](#), [cuLibraryGetKernel](#)

CUresult cuKernelGetName (const char **name, CUkernel hfunc)

Returns the function name for a CUkernel handle.

Parameters

name

- The returned name of the function

hfunc

- The function handle to retrieve the name for

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns in **name the function name associated with the kernel handle hfunc . The function name is returned as a null-terminated string. The returned name is only valid when the kernel handle is valid. If the library is unloaded or reloaded, one must call the API again to get the updated name. This API may return a mangled name if the function is not declared as having C linkage. If either **name or hfunc is NULL, [CUDA_ERROR_INVALID_VALUE](#) is returned.



Note:

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

CUresult cuKernelGetParamInfo (CUkernel kernel, 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

kernel

- The kernel to query

paramIndex

- The parameter index to query

paramOffset

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

paramSize

- Optionally returns 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` into `kernel`'s list of parameters, and returns in `paramOffset` and `paramSize` the offset and size, respectively, where the parameter will reside in the device-side parameter layout. 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 `kernel` takes. `paramSize` can be set to `NULL` if only the parameter offset is desired.



Note:

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

See also:

[`cuFuncGetParamInfo`](#)

CUresult cuKernelSetAttribute (CUfunction_attribute attrib, int val, CUkernel kernel, CUdevice dev)

Sets information about a kernel.

Parameters

attrib

- Attribute requested

val

- Value to set

kernel

- Kernel to set attribute of

dev

- Device to set attribute of

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

This call sets the value of a specified attribute `attrib` on the kernel `kernel` for the requested device `dev` to an integer value specified by `val`. This function returns `CUDA_SUCCESS` 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 (`CUDA_ERROR_INVALID_VALUE`).

Note that attributes set using `cuFuncSetAttribute()` will override the attribute set by this API irrespective of whether the call to `cuFuncSetAttribute()` is made before or after this API call. However, `cuKernelGetAttribute()` will always return the attribute value set by this API.

Supported attributes are:

- ▶ `CU_FUNC_ATTRIBUTE_MAX_DYNAMIC_SHARED_SIZE_BYTES`: This is the maximum size in bytes of dynamically-allocated shared memory. The value should contain the requested maximum size of dynamically-allocated shared memory. The sum of this value and the function attribute `CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES` cannot exceed the device attribute `CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK_OPTIN`. The maximal size of requestable dynamic shared memory may differ by GPU architecture.
- ▶ `CU_FUNC_ATTRIBUTE_PREFERRED_SHARED_MEMORY_CARVEOUT`: 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 `CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_MULTIPROCESSOR`. This is only a hint, and the driver can choose a different ratio if required to execute the function.
- ▶ `CU_FUNC_ATTRIBUTE_REQUIRED_CLUSTER_WIDTH`: 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 `CUDA_ERROR_NOT_PERMITTED`.
- ▶ `CU_FUNC_ATTRIBUTE_REQUIRED_CLUSTER_HEIGHT`: 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 `CUDA_ERROR_NOT_PERMITTED`.
- ▶ `CU_FUNC_ATTRIBUTE_REQUIRED_CLUSTER_DEPTH`: 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 `CUDA_ERROR_NOT_PERMITTED`.
- ▶ `CU_FUNC_ATTRIBUTE_NON_PORTABLE_CLUSTER_SIZE_ALLOWED`: Indicates whether the function can be launched with non-portable cluster size. 1 is allowed, 0 is disallowed.
- ▶ `CU_FUNC_ATTRIBUTE_CLUSTER_SCHEDULING_POLICY_PREFERENCE`: The block scheduling policy of a function. The value type is `CUclusterSchedulingPolicy`.



Note:

The API has stricter locking requirements in comparison to its legacy counterpart [cuFuncSetAttribute\(\)](#) 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:

[cuLibraryLoadData](#), [cuLibraryLoadFromFile](#), [cuLibraryUnload](#), [cuKernelGetAttribute](#), [cuLibraryGetKernel](#), [cuLaunchKernel](#), [cuKernelGetFunction](#), [cuLibraryGetModule](#), [cuModuleGetFunction](#), [cuFuncSetAttribute](#)

CUresult cuKernelSetCacheConfig (CUkernel kernel, CUfunc_cache config, CUdevice dev)

Sets the preferred cache configuration for a device kernel.

Parameters

kernel

- Kernel to configure cache for

config

- Requested cache configuration

dev

- Device to set attribute of

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

On devices where the L1 cache and shared memory use the same hardware resources, this sets through `config` the preferred cache configuration for the device kernel `kernel` on the requested device `dev`. This is only a preference. The driver will use the requested configuration if possible, but it is free to choose a different configuration if required to execute `kernel`. Any context-wide preference set via [cuCtxSetCacheConfig\(\)](#) will be overridden by this per-kernel setting.

Note that attributes set using [cuFuncSetCacheConfig\(\)](#) will override the attribute set by this API irrespective of whether the call to [cuFuncSetCacheConfig\(\)](#) is made before or after this API call.

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:

- ▶ [CU_FUNC_CACHE_PREFER_NONE](#): no preference for shared memory or L1 (default)
- ▶ [CU_FUNC_CACHE_PREFER_SHARED](#): prefer larger shared memory and smaller L1 cache
- ▶ [CU_FUNC_CACHE_PREFER_L1](#): prefer larger L1 cache and smaller shared memory
- ▶ [CU_FUNC_CACHE_PREFER_EQUAL](#): prefer equal sized L1 cache and shared memory

**Note:**

The API has stricter locking requirements in comparison to its legacy counterpart [cuFuncSetCacheConfig\(\)](#) due to device-wide semantics. If multiple threads are trying to set a config on the same device simultaneously, the cache config setting will depend on the interleavings chosen by the OS scheduler and memory consistency.

See also:

[cuLibraryLoadData](#), [cuLibraryLoadFromFile](#), [cuLibraryUnload](#), [cuLibraryGetKernel](#),
[cuKernelGetFunction](#), [cuLibraryGetModule](#), [cuModuleGetFunction](#), [cuFuncSetCacheConfig](#),
[cuCtxSetCacheConfig](#), [cuLaunchKernel](#)

CUresult cuLibraryEnumerateKernels (CUkernel *kernels, unsigned int numKernels, CUlibrary 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_VALUE](#)

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:

[cuLibraryGetKernelCount](#)

CUresult cuLibraryGetGlobal (CUdeviceptr *dptr, size_t *bytes, CUlibrary library, const char *name)

Returns a global device pointer.

Parameters

dptr

- Returned global device pointer for the requested context

bytes

- Returned global size in bytes

library

- Library to retrieve global from

name

- Name of global to retrieve

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_FOUND](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_CONTEXT_IS_DESTROYED](#)

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 context. If no global for the requested name `name` exists, the call returns [CUDA_ERROR_NOT_FOUND](#). One of the parameters `dptr` or `bytes` (not both) can be NULL in which case it is ignored.

See also:

[cuLibraryLoadData](#), [cuLibraryLoadFromFile](#), [cuLibraryUnload](#), [cuLibraryGetModule](#), [cuModuleGetGlobal](#)

CUresult cuLibraryGetKernel (CUkernel *pKernel, CUlibrary 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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_NOT_FOUND](#)

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 [CUDA_ERROR_NOT_FOUND](#).

See also:

[cuLibraryLoadData](#), [cuLibraryLoadFromFile](#), [cuLibraryUnload](#), [cuKernelGetFunction](#),
[cuLibraryGetModule](#), [cuModuleGetFunction](#)

CUresult cuLibraryGetKernelCount (unsigned int *count, CUlibrary lib)

Returns the number of kernels within a library.

Parameters**count**

- Number of kernels found within the library

lib

- Library to query

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

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

CUresult cuLibraryGetManaged (CUdeviceptr *dptr, size_t *bytes, CUlibrary 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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_FOUND](#)

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 [CUDA_ERROR_NOT_FOUND](#). 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 into atleast one context.

See also:

[cuLibraryLoadData](#), [cuLibraryLoadFromFile](#), [cuLibraryUnload](#)

CUresult cuLibraryGetModule (CUmodule *pMod, CUlibrary library)

Returns a module handle.

Parameters

pMod

- Returned module handle

library

- Library to retrieve module from

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_NOT_FOUND](#), [CUDA_ERROR_INVALID_CONTEXT](#),
[CUDA_ERROR_CONTEXT_IS_DESTROYED](#)

Description

Returns in `pMod` the module handle associated with the current context located in library `library`. If module handle is not found, the call returns [CUDA_ERROR_NOT_FOUND](#).

See also:

[cuLibraryLoadData](#), [cuLibraryLoadFromFile](#), [cuLibraryUnload](#), [cuModuleGetFunction](#)

CUresult cuLibraryGetUnifiedFunction (void **fptr, CUlibrary 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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_NOT_FOUND](#)

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 [CUDA_ERROR_NOT_FOUND](#). If there is no device with attribute [CU_DEVICE_ATTRIBUTE_UNIFIED_FUNCTION_POINTERS](#) present in the system, the call may return [CUDA_ERROR_NOT_FOUND](#).

See also:

[cuLibraryLoadData](#), [cuLibraryLoadFromFile](#), [cuLibraryUnload](#)

CUresult cuLibraryLoadData (CUlibrary *library, const void *code, CUjit_option *jitOptions, void **jitOptionsValues, unsigned int numJitOptions, CUlibraryOption *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

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_PTX,
CUDA_ERROR_UNSUPPORTED_PTX_VERSION, CUDA_ERROR_OUT_OF_MEMORY,
CUDA_ERROR_NO_BINARY_FOR_GPU,
CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND,
CUDA_ERROR_SHARED_OBJECT_INIT_FAILED,
CUDA_ERROR_JIT_COMPILER_NOT_FOUND, CUDA_ERROR_NOT_SUPPORTED

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 [cuLibraryUnload\(\)](#).
- ▶ 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.

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`.



Note:

If the library contains managed variables and no device in the system supports managed variables this call is expected to return [CUDA_ERROR_NOT_SUPPORTED](#)

See also:

[cuLibraryLoadFromFile](#), [cuLibraryUnload](#), [cuModuleLoad](#), [cuModuleLoadData](#), [cuModuleLoadDataEx](#)

CUresult cuLibraryLoadFromFile (CUlibrary *library, const char *fileName, CUjit_option *jitOptions, void **jitOptionsValues, unsigned int numJitOptions, CUlibraryOption *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

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_PTX,
CUDA_ERROR_UNSUPPORTED_PTX_VERSION, CUDA_ERROR_OUT_OF_MEMORY,
CUDA_ERROR_NO_BINARY_FOR_GPU,
CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND,
CUDA_ERROR_SHARED_OBJECT_INIT_FAILED,
CUDA_ERROR_JIT_COMPILER_NOT_FOUND, CUDA_ERROR_NOT_SUPPORTED

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 [cuLibraryUnload\(\)](#).
- ▶ 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.

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`.



Note:

If the library contains managed variables and no device in the system supports managed variables this call is expected to return [`CUDA\_ERROR\_NOT\_SUPPORTED`](#)

See also:

[`cuLibraryLoadData`](#), [`cuLibraryUnload`](#), [`cuModuleLoad`](#), [`cuModuleLoadData`](#), [`cuModuleLoadDataEx`](#)

CUresult cuLibraryUnload (CUlibrary library)

Unloads a library.

Parameters

library

- Library to unload

Returns

[`CUDA\_SUCCESS`](#), [`CUDA\_ERROR\_DEINITIALIZED`](#), [`CUDA\_ERROR\_NOT\_INITIALIZED`](#), [`CUDA\_ERROR\_INVALID\_VALUE`](#)

Description

Unloads the library specified with `library`

See also:

[`cuLibraryLoadData`](#), [`cuLibraryLoadFromFile`](#), [`cuModuleUnload`](#)

6.13. Memory Management

This section describes the memory management functions of the low-level CUDA driver application programming interface.

struct CUmemDecompressParams

Structure describing the parameters that compose a single decompression operation.

enum CUmemDecompressAlgorithm

Bitmasks for CU_DEVICE_ATTRIBUTE_MEM_DECOMPRESS_ALGORITHM_MASK.

Values

CU_MEM_DECOMPRESS_UNSUPPORTED = 0

Decompression is unsupported.

CU_MEM_DECOMPRESS_ALGORITHM_DEFLATE = 1<<0

Deflate is supported.

CU_MEM_DECOMPRESS_ALGORITHM_SNAPPY = 1<<1

Snappy is supported.

CUresult cuArray3DCreate (CUarray *pHandle, const CUDA_ARRAY3D_DESCRIPTOR *pAllocateArray)

Creates a 3D CUDA array.

Parameters

pHandle

- Returned array

pAllocateArray

- 3D array descriptor

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_UNKNOWN](#)

Description

Creates a CUDA array according to the CUDA_ARRAY3D_DESCRIPTOR structure pAllocateArray and returns a handle to the new CUDA array in *pHandle. The CUDA_ARRAY3D_DESCRIPTOR is defined as:

```
typedef struct {
    unsigned int Width;
    unsigned int Height;
    unsigned int Depth;
    CUarray_format Format;
    unsigned int NumChannels;
    unsigned int Flags;
} CUDA_ARRAY3D_DESCRIPTOR;
```

where:

- ▶ Width, Height, and Depth are the width, height, and depth of the CUDA array (in elements); the following types of CUDA arrays can be allocated:
 - ▶ A 1D array is allocated if Height and Depth extents are both zero.
 - ▶ A 2D array is allocated if only 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 Height is zero and the [CUDA_ARRAY3D_LAYERED](#) 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 [CUDA_ARRAY3D_LAYERED](#) 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 [CUDA_ARRAY3D_CUBEMAP](#) 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 [CUarray_cubemap_face](#).
 - ▶ A cubemap layered CUDA array is allocated if all three extents are non-zero, and both, [CUDA_ARRAY3D_CUBEMAP](#) and [CUDA_ARRAY3D_LAYERED](#) 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.
- ▶ Format specifies the format of the elements; [CUarray_format](#) is defined as:

```
typedef enum CUarray_format_enum {
    CU_AD_FORMAT_UNSIGNED_INT8 = 0x01,
    CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
    CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
    CU_AD_FORMAT_SIGNED_INT8 = 0x08,
    CU_AD_FORMAT_SIGNED_INT16 = 0x09,
    CU_AD_FORMAT_SIGNED_INT32 = 0x0a,
    CU_AD_FORMAT_HALF = 0x10,
    CU_AD_FORMAT_FLOAT = 0x20,
    CU_AD_FORMAT_NV12 = 0xb0,
    CU_AD_FORMAT_UNORM_INT8X1 = 0xc0,
    CU_AD_FORMAT_UNORM_INT8X2 = 0xc1,
    CU_AD_FORMAT_UNORM_INT8X4 = 0xc2,
    CU_AD_FORMAT_UNORM_INT16X1 = 0xc3,
    CU_AD_FORMAT_UNORM_INT16X2 = 0xc4,
    CU_AD_FORMAT_UNORM_INT16X4 = 0xc5,
    CU_AD_FORMAT_SNORM_INT8X1 = 0xc6,
    CU_AD_FORMAT_SNORM_INT8X2 = 0xc7,
    CU_AD_FORMAT_SNORM_INT8X4 = 0xc8,
    CU_AD_FORMAT_SNORM_INT16X1 = 0xc9,
    CU_AD_FORMAT_SNORM_INT16X2 = 0xca,
    CU_AD_FORMAT_SNORM_INT16X4 = 0xcb,
    CU_AD_FORMAT_BC1_UNORM = 0x91,
    CU_AD_FORMAT_BC1_UNORM_SRGB = 0x92,
    CU_AD_FORMAT_BC2_UNORM = 0x93,
    CU_AD_FORMAT_BC2_UNORM_SRGB = 0x94,
    CU_AD_FORMAT_BC3_UNORM = 0x95,
    CU_AD_FORMAT_BC3_UNORM_SRGB = 0x96,
```

```

CU_AD_FORMAT_BC4_UNORM = 0x97,
CU_AD_FORMAT_BC4_SNORM = 0x98,
CU_AD_FORMAT_BC5_UNORM = 0x99,
CU_AD_FORMAT_BC5_SNORM = 0x9a,
CU_AD_FORMAT_BC6H_UF16 = 0x9b,
CU_AD_FORMAT_BC6H_SF16 = 0x9c,
CU_AD_FORMAT_BC7_UNORM = 0x9d,
CU_AD_FORMAT_BC7_UNORM_SRGB = 0x9e,
CU_AD_FORMAT_P010 = 0x9f,
CU_AD_FORMAT_P016 = 0xa1,
CU_AD_FORMAT_NV16 = 0xa2,
CU_AD_FORMAT_P210 = 0xa3,
CU_AD_FORMAT_P216 = 0xa4,
CU_AD_FORMAT_YUY2 = 0xa5,
CU_AD_FORMAT_Y210 = 0xa6,
CU_AD_FORMAT_Y216 = 0xa7,
CU_AD_FORMAT_AYUV = 0xa8,
CU_AD_FORMAT_Y410 = 0xa9,
CU_AD_FORMAT_Y416 = 0xb1,
CU_AD_FORMAT_Y444_PLANAR8 = 0xb2,
CU_AD_FORMAT_Y444_PLANAR10 = 0xb3,
CU_AD_FORMAT_YUV444_8bit_SemiPlanar = 0xb4,
CU_AD_FORMAT_YUV444_16bit_SemiPlanar = 0xb5,
CU_AD_FORMAT_UNORM_INT_101010_2 = 0x50,
} CUarray_format;

```

- ▶ NumChannels specifies the number of packed components per CUDA array element; it may be 1, 2, or 4;
- ▶ Flags may be set to
 - ▶ [CUDA_ARRAY3D_LAYERED](#) to enable creation of layered CUDA arrays. If this flag is set, Depth specifies the number of layers, not the depth of a 3D array.
 - ▶ [CUDA_ARRAY3D_SURFACE_LDST](#) to enable surface references to be bound to the CUDA array. If this flag is not set, [cuSurfRefSetArray](#) will fail when attempting to bind the CUDA array to a surface reference.
 - ▶ [CUDA_ARRAY3D_CUBEMAP](#) to enable creation of cubemaps. If this flag is set, Width must be equal to Height, and Depth must be six. If the [CUDA_ARRAY3D_LAYERED](#) flag is also set, then Depth must be a multiple of six.
 - ▶ [CUDA_ARRAY3D_TEXTURE_GATHER](#) to indicate that the CUDA array will be used for texture gather. Texture gather can only be performed on 2D CUDA arrays.

Width, Height and Depth must meet certain size requirements as listed in the following table. All values are specified in elements. Note that for brevity's sake, the full name of the device attribute is not specified. For ex., TEXTURE1D_WIDTH refers to the device attribute [CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH](#).

Note that 2D CUDA arrays have different size requirements if the [CUDA_ARRAY3D_TEXTURE_GATHER](#) flag is set. Width and Height must not be greater than [CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_GATHER_WIDTH](#) and [CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_GATHER_HEIGHT](#) respectively, in that case.

CUDA array type	Valid extents that must always be met {(width range in elements), (height range), (depth range)}	Valid extents with CUDA_ARRAY3D_SURFACE_LDST set {(width range in elements), (height range), (depth range)}
1D	{ (1,TEXTURE1D_WIDTH), 0, 0 }	{ (1,SURFACE1D_WIDTH), 0, 0 }
2D	{ (1,TEXTURE2D_WIDTH), (1,TEXTURE2D_HEIGHT), 0 }	{ (1,SURFACE2D_WIDTH), (1,SURFACE2D_HEIGHT), 0 }
3D	{ (1,TEXTURE3D_WIDTH), (1,TEXTURE3D_HEIGHT), (1,TEXTURE3D_DEPTH) } OR { (1,TEXTURE3D_WIDTH_ALTERNATE), (1,TEXTURE3D_HEIGHT_ALTERNATE), (1,TEXTURE3D_DEPTH_ALTERNATE) }	{ (1,SURFACE3D_WIDTH), (1,SURFACE3D_HEIGHT), (1,SURFACE3D_DEPTH) }
1D Layered	{ (1,TEXTURE1D_LAYERED_WIDTH), 0, (1,TEXTURE1D_LAYERED_LAYERS) }	{ (1,SURFACE1D_LAYERED_WIDTH), 0, (1,SURFACE1D_LAYERED_LAYERS) }
2D Layered	{ (1,TEXTURE2D_LAYERED_WIDTH), (1,TEXTURE2D_LAYERED_HEIGHT), (1,TEXTURE2D_LAYERED_LAYERS) }	{ (1,SURFACE2D_LAYERED_WIDTH), (1,SURFACE2D_LAYERED_HEIGHT), (1,SURFACE2D_LAYERED_LAYERS) }
Cubemap	{ (1,TEXTURECUBEMAP_WIDTH), (1,TEXTURECUBEMAP_WIDTH), 6 }	{ (1,SURFACECUBEMAP_WIDTH), (1,SURFACECUBEMAP_WIDTH), 6 }
Cubemap Layered	{ (1,TEXTURECUBEMAP_LAYERED_WIDTH), (1,TEXTURECUBEMAP_LAYERED_WIDTH), (1,TEXTURECUBEMAP_LAYERED_LAYERS) }	{ (1,SURFACECUBEMAP_LAYERED_WIDTH), (1,SURFACECUBEMAP_LAYERED_WIDTH), (1,SURFACECUBEMAP_LAYERED_LAYERS) }

Here are examples of CUDA array descriptions:

Description for a CUDA array of 2048 floats:

```
↑ CUDA_ARRAY3D_DESCRIPTOR desc;
   desc.Format = CU_AD_FORMAT_FLOAT;
   desc.NumChannels = 1;
   desc.Width = 2048;
   desc.Height = 0;
   desc.Depth = 0;
```

Description for a 64 x 64 CUDA array of floats:

```
↑ CUDA_ARRAY3D_DESCRIPTOR desc;
   desc.Format = CU_AD_FORMAT_FLOAT;
   desc.NumChannels = 1;
   desc.Width = 64;
   desc.Height = 64;
   desc.Depth = 0;
```

Description for a width x height x depth CUDA array of 64-bit, 4x16-bit float16's:

```
↑ CUDA_ARRAY3D_DESCRIPTOR desc;
    desc.Format = CU_AD_FORMAT_HALF;
    desc.NumChannels = 4;
    desc.Width = width;
    desc.Height = height;
    desc.Depth = depth;
```



Note:

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

See also:

[cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaMalloc3DArray](#)

CUresult cuArray3DGetDescriptor (CUDA_ARRAY3D_DESCRIPTOR *pArrayDescriptor, CUarray hArray)

Get a 3D CUDA array descriptor.

Parameters

pArrayDescriptor

- Returned 3D array descriptor

hArray

- 3D array to get descriptor of

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_CONTEXT_IS_DESTROYED](#)

Description

Returns in *pArrayDescriptor a descriptor containing information on the format and dimensions of the CUDA array hArray. It is useful for subroutines that have been passed a CUDA array, but need to know the CUDA array parameters for validation or other purposes.

This function may be called on 1D and 2D arrays, in which case the `Height` and/or `Depth` members of the descriptor struct will be set to 0.



Note:

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

See also:

[cuArray3DCreate](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaArrayGetInfo](#)

CUresult cuArrayCreate (CUarray *pHandle, const CUDA_ARRAY_DESCRIPTOR *pAllocateArray)

Creates a 1D or 2D CUDA array.

Parameters

pHandle

- Returned array

pAllocateArray

- Array descriptor

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_UNKNOWN](#)

Description

Creates a CUDA array according to the `CUDA_ARRAY_DESCRIPTOR` structure `pAllocateArray` and returns a handle to the new CUDA array in `*pHandle`. The `CUDA_ARRAY_DESCRIPTOR` is defined as:

```
↑ typedef struct {
    unsigned int Width;
    unsigned int Height;
    CUarray_format Format;
    unsigned int NumChannels;
} CUDA_ARRAY_DESCRIPTOR;
```

where:

- Width, and Height are the width, and height of the CUDA array (in elements); the CUDA array is one-dimensional if height is 0, two-dimensional otherwise;
- Format specifies the format of the elements; `CUarray_format` is defined as:

```
typedef enum CUarray_format_enum {
    CU_AD_FORMAT_UNSIGNED_INT8 = 0x01,
    CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
    CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
    CU_AD_FORMAT_SIGNED_INT8 = 0x08,
    CU_AD_FORMAT_SIGNED_INT16 = 0x09,
    CU_AD_FORMAT_SIGNED_INT32 = 0x0a,
    CU_AD_FORMAT_HALF = 0x10,
    CU_AD_FORMAT_FLOAT = 0x20,
    CU_AD_FORMAT_NV12 = 0xb0,
    CU_AD_FORMAT_UNORM_INT8X1 = 0xc0,
    CU_AD_FORMAT_UNORM_INT8X2 = 0xc1,
    CU_AD_FORMAT_UNORM_INT8X4 = 0xc2,
    CU_AD_FORMAT_UNORM_INT16X1 = 0xc3,
    CU_AD_FORMAT_UNORM_INT16X2 = 0xc4,
    CU_AD_FORMAT_UNORM_INT16X4 = 0xc5,
    CU_AD_FORMAT_SNORM_INT8X1 = 0xc6,
    CU_AD_FORMAT_SNORM_INT8X2 = 0xc7,
    CU_AD_FORMAT_SNORM_INT8X4 = 0xc8,
    CU_AD_FORMAT_SNORM_INT16X1 = 0xc9,
    CU_AD_FORMAT_SNORM_INT16X2 = 0xca,
    CU_AD_FORMAT_SNORM_INT16X4 = 0xcb,
    CU_AD_FORMAT_BC1_UNORM = 0x91,
    CU_AD_FORMAT_BC1_UNORM_SRGB = 0x92,
    CU_AD_FORMAT_BC2_UNORM = 0x93,
    CU_AD_FORMAT_BC2_UNORM_SRGB = 0x94,
    CU_AD_FORMAT_BC3_UNORM = 0x95,
    CU_AD_FORMAT_BC3_UNORM_SRGB = 0x96,
    CU_AD_FORMAT_BC4_UNORM = 0x97,
    CU_AD_FORMAT_BC4_SNORM = 0x98,
    CU_AD_FORMAT_BC5_UNORM = 0x99,
    CU_AD_FORMAT_BC5_SNORM = 0x9a,
    CU_AD_FORMAT_BC6H_UF16 = 0x9b,
    CU_AD_FORMAT_BC6H_SF16 = 0x9c,
    CU_AD_FORMAT_BC7_UNORM = 0x9d,
    CU_AD_FORMAT_BC7_UNORM_SRGB = 0x9e,
    CU_AD_FORMAT_P010 = 0x9f,
    CU_AD_FORMAT_P016 = 0xa1,
    CU_AD_FORMAT_NV16 = 0xa2,
    CU_AD_FORMAT_P210 = 0xa3,
    CU_AD_FORMAT_P216 = 0xa4,
    CU_AD_FORMAT_YUY2 = 0xa5,
    CU_AD_FORMAT_Y210 = 0xa6,
    CU_AD_FORMAT_Y216 = 0xa7,
    CU_AD_FORMAT_AYUV = 0xa8,
    CU_AD_FORMAT_Y410 = 0xa9,
    CU_AD_FORMAT_Y416 = 0xb1,
    CU_AD_FORMAT_Y444_PLANAR8 = 0xb2,
    CU_AD_FORMAT_Y444_PLANAR10 = 0xb3,
    CU_AD_FORMAT_YUV444_8bit_SemiPlanar = 0xb4,
    CU_AD_FORMAT_YUV444_16bit_SemiPlanar = 0xb5,
    CU_AD_FORMAT_UNORM_INT_101010_2 = 0x50,
} CUarray_format;
```

- NumChannels specifies the number of packed components per CUDA array element; it may be 1, 2, or 4;

Here are examples of CUDA array descriptions:

Description for a CUDA array of 2048 floats:

```
↑ CUDA_ARRAY_DESCRIPTOR desc;
    desc.Format = CU_AD_FORMAT_FLOAT;
    desc.NumChannels = 1;
    desc.Width = 2048;
    desc.Height = 1;
```

Description for a 64 x 64 CUDA array of floats:

```
↑ CUDA_ARRAY_DESCRIPTOR desc;
    desc.Format = CU_AD_FORMAT_FLOAT;
    desc.NumChannels = 1;
    desc.Width = 64;
    desc.Height = 64;
```

Description for a width x height CUDA array of 64-bit, 4x16-bit float16's:

```
↑ CUDA_ARRAY_DESCRIPTOR desc;
    desc.Format = CU_AD_FORMAT_HALF;
    desc.NumChannels = 4;
    desc.Width = width;
    desc.Height = height;
```

Description for a width x height CUDA array of 16-bit elements, each of which is two 8-bit unsigned chars:

```
↑ CUDA_ARRAY_DESCRIPTOR arrayDesc;
    desc.Format = CU_AD_FORMAT_UNSIGNED_INT8;
    desc.NumChannels = 2;
    desc.Width = width;
    desc.Height = height;
```



Note:

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

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaMallocArray](#)

CUresult cuArrayDestroy (CUarray hArray)

Destroys a CUDA array.

Parameters

hArray

- Array to destroy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_ARRAY_IS_MAPPED](#), [CUDA_ERROR_CONTEXT_IS_DESTROYED](#)

Description

Destroys the CUDA array `hArray`.



Note:

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

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayGetDescriptor](#),
[cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#),
[cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#),
[cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#),
[cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#),
[cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#),
[cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#),
[cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaFreeArray](#)

CUresult cuArrayGetDescriptor (CUDA_ARRAY_DESCRIPTOR *pArrayDescriptor, CUarray hArray)

Get a 1D or 2D CUDA array descriptor.

Parameters

pArrayDescriptor

- Returned array descriptor

hArray

- Array to get descriptor of

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#)

Description

Returns in `*pArrayDescriptor` a descriptor containing information on the format and dimensions of the CUDA array `hArray`. It is useful for subroutines that have been passed a CUDA array, but need to know the CUDA array parameters for validation or other purposes.



Note:

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

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaArrayGetInfo](#)

CUresult cuArrayGetMemoryRequirements (CUDA_ARRAY_MEMORY_REQUIREMENTS *memoryRequirements, CUarray array, CUdevice device)

Returns the memory requirements of a CUDA array.

Parameters

memoryRequirements

- Pointer to `CUDA_ARRAY_MEMORY_REQUIREMENTS`

array

- CUDA array to get the memory requirements of

device

- Device to get the memory requirements for

Returns

[CUDA_SUCCESS](#) [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns the memory requirements of a CUDA array in `memoryRequirements`. If the CUDA array is not allocated with flag `CUDA_ARRAY3D_DEFERRED_MAPPING` `CUDA_ERROR_INVALID_VALUE` will be returned.

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

See also:

[cuMipmappedArrayGetMemoryRequirements](#), [cuMemMapArrayAsync](#)

CUresult cuArrayGetPlane (CUarray *pPlaneArray, CUarray hArray, unsigned int planeIdx)

Gets a CUDA array plane from a CUDA array.

Parameters

pPlaneArray

- Returned CUDA array referenced by the `planeIdx`

hArray

- Multiplanar CUDA array

planeIdx

- Plane index

Returns

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_INVALID_HANDLE`

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: `CU_AD_FORMAT_NV12`, then `CUDA_ERROR_INVALID_VALUE` is returned.

Note that if the `hArray` has format `CU_AD_FORMAT_NV12`, then passing in 0 for `planeIdx` returns a CUDA array of the same size as `hArray` but with one channel and `CU_AD_FORMAT_UNSIGNED_INT8` as its format. If 1 is passed for `planeIdx`, then

the returned CUDA array has half the height and width of `hArray` with two channels and `CU_AD_FORMAT_UNSIGNED_INT8` as its format.



Note:

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

See also:

[cuArrayCreate](#), [cuArrayGetPlane](#)

CUresult cuArrayGetSparseProperties (CUDA_ARRAY_SPARSE_PROPERTIES *sparseProperties, CUarray array)

Returns the layout properties of a sparse CUDA array.

Parameters

sparseProperties

- Pointer to `CUDA_ARRAY_SPARSE_PROPERTIES`

array

- CUDA array to get the sparse properties of

Returns

[CUDA_SUCCESS](#) [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns the layout properties of a sparse CUDA array in `sparseProperties`. If the CUDA array is not allocated with flag `CUDA_ARRAY3D_SPARSE`, `CUDA_ERROR_INVALID_VALUE` will be returned.

If the returned value in `CUDA_ARRAY_SPARSE_PROPERTIES::flags` contains `CUDA_ARRAY_SPARSE_PROPERTIES_SINGLE_MIPTAIL`, then `CUDA_ARRAY_SPARSE_PROPERTIES::mipTailSize` represents the total size of the array. Otherwise, it will be zero. Also, the returned value in `CUDA_ARRAY_SPARSE_PROPERTIES::mipTailFirstLevel` is always zero. Note that the `array` must have been allocated using [cuArrayCreate](#) or [cuArray3DCreate](#). For CUDA arrays obtained using [cuMipmappedArrayGetLevel](#), `CUDA_ERROR_INVALID_VALUE` will be returned. Instead, [cuMipmappedArrayGetSparseProperties](#) must be used to obtain the sparse properties of the entire CUDA mipmapped array to which `array` belongs to.

See also:

[cuMipmappedArrayGetSparseProperties](#), [cuMemMapArrayAsync](#)

CUresult cuDeviceGetByPCIBusId (CUdevice *dev, const char *pciBusId)

Returns a handle to a compute device.

Parameters

dev

- Returned device handle

pciBusId

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

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

Returns in *device a device handle given a PCI bus ID string.



Note:

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

See also:

[cuDeviceGet](#), [cuDeviceGetAttribute](#), [cuDeviceGetPCIBusId](#), [cudaDeviceGetByPCIBusId](#)

CUresult cuDeviceGetPCIBusId (char *pciBusId, int len, CUdevice dev)

Returns a PCI Bus Id string for the device.

Parameters

pciBusId

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

len

- Maximum length of string to store in name

dev

- Device to get identifier string for

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

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

**Note:**

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

See also:

[cuDeviceGet](#), [cuDeviceGetAttribute](#), [cuDeviceGetByPCIBusId](#), [cudaDeviceGetPCIBusId](#)

CUresult cuDeviceRegisterAsyncNotification (CUdevice device, CUasyncCallback callbackFunc, void *userData, CUasyncCallbackHandle *callback)

Registers a callback function to receive async notifications.

Parameters**device**

- The device on which to register the callback

callbackFunc

- The function to register as a callback

userData

- A generic pointer to user data. This is passed into the callback function.

callback

- A handle representing the registered callback instance

Returns

[CUDA_SUCCESS](#) [CUDA_ERROR_NOT_SUPPORTED](#) [CUDA_ERROR_INVALID_DEVICE](#)
[CUDA_ERROR_INVALID_VALUE](#) [CUDA_ERROR_NOT_PERMITTED](#)
[CUDA_ERROR_UNKNOWN](#)

Description

Registers `callbackFunc` to receive async notifications.

The `userData` parameter is passed to the callback function at async notification time. Likewise, `callback` is also passed to the callback function to distinguish between multiple registered callbacks.

The callback function being registered should be designed to return quickly (~10ms). Any long running tasks should be queued for execution on an application thread.

Callbacks may not call `cuDeviceRegisterAsyncNotification` or `cuDeviceUnregisterAsyncNotification`. Doing so will result in [CUDA_ERROR_NOT_PERMITTED](#). Async notification callbacks execute in an undefined order and may be serialized.

Returns in `*callback` a handle representing the registered callback instance.



Note:

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

See also:

[cuDeviceUnregisterAsyncNotification](#)

CUresult cuDeviceUnregisterAsyncNotification (CUdevice device, CUasyncCallbackHandle callback)

Unregisters an async notification callback.

Parameters

device

- The device from which to remove `callback`.

callback

- The callback instance to unregister from receiving async notifications.

Returns

[CUDA_SUCCESS](#) [CUDA_ERROR_NOT_SUPPORTED](#) [CUDA_ERROR_INVALID_DEVICE](#)
[CUDA_ERROR_INVALID_VALUE](#) [CUDA_ERROR_NOT_PERMITTED](#)
[CUDA_ERROR_UNKNOWN](#)

Description

Unregisters `callback` so that the corresponding callback function will stop receiving async notifications.

**Note:**

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

See also:

[cuDeviceRegisterAsyncNotification](#)

CUresult cuIpcCloseMemHandle (CUdeviceptr dptr)

Attempts to close memory mapped with [cuIpcOpenMemHandle](#).

Parameters

dptr

- Device pointer returned by [cuIpcOpenMemHandle](#)

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_MAP_FAILED](#),
[CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

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

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

IPC functionality is restricted to devices with support for unified addressing on Linux and Windows operating systems. IPC functionality on Windows is supported for compatibility purposes but not recommended as it comes with performance cost. Users can test their device for IPC functionality by calling [cuapiDeviceGetAttribute](#) with [CU_DEVICE_ATTRIBUTE_IPC_EVENT_SUPPORTED](#)

See also:

[cuMemAlloc](#), [cuMemFree](#), [cuIpcGetEventHandle](#), [cuIpcOpenEventHandle](#), [cuIpcGetMemHandle](#),
[cuIpcOpenMemHandle](#), [cuIpcCloseMemHandle](#)

CUresult cuIpcGetEventHandle (CUipcEventHandle *pHandle, CUevent event)

Gets an interprocess handle for a previously allocated event.

Parameters

pHandle

- Pointer to a user allocated CUipcEventHandle in which to return the opaque event handle

event

- Event allocated with [CU_EVENT_INTERPROCESS](#) and [CU_EVENT_DISABLE_TIMING](#) flags.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_MAP_FAILED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Takes as input a previously allocated event. This event must have been created with the [CU_EVENT_INTERPROCESS](#) and [CU_EVENT_DISABLE_TIMING](#) flags set. This opaque handle may be copied into other processes and opened with [cuIpcOpenEventHandle](#) to allow efficient hardware synchronization between GPU work in different processes.

After the event has been opened in the importing process, [cuEventRecord](#), [cuEventSynchronize](#), [cuStreamWaitEvent](#) and [cuEventQuery](#) may be used in either process. Performing operations on the imported event after the exported event has been freed with [cuEventDestroy](#) will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux and Windows operating systems. IPC functionality on Windows is supported for compatibility purposes but not recommended as it comes with performance cost. Users can test their device for IPC functionality by calling [cuapiDeviceGetAttribute](#) with [CU_DEVICE_ATTRIBUTE_IPC_EVENT_SUPPORTED](#)

See also:

[cuEventCreate](#), [cuEventDestroy](#), [cuEventSynchronize](#), [cuEventQuery](#), [cuStreamWaitEvent](#), [cuIpcOpenEventHandle](#), [cuIpcGetMemHandle](#), [cuIpcOpenMemHandle](#), [cuIpcCloseMemHandle](#), [cudaIpcGetEventHandle](#)

CUresult cuIpcGetMemHandle (CUipcMemHandle *pHandle, CUdeviceptr dptr)

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

Parameters

pHandle

- Pointer to user allocated CUipcMemHandle to return the handle in.

dptr

- Base pointer to previously allocated device memory

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_MAP_FAILED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

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

If a region of memory is freed with [cuMemFree](#) and a subsequent call to [cuMemAlloc](#) returns memory with the same device address, [cuIpcGetMemHandle](#) will return a unique handle for the new memory.

IPC functionality is restricted to devices with support for unified addressing on Linux and Windows operating systems. IPC functionality on Windows is supported for compatibility purposes but not recommended as it comes with performance cost. Users can test their device for IPC functionality by calling [cuapiDeviceGetAttribute](#) with [CU_DEVICE_ATTRIBUTE_IPC_EVENT_SUPPORTED](#)

See also:

[cuMemAlloc](#), [cuMemFree](#), [cuIpcGetEventHandle](#), [cuIpcOpenEventHandle](#), [cuIpcOpenMemHandle](#), [cuIpcCloseMemHandle](#), [cudaIpcGetMemHandle](#)

CUresult cuIpcOpenEventHandle (CUevent *phEvent, CUipcEventHandle handle)

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

Parameters

phEvent

- Returns the imported event

handle

- Interprocess handle to open

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_MAP_FAILED](#),
[CUDA_ERROR_PEER_ACCESS_UNSUPPORTED](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_INVALID_VALUE](#)

Description

Opens an interprocess event handle exported from another process with [cuIpcGetEventHandle](#). This function returns a [CUevent](#) that behaves like a locally created event with the [CU_EVENT_DISABLE_TIMING](#) flag specified. This event must be freed with [cuEventDestroy](#).

Performing operations on the imported event after the exported event has been freed with [cuEventDestroy](#) will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux and Windows operating systems. IPC functionality on Windows is supported for compatibility purposes but not recommended as it comes with performance cost. Users can test their device for IPC functionality by calling [cuapiDeviceGetAttribute](#) with [CU_DEVICE_ATTRIBUTE_IPC_EVENT_SUPPORTED](#)

See also:

[cuEventCreate](#), [cuEventDestroy](#), [cuEventSynchronize](#), [cuEventQuery](#), [cuStreamWaitEvent](#),
[cuIpcGetEventHandle](#), [cuIpcGetMemHandle](#), [cuIpcOpenMemHandle](#), [cuIpcCloseMemHandle](#),
[cudaIpcOpenEventHandle](#)

CUresult cuIpcOpenMemHandle (CUdeviceptr *pdptr, CUipcMemHandle handle, unsigned int Flags)

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

Parameters**pdptr**

- Returned device pointer

handle

- CUipcMemHandle to open

Flags

- Flags for this operation. Must be specified as [CU_IPC_MEM_LAZY_ENABLE_PEER_ACCESS](#)

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_MAP_FAILED](#),
[CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_TOO_MANY_PEERS](#),
[CUDA_ERROR_INVALID_VALUE](#)

Description

Maps memory exported from another process with [cuIpcGetMemHandle](#) into the current device address space. For contexts on different devices [cuIpcOpenMemHandle](#) can attempt to enable peer access between the devices as if the user called [cuCtxEnablePeerAccess](#). This behavior is controlled by the [CU_IPC_MEM_LAZY_ENABLE_PEER_ACCESS](#) flag. [cuDeviceCanAccessPeer](#) can determine if a mapping is possible.

Contexts that may open CUipcMemHandles are restricted in the following way. CUipcMemHandles from each [CUdevice](#) in a given process may only be opened by one [CUcontext](#) per [CUdevice](#) per other process.

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

Memory returned from [cuIpcOpenMemHandle](#) must be freed with [cuIpcCloseMemHandle](#).

Calling [cuMemFree](#) on an exported memory region before calling [cuIpcCloseMemHandle](#) in the importing context will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux and Windows operating systems. IPC functionality on Windows is supported for compatibility purposes but not recommended as it comes with performance cost. Users can test their device for IPC functionality by calling [cuapiDeviceGetAttribute](#) with [CU_DEVICE_ATTRIBUTE_IPC_EVENT_SUPPORTED](#)



Note:

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

See also:

[cuMemAlloc](#), [cuMemFree](#), [cuIpcGetEventHandle](#), [cuIpcOpenEventHandle](#), [cuIpcGetMemHandle](#),
[cuIpcCloseMemHandle](#), [cuCtxEnablePeerAccess](#), [cuDeviceCanAccessPeer](#), [cudaIpcOpenMemHandle](#)

CUresult cuMemAlloc (CUdeviceptr *dptr, size_t bytesize)

Allocates device memory.

Parameters

dptr

- Returned device pointer

bytesize

- Requested allocation size in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Allocates `bytesize` bytes of linear memory on the device and returns in `*dptr` a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. If `bytesize` is 0, [cuMemAlloc\(\)](#) returns [CUDA_ERROR_INVALID_VALUE](#).



Note:

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

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaMalloc](#)

CUresult cuMemAllocHost (void **pp, size_t bytesize)

Allocates page-locked host memory.

Parameters

pp

- Returned pointer to host memory

bytesize

- Requested allocation size in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Allocates `bytesize` 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 [cuMemcpy\(\)](#). 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

[CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES](#) is true, [cuMemAllocHost](#) may not page-lock the allocated memory.

Page-locking excessive amounts of memory with [cuMemAllocHost\(\)](#) 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 all host memory allocated using [cuMemAllocHost\(\)](#) will automatically be immediately accessible to all contexts on all devices which support unified addressing (as may be queried using [CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING](#)). The device pointer that may be used to access this host memory from those contexts is always equal to the returned host pointer `*pp`. See [Unified Addressing](#) for additional details.

**Note:**

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

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#),
[cuMemAlloc](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#),
[cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#),
[cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#),
[cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#),
[cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#),
[cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#),
[cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaMallocHost](#)

CUresult cuMemAllocManaged (CUdeviceptr *dptr, size_t bytesize, unsigned int flags)

Allocates memory that will be automatically managed by the Unified Memory system.

Parameters

dptr

- Returned device pointer

bytesize

- Requested allocation size in bytes

flags

- Must be one of [CU_MEM_ATTACH_GLOBAL](#) or [CU_MEM_ATTACH_HOST](#)

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_NOT_SUPPORTED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Allocates `bytesize` bytes of managed memory on the device and returns in `*dptr` a pointer to the allocated memory. If the device doesn't support allocating managed memory, [CUDA_ERROR_NOT_SUPPORTED](#) is returned. Support for managed memory can be queried using the device attribute [CU_DEVICE_ATTRIBUTE_MANAGED_MEMORY](#). The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. If `bytesize` is 0, [cuMemAllocManaged](#) returns [CUDA_ERROR_INVALID_VALUE](#). 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 [CU_MEM_ATTACH_GLOBAL](#) or [CU_MEM_ATTACH_HOST](#). If [CU_MEM_ATTACH_GLOBAL](#) is specified, then this memory is accessible from any stream on any device. If [CU_MEM_ATTACH_HOST](#) is specified, then the allocation should not be accessed from devices that have a zero value for the device attribute [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#); an explicit call to [cuStreamAttachMemAsync](#) will be required to enable access on such devices.

If the association is later changed via [cuStreamAttachMemAsync](#) to a single stream, the default association as specified during [cuMemAllocManaged](#) is restored when that stream is destroyed. For `__managed__` variables, the default association is always [CU_MEM_ATTACH_GLOBAL](#). 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 [cuMemAllocManaged](#) should be released with [cuMemFree](#).

Device memory oversubscription is possible for GPUs that have a non-zero value for the device attribute [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#). 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 [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#), 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 [cuMemAdvise](#). The application can also explicitly migrate memory to a desired processor's memory via [cuMemPrefetchAsync](#).

In a multi-GPU system where all of the GPUs have a zero value for the device attribute [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#) 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 [cuMemAllocManaged](#) 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 [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#) 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 [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#). 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 contexts created in that process on devices that support managed memory have to be peer-to-peer compatible with each other. Context creation will fail if a context is created on a device that supports managed memory and is not peer-to-peer compatible with any of the other managed memory supporting devices on which contexts were previously created, even if those contexts have

been destroyed. 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 error codes from previous, asynchronous launches.

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cuDeviceGetAttribute](#), [cuStreamAttachMemAsync](#), [cudaMallocManaged](#)

CUresult cuMemAllocPitch (CUdeviceptr *dptr, size_t *pPitch, size_t WidthInBytes, size_t Height, unsigned int ElementSizeBytes)

Allocates pitched device memory.

Parameters

dptr

- Returned device pointer

pPitch

- Returned pitch of allocation in bytes

WidthInBytes

- Requested allocation width in bytes

Height

- Requested allocation height in rows

ElementSizeBytes

- Size of largest reads/writes for range

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Allocates at least `WidthInBytes * Height` bytes of linear memory on the device and returns in `*dptr` 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. `ElementSizeBytes` specifies the size of the largest reads and writes that will be performed on the memory range. `ElementSizeBytes` may be 4, 8 or 16 (since coalesced memory transactions are not possible on other data sizes). If `ElementSizeBytes` is smaller than the actual read/write size of a kernel, the kernel will run correctly, but possibly at reduced speed. The pitch returned in `*pPitch` by [cuMemAllocPitch\(\)](#) 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;
```

The pitch returned by [cuMemAllocPitch\(\)](#) is guaranteed to work with [cuMemcpy2D\(\)](#) under all circumstances. For allocations of 2D arrays, it is recommended that programmers consider performing pitch allocations using [cuMemAllocPitch\(\)](#). Due to 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).

The byte alignment of the pitch returned by [cuMemAllocPitch\(\)](#) is guaranteed to match or exceed the alignment requirement for texture binding with [cuTexRefSetAddress2D\(\)](#).



Note:

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

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaMallocPitch](#)

CUresult cuMemBatchDecompressAsync (CUmemDecompressParams *paramsArray, size_t count, unsigned int flags, size_t *errorIndex, CUstream stream)

Submit a batch of `count` independent decompression operations.

Parameters

paramsArray

The array of structures describing the independent decompression operations.

count

The number of entries in `paramsArray` array.

flags

Must be 0.

errorIndex

The index into `paramsArray` of the decompression operation for which the error returned by this function pertains to. If `index` is `SIZE_MAX` and the value returned is not [CUDA_SUCCESS](#), then the error returned by this function should be considered a general error that does not pertain to a particular decompression operation. May be `NULL`, in which case, no index will be recorded in the event of error.

stream

The stream where the work will be enqueued.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#)

Description

Each of the `count` decompression operations is described by a single entry in the `paramsArray` array. Once the batch has been submitted, the function will return, and decompression will happen asynchronously w.r.t. the CPU. To the work completion tracking mechanisms in the CUDA driver, the batch will be considered a single unit of work and processed according to stream semantics, i.e., it is not possible to query the completion of individual decompression operations within a batch.

The memory pointed to by each of [CUmemDecompressParams.src](#), [CUmemDecompressParams.dst](#), and [CUmemDecompressParams.dstActBytes](#), must be capable of usage with the hardware decompress feature. That is, for each of said pointers, the pointer attribute `CU_POINTER_ATTRIBUTE_IS_MEM_DECOMPRESS_CAPABLE` should give a non-zero value. To ensure this, the memory backing the pointers should have been allocated using one of the following CUDA memory allocators: `* cuMemAlloc\(\) * cuMemCreate\(\)` with the usage flag [CU_MEM_CREATE_USAGE_HW_DECOMPRESS](#) `* cuMemAllocFromPoolAsync\(\)` from a pool that was created with the usage flag [CU_MEM_POOL_CREATE_USAGE_HW_DECOMPRESS](#)

Additionally, [CUMemDecompressParams.src](#), [CUMemDecompressParams.dst](#), and [CUMemDecompressParams.dstActBytes](#), must all be accessible from the device associated with the context where `stream` was created. For information on how to ensure this, see the documentation for the allocator of interest.



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.

See also:

[cuMemAlloc](#), [cuMemPoolCreate](#), [cuMemAllocFromPoolAsync](#)

CUresult cuMemcpy (CUdeviceptr dst, CUdeviceptr src, size_t ByteCount)

Copies memory.

Parameters

dst

- Destination unified virtual address space pointer

src

- Source unified virtual address space pointer

ByteCount

- Size of memory copy in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Copies data between two pointers. `dst` and `src` are base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy. Note that this function infers the type of the transfer (host to host, host to device, device to device, or device to host) from the pointer values. This function is only allowed in contexts which support unified addressing.



Note:

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

- ▶ 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:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaMemcpy](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#)

CUresult cuMemcpy2D (const CUDA_MEMCPY2D *pCopy)

Copies memory for 2D arrays.

Parameters

pCopy

- Parameters for the memory copy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Perform a 2D memory copy according to the parameters specified in pCopy. The `CUDA_MEMCPY2D` structure is defined as:

```
typedef struct CUDA_MEMCPY2D_st {
    unsigned int srcXInBytes, srcY;
    CUmemorytype srcMemoryType;
    const void *srcHost;
    CUdeviceptr srcDevice;
    CUarray srcArray;
    unsigned int srcPitch;

    unsigned int dstXInBytes, dstY;
    CUmemorytype dstMemoryType;
    void *dstHost;
    CUdeviceptr dstDevice;
    CUarray dstArray;
    unsigned int dstPitch;
}
```

```
    unsigned int WidthInBytes;
    unsigned int Height;
} CUDA_MEMCPY2D;
```

where:

- `srcMemoryType` and `dstMemoryType` specify the type of memory of the source and destination, respectively; `CUmemorytype_enum` is defined as:

```
↑ typedef enum CUmemorytype_enum {
    CU_MEMORYTYPE_HOST = 0x01,
    CU_MEMORYTYPE_DEVICE = 0x02,
    CU_MEMORYTYPE_ARRAY = 0x03,
    CU_MEMORYTYPE_UNIFIED = 0x04
} CUmemorytype;
```

If `srcMemoryType` is `CU_MEMORYTYPE_UNIFIED`, `srcDevice` and `srcPitch` specify the (unified virtual address space) base address of the source data and the bytes per row to apply. `srcArray` is ignored. This value may be used only if unified addressing is supported in the calling context.

If `srcMemoryType` is `CU_MEMORYTYPE_HOST`, `srcHost` and `srcPitch` specify the (host) base address of the source data and the bytes per row to apply. `srcArray` is ignored.

If `srcMemoryType` is `CU_MEMORYTYPE_DEVICE`, `srcDevice` and `srcPitch` specify the (device) base address of the source data and the bytes per row to apply. `srcArray` is ignored.

If `srcMemoryType` is `CU_MEMORYTYPE_ARRAY`, `srcArray` specifies the handle of the source data. `srcHost`, `srcDevice` and `srcPitch` are ignored.

If `dstMemoryType` is `CU_MEMORYTYPE_HOST`, `dstHost` and `dstPitch` specify the (host) base address of the destination data and the bytes per row to apply. `dstArray` is ignored.

If `dstMemoryType` is `CU_MEMORYTYPE_UNIFIED`, `dstDevice` and `dstPitch` specify the (unified virtual address space) base address of the source data and the bytes per row to apply. `dstArray` is ignored. This value may be used only if unified addressing is supported in the calling context.

If `dstMemoryType` is `CU_MEMORYTYPE_DEVICE`, `dstDevice` and `dstPitch` specify the (device) base address of the destination data and the bytes per row to apply. `dstArray` is ignored.

If `dstMemoryType` is `CU_MEMORYTYPE_ARRAY`, `dstArray` specifies the handle of the destination data. `dstHost`, `dstDevice` and `dstPitch` are ignored.

- `srcXInBytes` and `srcY` specify the base address of the source data for the copy.

For host pointers, the starting address is

```
↑ void* Start = (void*)((char*)srcHost+srcY*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
↑ CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
```

For CUDA arrays, `srcXInBytes` must be evenly divisible by the array element size.

- ▶ `dstXInBytes` and `dstY` specify the base address of the destination data for the copy.

For host pointers, the base address is

```
↑ void* dstStart = (void*)((char*)dstHost+dstY*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
↑ CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
```

For CUDA arrays, `dstXInBytes` must be evenly divisible by the array element size.

- ▶ `WidthInBytes` and `Height` specify the width (in bytes) and height of the 2D copy being performed.
- ▶ If specified, `srcPitch` must be greater than or equal to `WidthInBytes + srcXInBytes`, and `dstPitch` must be greater than or equal to `WidthInBytes + dstXInBytes`.

[`cuMemcpy2D\(\)`](#) returns an error if any pitch is greater than the maximum allowed (`CU_DEVICE_ATTRIBUTE_MAX_PITCH`). [`cuMemAllocPitch\(\)`](#) passes back pitches that always work with [`cuMemcpy2D\(\)`](#). On intra-device memory copies (device to device, CUDA array to device, CUDA array to CUDA array), [`cuMemcpy2D\(\)`](#) may fail for pitches not computed by [`cuMemAllocPitch\(\)`](#). [`cuMemcpy2DUnaligned\(\)`](#) does not have this restriction, but may run significantly slower in the cases where [`cuMemcpy2D\(\)`](#) would have returned an error code.



Note:

- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ This function exhibits [`synchronous`](#) behavior for most use cases.

See also:

[`cuArray3DCreate`](#), [`cuArray3DGetDescriptor`](#), [`cuArrayCreate`](#), [`cuArrayDestroy`](#), [`cuArrayGetDescriptor`](#), [`cuMemAlloc`](#), [`cuMemAllocHost`](#), [`cuMemAllocPitch`](#), [`cuMemcpy2DAsync`](#), [`cuMemcpy2DUnaligned`](#), [`cuMemcpy3D`](#), [`cuMemcpy3DAsync`](#), [`cuMemcpyAtoA`](#), [`cuMemcpyAtoD`](#), [`cuMemcpyAtoH`](#), [`cuMemcpyAtoHAsync`](#), [`cuMemcpyDtoA`](#), [`cuMemcpyDtoD`](#), [`cuMemcpyDtoDAsync`](#), [`cuMemcpyDtoH`](#), [`cuMemcpyDtoHAsync`](#), [`cuMemcpyHtoA`](#), [`cuMemcpyHtoAAsync`](#), [`cuMemcpyHtoD`](#), [`cuMemcpyHtoDAsync`](#), [`cuMemFree`](#), [`cuMemFreeHost`](#), [`cuMemGetAddressRange`](#), [`cuMemGetInfo`](#), [`cuMemHostAlloc`](#), [`cuMemHostGetDevicePointer`](#), [`cuMemsetD2D8`](#), [`cuMemsetD2D16`](#), [`cuMemsetD2D32`](#), [`cuMemsetD8`](#), [`cuMemsetD16`](#), [`cuMemsetD32`](#), [`cudaMemcpy2D`](#), [`cudaMemcpy2DToArray`](#), [`cudaMemcpy2DFromArray`](#)

CUresult cuMemcpy2DAsync (const CUDA_MEMCPY2D *pCopy, CUstream hStream)

Copies memory for 2D arrays.

Parameters

pCopy

- Parameters for the memory copy

hStream

- Stream identifier

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#)

Description

Perform a 2D memory copy according to the parameters specified in pCopy. The CUDA_MEMCPY2D structure is defined as:

```
typedef struct CUDA_MEMCPY2D_st {
    unsigned int srcXInBytes, srcY;
    CUmemorytype srcMemoryType;
    const void *srcHost;
    CUdeviceptr srcDevice;
    CUarray srcArray;
    unsigned int srcPitch;
    unsigned int dstXInBytes, dstY;
    CUmemorytype dstMemoryType;
    void *dstHost;
    CUdeviceptr dstDevice;
    CUarray dstArray;
    unsigned int dstPitch;
    unsigned int WidthInBytes;
    unsigned int Height;
} CUDA_MEMCPY2D;
```

where:

- srcMemoryType and dstMemoryType specify the type of memory of the source and destination, respectively; CUmemorytype_enum is defined as:

```
typedef enum CUmemorytype_enum {
    CU_MEMORYTYPE_HOST = 0x01,
    CU_MEMORYTYPE_DEVICE = 0x02,
    CU_MEMORYTYPE_ARRAY = 0x03,
    CU_MEMORYTYPE_UNIFIED = 0x04
} CUmemorytype;
```

If srcMemoryType is [CU_MEMORYTYPE_HOST](#), srcHost and srcPitch specify the (host) base address of the source data and the bytes per row to apply. srcArray is ignored.

If `srcMemoryType` is [CU_MEMORYTYPE_UNIFIED](#), `srcDevice` and `srcPitch` specify the (unified virtual address space) base address of the source data and the bytes per row to apply. `srcArray` is ignored. This value may be used only if unified addressing is supported in the calling context.

If `srcMemoryType` is [CU_MEMORYTYPE_DEVICE](#), `srcDevice` and `srcPitch` specify the (device) base address of the source data and the bytes per row to apply. `srcArray` is ignored.

If `srcMemoryType` is [CU_MEMORYTYPE_ARRAY](#), `srcArray` specifies the handle of the source data. `srcHost`, `srcDevice` and `srcPitch` are ignored.

If `dstMemoryType` is [CU_MEMORYTYPE_UNIFIED](#), `dstDevice` and `dstPitch` specify the (unified virtual address space) base address of the source data and the bytes per row to apply. `dstArray` is ignored. This value may be used only if unified addressing is supported in the calling context.

If `dstMemoryType` is [CU_MEMORYTYPE_HOST](#), `dstHost` and `dstPitch` specify the (host) base address of the destination data and the bytes per row to apply. `dstArray` is ignored.

If `dstMemoryType` is [CU_MEMORYTYPE_DEVICE](#), `dstDevice` and `dstPitch` specify the (device) base address of the destination data and the bytes per row to apply. `dstArray` is ignored.

If `dstMemoryType` is [CU_MEMORYTYPE_ARRAY](#), `dstArray` specifies the handle of the destination data. `dstHost`, `dstDevice` and `dstPitch` are ignored.

- `srcXInBytes` and `srcY` specify the base address of the source data for the copy.

For host pointers, the starting address is

```
↑ void* Start = (void*)((char*)srcHost+srcY*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
↑ CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
```

For CUDA arrays, `srcXInBytes` must be evenly divisible by the array element size.

- `dstXInBytes` and `dstY` specify the base address of the destination data for the copy.

For host pointers, the base address is

```
↑ void* dstStart = (void*)((char*)dstHost+dstY*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
↑ CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
```

For CUDA arrays, `dstXInBytes` must be evenly divisible by the array element size.

- `WidthInBytes` and `Height` specify the width (in bytes) and height of the 2D copy being performed.
- If specified, `srcPitch` must be greater than or equal to `WidthInBytes + srcXInBytes`, and `dstPitch` must be greater than or equal to `WidthInBytes + dstXInBytes`.
- If specified, `srcPitch` must be greater than or equal to `WidthInBytes + srcXInBytes`, and `dstPitch` must be greater than or equal to `WidthInBytes + dstXInBytes`.

- If specified, `srcHeight` must be greater than or equal to `Height + srcY`, and `dstHeight` must be greater than or equal to `Height + dstY`.

`cuMemcpy2DAsync()` returns an error if any pitch is greater than the maximum allowed (`CU_DEVICE_ATTRIBUTE_MAX_PITCH`). `cuMemAllocPitch()` passes back pitches that always work with `cuMemcpy2D()`. On intra-device memory copies (device to device, CUDA array to device, CUDA array to CUDA array), `cuMemcpy2DAsync()` may fail for pitches not computed by `cuMemAllocPitch()`.



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.

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#), [cudaMemcpy2DAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#)

CUresult cuMemcpy2DUnaligned (const CUDA_MEMCPY2D *pCopy)

Copies memory for 2D arrays.

Parameters

pCopy

- Parameters for the memory copy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Perform a 2D memory copy according to the parameters specified in `pCopy`. The `CUDA_MEMCPY2D` structure is defined as:

```
typedef struct CUDA_MEMCPY2D_st {
    unsigned int srcXInBytes, srcY;
    CUmemorytype srcMemoryType;
    const void *srcHost;
    CUdeviceptr srcDevice;
    CUarray srcArray;
    unsigned int srcPitch;
    unsigned int dstXInBytes, dstY;
    CUmemorytype dstMemoryType;
    void *dstHost;
    CUdeviceptr dstDevice;
    CUarray dstArray;
    unsigned int dstPitch;
    unsigned int WidthInBytes;
    unsigned int Height;
} CUDA_MEMCPY2D;
```

where:

- `srcMemoryType` and `dstMemoryType` specify the type of memory of the source and destination, respectively; `CUmemorytype_enum` is defined as:

```
typedef enum CUmemorytype_enum {
    CU_MEMORYTYPE_HOST = 0x01,
    CU_MEMORYTYPE_DEVICE = 0x02,
    CU_MEMORYTYPE_ARRAY = 0x03,
    CU_MEMORYTYPE_UNIFIED = 0x04
} CUmemorytype;
```

If `srcMemoryType` is `CU_MEMORYTYPE_UNIFIED`, `srcDevice` and `srcPitch` specify the (unified virtual address space) base address of the source data and the bytes per row to apply. `srcArray` is ignored. This value may be used only if unified addressing is supported in the calling context.

If `srcMemoryType` is `CU_MEMORYTYPE_HOST`, `srcHost` and `srcPitch` specify the (host) base address of the source data and the bytes per row to apply. `srcArray` is ignored.

If `srcMemoryType` is `CU_MEMORYTYPE_DEVICE`, `srcDevice` and `srcPitch` specify the (device) base address of the source data and the bytes per row to apply. `srcArray` is ignored.

If `srcMemoryType` is `CU_MEMORYTYPE_ARRAY`, `srcArray` specifies the handle of the source data. `srcHost`, `srcDevice` and `srcPitch` are ignored.

If `dstMemoryType` is `CU_MEMORYTYPE_UNIFIED`, `dstDevice` and `dstPitch` specify the (unified virtual address space) base address of the source data and the bytes per row to apply. `dstArray` is ignored. This value may be used only if unified addressing is supported in the calling context.

If `dstMemoryType` is `CU_MEMORYTYPE_HOST`, `dstHost` and `dstPitch` specify the (host) base address of the destination data and the bytes per row to apply. `dstArray` is ignored.

If `dstMemoryType` is [CU_MEMORYTYPE_DEVICE](#), `dstDevice` and `dstPitch` specify the (device) base address of the destination data and the bytes per row to apply. `dstArray` is ignored.

If `dstMemoryType` is [CU_MEMORYTYPE_ARRAY](#), `dstArray` specifies the handle of the destination data. `dstHost`, `dstDevice` and `dstPitch` are ignored.

- `srcXInBytes` and `srcY` specify the base address of the source data for the copy.

For host pointers, the starting address is

```
↑ void* Start = (void*)((char*)srcHost+srcY*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
↑ CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
```

For CUDA arrays, `srcXInBytes` must be evenly divisible by the array element size.

- `dstXInBytes` and `dstY` specify the base address of the destination data for the copy.

For host pointers, the base address is

```
↑ void* dstStart = (void*)((char*)dstHost+dstY*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
↑ CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
```

For CUDA arrays, `dstXInBytes` must be evenly divisible by the array element size.

- `WidthInBytes` and `Height` specify the width (in bytes) and height of the 2D copy being performed.
- If specified, `srcPitch` must be greater than or equal to `WidthInBytes` + `srcXInBytes`, and `dstPitch` must be greater than or equal to `WidthInBytes` + `dstXInBytes`.

[cuMemcpy2D\(\)](#) returns an error if any pitch is greater than the maximum allowed ([CU_DEVICE_ATTRIBUTE_MAX_PITCH](#)). [cuMemAllocPitch\(\)](#) passes back pitches that always work with [cuMemcpy2D\(\)](#). On intra-device memory copies (device to device, CUDA array to device, CUDA array to CUDA array), [cuMemcpy2D\(\)](#) may fail for pitches not computed by [cuMemAllocPitch\(\)](#). [cuMemcpy2DUnaligned\(\)](#) does not have this restriction, but may run significantly slower in the cases where [cuMemcpy2D\(\)](#) would have returned an error code.



Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits [synchronous](#) behavior for most use cases.

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#),

[cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaMemcpy2D](#), [cudaMemcpy2DToArray](#), [cudaMemcpy2DFromArray](#)

CUresult cuMemcpy3D (const CUDA_MEMCPY3D *pCopy)

Copies memory for 3D arrays.

Parameters

pCopy

- Parameters for the memory copy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Perform a 3D memory copy according to the parameters specified in pCopy. The CUDA_MEMCPY3D structure is defined as:

```
typedef struct CUDA_MEMCPY3D_st {
    unsigned int srcXInBytes, srcY, srcZ;
    unsigned int srcLOD;
    CUmemorytype srcMemoryType;
    const void *srcHost;
    CUdeviceptr srcDevice;
    CUarray srcArray;
    unsigned int srcPitch; // ignored when src is array
    unsigned int srcHeight; // ignored when src is array; may be 0

    if Depth==1

        unsigned int dstXInBytes, dstY, dstZ;
        unsigned int dstLOD;
        CUmemorytype dstMemoryType;
        void *dstHost;
        CUdeviceptr dstDevice;
        CUarray dstArray;
        unsigned int dstPitch; // ignored when dst is array
        unsigned int dstHeight; // ignored when dst is array; may be 0

    if Depth==1

        unsigned int WidthInBytes;
        unsigned int Height;
        unsigned int Depth;
    } CUDA_MEMCPY3D;
```

where:

- `srcMemoryType` and `dstMemoryType` specify the type of memory of the source and destination, respectively; `CUmemorytype_enum` is defined as:

```
typedef enum CUmemorytype_enum {
    CU_MEMORYTYPE_HOST = 0x01,
    CU_MEMORYTYPE_DEVICE = 0x02,
    CU_MEMORYTYPE_ARRAY = 0x03,
    CU_MEMORYTYPE_UNIFIED = 0x04
} CUmemorytype;
```

If `srcMemoryType` is `CU_MEMORYTYPE_UNIFIED`, `srcDevice` and `srcPitch` specify the (unified virtual address space) base address of the source data and the bytes per row to apply. `srcArray` is ignored. This value may be used only if unified addressing is supported in the calling context.

If `srcMemoryType` is `CU_MEMORYTYPE_HOST`, `srcHost`, `srcPitch` and `srcHeight` specify the (host) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. `srcArray` is ignored.

If `srcMemoryType` is `CU_MEMORYTYPE_DEVICE`, `srcDevice`, `srcPitch` and `srcHeight` specify the (device) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. `srcArray` is ignored.

If `srcMemoryType` is `CU_MEMORYTYPE_ARRAY`, `srcArray` specifies the handle of the source data. `srcHost`, `srcDevice`, `srcPitch` and `srcHeight` are ignored.

If `dstMemoryType` is `CU_MEMORYTYPE_UNIFIED`, `dstDevice` and `dstPitch` specify the (unified virtual address space) base address of the source data and the bytes per row to apply. `dstArray` is ignored. This value may be used only if unified addressing is supported in the calling context.

If `dstMemoryType` is `CU_MEMORYTYPE_HOST`, `dstHost` and `dstPitch` specify the (host) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. `dstArray` is ignored.

If `dstMemoryType` is `CU_MEMORYTYPE_DEVICE`, `dstDevice` and `dstPitch` specify the (device) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. `dstArray` is ignored.

If `dstMemoryType` is `CU_MEMORYTYPE_ARRAY`, `dstArray` specifies the handle of the destination data. `dstHost`, `dstDevice`, `dstPitch` and `dstHeight` are ignored.

- `srcXInBytes`, `srcY` and `srcZ` specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*)((char*)srcHost+(srcZ*srcHeight+srcY)*srcPitch +
srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice+(srcZ*srcHeight+srcY)*srcPitch+srcXInBytes;
```

For CUDA arrays, `srcXInBytes` must be evenly divisible by the array element size.

- ▶ `dstXInBytes`, `dstY` and `dstZ` specify the base address of the destination data for the copy.

For host pointers, the base address is

```
↑ void* dstStart = (void*)((char*)dstHost + (dstZ*dstHeight + dstY)*dstPitch +
dstXInBytes);
```

For device pointers, the starting address is

```
↑ CUdeviceptr dstStart = dstDevice + (dstZ*dstHeight + dstY)*dstPitch + dstXInBytes;
```

For CUDA arrays, `dstXInBytes` must be evenly divisible by the array element size.

- ▶ `WidthInBytes`, `Height` and `Depth` specify the width (in bytes), height and depth of the 3D copy being performed.
- ▶ If specified, `srcPitch` must be greater than or equal to `WidthInBytes + srcXInBytes`, and `dstPitch` must be greater than or equal to `WidthInBytes + dstXInBytes`.
- ▶ If specified, `srcHeight` must be greater than or equal to `Height + srcY`, and `dstHeight` must be greater than or equal to `Height + dstY`.

[`cuMemcpy3D\(\)`](#) returns an error if any pitch is greater than the maximum allowed (`CU_DEVICE_ATTRIBUTE_MAX_PITCH`).

The `srcLOD` and `dstLOD` members of the `CUDA_MEMCPY3D` structure must be set to 0.



Note:

- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ This function exhibits [`synchronous`](#) behavior for most use cases.

See also:

[`cuArray3DCreate`](#), [`cuArray3DGetDescriptor`](#), [`cuArrayCreate`](#), [`cuArrayDestroy`](#), [`cuArrayGetDescriptor`](#), [`cuMemAlloc`](#), [`cuMemAllocHost`](#), [`cuMemAllocPitch`](#), [`cuMemcpy2D`](#), [`cuMemcpy2DAsync`](#), [`cuMemcpy2DUnaligned`](#), [`cuMemcpy3DAsync`](#), [`cuMemcpyAtoA`](#), [`cuMemcpyAtoD`](#), [`cuMemcpyAtoH`](#), [`cuMemcpyAtoHAsync`](#), [`cuMemcpyDtoA`](#), [`cuMemcpyDtoD`](#), [`cuMemcpyDtoDAsync`](#), [`cuMemcpyDtoH`](#), [`cuMemcpyDtoHAsync`](#), [`cuMemcpyHtoA`](#), [`cuMemcpyHtoAAsync`](#), [`cuMemcpyHtoD`](#), [`cuMemcpyHtoDAsync`](#), [`cuMemFree`](#), [`cuMemFreeHost`](#), [`cuMemGetAddressRange`](#), [`cuMemGetInfo`](#), [`cuMemHostAlloc`](#), [`cuMemHostGetDevicePointer`](#), [`cuMemsetD2D8`](#), [`cuMemsetD2D16`](#), [`cuMemsetD2D32`](#), [`cuMemsetD8`](#), [`cuMemsetD16`](#), [`cuMemsetD32`](#), [`cudaMemcpy3D`](#)

CUresult cuMemcpy3DAsync (const CUDA_MEMCPY3D *pCopy, CUstream hStream)

Copies memory for 3D arrays.

Parameters

pCopy

- Parameters for the memory copy

hStream

- Stream identifier

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#)

Description

Perform a 3D memory copy according to the parameters specified in pCopy. The CUDA_MEMCPY3D structure is defined as:

```
↑      typedef struct CUDA_MEMCPY3D_st {
                unsigned int srcXInBytes, srcY, srcZ;
                unsigned int srcLOD;
                CUmemorytype srcMemoryType;
                const void *srcHost;
                CUdeviceptr srcDevice;
                CUarray srcArray;
                unsigned int srcPitch; // ignored when src is array
                unsigned int srcHeight; // ignored when src is array; may be 0
if Depth==1
                unsigned int dstXInBytes, dstY, dstZ;
                unsigned int dstLOD;
                CUmemorytype dstMemoryType;
                void *dstHost;
                CUdeviceptr dstDevice;
                CUarray dstArray;
                unsigned int dstPitch; // ignored when dst is array
                unsigned int dstHeight; // ignored when dst is array; may be 0
if Depth==1
                unsigned int WidthInBytes;
                unsigned int Height;
                unsigned int Depth;
                } CUDA_MEMCPY3D;
```

where:

- srcMemoryType and dstMemoryType specify the type of memory of the source and destination, respectively; CUmemorytype_enum is defined as:

```
↑      typedef enum CUmemorytype_enum {
                CU_MEMORYTYPE_HOST = 0x01,
                CU_MEMORYTYPE_DEVICE = 0x02,
```

```

    CU_MEMORYTYPE_ARRAY = 0x03,
    CU_MEMORYTYPE_UNIFIED = 0x04
} CUmemorytype;

```

If srcMemoryType is [CU_MEMORYTYPE_UNIFIED](#), srcDevice and srcPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. srcArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If srcMemoryType is [CU_MEMORYTYPE_HOST](#), srcHost, srcPitch and srcHeight specify the (host) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. srcArray is ignored.

If srcMemoryType is [CU_MEMORYTYPE_DEVICE](#), srcDevice, srcPitch and srcHeight specify the (device) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. srcArray is ignored.

If srcMemoryType is [CU_MEMORYTYPE_ARRAY](#), srcArray specifies the handle of the source data. srcHost, srcDevice, srcPitch and srcHeight are ignored.

If dstMemoryType is [CU_MEMORYTYPE_UNIFIED](#), dstDevice and dstPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. dstArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If dstMemoryType is [CU_MEMORYTYPE_HOST](#), dstHost and dstPitch specify the (host) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. dstArray is ignored.

If dstMemoryType is [CU_MEMORYTYPE_DEVICE](#), dstDevice and dstPitch specify the (device) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. dstArray is ignored.

If dstMemoryType is [CU_MEMORYTYPE_ARRAY](#), dstArray specifies the handle of the destination data. dstHost, dstDevice, dstPitch and dstHeight are ignored.

- srcXInBytes, srcY and srcZ specify the base address of the source data for the copy.

For host pointers, the starting address is

```

↑ void* Start = (void*)((char*)srcHost+(srcZ*srcHeight+srcY)*srcPitch +
srcXInBytes);

```

For device pointers, the starting address is

```

↑ CUdeviceptr Start = srcDevice+(srcZ*srcHeight+srcY)*srcPitch+srcXInBytes;

```

For CUDA arrays, srcXInBytes must be evenly divisible by the array element size.

- dstXInBytes, dstY and dstZ specify the base address of the destination data for the copy.

For host pointers, the base address is

```

↑ void* dstStart = (void*)((char*)dstHost+(dstZ*dstHeight+dstY)*dstPitch +
dstXInBytes);

```

For device pointers, the starting address is

```
↑ CUdeviceptr dstStart = dstDevice+(dstZ*dstHeight+dstY)*dstPitch+dstXInBytes;
```

For CUDA arrays, `dstXInBytes` must be evenly divisible by the array element size.

- ▶ `WidthInBytes`, `Height` and `Depth` specify the width (in bytes), height and depth of the 3D copy being performed.
- ▶ If specified, `srcPitch` must be greater than or equal to `WidthInBytes + srcXInBytes`, and `dstPitch` must be greater than or equal to `WidthInBytes + dstXInBytes`.
- ▶ If specified, `srcHeight` must be greater than or equal to `Height + srcY`, and `dstHeight` must be greater than or equal to `Height + dstY`.

[`cuMemcpy3DAsync\(\)`](#) returns an error if any pitch is greater than the maximum allowed (`CU_DEVICE_ATTRIBUTE_MAX_PITCH`).

The `srcLOD` and `dstLOD` members of the `CUDA_MEMCPY3D` structure must be set to 0.



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.

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#), [cudaMemcpy3DAsync](#)

CUresult cuMemcpy3DBatchAsync (size_t numOps, CUDA_MEMCPY3D_BATCH_OP *opList, size_t *failIdx, unsigned long long flags, CUstream hStream)

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.

hStream

- The stream to enqueue the operations in. Must not be default NULL stream.

Returns

CUDA_SUCCESS CUDA_ERROR_DEINITIALIZED CUDA_ERROR_NOT_INITIALIZED
CUDA_ERROR_INVALID_VALUE

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 `CUDA_MEMCPY3D_BATCH_OP::src` and `CUDA_MEMCPY3D_BATCH_OP::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 `CUDA_MEMCPY3D_BATCH_OP::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 `CUmemcpy3DOperand::type` is specified as CU_MEMCPY_OPERAND_TYPE_POINTER, then `CUmemcpy3DOperand::op::ptr` will be used. The `CUmemcpy3DOperand::op::ptr::ptr` field must contain the pointer where the copy should begin. The `CUmemcpy3DOperand::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 `CUDA_MEMCPY3D_BATCH_OP::extent::width`. The `CUmemcpy3DOperand::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 `CUDA_MEMCPY3D_BATCH_OP::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 [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#) is true or system-allocated pageable memory on devices where [CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS](#) is true, the `CUmemcpy3DOperand::op::ptr::locHint` field can be used to hint the location of the operand.

If an operand's type is specified as [CU_MEMCPY_OPERAND_TYPE_ARRAY](#), then `CUmemcpy3DOperand::op::array` will be used. The `CUmemcpy3DOperand::op::array::array` field specifies the CUDA array and `CUmemcpy3DOperand::op::array::offset` specifies the 3D offset into that array where the copy begins.

The `CUmemcpyAttributes::srcAccessOrder` indicates the source access ordering to be observed for copies associated with the attribute. If the source access order is set to [CU_MEMCPY_SRC_ACCESS_ORDER_STREAM](#), then the source will be accessed in stream order. If the source access order is set to [CU_MEMCPY_SRC_ACCESS_ORDER_DURING_API_CALL](#) 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 [CU_MEMCPY_SRC_ACCESS_ORDER_ANY](#) 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 `opList` must have a valid `srcAccessOrder` setting, otherwise this API will return [CUDA_ERROR_INVALID_VALUE](#).

The `CUmemcpyAttributes::flags` field can be used to specify certain flags for copies. Setting the [CU_MEMCPY_FLAG_PREFER_OVERLAP_WITH_COMPUTE](#) 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`.

CUresult cuMemcpy3DPeer (const CUDA_MEMCPY3D_PEER *pCopy)

Copies memory between contexts.

Parameters

pCopy

- Parameters for the memory copy

Returns

[`CUDA\_SUCCESS`](#), [`CUDA\_ERROR\_DEINITIALIZED`](#), [`CUDA\_ERROR\_NOT\_INITIALIZED`](#), [`CUDA\_ERROR\_INVALID\_CONTEXT`](#), [`CUDA\_ERROR\_INVALID\_VALUE`](#)

Description

Perform a 3D memory copy according to the parameters specified in `pCopy`. See the definition of the `CUDA_MEMCPY3D_PEER` structure for documentation of its parameters.



Note:

- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ This function exhibits [`synchronous`](#) behavior for most use cases.

See also:

[`cuMemcpyDtoD`](#), [`cuMemcpyPeer`](#), [`cuMemcpyDtoDAsync`](#), [`cuMemcpyPeerAsync`](#), [`cuMemcpy3DPeerAsync`](#), [`cudaMemcpy3DPeer`](#)

CUresult cuMemcpy3DPeerAsync (const CUDA_MEMCPY3D_PEER *pCopy, CUstream hStream)

Copies memory between contexts asynchronously.

Parameters

pCopy

- Parameters for the memory copy

hStream

- Stream identifier

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Perform a 3D memory copy according to the parameters specified in `pCopy`. See the definition of the `CUDA_MEMCPY3D_PEER` 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.

See also:

[cuMemcpyDtoD](#), [cuMemcpyPeer](#), [cuMemcpyDtoDAsync](#), [cuMemcpyPeerAsync](#),
[cuMemcpy3DPeerAsync](#), [cudaMemcpy3DPeerAsync](#)

CUresult cuMemcpyAsync (CUdeviceptr dst, CUdeviceptr src, size_t ByteCount, CUstream hStream)

Copies memory asynchronously.

Parameters**dst**

- Destination unified virtual address space pointer

src

- Source unified virtual address space pointer

ByteCount

- Size of memory copy in bytes

hStream

- Stream identifier

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#)

Description

Copies data between two pointers. `dst` and `src` are base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy. Note that this function infers the type of the transfer (host to host, host to device, device to device, or device to host) from the pointer values. This function is only allowed in contexts which support unified addressing.



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.
- ▶ 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:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#), [cudaMemcpyAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

CUresult cuMemcpyAtoA (CUarray dstArray, size_t dstOffset, CUarray srcArray, size_t srcOffset, size_t ByteCount)

Copies memory from Array to Array.

Parameters

dstArray

- Destination array

dstOffset

- Offset in bytes of destination array

srcArray

- Source array

srcOffset

- Offset in bytes of source array

ByteCount

- Size of memory copy in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Copies from one 1D CUDA array to another. `dstArray` and `srcArray` specify the handles of the destination and source CUDA arrays for the copy, respectively. `dstOffset` and `srcOffset` specify the destination and source offsets in bytes into the CUDA arrays. `ByteCount` is the number of bytes to be copied. The size of the elements in the CUDA arrays need not be the same format, but the elements must be the same size; and count must be evenly divisible by that size.

**Note:**

- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ This function exhibits [synchronous](#) behavior for most use cases.

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#),
[cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#),
[cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#),
[cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#),
[cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#),
[cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#),
[cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#),
[cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaMemcpyArrayToArray](#)

CUresult cuMemcpyAtoD (CUdeviceptr dstDevice, CUarray srcArray, size_t srcOffset, size_t ByteCount)

Copies memory from Array to Device.

Parameters**dstDevice**

- Destination device pointer

srcArray

- Source array

srcOffset

- Offset in bytes of source array

ByteCount

- Size of memory copy in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Copies from one 1D CUDA array to device memory. `dstDevice` specifies the base pointer of the destination and must be naturally aligned with the CUDA array elements. `srcArray` and `srcOffset` specify the CUDA array handle and the offset in bytes into the array where the copy is to begin. `ByteCount` specifies the number of bytes to copy and must be evenly divisible by the array element size.

**Note:**

- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ This function exhibits [synchronous](#) behavior for most use cases.

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaMemcpyFromArray](#)

CUresult cuMemcpyAtoH (void *dstHost, CUarray srcArray, size_t srcOffset, size_t ByteCount)

Copies memory from Array to Host.

Parameters**dstHost**

- Destination device pointer

srcArray

- Source array

srcOffset

- Offset in bytes of source array

ByteCount

- Size of memory copy in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Copies from one 1D CUDA array to host memory. `dstHost` specifies the base pointer of the destination. `srcArray` and `srcOffset` specify the CUDA array handle and starting offset in bytes of the source data. `ByteCount` specifies the number of bytes to copy.

**Note:**

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

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#),
[cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#),
[cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#),
[cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#),
[cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#),
[cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#),
[cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#),
[cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaMemcpyFromArray](#)

CUresult cuMemcpyAtoHAsync (void *dstHost, CUarray srcArray, size_t srcOffset, size_t ByteCount, CUstream hStream)

Copies memory from Array to Host.

Parameters

dstHost

- Destination pointer

srcArray

- Source array

srcOffset

- Offset in bytes of source array

ByteCount

- Size of memory copy in bytes

hStream

- Stream identifier

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#)

Description

Copies from one 1D CUDA array to host memory. `dstHost` specifies the base pointer of the destination. `srcArray` and `srcOffset` specify the CUDA array handle and starting offset in bytes of the source data. `ByteCount` specifies the number of bytes to copy.



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.
- ▶ 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:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#), [cudaMemcpyFromArrayAsync](#)

CUresult cuMemcpyBatchAsync (CUdeviceptr *dsts, CUdeviceptr *srcs, size_t *sizes, size_t count, CUmemcopyAttributes *attrs, size_t *attrIdxs, size_t numAttrs, size_t *failIdx, CUstream hStream)

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.

attrIdxs

- 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 `attrIdxs[k]` through `attrIdxs[k+1] - 1`. Also `attrs[numAttrs-1]` will apply to copies starting from `attrIdxs[numAttrs-1]` through `count - 1`.

numAttrs

- Size of `attrs` and `attrIdxs` 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.

hStream

- The stream to enqueue the operations in. Must not be legacy NULL stream.

Returns

CUDA_SUCCESS CUDA_ERROR_DEINITIALIZED CUDA_ERROR_NOT_INITIALIZED
CUDA_ERROR_INVALID_VALUE

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 [cuMemcpy3DBatchAsync](#).

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 contain 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 [CUmemcpyAttributes::srcAccessOrder](#) indicates the source access ordering to be observed for copies associated with the attribute. If the source access order is set to [CU_MEMCPY_SRC_ACCESS_ORDER_STREAM](#), then the source will be accessed in stream order. If the source access order is set to [CU_MEMCPY_SRC_ACCESS_ORDER_DURING_API_CALL](#) 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 [CU_MEMCPY_SRC_ACCESS_ORDER_ANY](#) 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 [CUmemcpyAttributes](#) corresponding to it including the appropriate `srcAccessOrder` setting, otherwise the API will return [CUDA_ERROR_INVALID_VALUE](#).

The [CUmemcpyAttributes::srcLocHint](#) and [CUmemcpyAttributes::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 [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#) is true or system-allocated pageable memory on devices where [CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS](#) is true. For other cases, these hints are ignored.

The [CUmemcpyAttributes::flags](#) field can be used to specify certain flags for copies. Setting the [CU_MEMCPY_FLAG_PREFER_OVERLAP_WITH_COMPUTE](#) 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`.

CUresult cuMemcpyDtoA (CUarray dstArray, size_t dstOffset, CUdeviceptr srcDevice, size_t ByteCount)

Copies memory from Device to Array.

Parameters

dstArray

- Destination array

dstOffset

- Offset in bytes of destination array

srcDevice

- Source device pointer

ByteCount

- Size of memory copy in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Copies from device memory to a 1D CUDA array. `dstArray` and `dstOffset` specify the CUDA array handle and starting index of the destination data. `srcDevice` specifies the base pointer of the source. `ByteCount` specifies the number of bytes to copy.



Note:

- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ This function exhibits [synchronous](#) behavior for most use cases.

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaMemcpyToArray](#)

CUresult cuMemcpyDtoD (CUdeviceptr dstDevice, CUdeviceptr srcDevice, size_t ByteCount)

Copies memory from Device to Device.

Parameters

dstDevice

- Destination device pointer

srcDevice

- Source device pointer

ByteCount

- Size of memory copy in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Copies from device memory to device memory. `dstDevice` and `srcDevice` are the base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy.

**Note:**

- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ This function exhibits [synchronous](#) behavior for most use cases.

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaMemcpy](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#)

CUresult cuMemcpyDtoDAsync (CUdeviceptr dstDevice, CUdeviceptr srcDevice, size_t ByteCount, CUstream hStream)

Copies memory from Device to Device.

Parameters

dstDevice

- Destination device pointer

srcDevice

- Source device pointer

ByteCount

- Size of memory copy in bytes

hStream

- Stream identifier

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#)

Description

Copies from device memory to device memory. `dstDevice` and `srcDevice` are the base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy.

**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.

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#), [cudaMemcpyAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

CUresult cuMemcpyDtoH (void *dstHost, CUdeviceptr srcDevice, size_t ByteCount)

Copies memory from Device to Host.

Parameters

dstHost

- Destination host pointer

srcDevice

- Source device pointer

ByteCount

- Size of memory copy in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Copies from device to host memory. `dstHost` and `srcDevice` specify the base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy.

**Note:**

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

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaMemcpy](#), [cudaMemcpyFromSymbol](#)

CUresult cuMemcpyDtoHAsync (void *dstHost, CUdeviceptr srcDevice, size_t ByteCount, CUstream hStream)

Copies memory from Device to Host.

Parameters

dstHost

- Destination host pointer

srcDevice

- Source device pointer

ByteCount

- Size of memory copy in bytes

hStream

- Stream identifier

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#)

Description

Copies from device to host memory. `dstHost` and `srcDevice` specify the base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy.



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.
- ▶ 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:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#), [cudaMemcpyAsync](#), [cudaMemcpyFromSymbolAsync](#)

CUresult cuMemcpyHtoA (CUarray dstArray, size_t dstOffset, const void *srcHost, size_t ByteCount)

Copies memory from Host to Array.

Parameters

dstArray

- Destination array

dstOffset

- Offset in bytes of destination array

srcHost

- Source host pointer

ByteCount

- Size of memory copy in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Copies from host memory to a 1D CUDA array. `dstArray` and `dstOffset` specify the CUDA array handle and starting offset in bytes of the destination data. `pSrc` specifies the base address of the source. `ByteCount` specifies the number of bytes to copy.



Note:

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

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaMemcpyToArray](#)

CUresult cuMemcpyHtoAAsync (CUarray dstArray, size_t dstOffset, const void *srcHost, size_t ByteCount, CUstream hStream)

Copies memory from Host to Array.

Parameters

dstArray

- Destination array

dstOffset

- Offset in bytes of destination array

srcHost

- Source host pointer

ByteCount

- Size of memory copy in bytes

hStream

- Stream identifier

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#)

Description

Copies from host memory to a 1D CUDA array. `dstArray` and `dstOffset` specify the CUDA array handle and starting offset in bytes of the destination data. `srcHost` specifies the base address of the source. `ByteCount` specifies the number of bytes to copy.

**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.
- ▶ 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:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#),
[cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#),
[cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#),
[cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#),
[cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#),
[cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#),
[cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#),
[cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#),
[cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#),
[cudaMemcpyToArrayAsync](#)

CUresult cuMemcpyHtoD (CUdeviceptr dstDevice, const void *srcHost, size_t ByteCount)

Copies memory from Host to Device.

Parameters

dstDevice

- Destination device pointer

srcHost

- Source host pointer

ByteCount

- Size of memory copy in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Copies from host memory to device memory. `dstDevice` and `srcHost` are the base addresses of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy.



Note:

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

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaMemcpy](#), [cudaMemcpyToSymbol](#)

CUresult cuMemcpyHtoDAsync (CUdeviceptr dstDevice, const void *srcHost, size_t ByteCount, CUstream hStream)

Copies memory from Host to Device.

Parameters

dstDevice

- Destination device pointer

srcHost

- Source host pointer

ByteCount

- Size of memory copy in bytes

hStream

- Stream identifier

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#)

Description

Copies from host memory to device memory. `dstDevice` and `srcHost` are the base addresses of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy.



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.
- ▶ 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:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#),

[cuMemcpyHtoD](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#), [cudaMemcpyAsync](#), [cudaMemcpyToSymbolAsync](#)

CUresult cuMemcpyPeer (CUdeviceptr dstDevice, CUcontext dstContext, CUdeviceptr srcDevice, CUcontext srcContext, size_t ByteCount)

Copies device memory between two contexts.

Parameters

dstDevice

- Destination device pointer

dstContext

- Destination context

srcDevice

- Source device pointer

srcContext

- Source context

ByteCount

- Size of memory copy in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Copies from device memory in one context to device memory in another context. `dstDevice` is the base device pointer of the destination memory and `dstContext` is the destination context. `srcDevice` is the base device pointer of the source memory and `srcContext` is the source pointer. `ByteCount` specifies the number of bytes to copy.



Note:

- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ This function exhibits [synchronous](#) behavior for most use cases.

See also:

[cuMemcpyDtoD](#), [cuMemcpy3DPeer](#), [cuMemcpyDtoDAsync](#), [cuMemcpyPeerAsync](#),
[cuMemcpy3DPeerAsync](#), [cudaMemcpyPeer](#)

CUresult cuMemcpyPeerAsync (CUdeviceptr dstDevice, CUcontext dstContext, CUdeviceptr srcDevice, CUcontext srcContext, size_t ByteCount, CUstream hStream)

Copies device memory between two contexts asynchronously.

Parameters

dstDevice

- Destination device pointer

dstContext

- Destination context

srcDevice

- Source device pointer

srcContext

- Source context

ByteCount

- Size of memory copy in bytes

hStream

- Stream identifier

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#)

Description

Copies from device memory in one context to device memory in another context. `dstDevice` is the base device pointer of the destination memory and `dstContext` is the destination context. `srcDevice` is the base device pointer of the source memory and `srcContext` is the source pointer. `ByteCount` specifies the number of bytes to copy.



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.

See also:

[cuMemcpyDtoD](#), [cuMemcpyPeer](#), [cuMemcpy3DPeer](#), [cuMemcpyDtoDAsync](#),
[cuMemcpy3DPeerAsync](#), [cudaMemcpyPeerAsync](#)

CUresult cuMemFree (CUdeviceptr dptr)

Frees device memory.

Parameters

dptr

- Pointer to memory to free

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Frees the memory space pointed to by `dptr`, which must have been returned by a previous call to one of the following memory allocation APIs - [cuMemAlloc\(\)](#), [cuMemAllocPitch\(\)](#), [cuMemAllocManaged\(\)](#), [cuMemAllocAsync\(\)](#), [cuMemAllocFromPoolAsync\(\)](#)

Note - This API will not perform any implicit synchronization when the pointer was allocated with [cuMemAllocAsync](#) or [cuMemAllocFromPoolAsync](#). Callers must ensure that all accesses to these pointer have completed before invoking [cuMemFree](#). For best performance and memory reuse, users should use [cuMemFreeAsync](#) to free memory allocated via the stream ordered memory allocator. For all other pointers, this API may perform implicit synchronization.



Note:

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

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#),
[cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemAllocManaged](#), [cuMemAllocAsync](#),
[cuMemAllocFromPoolAsync](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#),
[cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#),
[cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#),
[cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#),
[cuMemcpyHtoDAsync](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#),
[cuMemHostAlloc](#), [cuMemFreeAsync](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#),
[cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaFree](#)

CUresult cuMemFreeHost (void *p)

Frees page-locked host memory.

Parameters

p

- Pointer to memory to free

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Frees the memory space pointed to by `p`, which must have been returned by a previous call to [cuMemAllocHost\(\)](#).



Note:

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

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaFreeHost](#)

CUresult cuMemGetAddressRange (CUdeviceptr *pbase, size_t *psize, CUdeviceptr dptr)

Get information on memory allocations.

Parameters

pbase

- Returned base address

psize

- Returned size of device memory allocation

dptr

- Device pointer to query

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_NOT_FOUND](#),
[CUDA_ERROR_INVALID_VALUE](#)

Description

Returns the base address in *pbase and size in *psize of the allocation by [cuMemAlloc\(\)](#) or [cuMemAllocPitch\(\)](#) that contains the input pointer dptr. Both parameters pbase and psize are optional. If one of them is NULL, it is ignored.

**Note:**

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

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#),
[cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#),
[cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#),
[cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#),
[cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#),
[cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetInfo](#), [cuMemHostAlloc](#),
[cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#),
[cuMemsetD16](#), [cuMemsetD32](#)

CUresult cuMemGetHandleForAddressRange
 (void *handle, CUdeviceptr dptr, size_t size,
 CUmemRangeHandleType handleType, unsigned long long
 flags)

Retrieve handle for an address range.

Parameters**handle**

- Pointer to the location where the returned handle will be stored.

dptr

- Pointer to a valid CUDA device allocation. Must be aligned to host page size.

size

- Length of the address range. Must be aligned to host page size.

handleType

- Type of handle requested (defines type and size of the `handle` output parameter)

flags

- When requesting

CUmemRangeHandleType::CU_MEM_RANGE_HANDLE_TYPE_DMA_BUF_FD the value could be [CU_MEM_RANGE_FLAG_DMA_BUF_MAPPING_TYPE_PCIE](#), otherwise 0.

Returns

CUDA_SUCCESS CUDA_ERROR_INVALID_VALUE CUDA_ERROR_NOT_SUPPORTED

Description

Get a handle of the specified type to an address range. The address range must have been obtained by a prior call to either [cuMemAlloc](#) or [cuMemAddressReserve](#). If the address range was obtained via [cuMemAddressReserve](#), it must also be fully mapped via [cuMemMap](#). The address range must have been obtained by a prior call to either [cuMemAllocHost](#) or [cuMemHostAlloc](#) on Tegra.

Users must ensure the `dptr` and `size` are aligned to the host page size.

When requesting

CUmemRangeHandleType::CU_MEM_RANGE_HANDLE_TYPE_DMA_BUF_FD,

users are expected to query for `dma_buf` support for the platform by using

[CU_DEVICE_ATTRIBUTE_DMA_BUF_SUPPORTED](#) device attribute before calling this API.

The `handle` will be interpreted as a pointer to an integer to store the `dma_buf` file descriptor. Users must ensure the entire address range is backed and mapped when the address range is allocated by [cuMemAddressReserve](#). All the physical allocations backing the address range must be resident on the same device and have identical allocation properties. Users are also expected to retrieve a new handle every time the underlying physical allocation(s) corresponding to a previously queried VA range are changed.

For CUmemRangeHandleType::CU_MEM_RANGE_HANDLE_TYPE_DMA_BUF_FD, users may set flags to [CU_MEM_RANGE_FLAG_DMA_BUF_MAPPING_TYPE_PCIE](#). Which when set on a supported platform, will give a DMA_BUF handle mapped via PCIE BAR1 or will return an error otherwise.

CUresult cuMemGetInfo (size_t *free, size_t *total)

Gets free and total memory.

Parameters**free**

- Returned free memory in bytes

total

- Returned total memory in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

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.

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#),
[cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#),
[cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#),
[cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#),
[cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#),
[cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemHostAlloc](#),
[cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#),
[cuMemsetD16](#), [cuMemsetD32](#), [cudaMemGetInfo](#)

CUresult cuMemHostAlloc (void **pp, size_t bytesize, unsigned int Flags)

Allocates page-locked host memory.

Parameters**pp**

- Returned pointer to host memory

bytesize

- Requested allocation size in bytes

Flags

- Flags for allocation request

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Allocates `bytesize` 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 [cuMemcpyHtoD\(\)](#). 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

[CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES](#) is true, [cuMemHostAlloc](#) may not page-lock the allocated memory.

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 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.

- ▶ [CU_MEMHOSTALLOC_PORTABLE](#): The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
- ▶ [CU_MEMHOSTALLOC_DEVICEMAP](#): Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling [cuMemHostGetDevicePointer\(\)](#).
- ▶ [CU_MEMHOSTALLOC_WRITECOMBINED](#): 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 GPU 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.

The [CU_MEMHOSTALLOC_DEVICEMAP](#) flag may be specified on CUDA contexts for devices that do not support mapped pinned memory. The failure is deferred to [cuMemHostGetDevicePointer\(\)](#) because the memory may be mapped into other CUDA contexts via the [CU_MEMHOSTALLOC_PORTABLE](#) flag.

The memory allocated by this function must be freed with [cuMemFreeHost\(\)](#).

Note all host memory allocated using [cuMemHostAlloc\(\)](#) will automatically be immediately accessible to all contexts on all devices which support unified addressing (as may be queried using [CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING](#)). Unless the flag [CU_MEMHOSTALLOC_WRITECOMBINED](#) is specified, the device pointer that may be used to access this host memory from those contexts is always equal to the returned host pointer *pp. If the flag [CU_MEMHOSTALLOC_WRITECOMBINED](#) is specified, then the function [cuMemHostGetDevicePointer\(\)](#) must be used to query the device pointer, even if the context supports unified addressing. See [Unified Addressing](#) for additional details.



Note:

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

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaHostAlloc](#)

CUresult cuMemHostGetDevicePointer (CUdeviceptr *pdptr, void *p, unsigned int Flags)

Passes back device pointer of mapped pinned memory.

Parameters

pdptr

- Returned device pointer

p

- Host pointer

Flags

- Options (must be 0)

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Passes back the device pointer `pdptr` corresponding to the mapped, pinned host buffer `p` allocated by [cuMemHostAlloc](#).

[cuMemHostGetDevicePointer\(\)](#) will fail if the [CU_MEMHOSTALLOC_DEVICEMAP](#) flag was not specified at the time the memory was allocated, or if the function is called on a GPU that does not support mapped pinned memory.

For devices that have a non-zero value for the device attribute [CU_DEVICE_ATTRIBUTE_CAN_USE_HOST_POINTER_FOR_REGISTERED_MEM](#), the memory can also be accessed from the device using the host pointer `p`. The device pointer returned by [cuMemHostGetDevicePointer\(\)](#) may or may not match the original host pointer `p` 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 [cuMemHostGetDevicePointer\(\)](#) will match the original pointer `p`. If any device visible to the application has a zero value for the device attribute, the device pointer returned by [cuMemHostGetDevicePointer\(\)](#) will not match the original host pointer `p`, 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 one 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.

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#), [cudaHostGetDevicePointer](#)

CUresult cuMemHostGetFlags (unsigned int *pFlags, void *p)

Passes back flags that were used for a pinned allocation.

Parameters

pFlags

- Returned flags word

p

- Host pointer

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Passes back the flags `pFlags` that were specified when allocating the pinned host buffer `p` allocated by [cuMemHostAlloc](#).

[cuMemHostGetFlags\(\)](#) will fail if the pointer does not reside in an allocation performed by [cuMemAllocHost\(\)](#) or [cuMemHostAlloc\(\)](#).



Note:

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

See also:

[cuMemAllocHost](#), [cuMemHostAlloc](#), [cudaHostGetFlags](#)

CUresult cuMemHostRegister (void *p, size_t bytesize, unsigned int Flags)

Registers an existing host memory range for use by CUDA.

Parameters

p

- Host pointer to memory to page-lock

bytesize

- Size in bytes of the address range to page-lock

Flags

- Flags for allocation request

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE,
CUDA_ERROR_OUT_OF_MEMORY,
CUDA_ERROR_HOST_MEMORY_ALREADY_REGISTERED,
CUDA_ERROR_NOT_PERMITTED, CUDA_ERROR_NOT_SUPPORTED

Description

Page-locks the memory range specified by `p` and `bytesize` and maps it for the device(s) as specified by `Flags`. This memory range also is added to the same tracking mechanism as [cuMemHostAlloc](#) to automatically accelerate calls to functions such as [cuMemcpyHtoD\(\)](#). 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

CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES

is true, [cuMemHostRegister](#) will not page-lock the memory range specified by `ptr` but only populate unpopulated pages.

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

- ▶ CU_MEMHOSTREGISTER_PORTABLE: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
- ▶ CU_MEMHOSTREGISTER_DEVICEMAP: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling [cuMemHostGetDevicePointer\(\)](#).
- ▶ CU_MEMHOSTREGISTER_IOMEMORY: The pointer is treated as pointing to some I/O memory space, e.g. the PCI Express resource of a 3rd party device.
- ▶ CU_MEMHOSTREGISTER_READ_ONLY: The pointer is treated as pointing to memory that is considered read-only by the device. On platforms without CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES, 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 CU_DEVICE_ATTRIBUTE_READ_ONLY_HOST_REGISTER_SUPPORTED. Using this flag with a current context associated with a device that does not have this attribute set will cause [cuMemHostRegister](#) to error with `CUDA_ERROR_NOT_SUPPORTED`.

All of these flags are orthogonal to one another: a developer may page-lock memory that is portable or mapped with no restrictions.

The CU_MEMHOSTREGISTER_DEVICEMAP flag may be specified on CUDA contexts for devices that do not support mapped pinned memory. The failure is deferred to

[cuMemHostGetDevicePointer\(\)](#) because the memory may be mapped into other CUDA contexts via the [CU_MEMHOSTREGISTER_PORTABLE](#) flag.

For devices that have a non-zero value for the device attribute [CU_DEVICE_ATTRIBUTE_CAN_USE_HOST_POINTER_FOR_REGISTERED_MEM](#), the memory can also be accessed from the device using the host pointer `p`. The device pointer returned by [cuMemHostGetDevicePointer\(\)](#) 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 [cuMemHostGetDevicePointer\(\)](#) 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 [cuMemHostGetDevicePointer\(\)](#) 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 [cuMemHostUnregister\(\)](#).



Note:

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

See also:

[cuMemHostUnregister](#), [cuMemHostGetFlags](#), [cuMemHostGetDevicePointer](#), [cudaHostRegister](#)

CUresult cuMemHostUnregister (void *p)

Unregisters a memory range that was registered with [cuMemHostRegister](#).

Parameters

p

- Host pointer to memory to unregister

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_HOST_MEMORY_NOT_REGISTERED](#),

Description

Unmaps the memory range whose base address is specified by `p`, and makes it pageable again.

The base address must be the same one specified to [cuMemHostRegister\(\)](#).

**Note:**

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

See also:

[cuMemHostRegister](#), [cudaHostUnregister](#)

CUresult cuMemsetD16 (CUdeviceptr dstDevice, unsigned short us, size_t N)

Initializes device memory.

Parameters

dstDevice

- Destination device pointer

us

- Value to set

N

- Number of elements

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Sets the memory range of N 16-bit values to the specified value us. The dstDevice pointer must be two byte aligned.

**Note:**

- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ See also [memset synchronization details](#).

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#),

[cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#),
[cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#),
[cuMemsetD8Async](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#), [cudaMemset](#)

CUresult cuMemsetD16Async (CUdeviceptr dstDevice, unsigned short us, size_t N, CUstream hStream)

Sets device memory.

Parameters

dstDevice

- Destination device pointer

us

- Value to set

N

- Number of elements

hStream

- Stream identifier

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Sets the memory range of N 16-bit values to the specified value us. The dstDevice pointer must be two byte aligned.



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.

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#),
[cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#),
[cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#),
[cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#),
[cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#),
[cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#),
[cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#),

[cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD32](#), [cuMemsetD32Async](#), [cudaMemsetAsync](#)

CUresult cuMemsetD2D16 (CUdeviceptr dstDevice, size_t dstPitch, unsigned short us, size_t Width, size_t Height)

Initializes device memory.

Parameters

dstDevice

- Destination device pointer

dstPitch

- Pitch of destination device pointer(Unused if Height is 1)

us

- Value to set

Width

- Width of row

Height

- Number of rows

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Sets the 2D memory range of Width 16-bit values to the specified value us. Height specifies the number of rows to set, and dstPitch specifies the number of bytes between each row. The dstDevice pointer and dstPitch offset must be two byte aligned. This function performs fastest when the pitch is one that has been passed back by [cuMemAllocPitch\(\)](#).



Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- See also [memset synchronization details](#).

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#),

[cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#), [cudaMemset2D](#)

CUresult cuMemsetD2D16Async (CUdeviceptr dstDevice, size_t dstPitch, unsigned short us, size_t Width, size_t Height, CUstream hStream)

Sets device memory.

Parameters

dstDevice

- Destination device pointer

dstPitch

- Pitch of destination device pointer(Unused if Height is 1)

us

- Value to set

Width

- Width of row

Height

- Number of rows

hStream

- Stream identifier

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Sets the 2D memory range of Width 16-bit values to the specified value us. Height specifies the number of rows to set, and dstPitch specifies the number of bytes between each row. The dstDevice pointer and dstPitch offset must be two byte aligned. This function performs fastest when the pitch is one that has been passed back by [cuMemAllocPitch\(\)](#).



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.

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#), [cudaMemset2DAsync](#)

CUresult cuMemsetD2D32 (CUdeviceptr dstDevice, size_t dstPitch, unsigned int ui, size_t Width, size_t Height)

Initializes device memory.

Parameters

dstDevice

- Destination device pointer

dstPitch

- Pitch of destination device pointer(Unused if Height is 1)

ui

- Value to set

Width

- Width of row

Height

- Number of rows

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Sets the 2D memory range of Width 32-bit values to the specified value ui. Height specifies the number of rows to set, and dstPitch specifies the number of bytes between each row. The dstDevice pointer and dstPitch offset must be four byte aligned. This function performs fastest when the pitch is one that has been passed back by [cuMemAllocPitch\(\)](#).

**Note:**

- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ See also [memset synchronization details](#).

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#), [cudaMemset2D](#)

CUresult cuMemsetD2D32Async (CUdeviceptr dstDevice, size_t dstPitch, unsigned int ui, size_t Width, size_t Height, CUstream hStream)

Sets device memory.

Parameters

dstDevice

- Destination device pointer

dstPitch

- Pitch of destination device pointer(Unused if Height is 1)

ui

- Value to set

Width

- Width of row

Height

- Number of rows

hStream

- Stream identifier

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Sets the 2D memory range of `Width` 32-bit values to the specified value `ui`. `Height` specifies the number of rows to set, and `dstPitch` specifies the number of bytes between each row. The `dstDevice` pointer and `dstPitch` offset must be four byte aligned. This function performs fastest when the pitch is one that has been passed back by [cuMemAllocPitch\(\)](#).



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.

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#), [cudaMemset2DAsync](#)

CUresult cuMemsetD2D8 (CUdeviceptr dstDevice, size_t dstPitch, unsigned char uc, size_t Width, size_t Height)

Initializes device memory.

Parameters

dstDevice

- Destination device pointer

dstPitch

- Pitch of destination device pointer (Unused if `Height` is 1)

uc

- Value to set

Width

- Width of row

Height

- Number of rows

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Sets the 2D memory range of Width 8-bit values to the specified value `uc`. Height specifies the number of rows to set, and `dstPitch` specifies the number of bytes between each row. This function performs fastest when the pitch is one that has been passed back by [cuMemAllocPitch\(\)](#).



Note:

- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ See also [memset synchronization details](#).

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#), [cudaMemset2D](#)

CUresult cuMemsetD2D8Async (CUdeviceptr dstDevice, size_t dstPitch, unsigned char uc, size_t Width, size_t Height, CUstream hStream)

Sets device memory.

Parameters

dstDevice

- Destination device pointer

dstPitch

- Pitch of destination device pointer(Unused if Height is 1)

uc

- Value to set

Width

- Width of row

Height

- Number of rows

hStream

- Stream identifier

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Sets the 2D memory range of `Width` 8-bit values to the specified value `uc`. `Height` specifies the number of rows to set, and `dstPitch` specifies the number of bytes between each row. This function performs fastest when the pitch is one that has been passed back by [cuMemAllocPitch\(\)](#).

**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.

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#), [cudaMemset2DAsync](#)

CUresult cuMemsetD32 (CUdeviceptr dstDevice, unsigned int ui, size_t N)

Initializes device memory.

Parameters

dstDevice

- Destination device pointer

ui

- Value to set

N

- Number of elements

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Sets the memory range of N 32-bit values to the specified value ui. The dstDevice pointer must be four byte aligned.



Note:

- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ See also [memset synchronization details](#).

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32Async](#), [cudaMemset](#)

CUresult cuMemsetD32Async (CUdeviceptr dstDevice, unsigned int ui, size_t N, CUstream hStream)

Sets device memory.

Parameters

dstDevice

- Destination device pointer

ui

- Value to set

N

- Number of elements

hStream

- Stream identifier

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Sets the memory range of N 32-bit values to the specified value ui. The dstDevice pointer must be four byte aligned.



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.

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cudaMemsetAsync](#)

CUresult cuMemsetD8 (CUdeviceptr dstDevice, unsigned char uc, size_t N)

Initializes device memory.

Parameters

dstDevice

- Destination device pointer

uc

- Value to set

N

- Number of elements

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Sets the memory range of N 8-bit values to the specified value `uc`.



Note:

- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ See also [memset synchronization details](#).

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#), [cudaMemset](#)

CUresult cuMemsetD8Async (CUdeviceptr dstDevice, unsigned char uc, size_t N, CUstream hStream)

Sets device memory.

Parameters

dstDevice

- Destination device pointer

uc

- Value to set

N

- Number of elements

hStream

- Stream identifier

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Sets the memory range of N 8-bit values to the specified value uc.



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.

See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#), [cudaMemsetAsync](#)

CUresult cuMipmappedArrayCreate (CUmipmappedArray *pHandle, const CUDA_ARRAY3D_DESCRIPTOR *pMipmappedArrayDesc, unsigned int numMipmapLevels)

Creates a CUDA mipmapped array.

Parameters

pHandle

- Returned mipmapped array

pMipmappedArrayDesc

- mipmapped array descriptor

numMipmapLevels

- Number of mipmap levels

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_UNKNOWN](#)

Description

Creates a CUDA mipmapped array according to the [CUDA_ARRAY3D_DESCRIPTOR](#) structure `pMipmappedArrayDesc` and returns a handle to the new CUDA mipmapped array in `*pHandle`. `numMipmapLevels` specifies the number of mipmap levels to be allocated. This value is clamped to the range $[1, 1 + \text{floor}(\log_2(\max(\text{width}, \text{height}, \text{depth})))]$.

The [CUDA_ARRAY3D_DESCRIPTOR](#) is defined as:

```
typedef struct {
    unsigned int Width;
    unsigned int Height;
    unsigned int Depth;
    CUarray\_format Format;
    unsigned int NumChannels;
    unsigned int Flags;
} CUDA\_ARRAY3D\_DESCRIPTOR;
```

where:

- ▶ Width, Height, and Depth are the width, height, and depth of the CUDA array (in elements); the following types of CUDA arrays can be allocated:
 - ▶ A 1D mipmapped array is allocated if Height and Depth extents are both zero.
 - ▶ A 2D mipmapped array is allocated if only 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 Height is zero and the [CUDA_ARRAY3D_LAYERED](#) flag is set. Each layer is a 1D 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 [CUDA_ARRAY3D_LAYERED](#) flag is set. Each layer is a 2D 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 [CUDA_ARRAY3D_CUBEMAP](#) 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 [CUarray_cubemap_face](#).
- ▶ A cubemap layered CUDA mipmapped array is allocated if all three extents are non-zero, and both, [CUDA_ARRAY3D_CUBEMAP](#) and [CUDA_ARRAY3D_LAYERED](#) 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.
- ▶ Format specifies the format of the elements; [CUarray_format](#) is defined as:

```
typedef enum CUarray_format_enum {
    CU_AD_FORMAT_UNSIGNED_INT8 = 0x01,
    CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
    CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
    CU_AD_FORMAT_SIGNED_INT8 = 0x08,
    CU_AD_FORMAT_SIGNED_INT16 = 0x09,
    CU_AD_FORMAT_SIGNED_INT32 = 0x0a,
    CU_AD_FORMAT_HALF = 0x10,
    CU_AD_FORMAT_FLOAT = 0x20,
    CU_AD_FORMAT_NV12 = 0xb0,
    CU_AD_FORMAT_UNORM_INT8X1 = 0xc0,
    CU_AD_FORMAT_UNORM_INT8X2 = 0xc1,
    CU_AD_FORMAT_UNORM_INT8X4 = 0xc2,
    CU_AD_FORMAT_UNORM_INT16X1 = 0xc3,
    CU_AD_FORMAT_UNORM_INT16X2 = 0xc4,
    CU_AD_FORMAT_UNORM_INT16X4 = 0xc5,
    CU_AD_FORMAT_SNORM_INT8X1 = 0xc6,
    CU_AD_FORMAT_SNORM_INT8X2 = 0xc7,
    CU_AD_FORMAT_SNORM_INT8X4 = 0xc8,
    CU_AD_FORMAT_SNORM_INT16X1 = 0xc9,
    CU_AD_FORMAT_SNORM_INT16X2 = 0xca,
    CU_AD_FORMAT_SNORM_INT16X4 = 0xcb,
    CU_AD_FORMAT_BC1_UNORM = 0x91,
    CU_AD_FORMAT_BC1_UNORM_SRGB = 0x92,
    CU_AD_FORMAT_BC2_UNORM = 0x93,
    CU_AD_FORMAT_BC2_UNORM_SRGB = 0x94,
    CU_AD_FORMAT_BC3_UNORM = 0x95,
    CU_AD_FORMAT_BC3_UNORM_SRGB = 0x96,
    CU_AD_FORMAT_BC4_UNORM = 0x97,
    CU_AD_FORMAT_BC4_SNORM = 0x98,
    CU_AD_FORMAT_BC5_UNORM = 0x99,
    CU_AD_FORMAT_BC5_SNORM = 0x9a,
    CU_AD_FORMAT_BC6H_UF16 = 0x9b,
    CU_AD_FORMAT_BC6H_SF16 = 0x9c,
    CU_AD_FORMAT_BC7_UNORM = 0x9d,
    CU_AD_FORMAT_BC7_UNORM_SRGB = 0x9e,
    CU_AD_FORMAT_P010 = 0x9f,
    CU_AD_FORMAT_P016 = 0xa1,
    CU_AD_FORMAT_NV16 = 0xa2,
    CU_AD_FORMAT_P210 = 0xa3,
    CU_AD_FORMAT_P216 = 0xa4,
    CU_AD_FORMAT_YUY2 = 0xa5,
```

```

    CU_AD_FORMAT_Y210 = 0xa6,
    CU_AD_FORMAT_Y216 = 0xa7,
    CU_AD_FORMAT_AYUV = 0xa8,
    CU_AD_FORMAT_Y410 = 0xa9,
    CU_AD_FORMAT_Y416 = 0xb1,
    CU_AD_FORMAT_Y444_PLANAR8 = 0xb2,
    CU_AD_FORMAT_Y444_PLANAR10 = 0xb3,
    CU_AD_FORMAT_YUV444_8bit_SemiPlanar = 0xb4,
    CU_AD_FORMAT_YUV444_16bit_SemiPlanar = 0xb5,
    CU_AD_FORMAT_UNORM_INT_101010_2 = 0x50,
} CUarray_format;

```

- ▶ NumChannels specifies the number of packed components per CUDA array element; it may be 1, 2, or 4;
- ▶ Flags may be set to
 - ▶ [CUDA_ARRAY3D_LAYERED](#) to enable creation of layered CUDA mipmapped arrays. If this flag is set, Depth specifies the number of layers, not the depth of a 3D array.
 - ▶ [CUDA_ARRAY3D_SURFACE_LDST](#) to enable surface references to be bound to individual mipmap levels of the CUDA mipmapped array. If this flag is not set, [cuSurfRefSetArray](#) will fail when attempting to bind a mipmap level of the CUDA mipmapped array to a surface reference.
 - ▶ [CUDA_ARRAY3D_CUBEMAP](#) to enable creation of mipmapped cubemaps. If this flag is set, Width must be equal to Height, and Depth must be six. If the [CUDA_ARRAY3D_LAYERED](#) flag is also set, then Depth must be a multiple of six.
 - ▶ [CUDA_ARRAY3D_TEXTURE_GATHER](#) to indicate that the CUDA mipmapped array will be used for texture gather. Texture gather can only be performed on 2D CUDA mipmapped arrays.

Width, Height and Depth must meet certain size requirements as listed in the following table. All values are specified in elements. Note that for brevity's sake, the full name of the device attribute is not specified. For ex., TEXTURE1D_MIPMAPPED_WIDTH refers to the device attribute [CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_MIPMAPPED_WIDTH](#).

CUDA array type	Valid extents that must always be met {(width range in elements), (height range), (depth range)}	Valid extents with CUDA_ARRAY3D_SURFACE_LDST set {(width range in elements), (height range), (depth range)}
1D	{ (1,TEXTURE1D_MIPMAPPED_WIDTH), 0, 0 }	{ (1,SURFACE1D_WIDTH), 0, 0 }
2D	{ (1,TEXTURE2D_MIPMAPPED_WIDTH), (1,TEXTURE2D_MIPMAPPED_HEIGHT), 0 }	{ (1,SURFACE2D_WIDTH), (1,SURFACE2D_HEIGHT), 0 }
3D	{ (1,TEXTURE3D_WIDTH), (1,TEXTURE3D_HEIGHT), (1,TEXTURE3D_DEPTH) } OR	{ (1,SURFACE3D_WIDTH), (1,SURFACE3D_HEIGHT), (1,SURFACE3D_DEPTH) }

	<pre>{ (1,TEXTURE3D_WIDTH_ALTERNATE), (1,TEXTURE3D_HEIGHT_ALTERNATE), (1,TEXTURE3D_DEPTH_ALTERNATE) }</pre>
1D Layered	<pre>{ (1,TEXTURE1D_LAYERED_WIDTH), (1,SURFACE1D_LAYERED_WIDTH), 0, 0, (1,TEXTURE1D_LAYERED_LAYERS), (1,SURFACE1D_LAYERED_LAYERS) }</pre>
2D Layered	<pre>{ (1,TEXTURE2D_LAYERED_WIDTH), (1,SURFACE2D_LAYERED_WIDTH), (1,TEXTURE2D_LAYERED_HEIGHT), (1,SURFACE2D_LAYERED_HEIGHT), (1,TEXTURE2D_LAYERED_LAYERS), (1,SURFACE2D_LAYERED_LAYERS) }</pre>
Cubemap	<pre>{ (1,TEXTURECUBEMAP_WIDTH), (1,SURFACECUBEMAP_WIDTH), (1,TEXTURECUBEMAP_WIDTH), (1,SURFACECUBEMAP_WIDTH), 6 } 6 }</pre>
Cubemap Layered	<pre>{ (1,TEXTURECUBEMAP_LAYERED_WIDTH), (1,SURFACECUBEMAP_LAYERED_WIDTH), (1,TEXTURECUBEMAP_LAYERED_WIDTH), (1,SURFACECUBEMAP_LAYERED_WIDTH), (1,TEXTURECUBEMAP_LAYERED_LAYERS), (1,SURFACECUBEMAP_LAYERED_LAYERS) }</pre>

**Note:**

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

See also:

[cuMipmappedArrayDestroy](#), [cuMipmappedArrayGetLevel](#), [cuArrayCreate](#),
[cudaMallocMipmappedArray](#)

CUresult cuMipmappedArrayDestroy (CUmipmappedArray hMipmappedArray)

Destroys a CUDA mipmapped array.

Parameters

hMipmappedArray

- Mipmapped array to destroy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_ARRAY_IS_MAPPED](#), [CUDA_ERROR_CONTEXT_IS_DESTROYED](#)

Description

Destroys the CUDA mipmapped array `hMipmappedArray`.



Note:

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

See also:

[cuMipmappedArrayCreate](#), [cuMipmappedArrayGetLevel](#), [cuArrayCreate](#), [cudaFreeMipmappedArray](#)

CUresult cuMipmappedArrayGetLevel (CUarray *pLevelArray, CUmipmappedArray hMipmappedArray, unsigned int level)

Gets a mipmap level of a CUDA mipmapped array.

Parameters

pLevelArray

- Returned mipmap level CUDA array

hMipmappedArray

- CUDA mipmapped array

level

- Mipmap level

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#)

Description

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

If `level` is greater than the maximum number of levels in this mipmapped array, [CUDA_ERROR_INVALID_VALUE](#) is returned.



Note:

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

See also:

[cuMipmappedArrayCreate](#), [cuMipmappedArrayDestroy](#), [cuArrayCreate](#),
[cudaGetMipmappedArrayLevel](#)

CUresult cuMipmappedArrayGetMemoryRequirements (CUDA_ARRAY_MEMORY_REQUIREMENTS *memoryRequirements, CUmipmappedArray mipmap, CUdevice device)

Returns the memory requirements of a CUDA mipmapped array.

Parameters

memoryRequirements

- Pointer to CUDA_ARRAY_MEMORY_REQUIREMENTS

mipmap

- CUDA mipmapped array to get the memory requirements of

device

- Device to get the memory requirements for

Returns

[CUDA_SUCCESS](#) [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns the memory requirements of a CUDA mipmapped array in `memoryRequirements`. If the CUDA mipmapped array is not allocated with flag [CUDA_ARRAY3D_DEFERRED_MAPPING](#), [CUDA_ERROR_INVALID_VALUE](#) will be returned.

The returned value in [CUDA_ARRAY_MEMORY_REQUIREMENTS::size](#) represents the total size of the CUDA mipmapped array. The returned value in [CUDA_ARRAY_MEMORY_REQUIREMENTS::alignment](#) represents the alignment necessary for mapping the CUDA mipmapped array.

See also:

[cuArrayGetMemoryRequirements](#), [cuMemMapArrayAsync](#)

CUresult cuMipmappedArrayGetSparseProperties (CUDA_ARRAY_SPARSE_PROPERTIES *sparseProperties, CUmipmappedArray mipmap)

Returns the layout properties of a sparse CUDA mipmapped array.

Parameters

sparseProperties

- Pointer to CUDA_ARRAY_SPARSE_PROPERTIES

mipmap

- CUDA mipmapped array to get the sparse properties of

Returns

CUDA_SUCCESS CUDA_ERROR_INVALID_VALUE

Description

Returns the sparse array layout properties in `sparseProperties`. If the CUDA mipmapped array is not allocated with flag CUDA_ARRAY3D_SPARSE, CUDA_ERROR_INVALID_VALUE will be returned.

For non-layered CUDA mipmapped arrays, CUDA_ARRAY_SPARSE_PROPERTIES::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 CUDA_ARRAY_SPARSE_PROPERTIES::flags contains CUDA_ARRAY_SPARSE_PROPERTIES_SINGLE_MIPTAIL, then CUDA_ARRAY_SPARSE_PROPERTIES::mipTailSize specifies the size of the mip tail of all layers combined. Otherwise, CUDA_ARRAY_SPARSE_PROPERTIES::mipTailSize specifies mip tail size per layer. The returned value of CUDA_ARRAY_SPARSE_PROPERTIES::mipTailFirstLevel is valid only if CUDA_ARRAY_SPARSE_PROPERTIES::mipTailSize is non-zero.

See also:

cuArrayGetSparseProperties, cuMemMapArrayAsync

6.14. Virtual Memory Management

This section describes the virtual memory management functions of the low-level CUDA driver application programming interface.

CUresult cuMemAddressFree (CUdeviceptr ptr, size_t size)

Free an address range reservation.

Parameters

ptr

- Starting address of the virtual address range to free

size

- Size of the virtual address region to free

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED,
CUDA_ERROR_NOT_SUPPORTED

Description

Frees a virtual address range reserved by cuMemAddressReserve. The size must match what was given to memAddressReserve and the ptr given must match what was returned from memAddressReserve.

See also:

[cuMemAddressReserve](#)

CUresult cuMemAddressReserve (CUdeviceptr *ptr, size_t size, size_t alignment, CUdeviceptr addr, unsigned long long flags)

Allocate an address range reservation.

Parameters

ptr

- Resulting pointer to start of virtual address range allocated

size

- Size of the reserved virtual address range requested

alignment

- Alignment of the reserved virtual address range requested

addr

- Fixed starting address range requested

flags

- Currently unused, must be zero

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_PERMITTED, CUDA_ERROR_NOT_SUPPORTED

Description

Reserves a virtual address range based on the given parameters, giving the starting address of the range in `ptr`. This API requires a system that supports UVA. The size and address parameters must be a multiple of the host page size and the alignment must be a power of two or zero for default alignment.

See also:

[cuMemAddressFree](#)

CUresult cuMemCreate (CUmemGenericAllocationHandle *handle, size_t size, const CUmemAllocationProp *prop, unsigned long long flags)

Create a CUDA memory handle representing a memory allocation of a given size described by the given properties.

Parameters

handle

- Value of handle returned. All operations on this allocation are to be performed using this handle.

size

- Size of the allocation requested

prop

- Properties of the allocation to create.

flags

- flags for future use, must be zero now.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY,
CUDA_ERROR_INVALID_DEVICE, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED,
CUDA_ERROR_NOT_SUPPORTED

Description

This creates a memory allocation on the target device specified through the `prop` structure. The created allocation will not have any device or host mappings. The generic memory handle for the allocation can be mapped to the address space of calling

process via [cuMemMap](#). This handle cannot be transmitted directly to other processes (see [cuMemExportToShareableHandle](#)). On Windows, the caller must also pass an LPSECURITYATTRIBUTE in `prop` to be associated with this handle which limits or allows access to this handle for a recipient process (see [CUmemAllocationProp::win32HandleMetaData](#) for more). The `size` of this allocation must be a multiple of the value given via [cuMemGetAllocationGranularity](#) with the [CU_MEM_ALLOC_GRANULARITY_MINIMUM](#) flag. To create a CPU allocation targeting a specific host NUMA node, applications must set `CUmemAllocationProp::CUmemLocation::type` to [CU_MEM_LOCATION_TYPE_HOST_NUMA](#) and `CUmemAllocationProp::CUmemLocation::id` must specify the NUMA ID of the CPU. On systems where NUMA is not available `CUmemAllocationProp::CUmemLocation::id` must be set to 0. Specifying [CU_MEM_LOCATION_TYPE_HOST_NUMA_CURRENT](#) or [CU_MEM_LOCATION_TYPE_HOST](#) as the `CUmemLocation::type` will result in [CUDA_ERROR_INVALID_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``

If `CUmemAllocationProp::allocFlags::usage` contains [CU_MEM_CREATE_USAGE_TILE_POOL](#) flag then the memory allocation is intended only to be used as backing tile pool for sparse CUDA arrays and sparse CUDA mipmapped arrays. (see [cuMemMapArrayAsync](#)).



Note:

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

See also:

[cuMemRelease](#), [cuMemExportToShareableHandle](#), [cuMemImportFromShareableHandle](#)

CUresult cuMemExportToShareableHandle (void *shareableHandle, CUmemGenericAllocationHandle handle, CUmemAllocationHandleType handleType, unsigned long long flags)

Exports an allocation to a requested shareable handle type.

Parameters

shareableHandle

- Pointer to the location in which to store the requested handle type

handle

- CUDA handle for the memory allocation

handleType

- Type of shareable handle requested (defines type and size of the `shareableHandle` output parameter)

flags

- Reserved, must be zero

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED, CUDA_ERROR_NOT_SUPPORTED

Description

Given a CUDA memory handle, create a shareable memory allocation handle that can be used to share the memory with other processes. The recipient process can convert the shareable handle back into a CUDA memory handle using [cuMemImportFromShareableHandle](#) and map it with [cuMemMap](#). The implementation of what this handle is and how it can be transferred is defined by the requested handle type in `handleType`

Once all shareable handles are closed and the allocation is released, the allocated memory referenced will be released back to the OS and uses of the CUDA handle afterward will lead to undefined behavior.

This API can also be used in conjunction with other APIs (e.g. Vulkan, OpenGL) that support importing memory from the shareable type

See also:

[cuMemImportFromShareableHandle](#)

CUresult cuMemGetAccess (unsigned long long *flags, const CUmемLocation *location, CUdeviceptr ptr)

Get the access flags set for the given location and ptr.

Parameters

flags

- Flags set for this location

location

- Location in which to check the flags for

ptr

- Address in which to check the access flags for

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED, CUDA_ERROR_NOT_SUPPORTED

Description

See also:

[cuMemSetAccess](#)

CUresult cuMemGetAllocationGranularity (size_t *granularity, const CUmемAllocationProp *prop, CUmемAllocationGranularity_flags option)

Calculates either the minimal or recommended granularity.

Parameters

granularity

Returned granularity.

prop

Property for which to determine the granularity for

option

Determines which granularity to return

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED,
CUDA_ERROR_NOT_SUPPORTED

Description

Calculates either the minimal or recommended granularity for a given allocation specification and returns it in granularity. This granularity can be used as a multiple for alignment, size, or address mapping.

See also:

[cuMemCreate](#), [cuMemMap](#)

CUresult cuMemGetAllocationPropertiesFromHandle (CUmemAllocationProp *prop, CUmemGenericAllocationHandle handle)

Retrieve the contents of the property structure defining properties for this handle.

Parameters

prop

- Pointer to a properties structure which will hold the information about this handle

handle

- Handle which to perform the query on

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED,
CUDA_ERROR_NOT_SUPPORTED

Description

See also:

[cuMemCreate](#), [cuMemImportFromShareableHandle](#)

CUresult cuMemImportFromShareableHandle (CUmemGenericAllocationHandle *handle, void *osHandle, CUmemAllocationHandleType shHandleType)

Imports an allocation from a requested shareable handle type.

Parameters

handle

- CUDA Memory handle for the memory allocation.

osHandle

- Shareable Handle representing the memory allocation that is to be imported.

shHandleType

- handle type of the exported handle [CUmemAllocationHandleType](#).

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_PERMITTED](#),
[CUDA_ERROR_NOT_SUPPORTED](#)

Description

If the current process cannot support the memory described by this shareable handle, this API will error as [CUDA_ERROR_NOT_SUPPORTED](#).

If `shHandleType` is [CU_MEM_HANDLE_TYPE_FABRIC](#) and the importer process has not been granted access to the same IMEX channel as the exporter process, this API will error as [CUDA_ERROR_NOT_PERMITTED](#).



Note:

Importing shareable handles exported from some graphics APIs(Vulkan, OpenGL, etc) created on devices under an SLI group may not be supported, and thus this API will return [CUDA_ERROR_NOT_SUPPORTED](#). There is no guarantee that the contents of `handle` will be the same CUDA memory handle for the same given OS shareable handle, or the same underlying allocation.

See also:

[cuMemExportToShareableHandle](#), [cuMemMap](#), [cuMemRelease](#)

CUresult cuMemMap (CUdeviceptr ptr, size_t size, size_t offset, CUmemGenericAllocationHandle handle, unsigned long long flags)

Maps an allocation handle to a reserved virtual address range.

Parameters

ptr

- Address where memory will be mapped.

size

- Size of the memory mapping.

offset

handle from which to start mapping Note: currently must be zero.

- Offset into the memory represented by

handle

- Handle to a shareable memory

flags

- flags for future use, must be zero now.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_PERMITTED](#), [CUDA_ERROR_NOT_SUPPORTED](#)

- handle from which to start mapping
- Note: currently must be zero.

Description

Maps bytes of memory represented by `handle` starting from byte `offset` to `size` to address range `[addr, addr + size]`. This range must be an address reservation previously reserved with [cuMemAddressReserve](#), and `offset + size` must be less than the size of the memory allocation. Both `ptr`, `size`, and `offset` must be a multiple of the value given via [cuMemGetAllocationGranularity](#) with the [CU_MEM_ALLOC_GRANULARITY_MINIMUM](#) flag. If `handle` represents a multicast object, `ptr`, `size` and `offset` must be aligned to the value returned by [cuMulticastGetGranularity](#) with the flag [CU_MULTICAST_MINIMUM_GRANULARITY](#). For best performance however, it is recommended that `ptr`, `size` and `offset` be aligned to the value returned by [cuMulticastGetGranularity](#) with the flag [CU_MULTICAST_RECOMMENDED_GRANULARITY](#).

Please note calling [cuMemMap](#) does not make the address accessible, the caller needs to update accessibility of a contiguous mapped VA range by calling [cuMemSetAccess](#).

Once a recipient process obtains a shareable memory handle from [cuMemImportFromShareableHandle](#), the process must use [cuMemMap](#) to map the memory into its address ranges before setting accessibility with [cuMemSetAccess](#).

[cuMemMap](#) can only create mappings on VA range reservations that are not currently mapped.



Note:

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

See also:

[cuMemUnmap](#), [cuMemSetAccess](#), [cuMemCreate](#), [cuMemAddressReserve](#),
[cuMemImportFromShareableHandle](#)

CUresult cuMemMapArrayAsync (CUarrayMapInfo *mapInfoList, unsigned int count, CUstream hStream)

Maps or unmaps subregions of sparse CUDA arrays and sparse CUDA mipmapped arrays.

Parameters

mapInfoList

- List of CUarrayMapInfo

count

- Count of CUarrayMapInfo in mapInfoList

hStream

- Stream identifier for the stream to use for map or unmap operations

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#)

Description

Performs map or unmap operations on subregions of sparse CUDA arrays and sparse CUDA mipmapped arrays. Each operation is specified by a CUarrayMapInfo entry in the mapInfoList array of size count. The structure CUarrayMapInfo is defined as follow:

```
typedef struct CUarrayMapInfo_st {
    CUresourceType resourceType;
    union {
        CUmipmappedArray mipmap;
        CUarray array;
    } resource;

    CUarraySparseSubresourceType subresourceType;
    union {
        struct {
            unsigned int level;
            unsigned int layer;
        };
    };
};
```

```

        unsigned int offsetX;
        unsigned int offsetY;
        unsigned int offsetZ;
        unsigned int extentWidth;
        unsigned int extentHeight;
        unsigned int extentDepth;
    } sparseLevel;
    struct {
        unsigned int layer;
        unsigned long long offset;
        unsigned long long size;
    } mipTail;
} subresource;

CUmemOperationType memOperationType;

CUmemHandleType memHandleType;
union {
    CUmemGenericAllocationHandle memHandle;
} memHandle;

unsigned long long offset;
unsigned int deviceBitMask;
unsigned int flags;
unsigned int reserved[2];
} CUarrayMapInfo;

```

where [CUarrayMapInfo::resourceType](#) specifies the type of resource to be operated on. If [CUarrayMapInfo::resourceType](#) is set to [CUresourcetype::CU_RESOURCE_TYPE_ARRAY](#) then [CUarrayMapInfo::resource::array](#) must be set to a valid sparse CUDA array handle. The CUDA array must be either a 2D, 2D layered or 3D CUDA array and must have been allocated using [cuArrayCreate](#) or [cuArray3DCreate](#) with the flag [CUDA_ARRAY3D_SPARSE](#) or [CUDA_ARRAY3D_DEFERRED_MAPPING](#). For CUDA arrays obtained using [cuMipmappedArrayGetLevel](#), [CUDA_ERROR_INVALID_VALUE](#) will be returned. If [CUarrayMapInfo::resourceType](#) is set to [CUresourcetype::CU_RESOURCE_TYPE_MIPMAPPED_ARRAY](#) then [CUarrayMapInfo::resource::mipmap](#) must be set to a valid sparse CUDA mipmapped array handle. The CUDA mipmapped array must be either a 2D, 2D layered or 3D CUDA mipmapped array and must have been allocated using [cuMipmappedArrayCreate](#) with the flag [CUDA_ARRAY3D_SPARSE](#) or [CUDA_ARRAY3D_DEFERRED_MAPPING](#).

[CUarrayMapInfo::subresourceType](#) specifies the type of subresource within the resource.

[CUarraySparseSubresourceType_enum](#) is defined as:

```

typedef enum CUarraySparseSubresourceType_enum {
    CU_ARRAY_SPARSE_SUBRESOURCE_TYPE_SPARSE_LEVEL = 0,
    CU_ARRAY_SPARSE_SUBRESOURCE_TYPE_MIPTAIL = 1
} CUarraySparseSubresourceType;

```

where

[CUarraySparseSubresourceType::CU_ARRAY_SPARSE_SUBRESOURCE_TYPE_SPARSE_LEVEL](#) indicates a sparse-miplevel which spans at least one tile in every dimension. The remaining miplevels which are too small to span at least one tile in any dimension constitute the mip tail region as indicated by [CUarraySparseSubresourceType::CU_ARRAY_SPARSE_SUBRESOURCE_TYPE_MIPTAIL](#) subresource type.

If [CUarrayMapInfo::subresourceType](#) is set to

[CUarraySparseSubresourceType::CU_ARRAY_SPARSE_SUBRESOURCE_TYPE_SPARSE_LEVEL](#) then [CUarrayMapInfo::subresource::sparseLevel](#) struct must contain valid array subregion offsets and extents. The [CUarrayMapInfo::subresource::sparseLevel::offsetX](#), [CUarrayMapInfo::subresource::sparseLevel::offsetY](#) and [CUarrayMapInfo::subresource::sparseLevel::offsetZ](#) must specify valid X, Y and Z offsets respectively. The [CUarrayMapInfo::subresource::sparseLevel::extentWidth](#), [CUarrayMapInfo::subresource::sparseLevel::extentHeight](#) and [CUarrayMapInfo::subresource::sparseLevel::extentDepth](#) must specify valid width, height and depth extents respectively. These offsets and extents must be aligned to the corresponding tile dimension. For CUDA mipmapped arrays [CUarrayMapInfo::subresource::sparseLevel::level](#) must specify a valid mip level index. Otherwise, must be zero. For layered CUDA arrays and layered CUDA mipmapped arrays [CUarrayMapInfo::subresource::sparseLevel::layer](#) must specify a valid layer index. Otherwise, must be zero. [CUarrayMapInfo::subresource::sparseLevel::offsetZ](#) must be zero and [CUarrayMapInfo::subresource::sparseLevel::extentDepth](#) must be set to 1 for 2D and 2D layered CUDA arrays and CUDA mipmapped arrays. Tile extents can be obtained by calling [cuArrayGetSparseProperties](#) and [cuMipmappedArrayGetSparseProperties](#)

If [CUarrayMapInfo::subresourceType](#) is set to

[CUarraySparseSubresourceType::CU_ARRAY_SPARSE_SUBRESOURCE_TYPE_MIPTAIL](#) then [CUarrayMapInfo::subresource::mipTail](#) struct must contain valid mip tail offset in [CUarrayMapInfo::subresource::mipTail::offset](#) and size in [CUarrayMapInfo::subresource::mipTail::size](#). Both, mip tail offset and mip tail size must be aligned to the tile size. For layered CUDA mipmapped arrays which don't have the flag [CU_ARRAY_SPARSE_PROPERTIES_SINGLE_MIPTAIL](#) set in [CUDA_ARRAY_SPARSE_PROPERTIES::flags](#) as returned by [cuMipmappedArrayGetSparseProperties](#), [CUarrayMapInfo::subresource::mipTail::layer](#) must specify a valid layer index. Otherwise, must be zero.

If [CUarrayMapInfo::resource::array](#) or [CUarrayMapInfo::resource::mipmap](#) was created with [CUDA_ARRAY3D_DEFERRED_MAPPING](#) flag set the [CUarrayMapInfo::subresourceType](#) and the contents of [CUarrayMapInfo::subresource](#) will be ignored.

[CUarrayMapInfo::memOperationType](#) specifies the type of operation. [CUmemOperationType](#) is defined as:

```
typedef enum CUmemOperationType_enum {
    CU_MEM_OPERATION_TYPE_MAP = 1,
    CU_MEM_OPERATION_TYPE_UNMAP = 2
} CUmemOperationType;
```

If [CUarrayMapInfo::memOperationType](#) is set to

[CUmemOperationType::CU_MEM_OPERATION_TYPE_MAP](#) then the subresource will be mapped onto the tile pool memory specified by [CUarrayMapInfo::memHandle](#) at offset [CUarrayMapInfo::offset](#). The tile pool allocation has to be created by specifying the [CU_MEM_CREATE_USAGE_TILE_POOL](#) flag when calling [cuMemCreate](#). Also, [CUarrayMapInfo::memHandleType](#) must be set to [CUmemHandleType::CU_MEM_HANDLE_TYPE_GENERIC](#).

If [CUarrayMapInfo::memOperationType](#) is set to `CUmемOperationType::CU_MEM_OPERATION_TYPE_UNMAP` then an unmapping operation is performed. `CUarrayMapInfo::memHandle` must be `NULL`.

[CUarrayMapInfo::deviceBitMask](#) specifies the list of devices that must map or unmap physical memory. Currently, this mask must have exactly one bit set, and the corresponding device must match the device associated with the stream. If [CUarrayMapInfo::memOperationType](#) is set to `CUmемOperationType::CU_MEM_OPERATION_TYPE_MAP`, the device must also match the device associated with the tile pool memory allocation as specified by `CUarrayMapInfo::memHandle`.

[CUarrayMapInfo::flags](#) and [CUarrayMapInfo::reserved\[\]](#) are unused and must be set to zero.

See also:

[cuMipmappedArrayCreate](#), [cuArrayCreate](#), [cuArray3DCreate](#), [cuMemCreate](#),
[cuArrayGetSparseProperties](#), [cuMipmappedArrayGetSparseProperties](#)

CUresult cuMemRelease (CUmemGenericAllocationHandle handle)

Release a memory handle representing a memory allocation which was previously allocated through `cuMemCreate`.

Parameters

handle

Value of handle which was returned previously by `cuMemCreate`.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_PERMITTED](#),
[CUDA_ERROR_NOT_SUPPORTED](#)

Description

Frees the memory that was allocated on a device through `cuMemCreate`.

The memory allocation will be freed when all outstanding mappings to the memory are unmapped and when all outstanding references to the handle (including its shareable counterparts) are also released. The generic memory handle can be freed when there are still outstanding mappings made with this handle. Each time a recipient process imports a shareable handle, it needs to pair it with [cuMemRelease](#) for the handle to be freed. If `handle` is not a valid handle the behavior is undefined.



Note:

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

See also:

[cuMemCreate](#)

CUresult cuMemRetainAllocationHandle (CUmemGenericAllocationHandle *handle, void *addr)

Given an address `addr`, returns the allocation handle of the backing memory allocation.

Parameters

handle

CUDA Memory handle for the backing memory allocation.

addr

Memory address to query, that has been mapped previously.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_PERMITTED](#),
[CUDA_ERROR_NOT_SUPPORTED](#)

Description

The handle is guaranteed to be the same handle value used to map the memory. If the address requested is not mapped, the function will fail. The returned handle must be released with corresponding number of calls to [cuMemRelease](#).



Note:

The address `addr`, can be any address in a range previously mapped by [cuMemMap](#), and not necessarily the start address.

See also:

[cuMemCreate](#), [cuMemRelease](#), [cuMemMap](#)

CUresult cuMemSetAccess (CUdeviceptr ptr, size_t size, const CUmemAccessDesc *desc, size_t count)

Set the access flags for each location specified in `desc` for the given virtual address range.

Parameters

ptr

- Starting address for the virtual address range

size

- Length of the virtual address range

desc

mapping for each location specified

- Array of CUMemAccessDesc that describe how to change the

count

- Number of CUMemAccessDesc in desc

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_NOT_SUPPORTED](#)

- mapping for each location specified

Description

Given the virtual address range via `ptr` and `size`, and the locations in the array given by `desc` and `count`, set the access flags for the target locations. The range must be a fully mapped address range containing all allocations created by [cuMemMap](#) / [cuMemCreate](#). Users cannot specify [CU_MEM_LOCATION_TYPE_HOST_NUMA](#) accessibility for allocations created on with other location types. Note: When `CUMemAccessDesc::CUMemLocation::type` is [CU_MEM_LOCATION_TYPE_HOST_NUMA](#), `CUMemAccessDesc::CUMemLocation::id` is ignored. When setting the access flags for a virtual address range mapping a multicast object, `ptr` and `size` must be aligned to the value returned by [cuMulticastGetGranularity](#) with the flag `CU_MULTICAST_MINIMUM_GRANULARITY`. For best performance however, it is recommended that `ptr` and `size` be aligned to the value returned by [cuMulticastGetGranularity](#) with the flag `CU_MULTICAST_RECOMMENDED_GRANULARITY`.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits [synchronous](#) behavior for most use cases.

See also:

[cuMemSetAccess](#), [cuMemCreate](#), [:cuMemMap](#)

CUresult cuMemUnmap (CUdeviceptr ptr, size_t size)

Unmap the backing memory of a given address range.

Parameters**ptr**

- Starting address for the virtual address range to unmap

size

- Size of the virtual address range to unmap

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_PERMITTED](#), [CUDA_ERROR_NOT_SUPPORTED](#)

Description

The range must be the entire contiguous address range that was mapped to. In other words, [cuMemUnmap](#) cannot unmap a sub-range of an address range mapped by [cuMemCreate](#) / [cuMemMap](#). Any backing memory allocations will be freed if there are no existing mappings and there are no unreleased memory handles.

When [cuMemUnmap](#) returns successfully the address range is converted to an address reservation and can be used for a future calls to [cuMemMap](#). Any new mapping to this virtual address will need to have access granted through [cuMemSetAccess](#), as all mappings start with no accessibility setup.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits [synchronous](#) behavior for most use cases.

See also:

[cuMemCreate](#), [cuMemAddressReserve](#)

6.15. Stream Ordered Memory Allocator

This section describes the stream ordered memory allocator exposed by the low-level CUDA driver application programming interface.

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 [cuDeviceGetAttribute\(\)](#) with the device attribute [CU_DEVICE_ATTRIBUTE_MEMORY_POOLS_SUPPORTED](#)

CUresult cuMemAllocAsync (CUdeviceptr *dptr, size_t bytesize, CUstream hStream)

Allocates memory with stream ordered semantics.

Parameters

dptr

- Returned device pointer

bytesize

- Number of bytes to allocate

hStream

- The stream establishing the stream ordering contract and the memory pool to allocate from

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#) (default stream specified with no current context), [CUDA_ERROR_NOT_SUPPORTED](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

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 allocation operation completes. The allocation comes from the memory pool current to 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.

See also:

[cuMemAllocFromPoolAsync](#), [cuMemFreeAsync](#), [cuDeviceSetMemPool](#),
[cuDeviceGetDefaultMemPool](#), [cuDeviceGetMemPool](#), [cuMemPoolCreate](#), [cuMemPoolSetAccess](#),
[cuMemPoolSetAttribute](#)

CUresult cuMemAllocFromPoolAsync (CUdeviceptr *dptr, size_t bytesize, CUmemoryPool pool, CUstream hStream)

Allocates memory from a specified pool with stream ordered semantics.

Parameters

dptr

- Returned device pointer

bytesize

- Number of bytes to allocate

pool

- The pool to allocate from

hStream

- The stream establishing the stream ordering semantic

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#) (default stream specified with no current context),
[CUDA_ERROR_NOT_SUPPORTED](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

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 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:

[cuMemAllocAsync](#), [cuMemFreeAsync](#), [cuDeviceGetDefaultMemPool](#), [cuDeviceGetMemPool](#), [cuMemPoolCreate](#), [cuMemPoolSetAccess](#), [cuMemPoolSetAttribute](#)

CUresult cuMemFreeAsync (CUdeviceptr dptr, CUstream hStream)

Frees memory with stream ordered semantics.

Parameters

dptr

- memory to free

hStream

- The stream establishing the stream ordering contract.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#) (default stream specified with no current context), [CUDA_ERROR_NOT_SUPPORTED](#)

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.

CUresult cuMemPoolCreate (CUmemoryPool *pool, const CUmemPoolProps *poolProps)

Creates a memory pool.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_NOT_PERMITTED](#), [CUDA_ERROR_NOT_SUPPORTED](#)

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 `CUmemPoolProps::CUmemLocation::type` to [CU_MEM_LOCATION_TYPE_HOST_NUMA](#) and `CUmemPoolProps::CUmemLocation::id` must specify the NUMA ID of the host memory node. Specifying [CU_MEM_LOCATION_TYPE_HOST_NUMA_CURRENT](#) or [CU_MEM_LOCATION_TYPE_HOST](#) as the `CUmemPoolProps::CUmemLocation::type` will result in [CUDA_ERROR_INVALID_VALUE](#). By default, the pool's memory will be accessible from the device it is allocated on. In the case of pools created with [CU_MEM_LOCATION_TYPE_HOST_NUMA](#), 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 [CUmemPoolProps::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 `CU_MEM_HANDLE_TYPE_NONE` creates a memory pool that will not support IPC.

See also:

[cuDeviceSetMemPool](#), [cuDeviceGetMemPool](#), [cuDeviceGetDefaultMemPool](#),
[cuMemAllocFromPoolAsync](#), [cuMemPoolExportToShareableHandle](#)

CUresult cuMemPoolDestroy (CUmemoryPool pool)

Destroys the specified memory pool.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

If any pointers obtained from this pool haven't been freed or the pool has free operations that haven't completed when [cuMemPoolDestroy](#) 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:

[cuMemFreeAsync](#), [cuDeviceSetMemPool](#), [cuDeviceGetMemPool](#), [cuDeviceGetDefaultMemPool](#), [cuMemPoolCreate](#)

CUresult cuMemPoolExportPointer (CUmemPoolPtrExportData *shareData_out, CUdeviceptr ptr)

Export data to share a memory pool allocation between processes.

Parameters

shareData_out

- Returned export data

ptr

- pointer to memory being exported

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Constructs `shareData_out` for sharing a specific allocation from an already shared memory pool. The recipient process can import the allocation with the [cuMemPoolImportPointer](#) api. The data is not a handle and may be shared through any IPC mechanism.

See also:

[cuMemPoolExportToShareableHandle](#), [cuMemPoolImportFromShareableHandle](#), [cuMemPoolImportPointer](#)

CUresult cuMemPoolExportToShareableHandle (void *handle_out, CUmemoryPool pool, CUmemAllocationHandleType handleType, unsigned long long flags)

Exports a memory pool to the requested handle type.

Parameters

handle_out

- Returned OS handle

pool

- pool to export

handleType

- the type of handle to create

flags

- must be 0

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

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 [cuMemPoolImportFromShareableHandle](#). Individual pointers can then be shared with the [cuMemPoolExportPointer](#) and [cuMemPoolImportPointer](#) 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 CU_MEM_HANDLE_TYPE_NONE.

See also:

[cuMemPoolImportFromShareableHandle](#), [cuMemPoolExportPointer](#), [cuMemPoolImportPointer](#), [cuMemAllocAsync](#), [cuMemFreeAsync](#), [cuDeviceGetDefaultMemPool](#), [cuDeviceGetMemPool](#), [cuMemPoolCreate](#), [cuMemPoolSetAccess](#), [cuMemPoolSetAttribute](#)

CUresult cuMemPoolGetAccess (CUmemAccess_flags *flags, CUmemoryPool memPool, CUmemLocation *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:

[cuMemAllocAsync](#), [cuMemFreeAsync](#), [cuDeviceGetDefaultMemPool](#), [cuDeviceGetMemPool](#), [cuMemPoolCreate](#)

CUresult cuMemPoolGetAttribute (CUmemoryPool pool, CUmemPool_attribute attr, void *value)

Gets attributes of a memory pool.

Parameters

pool

- The memory pool to get attributes of

attr

- The attribute to get

value

- Retrieved value

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Supported attributes are:

- ▶ [CU_MEMPOOL_ATTR_RELEASE_THRESHOLD](#): (value type = `cuint64_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)
- ▶ [CU_MEMPOOL_ATTR_REUSE_FOLLOW_EVENT_DEPENDENCIES](#): (value type = `int`) Allow [cuMemAllocAsync](#) 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)
- ▶ [CU_MEMPOOL_ATTR_REUSE_ALLOW_OPPORTUNISTIC](#): (value type = `int`) Allow reuse of already completed frees when there is no dependency between the free and allocation. (default enabled)
- ▶ [CU_MEMPOOL_ATTR_REUSE_ALLOW_INTERNAL_DEPENDENCIES](#): (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 [cuMemFreeAsync](#) (default enabled).
- ▶ [CU_MEMPOOL_ATTR_RESERVED_MEM_CURRENT](#): (value type = `cuint64_t`) Amount of backing memory currently allocated for the mempool
- ▶ [CU_MEMPOOL_ATTR_RESERVED_MEM_HIGH](#): (value type = `cuint64_t`) High watermark of backing memory allocated for the mempool since the last time it was reset.
- ▶ [CU_MEMPOOL_ATTR_USED_MEM_CURRENT](#): (value type = `cuint64_t`) Amount of memory from the pool that is currently in use by the application.
- ▶ [CU_MEMPOOL_ATTR_USED_MEM_HIGH](#): (value type = `cuint64_t`) High watermark of the amount of memory from the pool that was in use by the application.

See also:

[cuMemAllocAsync](#), [cuMemFreeAsync](#), [cuDeviceGetDefaultMemPool](#), [cuDeviceGetMemPool](#), [cuMemPoolCreate](#)

CUresult cuMemPoolImportFromShareableHandle (CUmemoryPool *pool_out, void *handle, CUmemAllocationHandleType handleType, unsigned long long flags)

imports a memory pool from a shared handle.

Parameters

pool_out

- Returned memory pool

handle

- OS handle of the pool to open

handleType

- The type of handle being imported

flags

- must be 0

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Specific allocations can be imported from the imported pool with [cuMemPoolImportPointer](#).

If `handleType` is [CU_MEM_HANDLE_TYPE_FABRIC](#) and the importer process has not been granted access to the same IMEX channel as the exporter process, this API will error as [CUDA_ERROR_NOT_PERMITTED](#).



Note:

Imported memory pools do not support creating new allocations. As such imported memory pools may not be used in [cuDeviceSetMemPool](#) or [cuMemAllocFromPoolAsync](#) calls.

See also:

[cuMemPoolExportToShareableHandle](#), [cuMemPoolExportPointer](#), [cuMemPoolImportPointer](#)

CUresult cuMemPoolImportPointer (CUdeviceptr *ptr_out, CUmemoryPool pool, CUmemPoolPtrExportData *shareData)

Import a memory pool allocation from another process.

Parameters

ptr_out

- pointer to imported memory

pool

- pool from which to import

shareData

- data specifying the memory to import

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 `cuMemFree` or `cuMemFreeAsync`. If `cuMemFreeAsync` is used, the free must be completed on the importing process before the free operation on the exporting process.



Note:

The `cuMemFreeAsync` api may be used in the exporting process before the `cuMemFreeAsync` operation completes in its stream as long as the `cuMemFreeAsync` in the exporting process specifies a stream with a stream dependency on the importing process's `cuMemFreeAsync`.

See also:

[cuMemPoolExportToShareableHandle](#), [cuMemPoolImportFromShareableHandle](#), [cuMemPoolExportPointer](#)

CUresult cuMemPoolSetAccess (CUmemoryPool pool, const CUmemAccessDesc *map, size_t count)

Controls visibility of pools between devices.

Parameters

pool

- The pool being modified

map

- Array of access descriptors. Each descriptor instructs the access to enable for a single gpu.

count

- Number of descriptors in the map array.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

See also:

[cuMemAllocAsync](#), [cuMemFreeAsync](#), [cuDeviceGetDefaultMemPool](#), [cuDeviceGetMemPool](#), [cuMemPoolCreate](#)

CUresult cuMemPoolSetAttribute (CUmemoryPool pool, CUmemPool_attribute attr, void *value)

Sets attributes of a memory pool.

Parameters

pool

- The memory pool to modify

attr

- The attribute to modify

value

- Pointer to the value to assign

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Supported attributes are:

- ▶ [CU_MEMPOOL_ATTR_RELEASE_THRESHOLD](#): (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)
- ▶ [CU_MEMPOOL_ATTR_REUSE_FOLLOW_EVENT_DEPENDENCIES](#): (value type = `int`) Allow [cuMemAllocAsync](#) 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)
- ▶ [CU_MEMPOOL_ATTR_REUSE_ALLOW_OPPORTUNISTIC](#): (value type = `int`) Allow reuse of already completed frees when there is no dependency between the free and allocation. (default enabled)
- ▶ [CU_MEMPOOL_ATTR_REUSE_ALLOW_INTERNAL_DEPENDENCIES](#): (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 [cuMemFreeAsync](#) (default enabled).
- ▶ [CU_MEMPOOL_ATTR_RESERVED_MEM_HIGH](#): (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.
- ▶ [CU_MEMPOOL_ATTR_USED_MEM_HIGH](#): (value type = `cuuint64_t`) Reset the high watermark that tracks the amount of used memory that was allocated for the memory pool.

See also:

[cuMemAllocAsync](#), [cuMemFreeAsync](#), [cuDeviceGetDefaultMemPool](#), [cuDeviceGetMemPool](#), [cuMemPoolCreate](#)

CUresult cuMemPoolTrimTo (CUmemoryPool pool, size_t minBytesToKeep)

Tries to release memory back to the OS.

Parameters

pool

- The memory pool to trim

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

[CUDA_SUCCESS](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

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.

See also:

[cuMemAllocAsync](#), [cuMemFreeAsync](#), [cuDeviceGetDefaultMemPool](#), [cuDeviceGetMemPool](#), [cuMemPoolCreate](#)

6.16. Multicast Object Management

This section describes the CUDA multicast object operations exposed by the low-level CUDA driver application programming interface.

overview

A multicast object created via [cuMulticastCreate](#) enables certain memory operations to be broadcast to a team of devices. Devices can be added to a multicast object via [cuMulticastAddDevice](#). Memory can be bound on each participating device via either [cuMulticastBindMem](#) or [cuMulticastBindAddr](#). Multicast objects can be mapped into a device's virtual address space using the virtual memory management APIs (see [cuMemMap](#) and [cuMemSetAccess](#)).

Supported Platforms

Support for multicast on a specific device can be queried using the device attribute [CU_DEVICE_ATTRIBUTE_MULTICAST_SUPPORTED](#)

CUresult cuMulticastAddDevice (CUmemGenericAllocationHandle mcHandle, CUdevice dev)

Associate a device to a multicast object.

Parameters

mcHandle

Handle representing a multicast object.

dev

Device that will be associated to the multicast object.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_PERMITTED](#), [CUDA_ERROR_NOT_SUPPORTED](#)

Description

Associates a device to a multicast object. The added device will be a part of the multicast team of size specified by [CUmulticastObjectProp::numDevices](#) during [cuMulticastCreate](#). The association of the device to the multicast object is permanent during the life time of the multicast object. All devices must be added to the multicast team before any memory can be bound to any device in the team. Any calls to [cuMulticastBindMem](#) or [cuMulticastBindAddr](#) will block until all devices have been added. Similarly all devices must be added to the multicast team before a virtual address range can be mapped to the multicast object. A call to [cuMemMap](#) will block until all devices have been added.

See also:

[cuMulticastCreate](#), [cuMulticastBindMem](#), [cuMulticastBindAddr](#)

CUresult cuMulticastBindAddr (CUmemGenericAllocationHandle mcHandle, size_t mcOffset, CUdeviceptr memptr, size_t size, unsigned long long flags)

Bind a memory allocation represented by a virtual address to a multicast object.

Parameters

mcHandle

Handle representing a multicast object.

mcOffset

Offset into multicast va range for attachment.

memptr

Virtual address of the memory allocation.

size

Size of memory that will be bound to the multicast object.

flags

Flags for future use, must be zero now.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED, CUDA_ERROR_NOT_SUPPORTED, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_SYSTEM_NOT_READY

Description

Binds a memory allocation specified by its mapped address `memptr` to a multicast object represented by `mcHandle`. The memory must have been allocated via [cuMemCreate](#) or [cudaMallocAsync](#). The intended `size` of the bind, the offset in the multicast range `mcOffset` and `memptr` must be a multiple of the value returned by [cuMulticastGetGranularity](#) with the flag [CU_MULTICAST_GRANULARITY_MINIMUM](#). For best performance however, `size`, `mcOffset` and `memptr` should be aligned to the value returned by [cuMulticastGetGranularity](#) with the flag [CU_MULTICAST_GRANULARITY_RECOMMENDED](#).

The `size` cannot be larger than the size of the allocated memory. Similarly the `size + mcOffset` cannot be larger than the total size of the multicast object. The memory allocation must have been created on one of the devices that was added to the multicast team via [cuMulticastAddDevice](#). Externally shareable as well as imported multicast objects can be bound only to externally shareable memory. Note that this call will return `CUDA_ERROR_OUT_OF_MEMORY` if there are insufficient resources required to perform the bind. This call may also return

CUDA_ERROR_SYSTEM_NOT_READY if the necessary system software is not initialized or running.

See also:

[cuMulticastCreate](#), [cuMulticastAddDevice](#), [cuMemCreate](#)

CUresult cuMulticastBindMem
(CUmemGenericAllocationHandle mcHandle, size_t mcOffset, CUmemGenericAllocationHandle memHandle, size_t memOffset, size_t size, unsigned long long flags)

Bind a memory allocation represented by a handle to a multicast object.

Parameters

mcHandle

Handle representing a multicast object.

mcOffset

Offset into the multicast object for attachment.

memHandle

Handle representing a memory allocation.

memOffset

Offset into the memory for attachment.

size

Size of the memory that will be bound to the multicast object.

flags

Flags for future use, must be zero for now.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_PERMITTED](#), [CUDA_ERROR_NOT_SUPPORTED](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_SYSTEM_NOT_READY](#)

Description

Binds a memory allocation specified by `memHandle` and created via [cuMemCreate](#) to a multicast object represented by `mcHandle` and created via [cuMulticastCreate](#). The intended size of the bind, the offset in the multicast range `mcOffset` as well as the offset in the memory `memOffset` must be a multiple of the value returned by [cuMulticastGetGranularity](#) with the flag [CU_MULTICAST_GRANULARITY_MINIMUM](#). For best performance however, `size`, `mcOffset` and `memOffset` should be aligned to the granularity of

the memory allocation(see `::cuMemGetAllocationGranularity`) or to the value returned by `cuMulticastGetGranularity` with the flag `CU_MULTICAST_GRANULARITY_RECOMMENDED`.

The `size + memOffset` cannot be larger than the size of the allocated memory. Similarly the `size + mcOffset` cannot be larger than the size of the multicast object. The memory allocation must have been created on one of the devices that was added to the multicast team via `cuMulticastAddDevice`. Externally shareable as well as imported multicast objects can be bound only to externally shareable memory. Note that this call will return `CUDA_ERROR_OUT_OF_MEMORY` if there are insufficient resources required to perform the bind. This call may also return `CUDA_ERROR_SYSTEM_NOT_READY` if the necessary system software is not initialized or running.

See also:

`cuMulticastCreate`, `cuMulticastAddDevice`, `cuMemCreate`

CUresult cuMulticastCreate (CUmemGenericAllocationHandle *mcHandle, const CUmulticastObjectProp *prop)

Create a generic allocation handle representing a multicast object described by the given properties.

Parameters

mcHandle

Value of handle returned.

prop

Properties of the multicast object to create.

Returns

`CUDA_SUCCESS`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_OUT_OF_MEMORY`,
`CUDA_ERROR_INVALID_DEVICE`, `CUDA_ERROR_NOT_INITIALIZED`,
`CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_PERMITTED`,
`CUDA_ERROR_NOT_SUPPORTED`

Description

This creates a multicast object as described by `prop`. The number of participating devices is specified by `CUmulticastObjectProp::numDevices`. Devices can be added to the multicast object via `cuMulticastAddDevice`. All participating devices must be added to the multicast object before memory can be bound to it. Memory is bound to the multicast object via either `cuMulticastBindMem` or `cuMulticastBindAddr`, and can be unbound via `cuMulticastUnbind`. The total amount of memory that can be bound per device is specified by `:CUmulticastObjectProp::size`. This size must be a multiple of the value returned by `cuMulticastGetGranularity` with the flag `CU_MULTICAST_GRANULARITY_MINIMUM`. For best performance however,

the size should be aligned to the value returned by [cuMulticastGetGranularity](#) with the flag [CU_MULTICAST_GRANULARITY_RECOMMENDED](#).

After all participating devices have been added, multicast objects can also be mapped to a device's virtual address space using the virtual memory management APIs (see [cuMemMap](#) and [cuMemSetAccess](#)). Multicast objects can also be shared with other processes by requesting a shareable handle via [cuMemExportToShareableHandle](#). Note that the desired types of shareable handles must be specified in the bitmask [CUMulticastObjectProp::handleTypes](#). Multicast objects can be released using the virtual memory management API [cuMemRelease](#).

See also:

[cuMulticastAddDevice](#), [cuMulticastBindMem](#), [cuMulticastBindAddr](#), [cuMulticastUnbind](#)
[cuMemCreate](#), [cuMemRelease](#), [cuMemExportToShareableHandle](#),
[cuMemImportFromShareableHandle](#)

CUresult cuMulticastGetGranularity (size_t *granularity, const CUMulticastObjectProp *prop, CUMulticastGranularity_flags option)

Calculates either the minimal or recommended granularity for multicast object.

Parameters

granularity

Returned granularity.

prop

Properties of the multicast object.

option

Determines which granularity to return.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_PERMITTED](#),
[CUDA_ERROR_NOT_SUPPORTED](#)

Description

Calculates either the minimal or recommended granularity for a given set of multicast object properties and returns it in granularity. This granularity can be used as a multiple for size, bind offsets and address mappings of the multicast object.

See also:

[cuMulticastCreate](#), [cuMulticastBindMem](#), [cuMulticastBindAddr](#), [cuMulticastUnbind](#)

CUresult cuMulticastUnbind (CUmemGenericAllocationHandle mcHandle, CUdevice dev, size_t mcOffset, size_t size)

Unbind any memory allocations bound to a multicast object at a given offset and upto a given size.

Parameters

mcHandle

Handle representing a multicast object.

dev

Device that hosts the memory allocation.

mcOffset

Offset into the multicast object.

size

Desired size to unbind.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_PERMITTED](#), [CUDA_ERROR_NOT_SUPPORTED](#)

Description

Unbinds any memory allocations hosted on `dev` and bound to a multicast object at `mcOffset` and upto a given `size`. The intended `size` of the unbind and the offset in the multicast range (`mcOffset`) must be a multiple of the value returned by [cuMulticastGetGranularity](#) flag [CU_MULTICAST GRANULARITY MINIMUM](#). The `size + mcOffset` cannot be larger than the total size of the multicast object.



Note:

Warning: The `mcOffset` and the `size` must match the corresponding values specified during the bind call. Any other values may result in undefined behavior.

See also:

[cuMulticastBindMem](#), [cuMulticastBindAddr](#)

6.17. Unified Addressing

This section describes the unified addressing functions of the low-level CUDA driver 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 [cuDeviceGetAttribute\(\)](#) with the device attribute [CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING](#).

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 [cuPointerGetAttribute\(\)](#)

Since pointers are unique, it is not necessary to specify information about the pointers specified to the various copy functions in the CUDA API. The function [cuMemcpy\(\)](#) may be used to perform a copy between two pointers, ignoring whether they point to host or device memory (making [cuMemcpyHtoD\(\)](#), [cuMemcpyDtoD\(\)](#), and [cuMemcpyDtoH\(\)](#) unnecessary for devices supporting unified addressing). For multidimensional copies, the memory type [CU_MEMORYTYPE_UNIFIED](#) may be used to specify that the CUDA driver should infer the location of the pointer from its value.

Automatic Mapping of Host Allocated Host Memory

All host memory allocated in all contexts using [cuMemAllocHost\(\)](#) and [cuMemHostAlloc\(\)](#) is always directly accessible from all contexts on all devices that support unified addressing. This is the case regardless of whether or not the flags [CU_MEMHOSTALLOC_PORTABLE](#) and [CU_MEMHOSTALLOC_DEVICEMAP](#) 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, so it is not necessary to call [cuMemHostGetDevicePointer\(\)](#) to get the device pointer for these allocations.

Note that this is not the case for memory allocated using the flag [CU_MEMHOSTALLOC_WRITECOMBINED](#), as discussed below.

Automatic Registration of Peer Memory

Upon enabling direct access from a context that supports unified addressing to another peer context that supports unified addressing using [cuCtxEnablePeerAccess\(\)](#) all memory allocated in the peer context using [cuMemAlloc\(\)](#) and [cuMemAllocPitch\(\)](#) will immediately be accessible by the current context. The device pointer value through which any peer memory may be accessed in the current context is the same pointer value through which that memory may be accessed in the peer context.

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 [cuMemHostRegister\(\)](#) and host memory allocated using the flag [CU_MEMHOSTALLOC_WRITECOMBINED](#). 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 contexts that support unified addressing.

This device address may be queried using [cuMemHostGetDevicePointer\(\)](#) when a context using unified addressing is current. Either the host or the unified device pointer value may be used to refer to this memory through [cuMemcpy\(\)](#) and similar functions using the [CU_MEMORYTYPE_UNIFIED](#) memory type.

CUresult cuMemAdvise (CUdeviceptr devPtr, size_t count, CUmem_advise advice, CUdevice 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

Note there is a later version of this API, [cuMemAdvise_v2](#). It will supplant this version in 13.0, which is retained for minor version compatibility.

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 [cuMemAllocManaged](#) 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:

- ▶ [CU_MEM_ADVICE_SET_READ_MOSTLY](#): 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 [cuMemPrefetchAsync](#) 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 [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#). Also, if a context is created on a device that does not have the device attribute [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#) 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 [CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS](#) 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 [CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES](#), then setting this advice will not create a read-only copy when that device accesses this memory region.
- ▶ [CU_MEM_ADVICE_UNSET_READ_MOSTLY](#): Undoes the effect of [CU_MEM_ADVICE_SET_READ_MOSTLY](#) 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.
- ▶ [CU_MEM_ADVICE_SET_PREFERRED_LOCATION](#): This advice sets the preferred location for the data to be the memory belonging to `device`. Passing in `CU_DEVICE_CPU` 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 [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#). 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 [cuMemPrefetchAsync](#). 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 [CU_MEM_ADVISE_SET_READ_MOSTLY](#) 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 [CU_MEM_ADVISE_SET_READ_MOSTLY](#). If the memory region refers to valid system-allocated pageable memory, then `device` must have a non-zero value for the device attribute [CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS](#).

- ▶ [CU_MEM_ADVISE_UNSET_PREFERRED_LOCATION](#): Undoes the effect of [CU_MEM_ADVISE_SET_PREFERRED_LOCATION](#) and changes the preferred location to none.
- ▶ [CU_MEM_ADVISE_SET_ACCESSED_BY](#): This advice implies that the data will be accessed by `device`. Passing in [CU_DEVICE_CPU](#) for `device` will set the advice for the CPU. If `device` is a GPU, then the device attribute [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#) 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 [CU_MEM_ADVISE_SET_ACCESSED_BY](#) flag set for this data will now have its mapping updated to point to the page in host memory. If [CU_MEM_ADVISE_SET_READ_MOSTLY](#) 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 [CU_MEM_ADVISE_SET_PREFERRED_LOCATION](#) 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 [CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS](#). Additionally, if `device` has a non-zero value for the device attribute [CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES](#), then this call has no effect.
- ▶ [CU_MEM_ADVISE_UNSET_ACCESSED_BY](#): Undoes the effect of [CU_MEM_ADVISE_SET_ACCESSED_BY](#). 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 [CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS](#). Additionally, if `device` has a non-zero value for the device attribute [CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES](#), 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.

See also:

[cuMemcpy](#), [cuMemcpyPeer](#), [cuMemcpyAsync](#), [cuMemcpy3DPeerAsync](#), [cuMemPrefetchAsync](#), [cuMemAdvise_v2](#) [cudaMemAdvise](#)

CUresult cuMemAdvise_v2 (CUdeviceptr devPtr, size_t count, CUmem_advise advice, CUmemLocation 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

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 [cuMemAllocManaged](#) 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:

- ▶ `CU_MEM_ADVICE_SET_READ_MOSTLY`: 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 `cuMemPrefetchAsync` or `cuMemPrefetchAsync_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 `cuMemPrefetchAsync_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 `CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS`. Also, if a context is created on a device that does not have the device attribute `CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS` 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 `CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS` 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 `CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES`, then setting this advice will not create a read-only copy when that device accesses this memory region.
- ▶ `CU_MEM_ADVICE_UNSET_READ_MOSTLY`: Undoes the effect of `CU_MEM_ADVICE_SET_READ_MOSTLY` 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.
- ▶ `CU_MEM_ADVICE_SET_PREFERRED_LOCATION`: This advice sets the preferred location for the data to be the memory belonging to `location`. When `CUmemLocation::type` is `CU_MEM_LOCATION_TYPE_HOST`, `CUmemLocation::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 `CUmemLocation::type` to `CU_MEM_LOCATION_TYPE_HOST_NUMA` and `CUmemLocation::id` must specify the NUMA ID of the host NUMA node. If `CUmemLocation::type` is set to `CU_MEM_LOCATION_TYPE_HOST_NUMA_CURRENT`, `CUmemLocation::id` will be ignored and the the host NUMA node closest to the

calling thread's CPU will be used as the preferred location. If `CUmemLocation::type` is a `CU_MEM_LOCATION_TYPE_DEVICE`, then `CUmemLocation::id` must be a valid device ordinal and the device must have a non-zero value for the device attribute `CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS`. 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 `cuMemPrefetchAsync`. 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 `CU_MEM_ADVISE_SET_READ_MOSTLY` 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 processor as outlined in description for the advice `CU_MEM_ADVISE_SET_READ_MOSTLY`. If the memory region refers to valid system-allocated pageable memory, and `CUmemLocation::type` is `CU_MEM_LOCATION_TYPE_DEVICE` then `CUmemLocation::id` must be a valid device that has a non-zero value for the device attribute `CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS`.

- ▶ `CU_MEM_ADVISE_UNSET_PREFERRED_LOCATION`: Undoes the effect of `CU_MEM_ADVISE_SET_PREFERRED_LOCATION` and changes the preferred location to none. The `location` argument is ignored for this advice.
- ▶ `CU_MEM_ADVISE_SET_ACCESSED_BY`: This advice implies that the data will be accessed by processor `location`. The `CUmemLocation::type` must be either `CU_MEM_LOCATION_TYPE_DEVICE` with `CUmemLocation::id` representing a valid device ordinal or `CU_MEM_LOCATION_TYPE_HOST` and `CUmemLocation::id` will be ignored. All other location types are invalid. If `CUmemLocation::id` is a GPU, then the device attribute `CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS` 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 [CU_MEM_ADVISE_SET_ACCESSED_BY](#) flag set for this data will now have its mapping updated to point to the page in host memory. If [CU_MEM_ADVISE_SET_READ_MOSTLY](#) 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 `CUmemLocation::type` is [CU_MEM_LOCATION_TYPE_DEVICE](#) then device in `CUmemLocation::id` must have a non-zero value for the device attribute [CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS](#). Additionally, if `CUmemLocation::id` has a non-zero value for the device attribute [CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES](#), then this call has no effect.

- [CU_MEM_ADVISE_UNSET_ACCESSED_BY](#): Undoes the effect of [CU_MEM_ADVISE_SET_ACCESSED_BY](#). 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 `CUmemLocation::type` is [CU_MEM_LOCATION_TYPE_DEVICE](#) then device in `CUmemLocation::id` must have a non-zero value for the device attribute [CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS](#). Additionally, if `CUmemLocation::id` has a non-zero value for the device attribute [CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES](#), 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.

See also:

[cuMemcpy](#), [cuMemcpyPeer](#), [cuMemcpyAsync](#), [cuMemcpy3DPeerAsync](#), [cuMemPrefetchAsync](#), [cuMemAdvise](#) [cudaMemAdvise](#)

CUresult cuMemPrefetchAsync (CUdeviceptr devPtr, size_t count, CUdevice dstDevice, CUstream hStream)

Prefetches memory to the specified destination device.

Parameters

devPtr

- Pointer to be prefetched

count

- Size in bytes

dstDevice

- Destination device to prefetch to

hStream

- Stream to enqueue prefetch operation

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

Note there is a later version of this API, [cuMemPrefetchAsync_v2](#). It will supplant this version in 13.0, which is retained for minor version compatibility.

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. `hStream` is the stream in which the operation is enqueued. The memory range must refer to managed memory allocated via [cuMemAllocManaged](#) or declared via `__managed__` variables or it may also refer to system-allocated memory on systems with non-zero `CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS`.

Passing in `CU_DEVICE_CPU` for `dstDevice` will prefetch the data to host memory. If `dstDevice` is a GPU, then the device attribute [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#) must be non-zero.

Additionally, `hStream` must be associated with a device that has a non-zero value for the device attribute [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#).

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 [cuMemAllocManaged](#) allocations to host memory in order to make room. Device memory allocated using [cuMemAlloc](#) or [cuArrayCreate](#) 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 [cuMemAdvise](#) as described below:

If [CU_MEM_ADVISE_SET_READ_MOSTLY](#) was set on any subset of this memory range, then that subset will create a read-only copy of the pages on `dstDevice`.

If [CU_MEM_ADVISE_SET_PREFERRED_LOCATION](#) 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 [CU_MEM_ADVISE_SET_ACCESSED_BY](#) 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.

See also:

[cuMemcpy](#), [cuMemcpyPeer](#), [cuMemcpyAsync](#), [cuMemcpy3DPeerAsync](#), [cuMemAdvise](#), [cuMemPrefetchAsync](#) [cudaMemPrefetchAsync_v2](#)

CUresult cuMemPrefetchAsync_v2 (CUdeviceptr devPtr, size_t count, CUmemLocation location, unsigned int flags, CUstream hStream)

Prefetches memory to the specified destination location.

Parameters

devPtr

- Pointer to be prefetched

count

- Size in bytes

location**flags**

- flags for future use, must be zero now.

hStream

- Stream to enqueue prefetch operation

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

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. `hStream` is the stream in which the operation is enqueued. The memory range must refer to managed memory allocated via [cuMemAllocManaged](#) or declared via `__managed__` variables.

Specifying [CU_MEM_LOCATION_TYPE_DEVICE](#) for [CUMemLocation::type](#) will prefetch memory to GPU specified by device ordinal [CUMemLocation::id](#) which must have non-zero value for the device attribute [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#).

Additionally, `hStream` must be associated with a device that has a non-zero value for the device attribute [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#).

Specifying [CU_MEM_LOCATION_TYPE_HOST](#) as [CUMemLocation::type](#) will prefetch data to host memory. Applications can request prefetching memory to a specific host NUMA node by specifying [CU_MEM_LOCATION_TYPE_HOST_NUMA](#) for [CUMemLocation::type](#) and a valid host NUMA node id in [CUMemLocation::id](#). Users can also request prefetching memory to the host NUMA node closest to the current thread's CPU by specifying [CU_MEM_LOCATION_TYPE_HOST_NUMA_CURRENT](#) for [CUMemLocation::type](#). Note when [CUMemLocation::type](#) is either [CU_MEM_LOCATION_TYPE_HOST](#) OR [CU_MEM_LOCATION_TYPE_HOST_NUMA_CURRENT](#), [CUMemLocation::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 [cuMemAllocManaged](#) allocations to host memory in order to make room. Device memory allocated using [cuMemAlloc](#) or [cuArrayCreate](#) 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 [CU_MEM_ADVISE_SET_READ_MOSTLY](#) 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 [CU_MEM_ADVICE_SET_PREFERRED_LOCATION](#) 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 [CU_MEM_ADVICE_SET_ACCESSED_BY](#) 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.

See also:

[cuMemcpy](#), [cuMemcpyPeer](#), [cuMemcpyAsync](#), [cuMemcpy3DPeerAsync](#), [cuMemAdvise](#), [cuMemPrefetchAsync](#) [cudaMemPrefetchAsync_v2](#)

CUresult cuMemRangeGetAttribute (void *data, size_t dataSize, CUmем_range_attribute attribute, CUdeviceptr 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

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 [cuMemAllocManaged](#) or declared via `__managed__` variables.

The `attribute` parameter can take the following values:

- ▶ [CU_MEM_RANGE_ATTRIBUTE_READ_MOSTLY](#): 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.
- ▶ [CU_MEM_RANGE_ATTRIBUTE_PREFERRED_LOCATION](#): 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 `CU_DEVICE_CPU` if all pages in the memory range have the CPU as their preferred location, or it will be `CU_DEVICE_INVALID` 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.
- ▶ [CU_MEM_RANGE_ATTRIBUTE_ACCESSED_BY](#): 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 [CU_MEM_ADVISE_SET_ACCESSED_BY](#) 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, `CU_DEVICE_INVALID` 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, `CU_DEVICE_INVALID`, `CU_DEVICE_INVALID` }. 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.
- ▶ [CU_MEM_RANGE_ATTRIBUTE_LAST_PREFETCH_LOCATION](#): 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 [cuMemPrefetchAsync](#). This will either be a GPU id or `CU_DEVICE_CPU` 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, `CU_DEVICE_INVALID` will be returned. Note that this simply returns the last location that the

application requested to prefetch the memory range to. It gives no indication as to whether the prefetch operation to that location has completed or even begun.

- ▶ CU_MEM_RANGE_ATTRIBUTE_PREFERRED_LOCATION_TYPE: If this attribute is specified, `data` will be interpreted as a `CUmemLocationType`, and `dataSize` must be `sizeof(CUmemLocationType)`. The `CUmemLocationType` returned will be CU_MEM_LOCATION_TYPE_DEVICE if all pages in the memory range have the same GPU as their preferred location, or `CUmemLocationType` will be CU_MEM_LOCATION_TYPE_HOST if all pages in the memory range have the CPU as their preferred location, or it will be CU_MEM_LOCATION_TYPE_HOST_NUMA if all the pages in the memory range have the same host NUMA node ID as their preferred location or it will be CU_MEM_LOCATION_TYPE_INVALID 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.
- ▶ CU_MEM_RANGE_ATTRIBUTE_PREFERRED_LOCATION_ID: If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. If the CU_MEM_RANGE_ATTRIBUTE_PREFERRED_LOCATION_TYPE query for the same address range returns CU_MEM_LOCATION_TYPE_DEVICE, it will be a valid device ordinal or if it returns CU_MEM_LOCATION_TYPE_HOST_NUMA, it will be a valid host NUMA node ID or if it returns any other location type, the id should be ignored.
- ▶ CU_MEM_RANGE_ATTRIBUTE_LAST_PREFETCH_LOCATION_TYPE: If this attribute is specified, `data` will be interpreted as a `CUmemLocationType`, and `dataSize` must be `sizeof(CUmemLocationType)`. The result returned will be the last location to which all pages in the memory range were prefetched explicitly via `cuMemPrefetchAsync`. The `CUmemLocationType` returned will be CU_MEM_LOCATION_TYPE_DEVICE if the last prefetch location was a GPU or CU_MEM_LOCATION_TYPE_HOST if it was the CPU or CU_MEM_LOCATION_TYPE_HOST_NUMA 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 CU_MEM_LOCATION_TYPE_INVALID. 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.
- ▶ CU_MEM_RANGE_ATTRIBUTE_LAST_PREFETCH_LOCATION_ID: If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. If the CU_MEM_RANGE_ATTRIBUTE_LAST_PREFETCH_LOCATION_TYPE query for the same address range returns CU_MEM_LOCATION_TYPE_DEVICE, it will be a valid device ordinal or if it returns CU_MEM_LOCATION_TYPE_HOST_NUMA, 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.

See also:

[cuMemRangeGetAttributes](#), [cuMemPrefetchAsync](#), [cuMemAdvise](#), [cudaMemRangeGetAttribute](#)

CUresult cuMemRangeGetAttributes (void **data, size_t *dataSizes, CUmem_range_attribute *attributes, size_t numAttributes, CUdeviceptr 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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

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 [cuMemAllocManaged](#) 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 [cuMemRangeGetAttribute](#) for attribute descriptions and restrictions.

- ▶ [CU_MEM_RANGE_ATTRIBUTE_READ_MOSTLY](#)
- ▶ [CU_MEM_RANGE_ATTRIBUTE_PREFERRED_LOCATION](#)
- ▶ [CU_MEM_RANGE_ATTRIBUTE_ACCESSED_BY](#)
- ▶ [CU_MEM_RANGE_ATTRIBUTE_LAST_PREFETCH_LOCATION](#)
- ▶ [CU_MEM_RANGE_ATTRIBUTE_PREFERRED_LOCATION_TYPE](#)
- ▶ [CU_MEM_RANGE_ATTRIBUTE_PREFERRED_LOCATION_ID](#)
- ▶ [CU_MEM_RANGE_ATTRIBUTE_LAST_PREFETCH_LOCATION_TYPE](#)
- ▶ [CU_MEM_RANGE_ATTRIBUTE_LAST_PREFETCH_LOCATION_ID](#)



Note:

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

See also:

[cuMemRangeGetAttribute](#), [cuMemAdvise](#), [cuMemPrefetchAsync](#), [cudaMemRangeGetAttributes](#)

CUresult cuPointerGetAttribute (void *data, CUpointer_attribute attribute, CUdeviceptr ptr)

Returns information about a pointer.

Parameters

data

- Returned pointer attribute value

attribute

- Pointer attribute to query

ptr

- Pointer

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

The supported attributes are:

- ▶ [CU_POINTER_ATTRIBUTE_CONTEXT](#):

Returns in *data the [CUcontext](#) in which ptr was allocated or registered. The type of data must be [CUcontext](#) *.

If `ptr` was not allocated by, mapped by, or registered with a [CUcontext](#) which uses unified virtual addressing then [CUDA_ERROR_INVALID_VALUE](#) is returned.

► [CU_POINTER_ATTRIBUTE_MEMORY_TYPE](#):

Returns in `*data` the physical memory type of the memory that `ptr` addresses as a [CUmemorytype](#) enumerated value. The type of `data` must be unsigned int.

If `ptr` addresses device memory then `*data` is set to [CU_MEMORYTYPE_DEVICE](#). The particular [CUdevice](#) on which the memory resides is the [CUdevice](#) of the [CUcontext](#) returned by the [CU_POINTER_ATTRIBUTE_CONTEXT](#) attribute of `ptr`.

If `ptr` addresses host memory then `*data` is set to [CU_MEMORYTYPE_HOST](#).

If `ptr` was not allocated by, mapped by, or registered with a [CUcontext](#) which uses unified virtual addressing then [CUDA_ERROR_INVALID_VALUE](#) is returned.

If the current [CUcontext](#) does not support unified virtual addressing then [CUDA_ERROR_INVALID_CONTEXT](#) is returned.

► [CU_POINTER_ATTRIBUTE_DEVICE_POINTER](#):

Returns in `*data` the device pointer value through which `ptr` may be accessed by kernels running in the current [CUcontext](#). The type of `data` must be `CUdeviceptr *`.

If there exists no device pointer value through which kernels running in the current [CUcontext](#) may access `ptr` then [CUDA_ERROR_INVALID_VALUE](#) is returned.

If there is no current [CUcontext](#) then [CUDA_ERROR_INVALID_CONTEXT](#) is returned.

Except in the exceptional disjoint addressing cases discussed below, the value returned in `*data` will equal the input value `ptr`.

► [CU_POINTER_ATTRIBUTE_HOST_POINTER](#):

Returns in `*data` the host pointer value through which `ptr` may be accessed by the host program. The type of `data` must be `void **`. If there exists no host pointer value through which the host program may directly access `ptr` then [CUDA_ERROR_INVALID_VALUE](#) is returned.

Except in the exceptional disjoint addressing cases discussed below, the value returned in `*data` will equal the input value `ptr`.

► [CU_POINTER_ATTRIBUTE_P2P_TOKENS](#):

Returns in `*data` two tokens for use with the `nv-p2p.h` Linux kernel interface. `data` must be a struct of type `CUDA_POINTER_ATTRIBUTE_P2P_TOKENS`.

`ptr` must be a pointer to memory obtained from `:cuMemAlloc()`. Note that `p2pToken` and `vaSpaceToken` are only valid for the lifetime of the source allocation. A subsequent allocation at the same address may return completely different tokens. Querying this attribute has a side effect of setting the attribute [CU_POINTER_ATTRIBUTE_SYNC_MEMOPS](#) for the region of memory that `ptr` points to.

► [CU_POINTER_ATTRIBUTE_SYNC_MEMOPS](#):

A boolean attribute which when set, ensures that synchronous memory operations initiated on the region of memory that `ptr` points to will always synchronize. See further documentation in the section titled "API synchronization behavior" to learn more about cases when synchronous memory operations can exhibit asynchronous behavior.

► CU_POINTER_ATTRIBUTE_BUFFER_ID:

Returns in `*data` a buffer ID which is guaranteed to be unique within the process. `data` must point to an unsigned long long.

`ptr` must be a pointer to memory obtained from a CUDA memory allocation API. Every memory allocation from any of the CUDA memory allocation APIs will have a unique ID over a process lifetime. Subsequent allocations do not reuse IDs from previous freed allocations. IDs are only unique within a single process.

► CU_POINTER_ATTRIBUTE_IS_MANAGED:

Returns in `*data` a boolean that indicates whether the pointer points to managed memory or not.

If `ptr` is not a valid CUDA pointer then CUDA_ERROR_INVALID_VALUE is returned.

► CU_POINTER_ATTRIBUTE_DEVICE_ORDINAL:

Returns in `*data` an integer representing a device ordinal of a device against which the memory was allocated or registered.

► CU_POINTER_ATTRIBUTE_IS_LEGACY_CUDA_IPC_CAPABLE:

Returns in `*data` a boolean that indicates if this pointer maps to an allocation that is suitable for cudaIpcGetMemHandle.

► CU_POINTER_ATTRIBUTE_RANGE_START_ADDR:

Returns in `*data` the starting address for the allocation referenced by the device pointer `ptr`. Note that this is not necessarily the address of the mapped region, but the address of the mappable address range `ptr` references (e.g. from cuMemAddressReserve).

► CU_POINTER_ATTRIBUTE_RANGE_SIZE:

Returns in `*data` the size for the allocation referenced by the device pointer `ptr`. Note that this is not necessarily the size of the mapped region, but the size of the mappable address range `ptr` references (e.g. from cuMemAddressReserve). To retrieve the size of the mapped region, see cuMemGetAddressRange

► CU_POINTER_ATTRIBUTE_MAPPED:

Returns in `*data` a boolean that indicates if this pointer is in a valid address range that is mapped to a backing allocation.

► CU_POINTER_ATTRIBUTE_ALLOWED_HANDLE_TYPES:

Returns a bitmask of the allowed handle types for an allocation that may be passed to cuMemExportToShareableHandle.

► [CU_POINTER_ATTRIBUTE_MEMPOOL_HANDLE:](#)

Returns in *data the handle to the mempool that the allocation was obtained from.

► [CU_POINTER_ATTRIBUTE_IS_HW_DECOMPRESS_CAPABLE:](#)

Returns in *data a boolean that indicates whether the pointer points to memory that is capable to be used for hardware accelerated decompression.

Note that for most allocations in the unified virtual address space the host and device pointer for accessing the allocation will be the same. The exceptions to this are

- user memory registered using [cuMemHostRegister](#)
- host memory allocated using [cuMemHostAlloc](#) with the [CU_MEMHOSTALLOC_WRITECOMBINED](#) flag For these types of allocation there will exist separate, disjoint host and device addresses for accessing the allocation. In particular
 - The host address will correspond to an invalid unmapped device address (which will result in an exception if accessed from the device)
 - The device address will correspond to an invalid unmapped host address (which will result in an exception if accessed from the host). For these types of allocations, querying [CU_POINTER_ATTRIBUTE_HOST_POINTER](#) and [CU_POINTER_ATTRIBUTE_DEVICE_POINTER](#) may be used to retrieve the host and device addresses from either address.



Note:

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

See also:

[cuPointerSetAttribute](#), [cuMemAlloc](#), [cuMemFree](#), [cuMemAllocHost](#), [cuMemFreeHost](#), [cuMemHostAlloc](#), [cuMemHostRegister](#), [cuMemHostUnregister](#), [cudaPointerGetAttributes](#)

CUresult cuPointerGetAttributes (unsigned int numAttributes, CUpointer_attribute *attributes, void **data, CUdeviceptr ptr)

Returns information about a pointer.

Parameters

numAttributes

- Number of attributes to query

attributes

- An array of attributes to query (numAttributes and the number of attributes in this array should match)

data

- A two-dimensional array containing pointers to memory locations where the result of each attribute query will be written to.

ptr

- Pointer to query

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Description

The supported attributes are (refer to [cuPointerGetAttribute](#) for attribute descriptions and restrictions):

- ▶ CU_POINTER_ATTRIBUTE_CONTEXT
- ▶ CU_POINTER_ATTRIBUTE_MEMORY_TYPE
- ▶ CU_POINTER_ATTRIBUTE_DEVICE_POINTER
- ▶ CU_POINTER_ATTRIBUTE_HOST_POINTER
- ▶ CU_POINTER_ATTRIBUTE_SYNC_MEMOPS
- ▶ CU_POINTER_ATTRIBUTE_BUFFER_ID
- ▶ CU_POINTER_ATTRIBUTE_IS_MANAGED
- ▶ CU_POINTER_ATTRIBUTE_DEVICE_ORDINAL
- ▶ CU_POINTER_ATTRIBUTE_RANGE_START_ADDR
- ▶ CU_POINTER_ATTRIBUTE_RANGE_SIZE
- ▶ CU_POINTER_ATTRIBUTE_MAPPED
- ▶ CU_POINTER_ATTRIBUTE_IS_LEGACY_CUDA_IPC_CAPABLE
- ▶ CU_POINTER_ATTRIBUTE_ALLOWED_HANDLE_TYPES
- ▶ CU_POINTER_ATTRIBUTE_MEMPOOL_HANDLE
- ▶ CU_POINTER_ATTRIBUTE_IS_HW_DECOMPRESS_CAPABLE

Unlike [cuPointerGetAttribute](#), this function will not return an error when the `ptr` encountered is not a valid CUDA pointer. Instead, the attributes are assigned default NULL values and `CUDA_SUCCESS` is returned.

If `ptr` was not allocated by, mapped by, or registered with a [CUcontext](#) which uses UVA (Unified Virtual Addressing), CUDA_ERROR_INVALID_CONTEXT is returned.



Note:

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

See also:

[cuPointerGetAttribute](#), [cuPointerSetAttribute](#), [cudaPointerGetAttributes](#)

CUresult cuPointerSetAttribute (const void *value, CUpointer_attribute attribute, CUdeviceptr ptr)

Set attributes on a previously allocated memory region.

Parameters

value

- Pointer to memory containing the value to be set

attribute

- Pointer attribute to set

ptr

- Pointer to a memory region allocated using CUDA memory allocation APIs

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

The supported attributes are:

► [CU_POINTER_ATTRIBUTE_SYNC_MEMOPS](#):

A boolean attribute that can either be set (1) or unset (0). When set, the region of memory that `ptr` points to is guaranteed to always synchronize memory operations that are synchronous. If there are some previously initiated synchronous memory operations that are pending when this attribute is set, the function does not return until those memory operations are complete. See further documentation in the section titled "API synchronization behavior" to learn more about cases when synchronous memory operations can exhibit asynchronous behavior. `value` will be considered as a pointer to an unsigned integer to which this attribute is to be set.



Note:

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

See also:

[cuPointerGetAttribute](#), [cuPointerGetAttributes](#), [cuMemAlloc](#), [cuMemFree](#), [cuMemAllocHost](#), [cuMemFreeHost](#), [cuMemHostAlloc](#), [cuMemHostRegister](#), [cuMemHostUnregister](#)

6.18. Stream Management

This section describes the stream management functions of the low-level CUDA driver application programming interface.

CUresult cuStreamAddCallback (CUstream hStream, CUstreamCallback callback, void *userData, unsigned int flags)

Add a callback to a compute stream.

Parameters

hStream

- 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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_SUPPORTED](#)

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 [cuLaunchHostFunc](#). Additionally, this function is not supported with [cuStreamBeginCapture](#) and [cuStreamEndCapture](#), unlike [cuLaunchHostFunc](#).

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

The callback may be passed [CUDA_SUCCESS](#) or an error code. In the event of a device error, all subsequently executed callbacks will receive an appropriate [CUresult](#).

Callbacks must not make any CUDA API calls. Attempting to use a CUDA API will result in [CUDA_ERROR_NOT_PERMITTED](#). 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 host functions and stream callbacks have executed. Thus, for example, a callback might use global attached memory even if work has been added to another stream, if the work has been ordered behind the callback 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.

See also:

[cuStreamCreate](#), [cuStreamQuery](#), [cuStreamSynchronize](#), [cuStreamWaitEvent](#), [cuStreamDestroy](#), [cuMemAllocManaged](#), [cuStreamAttachMemAsync](#), [cuLaunchHostFunc](#), [cudaStreamAddCallback](#)

CUresult cuStreamAttachMemAsync (CUstream hStream, CUdeviceptr dptr, size_t length, unsigned int flags)

Attach memory to a stream asynchronously.

Parameters

hStream

- Stream in which to enqueue the attach operation

dptr

- Pointer to memory (must be a pointer to managed memory or to a valid host-accessible region of system-allocated pageable memory)

length

- Length of memory

flags

- Must be one of [CUMemAttach_flags](#)

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_NOT_SUPPORTED](#)

Description

Enqueues an operation in `hStream` to specify stream association of `length` bytes of memory starting from `dptr`. 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.

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

- ▶ managed memory declared using the `__managed__` keyword or allocated with [cuMemAllocManaged](#).
- ▶ 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 [CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS](#).

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 host allocations, `length` must be non-zero.

The stream association is specified using `flags` which must be one of [CUMemAttach_flags](#). If the [CU_MEM_ATTACH_GLOBAL](#) flag is specified, the memory can be accessed by any stream on any device. If the [CU_MEM_ATTACH_HOST](#) 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 [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#).

If the [CU_MEM_ATTACH_SINGLE](#) flag is specified and `hStream` is associated with a device that has a zero value for the device attribute [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#), the program makes a guarantee that it will only access the memory on the device from `hStream`. 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 `hStream` 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 [cuStreamAttachMemAsync](#) 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 `hStream` 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 [cuMemAllocManaged](#). For `__managed__` variables, the default association is always `CU_MEM_ATTACH_GLOBAL`. 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:

- ▶ This function uses standard [default stream](#) semantics.
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuStreamCreate](#), [cuStreamQuery](#), [cuStreamSynchronize](#), [cuStreamWaitEvent](#), [cuStreamDestroy](#), [cuMemAllocManaged](#), [cudaStreamAttachMemAsync](#)

CUresult cuStreamBeginCapture (CUstream hStream, CUstreamCaptureMode mode)

Begins graph capture on a stream.

Parameters

hStream

- 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 [cuThreadExchangeStreamCaptureMode](#).

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Begin graph capture on `hStream`. 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 [cuStreamEndCapture](#). Capture may not be initiated if `stream` is `CU_STREAM_LEGACY`. 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 [cuStreamIsCapturing](#). A unique id representing the capture sequence may be queried via [cuStreamGetCaptureInfo](#).

If `mode` is not `CU_STREAM_CAPTURE_MODE_RELAXED`, [cuStreamEndCapture](#) 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:

[cuStreamCreate](#), [cuStreamIsCapturing](#), [cuStreamEndCapture](#), [cuThreadExchangeStreamCaptureMode](#)

CUresult cuStreamBeginCaptureToGraph (CUstream hStream, CUgraph hGraph, const CUgraphNode *dependencies, const CUgraphEdgeData *dependencyData, size_t numDependencies, CUstreamCaptureMode mode)

Begins graph capture on a stream to an existing graph.

Parameters

hStream

- Stream in which to initiate capture.

hGraph

- 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 [cuThreadExchangeStreamCaptureMode](#).

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Begin graph capture on `hStream`, placing new nodes into an existing graph. When a stream is in capture mode, all operations pushed into the stream will not be executed, but will instead be captured into `hGraph`. The graph will not be instantiable until the user calls [cuStreamEndCapture](#).

Capture may not be initiated if `stream` is `CU_STREAM_LEGACY`. 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 [cuStreamIsCapturing](#). A unique id representing the capture sequence may be queried via [cuStreamGetCaptureInfo](#).

If `mode` is not `CU_STREAM_CAPTURE_MODE_RELAXED`, [cuStreamEndCapture](#) 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:

[cuStreamBeginCapture](#), [cuStreamCreate](#), [cuStreamIsCapturing](#), [cuStreamEndCapture](#), [cuThreadExchangeStreamCaptureMode](#), [cuGraphAddNode](#),

CUresult cuStreamCopyAttributes (CUstream dst, CUstream src)

Copies attributes from source stream to destination stream.

Parameters

dst

Destination stream

src

Source stream For list of attributes see CUstreamAttrID

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

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:

[CUaccessPolicyWindow](#)

CUresult cuStreamCreate (CUstream *phStream, unsigned int Flags)

Create a stream.

Parameters

phStream

- Returned newly created stream

Flags

- Parameters for stream creation

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Creates a stream and returns a handle in `phStream`. The `Flags` argument determines behaviors of the stream.

Valid values for `Flags` are:

- ▶ [`CU\_STREAM\_DEFAULT`](#): Default stream creation flag.
- ▶ [`CU\_STREAM\_NON\_BLOCKING`](#): 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.

See also:

[cuStreamDestroy](#), [cuStreamCreateWithPriority](#), [cuGreenCtxStreamCreate](#), [cuStreamGetPriority](#), [cuStreamGetFlags](#), [cuStreamGetDevice](#), [cuStreamWaitEvent](#), [cuStreamQuery](#), [cuStreamSynchronize](#), [cuStreamAddCallback](#), [cudaStreamCreate](#), [cudaStreamCreateWithFlags](#)

CUresult cuStreamCreateWithPriority (CUstream *phStream, unsigned int flags, int priority)

Create a stream with the given priority.

Parameters

phStream

- Returned newly created stream

flags

- Flags for stream creation. See [cuStreamCreate](#) for a list of valid flags

priority

- Stream priority. Lower numbers represent higher priorities. See [cuCtxGetStreamPriorityRange](#) for more information about meaningful stream priorities that can be passed.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Creates a stream with the specified priority and returns a handle in `phStream`. 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 [cuCtxGetStreamPriorityRange](#). If the specified priority is outside the numerical range returned by [cuCtxGetStreamPriorityRange](#), 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.
- ▶ 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:

[cuStreamDestroy](#), [cuStreamCreate](#), [cuGreenCtxStreamCreate](#), [cuStreamGetPriority](#), [cuCtxGetStreamPriorityRange](#), [cuStreamGetFlags](#), [cuStreamGetDevice](#), [cuStreamWaitEvent](#), [cuStreamQuery](#), [cuStreamSynchronize](#), [cuStreamAddCallback](#), [cudaStreamCreateWithPriority](#)

CUresult cuStreamDestroy (CUstream hStream)

Destroys a stream.

Parameters

hStream

- Stream to destroy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#)

Description

Destroys the stream specified by `hStream`.

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



Note:

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

See also:

[cuStreamCreate](#), [cuStreamWaitEvent](#), [cuStreamQuery](#), [cuStreamSynchronize](#), [cuStreamAddCallback](#), [cudaStreamDestroy](#)

CUresult cuStreamEndCapture (CUstream hStream, CUgraph *phGraph)

Ends capture on a stream, returning the captured graph.

Parameters

hStream

- Stream to query

phGraph

- The captured graph

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_STREAM_CAPTURE_WRONG_THREAD](#)

Description

End capture on `hStream`, returning the captured graph via `phGraph`. Capture must have been initiated on `hStream` via a call to [cuStreamBeginCapture](#). 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 [cuStreamBeginCapture](#) was not `CU_STREAM_CAPTURE_MODE_RELAXED`, this call must be from the same thread as [cuStreamBeginCapture](#).



Note:

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

See also:

[cuStreamCreate](#), [cuStreamBeginCapture](#), [cuStreamIsCapturing](#), [cuGraphDestroy](#)

CUresult cuStreamGetAttribute (CUstream hStream, CUstreamAttrID attr, CUstreamAttrValue *value_out)

Queries stream attribute.

Parameters

hStream

attr

value_out

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#)

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:

[CUaccessPolicyWindow](#)

CUresult cuStreamGetCaptureInfo (CUstream hStream, CUstreamCaptureStatus *captureStatus_out, cuuint64_t *id_out, CUgraph *graph_out, const CUgraphNode **dependencies_out, size_t *numDependencies_out)

Query a stream's capture state.

Parameters

hStream

- 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 [cuStreamEndCapture](#). 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 [cuStreamEndCapture](#) due to direct actions on the graph do not trigger [CUDA_ERROR_STREAM_CAPTURE_UNJOINED](#).

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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_STREAM_CAPTURE_IMPLICIT](#)

Description

Query stream state related to stream capture.

If called on [CU_STREAM_LEGACY](#) (the "null stream") while a stream not created with [CU_STREAM_NON_BLOCKING](#) is capturing, returns [CUDA_ERROR_STREAM_CAPTURE_IMPLICIT](#).

Valid data (other than capture status) is returned only if both of the following are true:

- ▶ the call returns [CUDA_SUCCESS](#)
- ▶ the returned capture status is [CU_STREAM_CAPTURE_STATUS_ACTIVE](#)

**Note:**

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuStreamGetCaptureInfo_v3](#), [cuStreamBeginCapture](#), [cuStreamIsCapturing](#),
[cuStreamUpdateCaptureDependencies](#)

```
CUresult cuStreamGetCaptureInfo_v3 (CUstream hStream,
CUstreamCaptureStatus *captureStatus_out, cuuint64_t
*id_out, CUgraph *graph_out, const CUgraphNode
**dependencies_out, const CUgraphEdgeData
**edgeData_out, size_t *numDependencies_out)
```

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

Parameters

hStream

- 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 [cuStreamEndCapture](#). 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 [cuStreamEndCapture](#) due to direct actions on the graph do not trigger [CUDA_ERROR_STREAM_CAPTURE_UNJOINED](#).

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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_STREAM_CAPTURE_IMPLICIT](#), [CUDA_ERROR_LOSSY_QUERY](#)

Description

Query stream state related to stream capture.

If called on [CU_STREAM_LEGACY](#) (the "null stream") while a stream not created with [CU_STREAM_NON_BLOCKING](#) is capturing, returns [CUDA_ERROR_STREAM_CAPTURE_IMPLICIT](#).

Valid data (other than capture status) is returned only if both of the following are true:

- ▶ the call returns [CUDA_SUCCESS](#)
- ▶ the returned capture status is [CU_STREAM_CAPTURE_STATUS_ACTIVE](#)

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 [CUDA_ERROR_LOSSY_QUERY](#).



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuStreamGetCaptureInfo](#) [cuStreamBeginCapture](#), [cuStreamIsCapturing](#),
[cuStreamUpdateCaptureDependencies](#)

CUresult cuStreamGetCtx (CUstream hStream, CUcontext *pctx)

Query the context associated with a stream.

Parameters

hStream

- Handle to the stream to be queried

pctx

- Returned context associated with the stream

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_SUPPORTED](#)

Description

Returns the CUDA context that the stream is associated with.

Note there is a later version of this API, [cuStreamGetCtx_v2](#). It will supplant this version in CUDA 13.0. It is recommended to use [cuStreamGetCtx_v2](#) till then as this version will return [CUDA_ERROR_NOT_SUPPORTED](#) for streams created via the API [cuGreenCtxStreamCreate](#).

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

- ▶ a stream created via any of the CUDA driver APIs such as [cuStreamCreate](#) and [cuStreamCreateWithPriority](#), or their runtime API equivalents such as [cudaStreamCreate](#), [cudaStreamCreateWithFlags](#) and [cudaStreamCreateWithPriority](#). The returned context is the context that was active in the calling thread when the stream was created. Passing an invalid handle will result in undefined behavior.
- ▶ any of the special streams such as the NULL stream, [CU_STREAM_LEGACY](#) and [CU_STREAM_PER_THREAD](#). The runtime API equivalents of these are also accepted, which are NULL, [cudaStreamLegacy](#) and [cudaStreamPerThread](#) respectively. Specifying any of the special handles will return the context current to the calling thread. If no context is current to the calling thread, [CUDA_ERROR_INVALID_CONTEXT](#) is returned.



Note:

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

See also:

[cuStreamDestroy](#), [cuStreamCreateWithPriority](#), [cuStreamGetPriority](#), [cuStreamGetFlags](#), [cuStreamGetDevice](#), [cuStreamWaitEvent](#), [cuStreamQuery](#), [cuStreamSynchronize](#), [cuStreamAddCallback](#), [cudaStreamCreate](#), [cuStreamGetCtx_v2](#), [cudaStreamCreateWithFlags](#)

CUresult cuStreamGetCtx_v2 (CUstream hStream, CUcontext *pCtx, CUgreenCtx *pGreenCtx)

Query the contexts associated with a stream.

Parameters

hStream

- Handle to the stream to be queried

pCtx

- Returned regular context associated with the stream

pGreenCtx

- Returned green context if the stream is associated with a green context or NULL if not

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#)

Description

Returns the contexts that the stream is associated with.

If the stream is associated with a green context, the API returns the green context in `pGreenCtx` and the primary context of the associated device in `pCtx`.

If the stream is associated with a regular context, the API returns the regular context in `pCtx` and NULL in `pGreenCtx`.

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

- ▶ a stream created via any of the CUDA driver APIs such as [cuStreamCreate](#), [cuStreamCreateWithPriority](#) and [cuGreenCtxStreamCreate](#), or their runtime API equivalents such as [cudaStreamCreate](#), [cudaStreamCreateWithFlags](#) and [cudaStreamCreateWithPriority](#). Passing an invalid handle will result in undefined behavior.
- ▶ any of the special streams such as the NULL stream, [CU_STREAM_LEGACY](#) and [CU_STREAM_PER_THREAD](#). The runtime API equivalents of these are also accepted, which are NULL, [cudaStreamLegacy](#) and [cudaStreamPerThread](#) respectively. If any of the special handles are specified, the API will operate on the context current to the calling thread. If a green context (that was converted via [cuCtxFromGreenCtx\(\)](#) before setting it current) is current to the calling thread, the API will return the green context in `pGreenCtx` and the primary context of the associated device in `pCtx`. If a regular context is current, the API returns the regular context in `pCtx` and NULL in `pGreenCtx`. Note that specifying [CU_STREAM_PER_THREAD](#) or [cudaStreamPerThread](#) will return [CUDA_ERROR_INVALID_HANDLE](#) if a green context is current to the calling thread. If no context is current to the calling thread, [CUDA_ERROR_INVALID_CONTEXT](#) is returned.

**Note:**

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

See also:

[cuStreamDestroy](#), [cuStreamCreate](#), [cuStreamCreateWithPriority](#), [cuGreenCtxStreamCreate](#),
[cuStreamGetPriority](#), [cuStreamGetFlags](#), [cuStreamGetDevice](#), [cuStreamWaitEvent](#), [cuStreamQuery](#),
[cuStreamSynchronize](#), [cuStreamAddCallback](#), [cudaStreamCreate](#), [cudaStreamCreateWithFlags](#),

CUresult cuStreamGetDevice (CUstream hStream, CUdevice *device)

Returns the device handle of the stream.

Parameters

hStream

- Handle to the stream to be queried

device

- Returns the device to which a stream belongs

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Returns in *device the device handle of the stream



Note:

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

See also:

[cuStreamDestroy](#), [cuStreamCreate](#), [cuGreenCtxStreamCreate](#), [cuStreamGetFlags](#)

CUresult cuStreamGetFlags (CUstream hStream, unsigned int *flags)

Query the flags of a given stream.

Parameters

hStream

- Handle to the stream to be queried

flags

- Pointer to an unsigned integer in which the stream's flags are returned The value returned in flags is a logical 'OR' of all flags that were used while creating this stream. See [cuStreamCreate](#) for the list of valid flags

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Query the flags of a stream created using [cuStreamCreate](#), [cuStreamCreateWithPriority](#) or [cuGreenCtxStreamCreate](#) and return the flags in `flags`.



Note:

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

See also:

[cuStreamDestroy](#), [cuStreamCreate](#), [cuGreenCtxStreamCreate](#), [cuStreamGetPriority](#),
[cudaStreamGetFlags](#) [cuStreamGetDevice](#)

CUresult cuStreamGetId (CUstream hStream, unsigned long long *streamId)

Returns the unique Id associated with the stream handle supplied.

Parameters

hStream

- Handle to the stream to be queried

streamId

- Pointer to store the Id of the stream

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#)

Description

Returns in `streamId` the unique Id which is associated with the given stream handle. 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 driver APIs such as [cuStreamCreate](#) and [cuStreamCreateWithPriority](#), or their runtime API equivalents such as [cudaStreamCreate](#), [cudaStreamCreateWithFlags](#) and [cudaStreamCreateWithPriority](#). Passing an invalid handle will result in undefined behavior.

- ▶ any of the special streams such as the NULL stream, [CU_STREAM_LEGACY](#) and [CU_STREAM_PER_THREAD](#). The runtime API equivalents of these are also accepted, which are NULL, [cudaStreamLegacy](#) and [cudaStreamPerThread](#) respectively.



Note:

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

See also:

[cuStreamDestroy](#), [cuStreamCreate](#), [cuStreamGetPriority](#), [cudaStreamGetId](#)

CUresult cuStreamGetPriority (CUstream hStream, int *priority)

Query the priority of a given 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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Query the priority of a stream created using [cuStreamCreate](#), [cuStreamCreateWithPriority](#) or [cuGreenCtxStreamCreate](#) and return the priority in `priority`. Note that if the stream was created with a priority outside the numerical range returned by [cuCtxGetStreamPriorityRange](#), this function returns the clamped priority. See [cuStreamCreateWithPriority](#) for details about priority clamping.



Note:

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

See also:

[cuStreamDestroy](#), [cuStreamCreate](#), [cuStreamCreateWithPriority](#), [cuGreenCtxStreamCreate](#), [cuCtxGetStreamPriorityRange](#), [cuStreamGetFlags](#), [cuStreamGetDevice](#), [cudaStreamGetPriority](#)

CUresult cuStreamIsCapturing (CUstream hStream, CUstreamCaptureStatus *captureStatus)

Returns a stream's capture status.

Parameters

hStream

- Stream to query

captureStatus

- Returns the stream's capture status

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_STREAM_CAPTURE_IMPLICIT](#)

Description

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

- ▶ [CU_STREAM_CAPTURE_STATUS_NONE](#): The stream is not capturing.
- ▶ [CU_STREAM_CAPTURE_STATUS_ACTIVE](#): The stream is capturing.
- ▶ [CU_STREAM_CAPTURE_STATUS_INVALIDATED](#): The stream was capturing but an error has invalidated the capture sequence. The capture sequence must be terminated with [cuStreamEndCapture](#) on the stream where it was initiated in order to continue using hStream.

Note that, if this is called on [CU_STREAM_LEGACY](#) (the "null stream") while a blocking stream in the same context is capturing, it will return [CUDA_ERROR_STREAM_CAPTURE_IMPLICIT](#) and *captureStatus 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:

[cuStreamCreate](#), [cuStreamBeginCapture](#), [cuStreamEndCapture](#)

CUresult cuStreamQuery (CUstream hStream)

Determine status of a compute stream.

Parameters

hStream

- Stream to query status of

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_READY](#)

Description

Returns [CUDA_SUCCESS](#) if all operations in the stream specified by `hStream` have completed, or [CUDA_ERROR_NOT_READY](#) if not.

For the purposes of Unified Memory, a return value of [CUDA_SUCCESS](#) is equivalent to having called [cuStreamSynchronize\(\)](#).



Note:

- ▶ This function uses standard [default stream](#) semantics.
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuStreamCreate](#), [cuStreamWaitEvent](#), [cuStreamDestroy](#), [cuStreamSynchronize](#), [cuStreamAddCallback](#), [cudaStreamQuery](#)

CUresult cuStreamSetAttribute (CUstream hStream, CUstreamAttrID attr, const CUstreamAttrValue *value)

Sets stream attribute.

Parameters

hStream

attr

value

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#)

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:

[CUaccessPolicyWindow](#)

CUresult cuStreamSynchronize (CUstream hStream)

Wait until a stream's tasks are completed.

Parameters

hStream

- Stream to wait for

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#)

Description

Waits until the device has completed all operations in the stream specified by `hStream`. If the context was created with the [CU_CTX_SCHED_BLOCKING_SYNC](#) flag, the CPU 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.

See also:

[cuStreamCreate](#), [cuStreamDestroy](#), [cuStreamWaitEvent](#), [cuStreamQuery](#), [cuStreamAddCallback](#), [cudaStreamSynchronize](#)

CUresult cuStreamUpdateCaptureDependencies (CUstream hStream, CUgraphNode *dependencies, size_t numDependencies, unsigned int flags)

Update the set of dependencies in a capturing stream (11.3+).

Parameters

hStream

- The stream to update

dependencies

- The set of dependencies to add

numDependencies

- The size of the dependencies array

flags

- See above

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_ILLEGAL_STATE](#)

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 [CU_STREAM_ADD_CAPTURE_DEPENDENCIES](#) and [CU_STREAM_SET_CAPTURE_DEPENDENCIES](#). 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 [CU_STREAM_ADD_CAPTURE_DEPENDENCIES](#).

Nodes that are removed from the dependency set via this API do not result in [CUDA_ERROR_STREAM_CAPTURE_UNJOINED](#) if they are unreachable from the stream at [cuStreamEndCapture](#).

Returns [CUDA_ERROR_ILLEGAL_STATE](#) if the stream is not capturing.

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

See also:

[cuStreamBeginCapture](#), [cuStreamGetCaptureInfo](#),

CUresult cuStreamUpdateCaptureDependencies_v2 (CUstream hStream, CUgraphNode *dependencies, const CUgraphEdgeData *dependencyData, size_t numDependencies, unsigned int flags)

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

Parameters

hStream

- 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_ILLEGAL_STATE](#)

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 along with the edge data for those dependencies.

Valid flags are [CU_STREAM_ADD_CAPTURE_DEPENDENCIES](#) and [CU_STREAM_SET_CAPTURE_DEPENDENCIES](#). 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 [CU_STREAM_ADD_CAPTURE_DEPENDENCIES](#).

Nodes that are removed from the dependency set via this API do not result in [CUDA_ERROR_STREAM_CAPTURE_UNJOINED](#) if they are unreachable from the stream at [cuStreamEndCapture](#).

Returns [CUDA_ERROR_ILLEGAL_STATE](#) if the stream is not capturing.

See also:

[cuStreamBeginCapture](#), [cuStreamGetCaptureInfo](#),

CUresult cuStreamWaitEvent (CUstream hStream, CUevent hEvent, unsigned int Flags)

Make a compute stream wait on an event.

Parameters

hStream

- Stream to wait

hEvent

- Event to wait on (may not be NULL)

Flags

- See CUevent_capture_flags

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#),

Description

Makes all future work submitted to `hStream` wait for all work captured in `hEvent`. See [cuEventRecord\(\)](#) for details on what is captured by an event. The synchronization will be performed efficiently on the device when applicable. `hEvent` may be from a different context or device than `hStream`.

flags include:

- ▶ [CU_EVENT_WAIT_DEFAULT](#): Default event creation flag.
- ▶ [CU_EVENT_WAIT_EXTERNAL](#): Event is captured in the graph as an external event node when performing stream capture. This flag is invalid outside of stream capture.



Note:

- ▶ This function uses standard [default stream](#) semantics.
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuStreamCreate](#), [cuEventRecord](#), [cuStreamQuery](#), [cuStreamSynchronize](#), [cuStreamAddCallback](#), [cuStreamDestroy](#), [cudaStreamWaitEvent](#)

CUresult cuThreadExchangeStreamCaptureMode (CUstreamCaptureMode *mode)

Swaps the stream capture interaction mode for a thread.

Parameters

mode

- Pointer to mode value to swap with the current mode

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

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:

```
↑
    CUstreamCaptureMode mode = desiredMode;
    cuThreadExchangeStreamCaptureMode(&mode);
    ...
    cuThreadExchangeStreamCaptureMode(&mode); // restore previous mode
```

During stream capture (see [cuStreamBeginCapture](#)), 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 [cuStreamBeginCapture](#) 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 [cuStreamBeginCapture-cuStreamEndCapture](#) sequence. This behavior can be controlled via this API and flags to [cuStreamBeginCapture](#).

A thread's mode is one of the following:

- ▶ **CU_STREAM_CAPTURE_MODE_GLOBAL**: This is the default mode. If the local thread has an ongoing capture sequence that was not initiated with **CU_STREAM_CAPTURE_MODE_RELAXED** at [cuStreamBeginCapture](#), or if any other thread has a concurrent capture sequence initiated with **CU_STREAM_CAPTURE_MODE_GLOBAL**, this thread is prohibited from potentially unsafe API calls.
- ▶ **CU_STREAM_CAPTURE_MODE_THREAD_LOCAL**: If the local thread has an ongoing capture sequence not initiated with **CU_STREAM_CAPTURE_MODE_RELAXED**, it is prohibited from potentially unsafe API calls. Concurrent capture sequences in other threads are ignored.
- ▶ **CU_STREAM_CAPTURE_MODE_RELAXED**: 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 [cuEventQuery](#) 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:

[cuStreamBeginCapture](#)

6.19. Event Management

This section describes the event management functions of the low-level CUDA driver application programming interface.

CUresult cuEventCreate (CUevent *phEvent, unsigned int Flags)

Creates an event.

Parameters

phEvent

- Returns newly created event

Flags

- Event creation flags

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Creates an event *phEvent for the current context with the flags specified via `Flags`. Valid flags include:

- ▶ [CU_EVENT_DEFAULT](#): Default event creation flag.
- ▶ [CU_EVENT_BLOCKING_SYNC](#): Specifies that the created event should use blocking synchronization. A CPU thread that uses [cuEventSynchronize\(\)](#) to wait on an event created with this flag will block until the event has actually been recorded.

- ▶ [CU_EVENT_DISABLE_TIMING](#): Specifies that the created event does not need to record timing data. Events created with this flag specified and the [CU_EVENT_BLOCKING_SYNC](#) flag not specified will provide the best performance when used with [cuStreamWaitEvent\(\)](#) and [cuEventQuery\(\)](#).
- ▶ [CU_EVENT_INTERPROCESS](#): Specifies that the created event may be used as an interprocess event by [cuIpcGetEventHandle\(\)](#). [CU_EVENT_INTERPROCESS](#) must be specified along with [CU_EVENT_DISABLE_TIMING](#).



Note:

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

See also:

[cuEventRecord](#), [cuEventQuery](#), [cuEventSynchronize](#), [cuEventDestroy](#), [cuEventElapsedTime](#), [cudaEventCreate](#), [cudaEventCreateWithFlags](#)

CUresult cuEventDestroy (CUevent hEvent)

Destroys an event.

Parameters

hEvent

- Event to destroy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#)

Description

Destroys the event specified by `hEvent`.

An event may be destroyed before it is complete (i.e., while [cuEventQuery\(\)](#) would return [CUDA_ERROR_NOT_READY](#)). 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.

See also:

[cuEventCreate](#), [cuEventRecord](#), [cuEventQuery](#), [cuEventSynchronize](#), [cuEventElapsedTime](#), [cudaEventDestroy](#)

CUresult cuEventElapsedTime (float *pMilliseconds, CUevent hStart, CUevent hEnd)

Computes the elapsed time between two events.

Parameters

pMilliseconds

- Time between hStart and hEnd in ms

hStart

- Starting event

hEnd

- Ending event

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_READY](#), [CUDA_ERROR_UNKNOWN](#)

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 [cuEventRecord\(\)](#) 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 [cuEventRecord\(\)](#) has not been called on either event then [CUDA_ERROR_INVALID_HANDLE](#) is returned. If [cuEventRecord\(\)](#) has been called on both events but one or both of them has not yet been completed (that is, [cuEventQuery\(\)](#) would return [CUDA_ERROR_NOT_READY](#) on at least one of the events), [CUDA_ERROR_NOT_READY](#) is returned. If either event was created with the [CU_EVENT_DISABLE_TIMING](#) flag, then this function will return [CUDA_ERROR_INVALID_HANDLE](#).

Note there is a later version of this API, [cuEventElapsedTime_v2](#). It will supplant this version in CUDA 13.0, which is retained for minor version compatibility.



Note:

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

See also:

[cuEventCreate](#), [cuEventRecord](#), [cuEventQuery](#), [cuEventSynchronize](#), [cuEventDestroy](#),
[cudaEventElapsedTime](#)

CUresult cuEventElapsedTime_v2 (float *pMilliseconds, CUevent hStart, CUevent hEnd)

Computes the elapsed time between two events.

Parameters

pMilliseconds

- Time between hStart and hEnd in ms

hStart

- Starting event

hEnd

- Ending event

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_NOT_READY](#), [CUDA_ERROR_UNKNOWN](#)

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 an elapsed time calculation only and any polling for completion on the events to be compared should be done with [cuEventQuery](#) 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 [cuEventRecord\(\)](#) 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 [cuEventRecord\(\)](#) has not been called on either event then [CUDA_ERROR_INVALID_HANDLE](#) is returned. If [cuEventRecord\(\)](#) has been called on both events but one or both of them has not yet been completed (that is, [cuEventQuery\(\)](#) would return [CUDA_ERROR_NOT_READY](#) on at least one of the events), [CUDA_ERROR_NOT_READY](#) is returned. If either event was created with the [CU_EVENT_DISABLE_TIMING](#) flag, then this function will return [CUDA_ERROR_INVALID_HANDLE](#).

**Note:**

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

See also:

[cuEventCreate](#), [cuEventRecord](#), [cuEventQuery](#), [cuEventSynchronize](#), [cuEventDestroy](#),
[cudaEventElapsedTime](#)

CUresult cuEventQuery (CUevent hEvent)

Queries an event's status.

Parameters

hEvent

- Event to query

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_NOT_READY](#)

Description

Queries the status of all work currently captured by `hEvent`. See [cuEventRecord\(\)](#) for details on what is captured by an event.

Returns [CUDA_SUCCESS](#) if all captured work has been completed, or [CUDA_ERROR_NOT_READY](#) if any captured work is incomplete.

For the purposes of Unified Memory, a return value of [CUDA_SUCCESS](#) is equivalent to having called [cuEventSynchronize\(\)](#).

**Note:**

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

See also:

[cuEventCreate](#), [cuEventRecord](#), [cuEventSynchronize](#), [cuEventDestroy](#), [cuEventElapsedTime](#),
[cudaEventQuery](#)

CUresult cuEventRecord (CUevent hEvent, CUstream hStream)

Records an event.

Parameters

hEvent

- Event to record

hStream

- Stream to record event for

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Captures in `hEvent` the contents of `hStream` at the time of this call. `hEvent` and `hStream` must be from the same context otherwise [CUDA_ERROR_INVALID_HANDLE](#) is returned. Calls such as [cuEventQuery\(\)](#) or [cuStreamWaitEvent\(\)](#) will then examine or wait for completion of the work that was captured. Uses of `hStream` after this call do not modify `hEvent`. See note on default stream behavior for what is captured in the default case.

[cuEventRecord\(\)](#) can be called multiple times on the same event and will overwrite the previously captured state. Other APIs such as [cuStreamWaitEvent\(\)](#) use the most recently captured state at the time of the API call, and are not affected by later calls to [cuEventRecord\(\)](#). Before the first call to [cuEventRecord\(\)](#), an event represents an empty set of work, so for example [cuEventQuery\(\)](#) would return [CUDA_SUCCESS](#).



Note:

- ▶ This function uses standard [default stream](#) semantics.
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuEventCreate](#), [cuEventQuery](#), [cuEventSynchronize](#), [cuStreamWaitEvent](#), [cuEventDestroy](#), [cuEventElapsedTime](#), [cudaEventRecord](#), [cuEventRecordWithFlags](#)

CUresult cuEventRecordWithFlags (CUevent hEvent, CUstream hStream, unsigned int flags)

Records an event.

Parameters

hEvent

- Event to record

hStream

- Stream to record event for

flags

- See CUevent_capture_flags

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Captures in `hEvent` the contents of `hStream` at the time of this call. `hEvent` and `hStream` must be from the same context otherwise [CUDA_ERROR_INVALID_HANDLE](#) is returned. Calls such as [cuEventQuery\(\)](#) or [cuStreamWaitEvent\(\)](#) will then examine or wait for completion of the work that was captured. Uses of `hStream` after this call do not modify `hEvent`. See note on default stream behavior for what is captured in the default case.

[cuEventRecordWithFlags\(\)](#) can be called multiple times on the same event and will overwrite the previously captured state. Other APIs such as [cuStreamWaitEvent\(\)](#) use the most recently captured state at the time of the API call, and are not affected by later calls to [cuEventRecordWithFlags\(\)](#). Before the first call to [cuEventRecordWithFlags\(\)](#), an event represents an empty set of work, so for example [cuEventQuery\(\)](#) would return [CUDA_SUCCESS](#).

flags include:

- ▶ [CU_EVENT_RECORD_DEFAULT](#): Default event creation flag.
- ▶ [CU_EVENT_RECORD_EXTERNAL](#): Event is captured in the graph as an external event node when performing stream capture. This flag is invalid outside of stream capture.



Note:

- ▶ This function uses standard [default stream](#) semantics.
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuEventCreate](#), [cuEventQuery](#), [cuEventSynchronize](#), [cuStreamWaitEvent](#), [cuEventDestroy](#),
[cuEventElapsedTime](#), [cuEventRecord](#), [cudaEventRecord](#)

CUresult cuEventSynchronize (CUevent hEvent)

Waits for an event to complete.

Parameters

hEvent

- Event to wait for

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#)

Description

Waits until the completion of all work currently captured in `hEvent`. See [cuEventRecord\(\)](#) for details on what is captured by an event.

Waiting for an event that was created with the [CU_EVENT_BLOCKING_SYNC](#) flag will cause the calling CPU thread to block until the event has been completed by the device. If the [CU_EVENT_BLOCKING_SYNC](#) 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.

See also:

[cuEventCreate](#), [cuEventRecord](#), [cuEventQuery](#), [cuEventDestroy](#), [cuEventElapsedTime](#),
[cudaEventSynchronize](#)

6.20. External Resource Interoperability

This section describes the external resource interoperability functions of the low-level CUDA driver application programming interface.

CUresult cuDestroyExternalMemory (CUexternalMemory extMem)

Destroys an external memory object.

Parameters

extMem

- External memory object to be destroyed

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_HANDLE](#)

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 [cuMemFree](#) and [cuMipmappedArrayDestroy](#) respectively.



Note:

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

See also:

[cuImportExternalMemory](#), [cuExternalMemoryGetMappedBuffer](#),
[cuExternalMemoryGetMappedMipmappedArray](#)

CUresult cuDestroyExternalSemaphore (CUexternalSemaphore extSem)

Destroys an external semaphore.

Parameters

extSem

- External semaphore to be destroyed

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_HANDLE](#)

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.

See also:

[cuImportExternalSemaphore](#), [cuSignalExternalSemaphoresAsync](#), [cuWaitExternalSemaphoresAsync](#)

CUresult cuExternalMemoryGetMappedBuffer (CUdeviceptr *devPtr, CUexternalMemory extMem, const CUDA_EXTERNAL_MEMORY_BUFFER_DESC *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

[CUDA_SUCCESS](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#)

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 `CUDA_EXTERNAL_MEMORY_BUFFER_DESC` structure is defined as follows:

```
↑
    typedef struct CUDA_EXTERNAL_MEMORY_BUFFER_DESC_st {
        unsigned long long offset;
        unsigned long long size;
        unsigned int flags;
    } CUDA\_EXTERNAL\_MEMORY\_BUFFER\_DESC;
```

where [CUDA_EXTERNAL_MEMORY_BUFFER_DESC::offset](#) is the offset in the memory object where the buffer's base address is. [CUDA_EXTERNAL_MEMORY_BUFFER_DESC::size](#) is the size of the buffer. [CUDA_EXTERNAL_MEMORY_BUFFER_DESC::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 [cuMemFree](#).



Note:

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

See also:

[cuImportExternalMemory](#), [cuDestroyExternalMemory](#),
[cuExternalMemoryGetMappedMipmappedArray](#)

```
CUresult cuExternalMemoryGetMappedMipmappedArray
(CUmipmappedArray *mipmap, CUexternalMemory
extMem, const
CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC
*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

[CUDA_SUCCESS](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#)

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 `CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC` is defined as follows:

```
↑
    typedef struct CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC_st {
        unsigned long long offset;
        CUDA_ARRAY3D_DESCRIPTOR arrayDesc;
        unsigned int numLevels;
    } CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC;
```

where `CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC::offset`

is the offset in the memory object where the base level of the mipmap chain is.

`CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC::arrayDesc` describes the format, dimensions and type of the base level of the mipmap chain. For further details on these parameters, please refer to the documentation for [cuMipmappedArrayCreate](#).

Note that if the mipmapped array is bound as a color target in the graphics API, then the flag `CUDA_ARRAY3D_COLOR_ATTACHMENT` must be specified in `CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC::arrayDesc::Flags`.

`CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC::numLevels` specifies the total number of levels in the mipmap chain.

If `extMem` was imported from a handle of type

`CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF`, then

`CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC::numLevels` must be equal to 1.

The returned CUDA mipmapped array must be freed using [cuMipmappedArrayDestroy](#).



Note:

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

See also:

[cuImportExternalMemory](#), [cuDestroyExternalMemory](#), [cuExternalMemoryGetMappedBuffer](#)

CUresult cuImportExternalMemory (CUexternalMemory *extMem_out, const CUDA_EXTERNAL_MEMORY_HANDLE_DESC *memHandleDesc)

Imports an external memory object.

Parameters

extMem_out

- Returned handle to an external memory object

memHandleDesc

- Memory import handle descriptor

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_OPERATING_SYSTEM](#)

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 `CUDA_EXTERNAL_MEMORY_HANDLE_DESC` structure is defined as follows:

```
typedef struct CUDA_EXTERNAL_MEMORY_HANDLE_DESC_st {
    CUexternalMemoryHandleType type;
    union {
        int fd;
        struct {
            void *handle;
            const void *name;
        } win32;
        const void *nvSciBufObject;
    } handle;
    unsigned long long size;
    unsigned int flags;
} CUDA_EXTERNAL_MEMORY_HANDLE_DESC;
```

where [CUDA_EXTERNAL_MEMORY_HANDLE_DESC::type](#) specifies the type of handle being imported. [CUexternalMemoryHandleType](#) is defined as:

```
typedef enum CUexternalMemoryHandleType_enum {
    CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD = 1,
    CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32 = 2,
    CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT = 3,
    CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP = 4,
    CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE = 5,
    CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE = 6,
    CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE_KMT = 7,
    CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF = 8,
} CUexternalMemoryHandleType;
```

If [CUDA_EXTERNAL_MEMORY_HANDLE_DESC::type](#) is [CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD](#), then

[CUDA_EXTERNAL_MEMORY_HANDLE_DESC::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 [CUDA_EXTERNAL_MEMORY_HANDLE_DESC::type](#) is

[CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32](#), then exactly one

of [CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::handle](#) and

[CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::name](#) must not be NULL. If

[CUDA_EXTERNAL_MEMORY_HANDLE_DESC::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 [CUDA_EXTERNAL_MEMORY_HANDLE_DESC::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 [`CUDA\_EXTERNAL\_MEMORY\_HANDLE\_DESC::type`](#) is [`CU\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_OPAQUE\_WIN32\_KMT`](#), then `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::handle` must be non-NULL and `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::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 [`CUDA\_EXTERNAL\_MEMORY\_HANDLE\_DESC::type`](#) is [`CU\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_D3D12\_HEAP`](#), then exactly one of `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::handle` and `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::name` must not be NULL. If `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::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 `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::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 [`CUDA\_EXTERNAL\_MEMORY\_HANDLE\_DESC::type`](#) is [`CU\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_D3D12\_RESOURCE`](#), then exactly one of `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::handle` and `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::name` must not be NULL. If `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::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 `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::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 [`CUDA\_EXTERNAL\_MEMORY\_HANDLE\_DESC::type`](#) is [`CU\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_D3D11\_RESOURCE`](#), then `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::handle` must represent a valid shared NT handle that is returned by `IDXGIResource1::CreateSharedHandle` when referring to a `ID3D11Resource` object. If `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::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 [`CUDA\_EXTERNAL\_MEMORY\_HANDLE\_DESC::type`](#) is [`CU\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_D3D11\_RESOURCE\_KMT`](#), then `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::handle` must represent a valid shared KMT handle that is returned by `IDXGIResource::GetSharedHandle` when referring to a `ID3D11Resource` object and `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::name` must be NULL.

If [`CUDA\_EXTERNAL\_MEMORY\_HANDLE\_DESC::type`](#) is [`CU\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_NVSCIBUF`](#), then

`CUDA_EXTERNAL_MEMORY_HANDLE_DESC::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 [cuWaitExternalSemaphoresAsync](#) or [cuSignalExternalSemaphoresAsync](#) as appropriate barriers to maintain coherence between CUDA and the other drivers. See [CUDA_EXTERNAL_SEMAPHORE_SIGNAL_SKIP_NVSCIBUF_MEMSYNC](#) and [CUDA_EXTERNAL_SEMAPHORE_WAIT_SKIP_NVSCIBUF_MEMSYNC](#) for memory synchronization.

The size of the memory object must be specified in [CUDA_EXTERNAL_MEMORY_HANDLE_DESC::size](#).

Specifying the flag [CUDA_EXTERNAL_MEMORY_DEDICATED](#) in [CUDA_EXTERNAL_MEMORY_HANDLE_DESC::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 [CUDA_EXTERNAL_MEMORY_HANDLE_DESC::type](#) is one of the following: [CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE](#), [CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE](#), [CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE_KMT](#)



Note:

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

[cuDestroyExternalMemory](#), [cuExternalMemoryGetMappedBuffer](#),
[cuExternalMemoryGetMappedMipmappedArray](#)

CUresult cuImportExternalSemaphore (CUexternalSemaphore *extSem_out, const CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC *semHandleDesc)

Imports an external semaphore.

Parameters

extSem_out

- Returned handle to an external semaphore

semHandleDesc

- Semaphore import handle descriptor

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_NOT_SUPPORTED](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_OPERATING_SYSTEM](#)

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 `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC` is defined as follows:

```
typedef struct CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_st {
    CUexternalSemaphoreHandleType type;
    union {
        int fd;
        struct {
            void *handle;
            const void *name;
        } win32;
        const void* NvSciSyncObj;
    } handle;
    unsigned int flags;
} CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC;
```

where `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type` specifies the type of handle being imported. `CUexternalSemaphoreHandleType` is defined as:

```
typedef enum CUexternalSemaphoreHandleType_enum {
    CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD = 1,
    CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32 = 2,
    CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT = 3,
    CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE = 4,
    CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE = 5,
    CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC = 6,
    CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX = 7,
    CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX_KMT = 8,
    CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_TIMELINE_SEMAPHORE_FD = 9,
    CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_TIMELINE_SEMAPHORE_WIN32 = 10
} CUexternalSemaphoreHandleType;
```

If `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type` is

[CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD](#), then

`CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::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 `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type` is

[CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32](#), then exactly one

of `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::handle` and `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::name` must not be NULL. If `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::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 `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::name` is not NULL, then it must name a valid synchronization object.

If `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type` is `CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT`, then `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::handle` must be non-NULL and `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::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 `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type` is `CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE`, then exactly one of `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::handle` and `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::name` must not be NULL. If `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::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 `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::name` is not NULL, then it must name a valid synchronization object that refers to a valid `ID3D12Fence` object.

If `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type` is `CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE`, then `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::handle` represents a valid shared NT handle that is returned by `ID3D11Fence::CreateSharedHandle`. If `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::name` is not NULL, then it must name a valid synchronization object that refers to a valid `ID3D11Fence` object.

If `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type` is `CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC`, then `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::nvSciSyncObj` represents a valid `NvSciSyncObj`.

`CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX`, then `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::handle` represents a valid shared NT handle that is returned by `IDXGIResource1::CreateSharedHandle` when referring to a `IDXGIKeyedMutex` object. If `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::name` is not NULL, then it must name a valid synchronization object that refers to a valid `IDXGIKeyedMutex` object.

If `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type` is `CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX_KMT`, then `CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::handle` represents a valid shared KMT handle that is returned by `IDXGIResource::GetSharedHandle` when referring to a

IDXGIKeyedMutex object and

CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::name must be NULL.

If [CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type](#) is

[CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_TIMELINE_SEMAPHORE_FD](#), then

CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::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 [CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type](#) is

[CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_TIMELINE_SEMAPHORE_WIN32](#), then

exactly one of CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::handle and CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::name must not be NULL. If CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::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 CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::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.

See also:

[cuDestroyExternalSemaphore](#), [cuSignalExternalSemaphoresAsync](#), [cuWaitExternalSemaphoresAsync](#)

CUresult cuSignalExternalSemaphoresAsync
 (const CUexternalSemaphore *extSemArray, const
 CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS
 *paramsArray, unsigned int numExtSems, CUstream
 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

[CUDA_SUCCESS](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_SUPPORTED](#)

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:

[CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD](#),
[CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32](#),
[CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT](#) then signaling the semaphore will set it to the signaled state.

If the semaphore object is any one of the following types:

[CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE](#),
[CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE](#),
[CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_TIMELINE_SEMAPHORE_FD](#),
[CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_TIMELINE_SEMAPHORE_WIN32](#) then the semaphore will be set to the value specified in
[CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS::params::fence::value](#).

If the semaphore object is of the type

[CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC](#) this API sets [CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS::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 [CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS::params::nvSciSync::fence](#).

By default, signaling such an external semaphore object causes appropriate memory synchronization operations to be performed over all external memory objects that are imported as [CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF](#). 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 [CUDA_EXTERNAL_SEMAPHORE_SIGNAL_SKIP_NVSCIBUF_MEMSYNC](#), 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 [CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC](#), if the [NvSciSyncAttrList](#) used to create the [NvSciSyncObj](#) had not set the flags in

[cuDeviceGetNvSciSyncAttributes](#) to `CUDA_NVSCISYNC_ATTR_SIGNAL`, this API will return `CUDA_ERROR_NOT_SUPPORTED`. `NvSciSyncFence` associated with semaphore object of the type [CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC](#) 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:

[CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX](#),
[CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX_KMT](#)

then the keyed mutex will be released with the key specified in

`CUDA_EXTERNAL_SEMAPHORE_PARAMS::params::keyedmutex::key`.



Note:

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

See also:

[cuImportExternalSemaphore](#), [cuDestroyExternalSemaphore](#), [cuWaitExternalSemaphoresAsync](#)

```
CUresult cuWaitExternalSemaphoresAsync (const
CUexternalSemaphore *extSemArray, const
CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS
*paramsArray, unsigned int numExtSems, CUstream
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

CUDA_SUCCESS, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_SUPPORTED, CUDA_ERROR_TIMEOUT

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:

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD,

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32,

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT 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:

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE,

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE,

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_TIMELINE_SEMAPHORE_FD,

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_TIMELINE_SEMAPHORE_WIN32 then

waiting on the semaphore will wait until the value of the semaphore is greater than or equal to `CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS::params::fence::value`.

If the semaphore object is of the type

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC then, waiting on the semaphore will wait until the

`CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS::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 CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF. 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

CUDA_EXTERNAL_SEMAPHORE_WAIT_SKIP_NVSCIBUF_MEMSYNC, 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 CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC, if the `NvSciSyncAttrList`

used to create the `NvSciSyncObj` had not set the flags in `cuDeviceGetNvSciSyncAttributes` to `CUDA_NVSCISYNC_ATTR_WAIT`, this API will return `CUDA_ERROR_NOT_SUPPORTED`.

If the semaphore object is any one of the following types:

[`CU\_EXTERNAL\_SEMAPHORE\_HANDLE\_TYPE\_D3D11\_KEYED\_MUTEX`](#),

[`CU\_EXTERNAL\_SEMAPHORE\_HANDLE\_TYPE\_D3D11\_KEYED\_MUTEX\_KMT`](#)

then the keyed mutex will be acquired when it is released with the key specified in

`CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS::params::keyedmutex::key` or until the timeout specified by

`CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS::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.

See also:

[`cuImportExternalSemaphore`](#), [`cuDestroyExternalSemaphore`](#), [`cuSignalExternalSemaphoresAsync`](#)

6.21. Stream Memory Operations

This section describes the stream memory operations of the low-level CUDA driver application programming interface.

Support for the [`CU\_STREAM\_WAIT\_VALUE\_NOR`](#) flag can be queried with `CU_DEVICE_ATTRIBUTE_CAN_USE_STREAM_WAIT_VALUE_NOR_V2`.

Support for the [`cuStreamWriteValue64\(\)`](#) and [`cuStreamWaitValue64\(\)`](#) functions, as well as for the [`CU\_STREAM\_MEM\_OP\_WAIT\_VALUE\_64`](#) and [`CU\_STREAM\_MEM\_OP\_WRITE\_VALUE\_64`](#) flags, can be queried with `CU_DEVICE_ATTRIBUTE_CAN_USE_64_BIT_STREAM_MEM_OPS`.

Support for both [`CU\_STREAM\_WAIT\_VALUE\_FLUSH`](#) and [`CU\_STREAM\_MEM\_OP\_FLUSH\_REMOTE\_WRITES`](#) requires dedicated platform hardware features and can be queried with [`cuDeviceGetAttribute\(\)`](#) and [`CU\_DEVICE\_ATTRIBUTE\_CAN\_FLUSH\_REMOTE\_WRITES`](#).

Note that all memory pointers passed as parameters to these operations are device pointers. Where necessary a device pointer should be obtained, for example with [`cuMemHostGetDevicePointer\(\)`](#).

None of the operations accepts pointers to managed memory buffers ([`cuMemAllocManaged`](#)).



Note:

Warning: Improper use of these APIs may deadlock the application. Synchronization ordering established through these APIs is not visible to CUDA. CUDA tasks that are (even indirectly) ordered by these APIs should also have that order expressed with CUDA-visible dependencies such as events. This ensures that the scheduler does not serialize them in an improper order.

CUresult cuStreamBatchMemOp (CUstream stream, unsigned int count, CUstreamBatchMemOpParams *paramArray, unsigned int flags)

Batch operations to synchronize the stream via memory operations.

Parameters

stream

The stream to enqueue the operations in.

count

The number of operations in the array. Must be less than 256.

paramArray

The types and parameters of the individual operations.

flags

Reserved for future expansion; must be 0.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_SUPPORTED](#)

Description

This is a batch version of [cuStreamWaitValue32\(\)](#) and [cuStreamWriteValue32\(\)](#). Batching operations may avoid some performance overhead in both the API call and the device execution versus adding them to the stream in separate API calls. The operations are enqueued in the order they appear in the array.

See [CUstreamBatchMemOpType](#) for the full set of supported operations, and [cuStreamWaitValue32\(\)](#), [cuStreamWaitValue64\(\)](#), [cuStreamWriteValue32\(\)](#), and [cuStreamWriteValue64\(\)](#) for details of specific operations.

See related APIs for details on querying support for specific operations.



Note:

Warning: Improper use of this API may deadlock the application. Synchronization ordering established through this API is not visible to CUDA. CUDA tasks that are (even indirectly) ordered by this API should also have that order expressed with CUDA-visible dependencies such as events. This ensures that the scheduler does not serialize them in an improper order. For more information, see the

Stream Memory Operations section in the programming guide(<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>).



Note:

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

See also:

[cuStreamWaitValue32](#), [cuStreamWaitValue64](#), [cuStreamWriteValue32](#), [cuStreamWriteValue64](#), [cuMemHostRegister](#)

CUresult cuStreamWaitValue32 (CUstream stream, CUdeviceptr addr, cuuint32_t value, unsigned int flags)

Wait on a memory location.

Parameters

stream

The stream to synchronize on the memory location.

addr

The memory location to wait on.

value

The value to compare with the memory location.

flags

See [CUstreamWaitValue](#) flags.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_SUPPORTED](#)

Description

Enqueues a synchronization of the stream on the given memory location. Work ordered after the operation will block until the given condition on the memory is satisfied. By default, the condition is to wait for $(\text{int32_t})(*\text{addr} - \text{value}) \geq 0$, a cyclic greater-or-equal. Other condition types can be specified via flags.

If the memory was registered via [cuMemHostRegister\(\)](#), the device pointer should be obtained with [cuMemHostGetDevicePointer\(\)](#). This function cannot be used with managed memory ([cuMemAllocManaged](#)).

Support for CU_STREAM_WAIT_VALUE_NOR can be queried with [cuDeviceGetAttribute\(\)](#) and CU_DEVICE_ATTRIBUTE_CAN_USE_STREAM_WAIT_VALUE_NOR_V2.



Note:

Warning: Improper use of this API may deadlock the application. Synchronization ordering established through this API is not visible to CUDA. CUDA tasks that are (even indirectly) ordered by this API should also have that order expressed with CUDA-visible dependencies such as events. This ensures that the scheduler does not serialize them in an improper order.



Note:

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

See also:

[cuStreamWaitValue64](#), [cuStreamWriteValue32](#), [cuStreamWriteValue64](#), [cuStreamBatchMemOp](#), [cuMemHostRegister](#), [cuStreamWaitEvent](#)

CUresult cuStreamWaitValue64 (CUstream stream, CUdeviceptr addr, cuuint64_t value, unsigned int flags)

Wait on a memory location.

Parameters

stream

The stream to synchronize on the memory location.

addr

The memory location to wait on.

value

The value to compare with the memory location.

flags

See [CUstreamWaitValue_flags](#).

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_SUPPORTED](#)

Description

Enqueues a synchronization of the stream on the given memory location. Work ordered after the operation will block until the given condition on the memory is satisfied. By default, the condition is to wait for $(\text{int64_t})(*\text{addr} - \text{value}) \geq 0$, a cyclic greater-or-equal. Other condition types can be specified via `flags`.

If the memory was registered via [cuMemHostRegister\(\)](#), the device pointer should be obtained with [cuMemHostGetDevicePointer\(\)](#).

Support for this can be queried with [cuDeviceGetAttribute\(\)](#) and [CU_DEVICE_ATTRIBUTE_CAN_USE_64_BIT_STREAM_MEM_OPS](#).



Note:

Warning: Improper use of this API may deadlock the application. Synchronization ordering established through this API is not visible to CUDA. CUDA tasks that are (even indirectly) ordered by this API should also have that order expressed with CUDA-visible dependencies such as events. This ensures that the scheduler does not serialize them in an improper order.



Note:

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

See also:

[cuStreamWaitValue32](#), [cuStreamWriteValue32](#), [cuStreamWriteValue64](#), [cuStreamBatchMemOp](#), [cuMemHostRegister](#), [cuStreamWaitEvent](#)

CUresult cuStreamWriteValue32 (CUstream stream, CUdeviceptr addr, cuuint32_t value, unsigned int flags)

Write a value to memory.

Parameters

stream

The stream to do the write in.

addr

The device address to write to.

value

The value to write.

flags

See [CUstreamWriteValue](#) flags.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_SUPPORTED](#)

Description

Write a value to memory.

If the memory was registered via [cuMemHostRegister\(\)](#), the device pointer should be obtained with [cuMemHostGetDevicePointer\(\)](#). This function cannot be used with managed memory ([cuMemAllocManaged](#)).



Note:

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

See also:

[cuStreamWriteValue64](#), [cuStreamWaitValue32](#), [cuStreamWaitValue64](#), [cuStreamBatchMemOp](#), [cuMemHostRegister](#), [cuEventRecord](#)

CUresult cuStreamWriteValue64 (CUstream stream, CUdeviceptr addr, cuuint64_t value, unsigned int flags)

Write a value to memory.

Parameters

stream

The stream to do the write in.

addr

The device address to write to.

value

The value to write.

flags

See [CUstreamWriteValue_flags](#).

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_SUPPORTED](#)

Description

Write a value to memory.

If the memory was registered via [cuMemHostRegister\(\)](#), the device pointer should be obtained with [cuMemHostGetDevicePointer\(\)](#).

Support for this can be queried with [cuDeviceGetAttribute\(\)](#) and [CU_DEVICE_ATTRIBUTE_CAN_USE_64_BIT_STREAM_MEM_OPS](#).



Note:

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

See also:

[cuStreamWriteValue32](#), [cuStreamWaitValue32](#), [cuStreamWaitValue64](#), [cuStreamBatchMemOp](#), [cuMemHostRegister](#), [cuEventRecord](#)

6.22. Execution Control

This section describes the execution control functions of the low-level CUDA driver application programming interface.

CUresult cuFuncGetAttribute (int *pi, CUfunction_attribute attrib, CUfunction hfunc)

Returns information about a function.

Parameters

pi

- Returned attribute value

attrib

- Attribute requested

hfunc

- Function to query attribute of

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_FUNCTION_NOT_LOADED](#)

Description

Returns in *pi the integer value of the attribute attrib on the kernel given by hfunc. The supported attributes are:

- ▶ [CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK](#): 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.
- ▶ [CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES](#): 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.
- ▶ [CU_FUNC_ATTRIBUTE_CONST_SIZE_BYTES](#): The size in bytes of user-allocated constant memory required by this function.

- ▶ CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES: The size in bytes of local memory used by each thread of this function.
- ▶ CU_FUNC_ATTRIBUTE_NUM_REGS: The number of registers used by each thread of this function.
- ▶ CU_FUNC_ATTRIBUTE_PTX_VERSION: 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. Note that this may return the undefined value of 0 for cubins compiled prior to CUDA 3.0.
- ▶ CU_FUNC_ATTRIBUTE_BINARY_VERSION: 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. Note that this will return a value of 10 for legacy cubins that do not have a properly-encoded binary architecture version.
- ▶ CU_FUNC_CACHE_MODE_CA: The attribute to indicate whether the function has been compiled with user specified option "-Xptxas --dlcm=ca" set .
- ▶ CU_FUNC_ATTRIBUTE_MAX_DYNAMIC_SHARED_SIZE_BYTES: The maximum size in bytes of dynamically-allocated shared memory.
- ▶ CU_FUNC_ATTRIBUTE_PREFERRED_SHARED_MEMORY_CARVEOUT: Preferred shared memory-L1 cache split ratio in percent of total shared memory.
- ▶ CU_FUNC_ATTRIBUTE_CLUSTER_SIZE_MUST_BE_SET: If this attribute is set, the kernel must launch with a valid cluster size specified.
- ▶ CU_FUNC_ATTRIBUTE_REQUIRED_CLUSTER_WIDTH: The required cluster width in blocks.
- ▶ CU_FUNC_ATTRIBUTE_REQUIRED_CLUSTER_HEIGHT: The required cluster height in blocks.
- ▶ CU_FUNC_ATTRIBUTE_REQUIRED_CLUSTER_DEPTH: The required cluster depth in blocks.
- ▶ CU_FUNC_ATTRIBUTE_NON_PORTABLE_CLUSTER_SIZE_ALLOWED: Indicates 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. 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.
- ▶ CU_FUNC_ATTRIBUTE_CLUSTER_SCHEDULING_POLICY_PREFERENCE: The block scheduling policy of a function. The value type is `CUclusterSchedulingPolicy`.

With a few exceptions, function attributes may also be queried on unloaded function handles returned from `cuModuleEnumerateFunctions`. CUDA_ERROR_FUNCTION_NOT_LOADED is returned if the attribute requires a fully loaded function but the function is not loaded. The loading state of a function

may be queried using `cuFuncIsLoaded`. `cuFuncLoad` may be called to explicitly load a function before querying the following attributes that require the function to be loaded:

- ▶ [`CU\_FUNC\_ATTRIBUTE\_MAX\_THREADS\_PER\_BLOCK`](#)
- ▶ [`CU\_FUNC\_ATTRIBUTE\_CONST\_SIZE\_BYTES`](#)
- ▶ [`CU\_FUNC\_ATTRIBUTE\_MAX\_DYNAMIC\_SHARED\_SIZE\_BYTES`](#)



Note:

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

See also:

[`cuCtxGetCacheConfig`](#), [`cuCtxSetCacheConfig`](#), [`cuFuncSetCacheConfig`](#), [`cuLaunchKernel`](#),
[`cudaFuncGetAttributes`](#), [`cudaFuncSetAttribute`](#), [`cuFuncIsLoaded`](#), [`cuFuncLoad`](#), [`cuKernelGetAttribute`](#)

CUresult cuFuncGetModule (CUmodule *hmod, CUfunction hfunc)

Returns a module handle.

Parameters

hmod

- Returned module handle

hfunc

- Function to retrieve module for

Returns

[`CUDA\_SUCCESS`](#), [`CUDA\_ERROR\_DEINITIALIZED`](#), [`CUDA\_ERROR\_NOT\_INITIALIZED`](#),
[`CUDA\_ERROR\_INVALID\_CONTEXT`](#), [`CUDA\_ERROR\_INVALID\_VALUE`](#),
[`CUDA\_ERROR\_NOT\_FOUND`](#)

Description

Returns in `*hmod` the handle of the module that function `hfunc` is located in. The lifetime of the module corresponds to the lifetime of the context it was loaded in or until the module is explicitly unloaded.

The CUDA runtime manages its own modules loaded into the primary context. If the handle returned by this API refers to a module loaded by the CUDA runtime, calling [`cuModuleUnload\(\)`](#) on that module will result in undefined behavior.



Note:

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

CUresult cuFuncGetName (const char **name, CUfunction hfunc)

Returns the function name for a CUfunction handle.

Parameters

name

- The returned name of the function

hfunc

- The function handle to retrieve the name for

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

Returns in **name the function name associated with the function handle hfunc . The function name is returned as a null-terminated string. The returned name is only valid when the function handle is valid. If the module is unloaded or reloaded, one must call the API again to get the updated name. This API may return a mangled name if the function is not declared as having C linkage. If either **name or hfunc is NULL, [CUDA_ERROR_INVALID_VALUE](#) is returned.



Note:

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

CUresult cuFuncGetParamInfo (CUfunction 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

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

paramSize

- Optionally returns 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` into `func`'s list of parameters, and returns in `paramOffset` and `paramSize` the offset and size, respectively, where the parameter will reside in the device-side parameter layout. 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. `paramSize` can be set to NULL if only the parameter offset is desired.



Note:

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

See also:

[cuKernelGetParamInfo](#)

CUresult cuFuncIsLoaded (CUfunctionLoadingState *state, CUfunction function)

Returns if the function is loaded.

Parameters**state**

- returned loading state

function

- the function to check

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns in `state` the loading state of `function`.

See also:

[cuFuncLoad](#), [cuModuleEnumerateFunctions](#)

CUresult cuFuncLoad (CUfunction function)

Loads a function.

Parameters

function

- the function to load

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Finalizes function loading for `function`. Calling this API with a fully loaded function has no effect.

See also:

[cuModuleEnumerateFunctions](#), [cuFuncIsLoaded](#)

CUresult cuFuncSetAttribute (CUfunction hfunc, CUfunction_attribute attrib, int value)

Sets information about a function.

Parameters

hfunc

- Function to query attribute of

attrib

- Attribute requested

value

- The value to set

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

This call sets the value of a specified attribute `attrib` on the kernel given by `hfunc` to an integer value specified by `val`. This function returns `CUDA_SUCCESS` 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 (CUDA_ERROR_INVALID_VALUE)

Supported attributes for the `cuFuncSetAttribute` call are:

- ▶ [CU_FUNC_ATTRIBUTE_MAX_DYNAMIC_SHARED_SIZE_BYTES](#): This maximum size in bytes of dynamically-allocated shared memory. The value should contain the requested maximum size of dynamically-allocated shared memory. The sum of this value and the function attribute [CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES](#) cannot exceed the device attribute [CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK_OPTIN](#). The maximal size of requestable dynamic shared memory may differ by GPU architecture.
- ▶ [CU_FUNC_ATTRIBUTE_PREFERRED_SHARED_MEMORY_CARVEOUT](#): 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 [CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_MULTIPROCESSOR](#). This is only a hint, and the driver can choose a different ratio if required to execute the function.
- ▶ [CU_FUNC_ATTRIBUTE_REQUIRED_CLUSTER_WIDTH](#): 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 CUDA_ERROR_NOT_PERMITTED.
- ▶ [CU_FUNC_ATTRIBUTE_REQUIRED_CLUSTER_HEIGHT](#): 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 CUDA_ERROR_NOT_PERMITTED.
- ▶ [CU_FUNC_ATTRIBUTE_REQUIRED_CLUSTER_DEPTH](#): 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 CUDA_ERROR_NOT_PERMITTED.
- ▶ [CU_FUNC_ATTRIBUTE_NON_PORTABLE_CLUSTER_SIZE_ALLOWED](#): Indicates whether the function can be launched with non-portable cluster size. 1 is allowed, 0 is disallowed.
- ▶ [CU_FUNC_ATTRIBUTE_CLUSTER_SCHEDULING_POLICY_PREFERENCE](#): The block scheduling policy of a function. The value type is `CUclusterSchedulingPolicy`.



Note:

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

See also:

[cuCtxGetCacheConfig](#), [cuCtxSetCacheConfig](#), [cuFuncSetCacheConfig](#), [cuLaunchKernel](#), [cudaFuncGetAttributes](#), [cudaFuncSetAttribute](#), [cuKernelSetAttribute](#)

CUresult cuFuncSetCacheConfig (CUfunction hfunc, CUfunc_cache config)

Sets the preferred cache configuration for a device function.

Parameters

hfunc

- Kernel to configure cache for

config

- Requested cache configuration

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#)

Description

On devices where the L1 cache and shared memory use the same hardware resources, this sets through `config` the preferred cache configuration for the device function `hfunc`. This is only a preference. The driver will use the requested configuration if possible, but it is free to choose a different configuration if required to execute `hfunc`. Any context-wide preference set via [cuCtxSetCacheConfig\(\)](#) will be overridden by this per-function setting unless the per-function setting is [CU_FUNC_CACHE_PREFER_NONE](#). In that case, the current context-wide setting will be used.

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:

- ▶ [CU_FUNC_CACHE_PREFER_NONE](#): no preference for shared memory or L1 (default)
- ▶ [CU_FUNC_CACHE_PREFER_SHARED](#): prefer larger shared memory and smaller L1 cache
- ▶ [CU_FUNC_CACHE_PREFER_L1](#): prefer larger L1 cache and smaller shared memory
- ▶ [CU_FUNC_CACHE_PREFER_EQUAL](#): prefer equal sized L1 cache and shared memory



Note:

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

See also:

[cuCtxGetCacheConfig](#), [cuCtxSetCacheConfig](#), [cuFuncGetAttribute](#), [cuLaunchKernel](#), [cudaFuncSetCacheConfig](#), [cuKernelSetCacheConfig](#)

CUresult cuLaunchCooperativeKernel (CUfunction f, unsigned int gridDimX, unsigned int gridDimY, unsigned int gridDimZ, unsigned int blockDimX, unsigned int blockDimY, unsigned int blockDimZ, unsigned int sharedMemBytes, CUstream hStream, void **kernelParams)

Launches a CUDA function CUfunction or a CUDA kernel CUkernel where thread blocks can cooperate and synchronize as they execute.

Parameters

f

- Function [CUfunction](#) or Kernel [CUkernel](#) to launch

gridDimX

- Width of grid in blocks

gridDimY

- Height of grid in blocks

gridDimZ

- Depth of grid in blocks

blockDimX

- X dimension of each thread block

blockDimY

- Y dimension of each thread block

blockDimZ

- Z dimension of each thread block

sharedMemBytes

- Dynamic shared-memory size per thread block in bytes

hStream

- Stream identifier

kernelParams

- Array of pointers to kernel parameters

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_IMAGE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_LAUNCH_FAILED](#), [CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES](#), [CUDA_ERROR_LAUNCH_TIMEOUT](#), [CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING](#),

CUDA_ERROR_COOPERATIVE_LAUNCH_TOO_LARGE, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED, CUDA_ERROR_NOT_FOUND

Description

Invokes the function [CUfunction](#) or the kernel [CUkernel](#) `f` on a `gridDimX x gridDimY x gridDimZ` grid of blocks. Each block contains `blockDimX x blockDimY x blockDimZ` threads. `sharedMemBytes` sets the amount of dynamic shared memory that will be available to each thread block.

The device on which this kernel is invoked must have a non-zero value for the device attribute [CU_DEVICE_ATTRIBUTE_COOPERATIVE_LAUNCH](#).

The total number of blocks launched cannot exceed the maximum number of blocks per multiprocessor as returned by [cuOccupancyMaxActiveBlocksPerMultiprocessor](#) (or [cuOccupancyMaxActiveBlocksPerMultiprocessorWithFlags](#)) times the number of multiprocessors as specified by the device attribute [CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT](#).

The kernel cannot make use of CUDA dynamic parallelism.

Kernel parameters must be specified via `kernelParams`. If `f` has `N` parameters, then `kernelParams` needs to be an array of `N` pointers. Each of `kernelParams[0]` through `kernelParams[N-1]` must point to a region of memory from which the actual kernel 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.

Calling [cuLaunchCooperativeKernel\(\)](#) sets persistent function state that is the same as function state set through [cuLaunchKernel](#) API

When the kernel `f` is launched via [cuLaunchCooperativeKernel\(\)](#), the previous block shape, shared size and parameter info associated with `f` is overwritten.

Note that to use [cuLaunchCooperativeKernel\(\)](#), the kernel `f` must either have been compiled with toolchain version 3.2 or later so that it will contain kernel parameter information, or have no kernel parameters. If either of these conditions is not met, then [cuLaunchCooperativeKernel\(\)](#) will return [CUDA_ERROR_INVALID_IMAGE](#).

Note that the API can also be used to launch context-less kernel [CUkernel](#) by querying the handle using [cuLibraryGetKernel\(\)](#) and then passing it to the API by casting to [CUfunction](#). Here, the context to launch the kernel on will either be taken from the specified stream `hStream` or the current context in case of NULL stream.



Note:

- ▶ This function uses standard [default stream](#) semantics.
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuCtxGetCacheConfig](#), [cuCtxSetCacheConfig](#), [cuFuncSetCacheConfig](#), [cuFuncGetAttribute](#), [cuLaunchCooperativeKernelMultiDevice](#), [cudaLaunchCooperativeKernel](#), [cuLibraryGetKernel](#), [cuKernelSetCacheConfig](#), [cuKernelGetAttribute](#), [cuKernelSetAttribute](#)

CUresult cuLaunchCooperativeKernelMultiDevice (CUDA_LAUNCH_PARAMS *launchParamsList, unsigned int numDevices, unsigned int flags)

Launches CUDA 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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_IMAGE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_LAUNCH_FAILED](#), [CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES](#), [CUDA_ERROR_LAUNCH_TIMEOUT](#), [CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING](#), [CUDA_ERROR_COOPERATIVE_LAUNCH_TOO_LARGE](#), [CUDA_ERROR_SHARED_OBJECT_INIT_FAILED](#)

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 [CU_DEVICE_ATTRIBUTE_COOPERATIVE_MULTI_DEVICE_LAUNCH](#).

All kernels launched must be identical with respect to the compiled code. Note that any `__device__`, `__constant__` or `__managed__` variables present in the module that owns the kernel launched on each

device, are independently instantiated on every device. It is the application's responsibility to ensure these variables are initialized and used appropriately.

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

The streams used to launch these kernels must have been created via either [cuStreamCreate](#) or [cuStreamCreateWithPriority](#). The NULL stream or [CU_STREAM_LEGACY](#) or [CU_STREAM_PER_THREAD](#) cannot be used.

The total number of blocks launched per kernel cannot exceed the maximum number of blocks per multiprocessor as returned by [cuOccupancyMaxActiveBlocksPerMultiprocessor](#) (or [cuOccupancyMaxActiveBlocksPerMultiprocessorWithFlags](#)) times the number of multiprocessors as specified by the device attribute [CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT](#). 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 kernels cannot make use of CUDA dynamic parallelism.

The `CUDA_LAUNCH_PARAMS` structure is defined as:

```
typedef struct CUDA_LAUNCH_PARAMS_st
{
    CUfunction function;
    unsigned int gridDimX;
    unsigned int gridDimY;
    unsigned int gridDimZ;
    unsigned int blockDimX;
    unsigned int blockDimY;
    unsigned int blockDimZ;
    unsigned int sharedMemBytes;
    CUstream hStream;
    void **kernelParams;
} CUDA_LAUNCH_PARAMS;
```

where:

- ▶ [CUDA_LAUNCH_PARAMS::function](#) specifies the kernel to be launched. All functions must be identical with respect to the compiled code. Note that you can also specify context-less kernel [CUkernel](#) by querying the handle using [cuLibraryGetKernel\(\)](#) and then casting to [CUfunction](#). In this case, the context to launch the kernel on be taken from the specified stream [CUDA_LAUNCH_PARAMS::hStream](#).
- ▶ [CUDA_LAUNCH_PARAMS::gridDimX](#) is the width of the grid in blocks. This must match across all kernels launched.
- ▶ [CUDA_LAUNCH_PARAMS::gridDimY](#) is the height of the grid in blocks. This must match across all kernels launched.
- ▶ [CUDA_LAUNCH_PARAMS::gridDimZ](#) is the depth of the grid in blocks. This must match across all kernels launched.
- ▶ [CUDA_LAUNCH_PARAMS::blockDimX](#) is the X dimension of each thread block. This must match across all kernels launched.

- ▶ [CUDA_LAUNCH_PARAMS::blockDimX](#) is the X dimension of each thread block. This must match across all kernels launched.
- ▶ [CUDA_LAUNCH_PARAMS::blockDimZ](#) is the Z dimension of each thread block. This must match across all kernels launched.
- ▶ [CUDA_LAUNCH_PARAMS::sharedMemBytes](#) is the dynamic shared-memory size per thread block in bytes. This must match across all kernels launched.
- ▶ [CUDA_LAUNCH_PARAMS::hStream](#) is the handle to the stream to perform the launch in. This cannot be the NULL stream or [CU_STREAM_LEGACY](#) or [CU_STREAM_PER_THREAD](#). The CUDA context associated with this stream must match that associated with [CUDA_LAUNCH_PARAMS::function](#).
- ▶ [CUDA_LAUNCH_PARAMS::kernelParams](#) is an array of pointers to kernel parameters. If [CUDA_LAUNCH_PARAMS::function](#) has N parameters, then [CUDA_LAUNCH_PARAMS::kernelParams](#) needs to be an array of N pointers. Each of [CUDA_LAUNCH_PARAMS::kernelParams\[0\]](#) through [CUDA_LAUNCH_PARAMS::kernelParams\[N-1\]](#) must point to a region of memory from which the actual kernel 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.

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 [CUDA_COOPERATIVE_LAUNCH_MULTI_DEVICE_NO_PRE_LAUNCH_SYNC](#). 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 [CUDA_COOPERATIVE_LAUNCH_MULTI_DEVICE_NO_POST_LAUNCH_SYNC](#). 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.

Calling [cuLaunchCooperativeKernelMultiDevice\(\)](#) sets persistent function state that is the same as function state set through [cuLaunchKernel](#) API when called individually for each element in `launchParamsList`.

When kernels are launched via [cuLaunchCooperativeKernelMultiDevice\(\)](#), the previous block shape, shared size and parameter info associated with each [CUDA_LAUNCH_PARAMS::function](#) in `launchParamsList` is overwritten.

Note that to use [cuLaunchCooperativeKernelMultiDevice\(\)](#), the kernels must either have been compiled with toolchain version 3.2 or later so that it will contain kernel parameter information, or have no kernel parameters. If either of these conditions is not met, then [cuLaunchCooperativeKernelMultiDevice\(\)](#) will return [CUDA_ERROR_INVALID_IMAGE](#).

**Note:**

- ▶ This function uses standard [default stream](#) semantics.
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuCtxGetCacheConfig](#), [cuCtxSetCacheConfig](#), [cuFuncSetCacheConfig](#), [cuFuncGetAttribute](#),
[cuLaunchCooperativeKernel](#), [cudaLaunchCooperativeKernelMultiDevice](#)

CUresult cuLaunchHostFunc (CUstream hStream, CUhostFn fn, void *userData)

Enqueues a host function call in a stream.

Parameters

hStream

- Stream to enqueue function call in

fn

- The function to call once preceding stream operations are complete

userData

- User-specified data to be passed to the function

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_NOT_SUPPORTED](#)

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 [CUDA_ERROR_NOT_PERMITTED](#), but this is not required. The host function must not perform any synchronization that may depend on outstanding CUDA work not mandated to run earlier. Host functions without a mandated order (such as in independent streams) execute in undefined order and may be serialized.

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

- ▶ The stream is considered idle for the duration of the function's execution. Thus, for example, the function may always use memory attached to the stream it was enqueued in.

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

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



Note:

- ▶ This function uses standard [default stream](#) semantics.
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuStreamCreate](#), [cuStreamQuery](#), [cuStreamSynchronize](#), [cuStreamWaitEvent](#), [cuStreamDestroy](#), [cuMemAllocManaged](#), [cuStreamAttachMemAsync](#), [cuStreamAddCallback](#)

CUresult cuLaunchKernel (CUfunction f, unsigned int gridDimX, unsigned int gridDimY, unsigned int gridDimZ, unsigned int blockDimX, unsigned int blockDimY, unsigned int blockDimZ, unsigned int sharedMemBytes, CUstream hStream, void **kernelParams, void **extra)

Launches a CUDA function CUfunction or a CUDA kernel CUkernel.

Parameters

f

- Function [CUfunction](#) or Kernel [CUkernel](#) to launch

gridDimX

- Width of grid in blocks

gridDimY

- Height of grid in blocks

gridDimZ

- Depth of grid in blocks

blockDimX

- X dimension of each thread block

blockDimY

- Y dimension of each thread block

blockDimZ

- Z dimension of each thread block

sharedMemBytes

- Dynamic shared-memory size per thread block in bytes

hStream

- Stream identifier

kernelParams

- Array of pointers to kernel parameters

extra

- Extra options

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_INVALID_IMAGE, CUDA_ERROR_INVALID_VALUE,
CUDA_ERROR_LAUNCH_FAILED, CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES,
CUDA_ERROR_LAUNCH_TIMEOUT,
CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING,
CUDA_ERROR_SHARED_OBJECT_INIT_FAILED, CUDA_ERROR_NOT_FOUND

Description

Invokes the function [CUfunction](#) or the kernel [CUkernel](#) `f` on a `gridDimX` x `gridDimY` x `gridDimZ` grid of blocks. Each block contains `blockDimX` x `blockDimY` x `blockDimZ` threads. `sharedMemBytes` sets the amount of dynamic shared memory that will be available to each thread block.

Kernel parameters to `f` can be specified in one of two ways:

- 1) Kernel parameters can be specified via `kernelParams`. If `f` has `N` parameters, then `kernelParams` needs to be an array of `N` pointers. Each of `kernelParams[0]` through `kernelParams[N-1]` must point to a region of memory from which the actual kernel 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 the `extra` parameter. This places the burden on the application of knowing each kernel parameter's

size and alignment/padding within the buffer. Here is an example of using the `extra` parameter in this manner:

```
↑
size_t argBufferSize;
char argBuffer[256];

// populate argBuffer and argBufferSize

void *config[] = {
    CU_LAUNCH_PARAM_BUFFER_POINTER, argBuffer,
    CU_LAUNCH_PARAM_BUFFER_SIZE, &argBufferSize,
    CU_LAUNCH_PARAM_END
};
status = cuLaunchKernel(f, gx, gy, gz, bx, by, bz, sh, s, NULL, config);
```

The `extra` parameter exists to allow `cuLaunchKernel` 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 `f`;
- ▶ `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 `CUDA_ERROR_INVALID_VALUE` will be returned if kernel parameters are specified with both `kernelParams` and `extra` (i.e. both `kernelParams` and `extra` are non-`NULL`).

Calling `cuLaunchKernel()` invalidates the persistent function state set through the following deprecated APIs: `cuFuncSetBlockShape()`, `cuFuncSetSharedSize()`, `cuParamSetSize()`, `cuParamSeti()`, `cuParamSetf()`, `cuParamSetv()`.

Note that to use `cuLaunchKernel()`, the kernel `f` must either have been compiled with toolchain version 3.2 or later so that it will contain kernel parameter information, or have no kernel parameters. If either of these conditions is not met, then `cuLaunchKernel()` will return `CUDA_ERROR_INVALID_IMAGE`.

Note that the API can also be used to launch context-less kernel `CUkernel` by querying the handle using `cuLibraryGetKernel()` and then passing it to the API by casting to `CUfunction`. Here, the context to launch the kernel on will either be taken from the specified stream `hStream` or the current context in case of `NULL` stream.



Note:

- ▶ This function uses standard [default stream](#) semantics.
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuCtxGetCacheConfig](#), [cuCtxSetCacheConfig](#), [cuFuncSetCacheConfig](#), [cuFuncGetAttribute](#),
[cudaLaunchKernel](#), [cuLibraryGetKernel](#), [cuKernelSetCacheConfig](#), [cuKernelGetAttribute](#),
[cuKernelSetAttribute](#)

CUresult cuLaunchKernelEx (const CUlaunchConfig *config, CUfunction f, void **kernelParams, void **extra)

Launches a CUDA function CUfunction or a CUDA kernel CUKernel with launch-time configuration.

Parameters

config

- Config to launch

f

- Function [CUfunction](#) or Kernel [CUKernel](#) to launch

kernelParams

- Array of pointers to kernel parameters

extra

- Extra options

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_INVALID_IMAGE](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_LAUNCH_FAILED](#), [CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES](#),
[CUDA_ERROR_LAUNCH_TIMEOUT](#),
[CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING](#),
[CUDA_ERROR_COOPERATIVE_LAUNCH_TOO_LARGE](#),
[CUDA_ERROR_SHARED_OBJECT_INIT_FAILED](#), [CUDA_ERROR_NOT_FOUND](#)

Description

Invokes the function [CUfunction](#) or the kernel [CUKernel](#) f with the specified launch-time configuration config.

The [CUlaunchConfig](#) structure is defined as:

```
typedef struct CUlaunchConfig_st {
    unsigned int gridDimX;
    unsigned int gridDimY;
    unsigned int gridDimZ;
    unsigned int blockDimX;
    unsigned int blockDimY;
    unsigned int blockDimZ;
    unsigned int sharedMemBytes;
    CUstream hStream;
    CUlaunchAttribute *attrs;
    unsigned int numAttrs;
} CUlaunchConfig;
```

where:

- ▶ [CUlaunchConfig::gridDimX](#) is the width of the grid in blocks.
- ▶ [CUlaunchConfig::gridDimY](#) is the height of the grid in blocks.
- ▶ [CUlaunchConfig::gridDimZ](#) is the depth of the grid in blocks.
- ▶ [CUlaunchConfig::blockDimX](#) is the X dimension of each thread block.
- ▶ [CUlaunchConfig::blockDimY](#) is the Y dimension of each thread block.
- ▶ [CUlaunchConfig::blockDimZ](#) is the Z dimension of each thread block.
- ▶ [CUlaunchConfig::sharedMemBytes](#) is the dynamic shared-memory size per thread block in bytes.
- ▶ [CUlaunchConfig::hStream](#) is the handle to the stream to perform the launch in. The CUDA context associated with this stream must match that associated with function f.
- ▶ [CUlaunchConfig::attrs](#) is an array of [CUlaunchConfig::numAttrs](#) contiguous [CUlaunchAttribute](#) elements. The value of this pointer is not considered if [CUlaunchConfig::numAttrs](#) is zero. However, in that case, it is recommended to set the pointer to NULL.
- ▶ [CUlaunchConfig::numAttrs](#) is the number of attributes populating the first [CUlaunchConfig::numAttrs](#) positions of the [CUlaunchConfig::attrs](#) array.

Launch-time configuration is specified by adding entries to [CUlaunchConfig::attrs](#). Each entry is an attribute ID and a corresponding attribute value.

The [CUlaunchAttribute](#) structure is defined as:

```
typedef struct CUlaunchAttribute_st {
    CUlaunchAttributeID id;
    CUlaunchAttributeValue value;
} CUlaunchAttribute;
```

where:

- ▶ [CUlaunchAttribute::id](#) is a unique enum identifying the attribute.
- ▶ [CUlaunchAttribute::value](#) is a union that hold the attribute value.

An example of using the `config` parameter:

```
CUlaunchAttribute coopAttr = {.id = CU_LAUNCH_ATTRIBUTE_COOPERATIVE,
                              .value = 1};
CUlaunchConfig config = {... // set block and grid dimensions
                        .attrs = &coopAttr,
                        .numAttrs = 1};

cuLaunchKernelEx(&config, kernel, NULL, NULL);
```

The [CUlaunchAttributeID](#) enum is defined as:

```
typedef enum CUlaunchAttributeID enum {
    CU_LAUNCH_ATTRIBUTE_IGNORE = 0,
    CU_LAUNCH_ATTRIBUTE_ACCESS_POLICY_WINDOW = 1,
    CU_LAUNCH_ATTRIBUTE_COOPERATIVE = 2,
    CU_LAUNCH_ATTRIBUTE_SYNCHRONIZATION_POLICY = 3,
    CU_LAUNCH_ATTRIBUTE_CLUSTER_DIMENSION = 4,
    CU_LAUNCH_ATTRIBUTE_CLUSTER_SCHEDULING_POLICY_PREFERENCE = 5,
    CU_LAUNCH_ATTRIBUTE_PROGRAMMATIC_STREAM_SERIALIZATION = 6,
    CU_LAUNCH_ATTRIBUTE_PROGRAMMATIC_EVENT = 7,
    CU_LAUNCH_ATTRIBUTE_PRIORITY = 8,
    CU_LAUNCH_ATTRIBUTE_MEM_SYNC_DOMAIN_MAP = 9,
    CU_LAUNCH_ATTRIBUTE_MEM_SYNC_DOMAIN = 10,
    CU_LAUNCH_ATTRIBUTE_PREFERRED_CLUSTER_DIMENSION = 11,
```

```

    CU_LAUNCH_ATTRIBUTE_LAUNCH_COMPLETION_EVENT = 12,
    CU_LAUNCH_ATTRIBUTE_DEVICE_UPDATABLE_KERNEL_NODE = 13,
} CULaunchAttributeID;

```

and the corresponding [CULaunchAttributeValue](#) union as :

```

typedef union CULaunchAttributeValue_union {
    CUaccessPolicyWindow accessPolicyWindow;
    int cooperative;
    CUsynchronizationPolicy syncPolicy;
    struct {
        unsigned int x;
        unsigned int y;
        unsigned int z;
    } clusterDim;
    CUclusterSchedulingPolicy clusterSchedulingPolicyPreference;
    int programmaticStreamSerializationAllowed;
    struct {
        CUevent event;
        int flags;
        int triggerAtBlockStart;
    } programmaticEvent;
    int priority;
    CULaunchMemSyncDomainMap memSyncDomainMap;
    CULaunchMemSyncDomain memSyncDomain;
    struct {
        unsigned int x;
        unsigned int y;
        unsigned int z;
    } preferredClusterDim;
    struct {
        CUevent event;
        int flags;
    } launchCompletionEvent;
    struct {
        int deviceUpdatable;
        CUgraphDeviceNode devNode;
    } deviceUpdatableKernelNode;
} CULaunchAttributeValue;

```

Setting [CU_LAUNCH_ATTRIBUTE_COOPERATIVE](#) to a non-zero value causes the kernel launch to be a cooperative launch, with exactly the same usage and semantics of [cuLaunchCooperativeKernel](#).

Setting [CU_LAUNCH_ATTRIBUTE_PROGRAMMATIC_STREAM_SERIALIZATION](#) to a non-zero values causes the kernel to use programmatic means to resolve its stream dependency -- enabling the CUDA runtime to opportunistically allow the grid's execution to overlap with the previous kernel in the stream, if that kernel requests the overlap.

[CU_LAUNCH_ATTRIBUTE_PROGRAMMATIC_EVENT](#) records an event along with the kernel launch. 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 through PTX `launchdep.release` or CUDA builtin function [cudaTriggerProgrammaticLaunchCompletion\(\)](#). A trigger can also be inserted at the beginning of each block's execution if `triggerAtBlockStart` is set to non-0. Note that dependents (including the CPU thread calling [cuEventSynchronize\(\)](#)) are not guaranteed to observe the release precisely when it is released. For example, [cuEventSynchronize\(\)](#) 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. The event supplied must not be an interprocess or interop event. The event must disable timing (i.e. created with [CU_EVENT_DISABLE_TIMING](#) flag set).

[CU_LAUNCH_ATTRIBUTE_LAUNCH_COMPLETION_EVENT](#) records an event along with the kernel launch. 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, for example if they run on different GPUs.
- ▶ If B is a higher priority than A.

Exercise caution if such an ordering inversion could lead to deadlock. The event supplied must not be an interprocess or interop event. The event must disable timing (i.e. must be created with the [CU_EVENT_DISABLE_TIMING](#) flag set).

Setting [CU_LAUNCH_ATTRIBUTE_DEVICE_UPDATABLE_KERNEL_NODE](#) to 1 on a captured launch causes the resulting kernel node to be device-updatable. This attribute is specific to graphs, and passing it to a launch in a non-capturing stream results in an error. Passing a value other than 0 or 1 is not allowed.

On success, a handle will be returned via `CUlaunchAttributeValue::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](#).

Kernel nodes which are device-updatable have additional restrictions compared to regular kernel nodes. Firstly, device-updatable nodes cannot be removed from their graph via [cuGraphDestroyNode](#). Additionally, once opted-in to this functionality, a node cannot opt out, and any attempt to set the attribute to 0 will result in an error. Graphs containing one or more device-updatable node also do not allow multiple instantiation.

[CU_LAUNCH_ATTRIBUTE_PREFERRED_CLUSTER_DIMENSION](#) allows 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 [CU_LAUNCH_ATTRIBUTE_CLUSTER_DIMENSION](#) 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 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.

The effect of other attributes is consistent with their effect when set via persistent APIs.

See [cuStreamSetAttribute](#) for

- ▶ [CU_LAUNCH_ATTRIBUTE_ACCESS_POLICY_WINDOW](#)
- ▶ [CU_LAUNCH_ATTRIBUTE_SYNCHRONIZATION_POLICY](#)

See [cuFuncSetAttribute](#) for

- ▶ [CU_LAUNCH_ATTRIBUTE_CLUSTER_DIMENSION](#)
- ▶ [CU_LAUNCH_ATTRIBUTE_CLUSTER_SCHEDULING_POLICY_PREFERENCE](#)

Kernel parameters to `f` can be specified in the same ways that they can be using [cuLaunchKernel](#).

Note that the API can also be used to launch context-less kernel [CUkernel](#) by querying the handle using [cuLibraryGetKernel\(\)](#) and then passing it to the API by casting to [CUfunction](#). Here, the context to launch the kernel on will either be taken from the specified stream [CUlaunchConfig::hStream](#) or the current context in case of NULL stream.



Note:

- ▶ This function uses standard [default stream](#) semantics.
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuCtxGetCacheConfig](#), [cuCtxSetCacheConfig](#), [cuFuncSetCacheConfig](#), [cuFuncGetAttribute](#), [cudaLaunchKernel](#), [cudaLaunchKernelEx](#), [cuLibraryGetKernel](#), [cuKernelSetCacheConfig](#), [cuKernelGetAttribute](#), [cuKernelSetAttribute](#)

6.23. Execution Control [DEPRECATED]

This section describes the deprecated execution control functions of the low-level CUDA driver application programming interface.

CUresult cuFuncSetBlockShape (CUfunction hfunc, int x, int y, int z)

Sets the block-dimensions for the function.

Parameters

hfunc

- Kernel to specify dimensions of

x

- X dimension

y

- Y dimension

z

- Z dimension

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Specifies the x, y, and z dimensions of the thread blocks that are created when the kernel given by `hfunc` is launched.



Note:

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

See also:

[cuFuncSetSharedSize](#), [cuFuncSetCacheConfig](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSeti](#), [cuParamSetf](#), [cuParamSetv](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#), [cuLaunchKernel](#)

CUresult cuFuncSetSharedMemConfig (CUfunction hfunc, CUsharedconfig config)

Sets the shared memory configuration for a device function.

Parameters

hfunc

- kernel to be given a shared memory config

config

- requested shared memory configuration

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#)

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 [cuFuncSetSharedMemConfig](#) will override the context wide setting set with [cuCtxSetSharedMemConfig](#).

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:

- ▶ [CU_SHARED_MEM_CONFIG_DEFAULT_BANK_SIZE](#): use the context's shared memory configuration when launching this function.
- ▶ [CU_SHARED_MEM_CONFIG_FOUR_BYTE_BANK_SIZE](#): set shared memory bank width to be natively four bytes when launching this function.
- ▶ [CU_SHARED_MEM_CONFIG_EIGHT_BYTE_BANK_SIZE](#): set shared memory bank width to be natively eight bytes when launching this function.



Note:

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

See also:

[cuCtxGetCacheConfig](#), [cuCtxSetCacheConfig](#), [cuCtxGetSharedMemConfig](#),
[cuCtxSetSharedMemConfig](#), [cuFuncGetAttribute](#), [cuLaunchKernel](#), [cudaFuncSetSharedMemConfig](#)

CUresult cuFuncSetSharedSize (CUfunction hfunc, unsigned int bytes)

Sets the dynamic shared-memory size for the function.

Parameters

hfunc

- Kernel to specify dynamic shared-memory size for

bytes

- Dynamic shared-memory size per thread in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Sets through `bytes` the amount of dynamic shared memory that will be available to each thread block when the kernel given by `hfunc` is launched.



Note:

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

See also:

[cuFuncSetBlockShape](#), [cuFuncSetCacheConfig](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSeti](#),
[cuParamSetf](#), [cuParamSetv](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#), [cuLaunchKernel](#)

CUresult cuLaunch (CUfunction f)

Launches a CUDA function.

Parameters

f

- Kernel to launch

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_LAUNCH_FAILED](#), [CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES](#),
[CUDA_ERROR_LAUNCH_TIMEOUT](#),
[CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING](#),
[CUDA_ERROR_SHARED_OBJECT_INIT_FAILED](#)

Description

Deprecated

Invokes the kernel `f` on a 1 x 1 x 1 grid of blocks. The block contains the number of threads specified by a previous call to [cuFuncSetBlockShape\(\)](#).

The block shape, dynamic shared memory size, and parameter information must be set using [cuFuncSetBlockShape\(\)](#), [cuFuncSetSharedSize\(\)](#), [cuParamSetSize\(\)](#), [cuParamSeti\(\)](#), [cuParamSetf\(\)](#), and [cuParamSetv\(\)](#) prior to calling this function.

Launching a function via [cuLaunchKernel\(\)](#) invalidates the function's block shape, dynamic shared memory size, and parameter information. After launching via `cuLaunchKernel`, this state must be re-initialized prior to calling this function. Failure to do so results in undefined behavior.



Note:

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

See also:

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSetf](#),
[cuParamSeti](#), [cuParamSetv](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#), [cuLaunchKernel](#)

CUresult cuLaunchGrid (CUfunction f, int grid_width, int grid_height)

Launches a CUDA function.

Parameters

f

- Kernel to launch

grid_width

- Width of grid in blocks

grid_height

- Height of grid in blocks

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_LAUNCH_FAILED](#), [CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES](#), [CUDA_ERROR_LAUNCH_TIMEOUT](#), [CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING](#), [CUDA_ERROR_SHARED_OBJECT_INIT_FAILED](#)

Description

Deprecated

Invokes the kernel `f` on a `grid_width` x `grid_height` grid of blocks. Each block contains the number of threads specified by a previous call to [cuFuncSetBlockShape\(\)](#).

The block shape, dynamic shared memory size, and parameter information must be set using [cuFuncSetBlockShape\(\)](#), [cuFuncSetSharedSize\(\)](#), [cuParamSetSize\(\)](#), [cuParamSeti\(\)](#), [cuParamSetf\(\)](#), and [cuParamSetv\(\)](#) prior to calling this function.

Launching a function via [cuLaunchKernel\(\)](#) invalidates the function's block shape, dynamic shared memory size, and parameter information. After launching via `cuLaunchKernel`, this state must be re-initialized prior to calling this function. Failure to do so results in undefined behavior.



Note:

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

See also:

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSetf](#), [cuParamSeti](#), [cuParamSetv](#), [cuLaunch](#), [cuLaunchGridAsync](#), [cuLaunchKernel](#)

CUresult cuLaunchGridAsync (CUfunction f, int grid_width, int grid_height, CUstream hStream)

Launches a CUDA function.

Parameters

f

- Kernel to launch

grid_width

- Width of grid in blocks

grid_height

- Height of grid in blocks

hStream

- Stream identifier

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_LAUNCH_FAILED](#), [CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES](#), [CUDA_ERROR_LAUNCH_TIMEOUT](#), [CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING](#), [CUDA_ERROR_SHARED_OBJECT_INIT_FAILED](#)

Description

Deprecated

Invokes the kernel `f` on a `grid_width` x `grid_height` grid of blocks. Each block contains the number of threads specified by a previous call to [cuFuncSetBlockShape\(\)](#).

The block shape, dynamic shared memory size, and parameter information must be set using [cuFuncSetBlockShape\(\)](#), [cuFuncSetSharedSize\(\)](#), [cuParamSetSize\(\)](#), [cuParamSeti\(\)](#), [cuParamSetf\(\)](#), and [cuParamSetv\(\)](#) prior to calling this function.

Launching a function via [cuLaunchKernel\(\)](#) invalidates the function's block shape, dynamic shared memory size, and parameter information. After launching via `cuLaunchKernel`, this state must be re-initialized prior to calling this function. Failure to do so results in undefined behavior.



Note:

- ▶ In certain cases where cubins are created with no ABI (i.e., using `ptxas --abi-compile no`), this function may serialize kernel launches. The CUDA driver retains asynchronous behavior by growing the per-thread stack as needed per launch and not shrinking it afterwards.
- ▶ This function uses standard [default stream](#) semantics.

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

See also:

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSetf](#), [cuParamSeti](#), [cuParamSetv](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchKernel](#)

CUresult cuParamSetf (CUfunction hfunc, int offset, float value)

Adds a floating-point parameter to the function's argument list.

Parameters

hfunc

- Kernel to add parameter to

offset

- Offset to add parameter to argument list

value

- Value of parameter

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Sets a floating-point parameter that will be specified the next time the kernel corresponding to `hfunc` will be invoked. `offset` is a byte offset.



Note:

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

See also:

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSeti](#), [cuParamSetv](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#), [cuLaunchKernel](#)

CUresult cuParamSeti (CUfunction hfunc, int offset, unsigned int value)

Adds an integer parameter to the function's argument list.

Parameters

hfunc

- Kernel to add parameter to

offset

- Offset to add parameter to argument list

value

- Value of parameter

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Sets an integer parameter that will be specified the next time the kernel corresponding to `hfunc` will be invoked. `offset` is a byte offset.



Note:

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

See also:

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSetf](#), [cuParamSetv](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#), [cuLaunchKernel](#)

CUresult cuParamSetSize (CUfunction hfunc, unsigned int numbytes)

Sets the parameter size for the function.

Parameters

hfunc

- Kernel to set parameter size for

numbytes

- Size of parameter list in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Sets through `numbytes` the total size in bytes needed by the function parameters of the kernel corresponding to `hfunc`.



Note:

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

See also:

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuFuncGetAttribute](#), [cuParamSetf](#), [cuParamSeti](#), [cuParamSetv](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#), [cuLaunchKernel](#)

CUresult cuParamSetTexRef (CUfunction hfunc, int texunit, CUtexref hTexRef)

Adds a texture-reference to the function's argument list.

Parameters

hfunc

- Kernel to add texture-reference to

texunit

- Texture unit (must be [CU_PARAM_TR_DEFAULT](#))

hTexRef

- Texture-reference to add to argument list

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Makes the CUDA array or linear memory bound to the texture reference `hTexRef` available to a device program as a texture. In this version of CUDA, the texture-reference must be obtained via [cuModuleGetTexRef\(\)](#) and the `texunit` parameter must be set to [CU_PARAM_TR_DEFAULT](#).



Note:

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

CUresult cuParamSetv (CUfunction hfunc, int offset, void *ptr, unsigned int numbytes)

Adds arbitrary data to the function's argument list.

Parameters

hfunc

- Kernel to add data to

offset

- Offset to add data to argument list

ptr

- Pointer to arbitrary data

numbytes

- Size of data to copy in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Copies an arbitrary amount of data (specified in `numbytes`) from `ptr` into the parameter space of the kernel corresponding to `hfunc`. `offset` is a byte offset.



Note:

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

See also:

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSetf](#), [cuParamSeti](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#), [cuLaunchKernel](#)

6.24. Graph Management

This section describes the graph management functions of the low-level CUDA driver application programming interface.

CUresult cuDeviceGetGraphMemAttribute (CUdevice device, CUgraphMem_attribute 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

Valid attributes are:

- ▶ [CU_GRAPH_MEM_ATTR_USED_MEM_CURRENT](#): Amount of memory, in bytes, currently associated with graphs
- ▶ [CU_GRAPH_MEM_ATTR_USED_MEM_HIGH](#): High watermark of memory, in bytes, associated with graphs since the last time it was reset. High watermark can only be reset to zero.
- ▶ [CU_GRAPH_MEM_ATTR_RESERVED_MEM_CURRENT](#): Amount of memory, in bytes, currently allocated for use by the CUDA graphs asynchronous allocator.
- ▶ [CU_GRAPH_MEM_ATTR_RESERVED_MEM_HIGH](#): High watermark of memory, in bytes, currently allocated for use by the CUDA graphs asynchronous allocator.

See also:

[cuDeviceSetGraphMemAttribute](#), [cuGraphAddMemAllocNode](#), [cuGraphAddMemFreeNode](#)

CUresult cuDeviceGraphMemTrim (CUdevice 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_DEVICE](#)

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.

See also:

[cuGraphAddMemAllocNode](#), [cuGraphAddMemFreeNode](#), [cuDeviceSetGraphMemAttribute](#), [cuDeviceGetGraphMemAttribute](#)

CUresult cuDeviceSetGraphMemAttribute (CUdevice device, CUgraphMem_attribute 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

Valid attributes are:

- [CU_GRAPH_MEM_ATTR_USED_MEM_HIGH](#): High watermark of memory, in bytes, associated with graphs since the last time it was reset. High watermark can only be reset to zero.

- [CU_GRAPH_MEM_ATTR_RESERVED_MEM_HIGH](#): High watermark of memory, in bytes, currently allocated for use by the CUDA graphs asynchronous allocator.

See also:

[cuDeviceGetGraphMemAttribute](#), [cuGraphAddMemAllocNode](#), [cuGraphAddMemFreeNode](#)

CUresult cuGraphAddBatchMemOpNode (CUgraphNode *phGraphNode, CUgraph hGraph, const CUgraphNode *dependencies, size_t numDependencies, const CUDA_BATCH_MEM_OP_NODE_PARAMS *nodeParams)

Creates a batch memory operation node and adds it to a graph.

Parameters

phGraphNode

- Returns newly created node

hGraph

- Graph to which to add the node

dependencies

- Dependencies of the node

numDependencies

- Number of dependencies

nodeParams

- Parameters for the node

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_NOT_SUPPORTED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Creates a new batch memory operation node and adds it to hGraph 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 phGraphNode.

When the node is added, the paramArray inside `nodeParams` is copied and therefore it can be freed after the call returns.



Note:

Warning: Improper use of this API may deadlock the application. Synchronization ordering established through this API is not visible to CUDA. CUDA tasks that are (even indirectly) ordered by this API should also have that order expressed with CUDA-visible dependencies such as events. This ensures that the scheduler does not serialize them in an improper order. For more information, see the Stream Memory Operations section in the programming guide(<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>).



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphAddNode](#), [cuStreamBatchMemOp](#), [cuStreamWaitValue32](#), [cuStreamWriteValue32](#), [cuStreamWaitValue64](#), [cuStreamWriteValue64](#), [cuGraphBatchMemOpNodeGetParams](#), [cuGraphBatchMemOpNodeSetParams](#), [cuGraphCreate](#), [cuGraphDestroyNode](#), [cuGraphAddChildGraphNode](#), [cuGraphAddEmptyNode](#), [cuGraphAddKernelNode](#), [cuGraphAddMemcpyNode](#), [cuGraphAddMemsetNode](#)

CUresult cuGraphAddChildGraphNode (CUgraphNode *phGraphNode, CUgraph hGraph, const CUgraphNode *dependencies, size_t numDependencies, CUgraph childGraph)

Creates a child graph node and adds it to a graph.

Parameters

phGraphNode

- Returns newly created node

hGraph

- Graph to which to add the node

dependencies

- Dependencies of the node

numDependencies

- Number of dependencies

childGraph

- The graph to clone into this node

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

Creates a new node which executes an embedded graph, and adds it to `hGraph` with `numDependencies` dependencies specified via `dependencies`. 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`.

If `hGraph` contains allocation or free nodes, this call will return an error.

The node executes an embedded child graph. The child graph is cloned in this call.

**Note:**

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphAddNode](#), [cuGraphChildGraphNodeGetGraph](#), [cuGraphCreate](#), [cuGraphDestroyNode](#), [cuGraphAddEmptyNode](#), [cuGraphAddKernelNode](#), [cuGraphAddHostNode](#), [cuGraphAddMemcpyNode](#), [cuGraphAddMemsetNode](#), [cuGraphClone](#)

CUresult cuGraphAddDependencies (CUgraph hGraph, const CUgraphNode *from, const CUgraphNode *to, size_t numDependencies)

Adds dependency edges to a graph.

Parameters**hGraph**

- 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

The number of dependencies to be added is defined by `numDependencies`. Elements in `from` and `to` at corresponding indices define a dependency. Each node in `from` and `to` must belong to `hGraph`.

If `numDependencies` is 0, elements in `from` and `to` 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.

See also:

[cuGraphRemoveDependencies](#), [cuGraphGetEdges](#), [cuGraphNodeGetDependencies](#),
[cuGraphNodeGetDependentNodes](#)

CUresult cuGraphAddDependencies_v2 (CUgraph hGraph, const CUGraphNode *from, const CUGraphNode *to, const CUgraphEdgeData *edgeData, size_t numDependencies)

Adds dependency edges to a graph (12.3+).

Parameters**hGraph**

- 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

The number of dependencies to be added is defined by `numDependencies`. Elements in `from` and `to` at corresponding indices define a dependency. Each node in `from` and `to` must belong to `hGraph`.

If `numDependencies` is 0, elements in `from` and `to` 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.

See also:

[cuGraphRemoveDependencies](#), [cuGraphGetEdges](#), [cuGraphNodeGetDependencies](#),
[cuGraphNodeGetDependentNodes](#)

CUresult cuGraphAddEmptyNode (CUgraphNode *phGraphNode, CUgraph hGraph, const CUgraphNode *dependencies, size_t numDependencies)

Creates an empty node and adds it to a graph.

Parameters

phGraphNode

- Returns newly created node

hGraph

- Graph to which to add the node

dependencies

- Dependencies of the node

numDependencies

- Number of dependencies

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#),

Description

Creates a new node which performs no operation, and adds it to `hGraph` with `numDependencies` dependencies specified via `dependencies`. 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`.

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 error codes from previous, asynchronous launches.

See also:

[cuGraphAddNode](#), [cuGraphCreate](#), [cuGraphDestroyNode](#), [cuGraphAddChildGraphNode](#),
[cuGraphAddKernelNode](#), [cuGraphAddHostNode](#), [cuGraphAddMemcpyNode](#),
[cuGraphAddMemsetNode](#)

CUresult cuGraphAddEventRecordNode (CUgraphNode *phGraphNode, CUgraph hGraph, const CUgraphNode *dependencies, size_t numDependencies, CUevent event)

Creates an event record node and adds it to a graph.

Parameters

phGraphNode

- Returns newly created node

hGraph

- Graph to which to add the node

dependencies

- Dependencies of the node

numDependencies

- Number of dependencies

event

- Event for the node

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_NOT_SUPPORTED](#), [CUDA_ERROR_INVALID_VALUE](#)

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.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphAddNode](#), [cuGraphAddEventWaitNode](#), [cuEventRecordWithFlags](#), [cuStreamWaitEvent](#), [cuGraphCreate](#), [cuGraphDestroyNode](#), [cuGraphAddChildGraphNode](#), [cuGraphAddEmptyNode](#), [cuGraphAddKernelNode](#), [cuGraphAddMemcpyNode](#), [cuGraphAddMemsetNode](#)

CUresult cuGraphAddEventWaitNode (CUgraphNode *phGraphNode, CUgraph hGraph, const CUgraphNode *dependencies, size_t numDependencies, CUevent event)

Creates an event wait node and adds it to a graph.

Parameters

phGraphNode

- Returns newly created node

hGraph

- Graph to which to add the node

dependencies

- Dependencies of the node

numDependencies

- Number of dependencies

event

- Event for the node

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_NOT_SUPPORTED](#), [CUDA_ERROR_INVALID_VALUE](#)

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. `event` may be from a different context or device than the launch stream.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphAddNode](#), [cuGraphAddEventRecordNode](#), [cuEventRecordWithFlags](#), [cuStreamWaitEvent](#), [cuGraphCreate](#), [cuGraphDestroyNode](#), [cuGraphAddChildGraphNode](#), [cuGraphAddEmptyNode](#), [cuGraphAddKernelNode](#), [cuGraphAddMemcpyNode](#), [cuGraphAddMemsetNode](#)

CUresult cuGraphAddExternalSemaphoresSignalNode
(CUgraphNode *phGraphNode, CUgraph hGraph, const
CUgraphNode *dependencies, size_t numDependencies,
const CUDA_EXT_SEM_SIGNAL_NODE_PARAMS
***nodeParams)**

Creates an external semaphore signal node and adds it to a graph.

Parameters

phGraphNode

- Returns newly created node

hGraph

- Graph to which to add the node

dependencies

- Dependencies of the node

numDependencies

- Number of dependencies

nodeParams

- Parameters for the node

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_NOT_SUPPORTED, CUDA_ERROR_INVALID_VALUE

Description

Creates a new external semaphore signal node and adds it to `hGraph` 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 `phGraphNode`.

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.

See also:

[cuGraphAddNode](#), [cuGraphExternalSemaphoresSignalNodeGetParams](#),
[cuGraphExternalSemaphoresSignalNodeSetParams](#),
[cuGraphExecExternalSemaphoresSignalNodeSetParams](#), [cuGraphAddExternalSemaphoresWaitNode](#),
[cuImportExternalSemaphore](#), [cuSignalExternalSemaphoresAsync](#), [cuWaitExternalSemaphoresAsync](#),
[cuGraphCreate](#), [cuGraphDestroyNode](#), [cuGraphAddEventRecordNode](#), [cuGraphAddEventWaitNode](#),
[cuGraphAddChildGraphNode](#), [cuGraphAddEmptyNode](#), [cuGraphAddKernelNode](#),
[cuGraphAddMemcpyNode](#), [cuGraphAddMemsetNode](#)

```
CUresult cuGraphAddExternalSemaphoresWaitNode
(CUgraphNode *phGraphNode, CUgraph hGraph, const
CUgraphNode *dependencies, size_t numDependencies,
const CUDA_EXT_SEM_WAIT_NODE_PARAMS
*nodeParams)
```

Creates an external semaphore wait node and adds it to a graph.

Parameters

phGraphNode

- Returns newly created node

hGraph

- Graph to which to add the node

dependencies

- Dependencies of the node

numDependencies

- Number of dependencies

nodeParams

- Parameters for the node

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_NOT_SUPPORTED, CUDA_ERROR_INVALID_VALUE

Description

Creates a new external semaphore wait node and adds it to `hGraph` 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 `phGraphNode`.

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.

See also:

[cuGraphAddNode](#), [cuGraphExternalSemaphoresWaitNodeGetParams](#),
[cuGraphExternalSemaphoresWaitNodeSetParams](#),
[cuGraphExecExternalSemaphoresWaitNodeSetParams](#), [cuGraphAddExternalSemaphoresSignalNode](#),
[cuImportExternalSemaphore](#), [cuSignalExternalSemaphoresAsync](#), [cuWaitExternalSemaphoresAsync](#),
[cuGraphCreate](#), [cuGraphDestroyNode](#), [cuGraphAddEventRecordNode](#), [cuGraphAddEventWaitNode](#),
[cuGraphAddChildGraphNode](#), [cuGraphAddEmptyNode](#), [cuGraphAddKernelNode](#),
[cuGraphAddMemcpyNode](#), [cuGraphAddMemsetNode](#)

**CUresult cuGraphAddHostNode (CUgraphNode
*phGraphNode, CUgraph hGraph, const CUgraphNode
*dependencies, size_t numDependencies, const
CUDA_HOST_NODE_PARAMS *nodeParams)**

Creates a host execution node and adds it to a graph.

Parameters

phGraphNode

- Returns newly created node

hGraph

- Graph to which to add the node

dependencies

- Dependencies of the node

numDependencies

- Number of dependencies

nodeParams

- Parameters for the host node

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_NOT_SUPPORTED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Creates a new CPU execution node and adds it to hGraph 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 phGraphNode.

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.

See also:

[cuGraphAddNode](#), [cuLaunchHostFunc](#), [cuGraphHostNodeGetParams](#), [cuGraphHostNodeSetParams](#), [cuGraphCreate](#), [cuGraphDestroyNode](#), [cuGraphAddChildGraphNode](#), [cuGraphAddEmptyNode](#), [cuGraphAddKernelNode](#), [cuGraphAddMemcpyNode](#), [cuGraphAddMemsetNode](#)

```
CUresult cuGraphAddKernelNode (CUgraphNode
*phGraphNode, CUgraph hGraph, const CUgraphNode
*dependencies, size_t numDependencies, const
CUDA_KERNEL_NODE_PARAMS *nodeParams)
```

Creates a kernel execution node and adds it to a graph.

Parameters

phGraphNode

- Returns newly created node

hGraph

- Graph to which to add the node

dependencies

- Dependencies of the node

numDependencies

- Number of dependencies

nodeParams

- Parameters for the GPU execution node

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Creates a new kernel execution node and adds it to hGraph 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 phGraphNode.

The `CUDA_KERNEL_NODE_PARAMS` structure is defined as:

```
typedef struct CUDA_KERNEL_NODE_PARAMS_st {
    CUfunction func;
    unsigned int gridDimX;
    unsigned int gridDimY;
    unsigned int gridDimZ;
    unsigned int blockDimX;
    unsigned int blockDimY;
    unsigned int blockDimZ;
    unsigned int sharedMemBytes;
    void **kernelParams;
    void **extra;
    CUkernel kern;
    CUcontext ctx;
} CUDA_KERNEL_NODE_PARAMS;
```

When the graph is launched, the node will invoke kernel `func` on a (`gridDimX` x `gridDimY` x `gridDimZ`) grid of blocks. Each block contains (`blockDimX` x `blockDimY` x `blockDimZ`) threads.

`sharedMemBytes` 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 for non-cooperative kernels 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 `CUDA_ERROR_INVALID_VALUE` will be returned if kernel parameters are specified with both `kernelParams` and `extra` (i.e. both `kernelParams` and `extra` are non-NULL). `CUDA_ERROR_INVALID_VALUE` will be returned if `extra` is used for a cooperative kernel.

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.

See also:

[cuGraphAddNode](#), [cuLaunchKernel](#), [cuLaunchCooperativeKernel](#), [cuGraphKernelNodeGetParams](#), [cuGraphKernelNodeSetParams](#), [cuGraphCreate](#), [cuGraphDestroyNode](#), [cuGraphAddChildGraphNode](#), [cuGraphAddEmptyNode](#), [cuGraphAddHostNode](#), [cuGraphAddMemcpyNode](#), [cuGraphAddMemsetNode](#)

CUresult cuGraphAddMemAllocNode (CUgraphNode *phGraphNode, CUgraph hGraph, const CUgraphNode *dependencies, size_t numDependencies, CUDA_MEM_ALLOC_NODE_PARAMS *nodeParams)

Creates an allocation node and adds it to a graph.

Parameters

phGraphNode

- Returns newly created node

hGraph

- Graph to which to add the node

dependencies

- Dependencies of the node

numDependencies

- Number of dependencies

nodeParams

- Parameters for the node

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_NOT_SUPPORTED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Creates a new allocation node and adds it to `hGraph` 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 `phGraphNode`.

When [cuGraphAddMemAllocNode](#) 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 [cuGraphAddMemFreeNode](#), 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 [cuMemFreeAsync](#) or [cuMemFree](#);
- ▶ launching a graph with a free node for that allocation; or
- ▶ specifying [CUDA_GRAPH_INSTANTIATE_FLAG_AUTO_FREE_ON_LAUNCH](#) during instantiation, which makes each launch behave as though it called [cuMemFreeAsync](#) for every unfreed allocation.

It is not possible to free an allocation in both the owning graph and another graph. If the allocation is freed in the same graph, a free node cannot be added to another graph. If the allocation is freed in another graph, a free node can no longer be added to the owning graph.

The following restrictions apply to graphs which contain allocation and/or memory free nodes:

- ▶ Nodes and edges of the graph cannot be deleted.
- ▶ The graph cannot be used in a child node.
- ▶ Only one instantiation of the graph may exist at any point in time.
- ▶ The graph cannot be cloned.



Note:

- ▶ Graph objects are not threadsafe. [More here.](#)

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

See also:

[cuGraphAddNode](#), [cuGraphAddMemFreeNode](#), [cuGraphMemAllocNodeGetParams](#),
[cuDeviceGraphMemTrim](#), [cuDeviceGetGraphMemAttribute](#), [cuDeviceSetGraphMemAttribute](#),
[cuMemAllocAsync](#), [cuMemFreeAsync](#), [cuGraphCreate](#), [cuGraphDestroyNode](#),
[cuGraphAddChildGraphNode](#), [cuGraphAddEmptyNode](#), [cuGraphAddEventRecordNode](#),
[cuGraphAddEventWaitNode](#), [cuGraphAddExternalSemaphoresSignalNode](#),
[cuGraphAddExternalSemaphoresWaitNode](#), [cuGraphAddKernelNode](#), [cuGraphAddMemcpyNode](#),
[cuGraphAddMemsetNode](#)

**CUresult cuGraphAddMemcpyNode (CUgraphNode
 *phGraphNode, CUgraph hGraph, const CUgraphNode
 *dependencies, size_t numDependencies, const
 CUDA_MEMCPY3D *copyParams, CUcontext ctx)**

Creates a memcpy node and adds it to a graph.

Parameters

phGraphNode

- Returns newly created node

hGraph

- Graph to which to add the node

dependencies

- Dependencies of the node

numDependencies

- Number of dependencies

copyParams

- Parameters for the memory copy

ctx

- Context on which to run the node

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#)

Description

Creates a new memcpy node and adds it to hGraph with numDependencies dependencies specified via dependencies. 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.

When the graph is launched, the node will perform the memcpy described by `copyParams`. See [cuMemcpy3D\(\)](#) 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 [CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS](#). If one or more of the operands refer to managed memory, then using the memory type [CU_MEMORYTYPE_UNIFIED](#) is disallowed for those operand(s). The managed memory will be treated as residing on either the host or the device, depending on which memory type is specified.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphAddNode](#), [cuMemcpy3D](#), [cuGraphMemcpyNodeGetParams](#),
[cuGraphMemcpyNodeSetParams](#), [cuGraphCreate](#), [cuGraphDestroyNode](#),
[cuGraphAddChildGraphNode](#), [cuGraphAddEmptyNode](#), [cuGraphAddKernelNode](#),
[cuGraphAddHostNode](#), [cuGraphAddMemsetNode](#)

CUresult cuGraphAddMemFreeNode (CUgraphNode *phGraphNode, CUgraph hGraph, const CUgraphNode *dependencies, size_t numDependencies, CUdeviceptr dptr)

Creates a memory free node and adds it to a graph.

Parameters

phGraphNode

- Returns newly created node

hGraph

- Graph to which to add the node

dependencies

- Dependencies of the node

numDependencies

- Number of dependencies

dptr

- Address of memory to free

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_NOT_SUPPORTED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Creates a new memory free node and adds it to `hGraph` 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 `phGraphNode`.

[cuGraphAddMemFreeNode](#) will return [CUDA_ERROR_INVALID_VALUE](#) if the user attempts to free:

- ▶ an allocation twice in the same graph.
- ▶ an address that was not returned by an allocation node.
- ▶ an invalid address.

The following restrictions apply to graphs which contain allocation and/or memory free nodes:

- ▶ Nodes and edges of the graph cannot be deleted.
- ▶ The graph cannot be used in a child node.
- ▶ Only one instantiation of the graph may exist at any point in time.
- ▶ The graph cannot be cloned.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphAddNode](#), [cuGraphAddMemAllocNode](#), [cuGraphMemFreeNodeGetParams](#), [cuDeviceGraphMemTrim](#), [cuDeviceGetGraphMemAttribute](#), [cuDeviceSetGraphMemAttribute](#), [cuMemAllocAsync](#), [cuMemFreeAsync](#), [cuGraphCreate](#), [cuGraphDestroyNode](#), [cuGraphAddChildGraphNode](#), [cuGraphAddEmptyNode](#), [cuGraphAddEventRecordNode](#), [cuGraphAddEventWaitNode](#), [cuGraphAddExternalSemaphoresSignalNode](#), [cuGraphAddExternalSemaphoresWaitNode](#), [cuGraphAddKernelNode](#), [cuGraphAddMemcpyNode](#), [cuGraphAddMemsetNode](#)

```
CUresult cuGraphAddMemsetNode (CUgraphNode
*phGraphNode, CUgraph hGraph, const CUgraphNode
*dependencies, size_t numDependencies, const
CUDA_MEMSET_NODE_PARAMS *memsetParams,
CUcontext ctx)
```

Creates a memset node and adds it to a graph.

Parameters

phGraphNode

- Returns newly created node

hGraph

- Graph to which to add the node

dependencies

- Dependencies of the node

numDependencies

- Number of dependencies

memsetParams

- Parameters for the memory set

ctx

- Context on which to run the node

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_CONTEXT

Description

Creates a new memset node and adds it to hGraph with numDependencies dependencies specified via dependencies. 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 element size must be 1, 2, or 4 bytes. When the graph is launched, the node will perform the memset described by memsetParams.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphAddNode](#), [cuMemsetD2D32](#), [cuGraphMemsetNodeGetParams](#),
[cuGraphMemsetNodeSetParams](#), [cuGraphCreate](#), [cuGraphDestroyNode](#),
[cuGraphAddChildGraphNode](#), [cuGraphAddEmptyNode](#), [cuGraphAddKernelNode](#),
[cuGraphAddHostNode](#), [cuGraphAddMemcpyNode](#)

**CUresult cuGraphAddNode (CUgraphNode
 *phGraphNode, CUgraph hGraph, const CUgraphNode
 *dependencies, size_t numDependencies,
 CUgraphNodeParams *nodeParams)**

Adds a node of arbitrary type to a graph.

Parameters

phGraphNode

- Returns newly created node

hGraph

- Graph to which to add the node

dependencies

- Dependencies of the node

numDependencies

- Number of dependencies

nodeParams

- Specification of the node

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_CONTEXT](#),
[CUDA_ERROR_NOT_SUPPORTED](#)

Description

Creates a new node in hGraph described by nodeParams with numDependencies dependencies specified via dependencies. numDependencies may be 0. dependencies may be null if numDependencies is 0. dependencies 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.

See also:

[cuGraphCreate](#), [cuGraphNodeSetParams](#), [cuGraphExecNodeSetParams](#)

CUresult cuGraphAddNode_v2 (CUgraphNode *phGraphNode, CUgraph hGraph, const CUgraphNode *dependencies, const CUgraphEdgeData *dependencyData, size_t numDependencies, CUgraphNodeParams *nodeParams)

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

Parameters

phGraphNode

- Returns newly created node

hGraph

- Graph to which to add the node

dependencies

- 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_NOT_SUPPORTED](#)

Description

Creates a new node in hGraph described by nodeParams with numDependencies dependencies specified via dependencies. numDependencies may be 0. dependencies may be null if numDependencies is 0. dependencies 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.

See also:

[cuGraphCreate](#), [cuGraphNodeSetParams](#), [cuGraphExecNodeSetParams](#)

CUresult cuGraphBatchMemOpNodeGetParams (CUgraphNode hNode, CUDA_BATCH_MEM_OP_NODE_PARAMS *nodeParams_out)

Returns a batch mem op node's parameters.

Parameters

hNode

- Node to get the parameters for

nodeParams_out

- Pointer to return the parameters

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#)

Description

Returns the parameters of batch mem op node `hNode` in `nodeParams_out`. The `paramArray` returned in `nodeParams_out` is 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 [cuGraphBatchMemOpNodeSetParams](#) 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.

See also:

[cuStreamBatchMemOp](#), [cuGraphAddBatchMemOpNode](#), [cuGraphBatchMemOpNodeSetParams](#)

CUresult cuGraphBatchMemOpNodeSetParams
(CUgraphNode hNode, const
CUDA_BATCH_MEM_OP_NODE_PARAMS
***nodeParams)**

Sets a batch mem op node's parameters.

Parameters

hNode

- Node to set the parameters for

nodeParams

- Parameters to copy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Sets the parameters of batch mem op node `hNode` to `nodeParams`.

The `paramArray` inside `nodeParams` is copied and therefore it can be freed after the call returns.



Note:

- ▶ Graph objects are not threadsafe. [More here.](#)
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphNodeSetParams](#), [cuStreamBatchMemOp](#), [cuGraphAddBatchMemOpNode](#),
[cuGraphBatchMemOpNodeGetParams](#)

CUresult cuGraphChildGraphNodeGetGraph (CUgraphNode hNode, CUgraph *phGraph)

Gets a handle to the embedded graph of a child graph node.

Parameters

hNode

- Node to get the embedded graph for

phGraph

- Location to store a handle to the graph

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#),

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.

See also:

[cuGraphAddChildGraphNode](#), [cuGraphNodeFindInClone](#)

CUresult cuGraphClone (CUgraph *phGraphClone, CUgraph originalGraph)

Clones a graph.

Parameters

phGraphClone

- Returns newly created cloned graph

originalGraph

- Graph to clone

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

This function creates a copy of `originalGraph` and returns it in `phGraphClone`. All parameters are copied into the cloned graph. The original graph may be modified after this call without affecting the clone.

Child graph nodes in the original graph are recursively copied into the clone.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphCreate](#), [cuGraphNodeFindInClone](#)

CUresult cuGraphConditionalHandleCreate (CUgraphConditionalHandle *pHandle_out, CUgraph hGraph, CUcontext ctx, unsigned int defaultLaunchValue, unsigned int flags)

Create a conditional handle.

Parameters

pHandle_out

- Pointer used to return the handle to the caller.

hGraph

- Graph which will contain the conditional node using this handle.

ctx

- Context for the handle and associated conditional node.

defaultLaunchValue

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

flags

- Currently must be `CU_GRAPH_COND_ASSIGN_DEFAULT` 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.

Handles can only be set from the context with which they are associated.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphAddNode](#)

CUresult cuGraphCreate (CUgraph *phGraph, unsigned int flags)

Creates a graph.

Parameters

phGraph

- Returns newly created graph

flags

- Graph creation flags, must be 0

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Creates an empty graph, which is returned via `phGraph`.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).

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

See also:

[cuGraphAddChildGraphNode](#), [cuGraphAddEmptyNode](#), [cuGraphAddKernelNode](#),
[cuGraphAddHostNode](#), [cuGraphAddMemcpyNode](#), [cuGraphAddMemsetNode](#), [cuGraphInstantiate](#),
[cuGraphDestroy](#), [cuGraphGetNodes](#), [cuGraphGetRootNodes](#), [cuGraphGetEdges](#), [cuGraphClone](#)

CUresult cuGraphDebugDotPrint (CUgraph hGraph, const char *path, unsigned int flags)

Write a DOT file describing graph structure.

Parameters

hGraph

- The graph to create a DOT file from

path

- The path to write the DOT file to

flags

- Flags from CUgraphDebugDot_flags for specifying which additional node information to write

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OPERATING_SYSTEM](#)

Description

Using the provided `hGraph`, 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.

CUresult cuGraphDestroy (CUgraph hGraph)

Destroys a graph.

Parameters

hGraph

- Graph to destroy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#)

Description

Destroys the graph specified by `hGraph`, 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.

See also:

[cuGraphCreate](#)

CUresult cuGraphDestroyNode (CUgraphNode hNode)

Remove a node from the graph.

Parameters

hNode

- Node to remove

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Removes `hNode` from its graph. This operation also severs any dependencies of other nodes on `hNode` and vice versa.

Nodes which belong to a graph which contains allocation or free nodes cannot be destroyed. 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.

See also:

[cuGraphAddChildGraphNode](#), [cuGraphAddEmptyNode](#), [cuGraphAddKernelNode](#),
[cuGraphAddHostNode](#), [cuGraphAddMemcpyNode](#), [cuGraphAddMemsetNode](#)

CUresult cuGraphEventRecordNodeGetEvent (CUgraphNode hNode, CUevent *event_out)

Returns the event associated with an event record node.

Parameters

hNode

- Node to get the event for

event_out

- Pointer to return the event

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#)

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.

See also:

[cuGraphAddEventRecordNode](#), [cuGraphEventRecordNodeSetEvent](#),
[cuGraphEventWaitNodeGetEvent](#), [cuEventRecordWithFlags](#), [cuStreamWaitEvent](#)

CUresult cuGraphEventRecordNodeSetEvent (CUgraphNode hNode, CUevent event)

Sets an event record node's event.

Parameters

hNode

- Node to set the event for

event

- Event to use

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

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.

See also:

[cuGraphNodeSetParams](#), [cuGraphAddEventRecordNode](#), [cuGraphEventRecordNodeGetEvent](#), [cuGraphEventWaitNodeSetEvent](#), [cuEventRecordWithFlags](#), [cuStreamWaitEvent](#)

CUresult cuGraphEventWaitNodeGetEvent (CUgraphNode hNode, CUevent *event_out)

Returns the event associated with an event wait node.

Parameters

hNode

- Node to get the event for

event_out

- Pointer to return the event

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

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.

See also:

[cuGraphAddEventWaitNode](#), [cuGraphEventWaitNodeSetEvent](#), [cuGraphEventRecordNodeGetEvent](#), [cuEventRecordWithFlags](#), [cuStreamWaitEvent](#)

CUresult cuGraphEventWaitNodeSetEvent (CUgraphNode hNode, CUevent event)

Sets an event wait node's event.

Parameters

hNode

- Node to set the event for

event

- Event to use

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

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.

See also:

[cuGraphNodeSetParams](#), [cuGraphAddEventWaitNode](#), [cuGraphEventWaitNodeGetEvent](#), [cuGraphEventRecordNodeSetEvent](#), [cuEventRecordWithFlags](#), [cuStreamWaitEvent](#)

CUresult cuGraphExecBatchMemOpNodeSetParams (CUgraphExec hGraphExec, CUgraphNode hNode, const CUDA_BATCH_MEM_OP_NODE_PARAMS *nodeParams)

Sets the parameters for a batch mem op node in the given graphExec.

Parameters

hGraphExec

- The executable graph in which to set the specified node

hNode

- Batch mem op node from the graph from which graphExec was instantiated

nodeParams

- Updated Parameters to set

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE,

Description

Sets the parameters of a batch mem op 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 following fields on operations may be modified on an executable graph:

`op.waitValue.address` `op.waitValue.value[64]` `op.waitValue.flags` bits corresponding to wait type (i.e. `CU_STREAM_WAIT_VALUE_FLUSH` bit cannot be modified) `op.writeValue.address` `op.writeValue.value[64]`

Other fields, such as the context, count or type of operations, and other types of operations such as membars, may not be modified.

`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.

The `paramArray` inside `nodeParams` is copied and therefore it can be freed after the call returns.



Note:

- ▶ Graph objects are not threadsafe. [More here.](#)
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphExecNodeSetParams](#), [cuStreamBatchMemOp](#), [cuGraphAddBatchMemOpNode](#),
[cuGraphBatchMemOpNodeGetParams](#), [cuGraphBatchMemOpNodeSetParams](#), [cuGraphInstantiate](#)

CUresult cuGraphExecChildGraphNodeSetParams (CUgraphExec hGraphExec, CUGraphNode hNode, CUgraph 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

hNode

- Host node from the graph which was used to instantiate graphExec

childGraph

- The graph supplying the updated parameters

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

Updates the work represented by hNode in hGraphExec as though the nodes contained in hNode's graph had the parameters contained in childGraph's nodes at instantiation. hNode must remain in the graph which was used to instantiate hGraphExec. Changed edges to and from hNode are ignored.

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.

The topology of childGraph, as well as the node insertion order, must match that of the graph contained in hNode. See [cuGraphExecUpdate\(\)](#) 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.

See also:

[cuGraphExecNodeSetParams](#), [cuGraphAddChildGraphNode](#),
[cuGraphChildGraphNodeGetGraph](#), [cuGraphExecKernelNodeSetParams](#),
[cuGraphExecMemcpyNodeSetParams](#), [cuGraphExecMemsetNodeSetParams](#),
[cuGraphExecHostNodeSetParams](#), [cuGraphExecEventRecordNodeSetEvent](#),
[cuGraphExecEventWaitNodeSetEvent](#), [cuGraphExecExternalSemaphoresSignalNodeSetParams](#),
[cuGraphExecExternalSemaphoresWaitNodeSetParams](#), [cuGraphExecUpdate](#), [cuGraphInstantiate](#)

CUresult cuGraphExecDestroy (CUgraphExec hGraphExec)

Destroys an executable graph.

Parameters

hGraphExec

- Executable graph to destroy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#)

Description

Destroys the executable graph specified by `hGraphExec`, as well as all of its executable nodes. If the executable graph is in-flight, it will not be terminated, but rather freed asynchronously on completion.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphInstantiate](#), [cuGraphUpload](#), [cuGraphLaunch](#)

CUresult cuGraphExecEventRecordNodeSetEvent (CUgraphExec hGraphExec, CUgraphNode hNode, CUevent 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),

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.

See also:

[cuGraphExecNodeSetParams](#), [cuGraphAddEventRecordNode](#),
[cuGraphEventRecordNodeGetEvent](#), [cuGraphEventWaitNodeSetEvent](#),
[cuEventRecordWithFlags](#), [cuStreamWaitEvent](#), [cuGraphExecKernelNodeSetParams](#),
[cuGraphExecMemcpyNodeSetParams](#), [cuGraphExecMemsetNodeSetParams](#),
[cuGraphExecHostNodeSetParams](#), [cuGraphExecChildGraphNodeSetParams](#),
[cuGraphExecEventWaitNodeSetEvent](#), [cuGraphExecExternalSemaphoresSignalNodeSetParams](#),
[cuGraphExecExternalSemaphoresWaitNodeSetParams](#), [cuGraphExecUpdate](#), [cuGraphInstantiate](#)

CUresult cuGraphExecEventWaitNodeSetEvent (CUgraphExec hGraphExec, CUgraphNode hNode, CUevent 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),

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.

See also:

[cuGraphExecNodeSetParams](#), [cuGraphAddEventWaitNode](#),
[cuGraphEventWaitNodeGetEvent](#), [cuGraphEventRecordNodeSetEvent](#),
[cuEventRecordWithFlags](#), [cuStreamWaitEvent](#), [cuGraphExecKernelNodeSetParams](#),
[cuGraphExecMemcpyNodeSetParams](#), [cuGraphExecMemsetNodeSetParams](#),
[cuGraphExecHostNodeSetParams](#), [cuGraphExecChildGraphNodeSetParams](#),
[cuGraphExecEventRecordNodeSetEvent](#), [cuGraphExecExternalSemaphoresSignalNodeSetParams](#),
[cuGraphExecExternalSemaphoresWaitNodeSetParams](#), [cuGraphExecUpdate](#), [cuGraphInstantiate](#)

CUresult

cuGraphExecExternalSemaphoresSignalNodeSetParams
 (CUgraphExec hGraphExec, CUgraphNode hNode,
 const CUDA_EXT_SEM_SIGNAL_NODE_PARAMS
 *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

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE,

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.

See also:

[cuGraphExecNodeSetParams](#), [cuGraphAddExternalSemaphoresSignalNode](#),
[cuImportExternalSemaphore](#), [cuSignalExternalSemaphoresAsync](#), [cuWaitExternalSemaphoresAsync](#),
[cuGraphExecKernelNodeSetParams](#), [cuGraphExecMemcpyNodeSetParams](#),

[cuGraphExecMemsetNodeSetParams](#), [cuGraphExecHostNodeSetParams](#),
[cuGraphExecChildGraphNodeSetParams](#), [cuGraphExecEventRecordNodeSetEvent](#),
[cuGraphExecEventWaitNodeSetEvent](#), [cuGraphExecExternalSemaphoresWaitNodeSetParams](#),
[cuGraphExecUpdate](#), [cuGraphInstantiate](#)

CUresult

cuGraphExecExternalSemaphoresWaitNodeSetParams
 (CUgraphExec hGraphExec, CUgraphNode hNode,
 const CUDA_EXT_SEM_WAIT_NODE_PARAMS
 *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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),

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.

See also:

[cuGraphExecNodeSetParams](#), [cuGraphAddExternalSemaphoresWaitNode](#),
[cuImportExternalSemaphore](#), [cuSignalExternalSemaphoresAsync](#), [cuWaitExternalSemaphoresAsync](#),
[cuGraphExecKernelNodeSetParams](#), [cuGraphExecMemcpyNodeSetParams](#),
[cuGraphExecMemsetNodeSetParams](#), [cuGraphExecHostNodeSetParams](#),
[cuGraphExecChildGraphNodeSetParams](#), [cuGraphExecEventRecordNodeSetEvent](#),
[cuGraphExecEventWaitNodeSetEvent](#), [cuGraphExecExternalSemaphoresSignalNodeSetParams](#),
[cuGraphExecUpdate](#), [cuGraphInstantiate](#)

CUresult cuGraphExecGetFlags (CUgraphExec hGraphExec, cuuint64_t *flags)

Query the instantiation flags of an executable graph.

Parameters

hGraphExec

- The executable graph to query

flags

- Returns the instantiation flags

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

Returns the flags that were passed to instantiation for the given executable graph.

[CUDA_GRAPH_INSTANTIATE_FLAG_UPLOAD](#) 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.

See also:

[cuGraphInstantiate](#), [cuGraphInstantiateWithParams](#)

CUresult cuGraphExecHostNodeSetParams (CUgraphExec hGraphExec, CUgraphNode hNode, const CUDA_HOST_NODE_PARAMS *nodeParams)

Sets the parameters for a host node in the given graphExec.

Parameters

hGraphExec

- The executable graph in which to set the specified node

hNode

- Host node from the graph which was used to instantiate graphExec

nodeParams

- The updated parameters to set

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

Updates the work represented by hNode in hGraphExec as though hNode had contained nodeParams at instantiation. hNode must remain in the graph which was used to instantiate hGraphExec. Changed edges to and from hNode are ignored.

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.

See also:

[cuGraphExecNodeSetParams](#), [cuGraphAddHostNode](#), [cuGraphHostNodeSetParams](#),
[cuGraphExecKernelNodeSetParams](#), [cuGraphExecMemcpyNodeSetParams](#),
[cuGraphExecMemsetNodeSetParams](#), [cuGraphExecChildGraphNodeSetParams](#),
[cuGraphExecEventRecordNodeSetEvent](#), [cuGraphExecEventWaitNodeSetEvent](#),
[cuGraphExecExternalSemaphoresSignalNodeSetParams](#),
[cuGraphExecExternalSemaphoresWaitNodeSetParams](#), [cuGraphExecUpdate](#), [cuGraphInstantiate](#)

CUresult cuGraphExecKernelNodeSetParams (CUgraphExec hGraphExec, CUgraphNode hNode, const CUDA_KERNEL_NODE_PARAMS *nodeParams)

Sets the parameters for a kernel node in the given graphExec.

Parameters

hGraphExec

- The executable graph in which to set the specified node

hNode

- kernel node from the graph from which graphExec was instantiated

nodeParams

- Updated Parameters to set

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

Sets the parameters of a kernel 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. All `nodeParams` fields may change, but the following restrictions apply to `func` updates:

- ▶ 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.
- ▶ 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 context 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. `hNode` is also not modified by this call.

If `hNode` 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.

See also:

[cuGraphExecNodeSetParams](#), [cuGraphAddKernelNode](#), [cuGraphKernelNodeSetParams](#),
[cuGraphExecMemcpyNodeSetParams](#), [cuGraphExecMemsetNodeSetParams](#),
[cuGraphExecHostNodeSetParams](#), [cuGraphExecChildGraphNodeSetParams](#),
[cuGraphExecEventRecordNodeSetEvent](#), [cuGraphExecEventWaitNodeSetEvent](#),
[cuGraphExecExternalSemaphoresSignalNodeSetParams](#),
[cuGraphExecExternalSemaphoresWaitNodeSetParams](#), [cuGraphExecUpdate](#), [cuGraphInstantiate](#)

CUresult cuGraphExecMemcpyNodeSetParams (CUgraphExec hGraphExec, CUgraphNode hNode, const CUDA_MEMCPY3D *copyParams, CUcontext ctx)

Sets the parameters for a memcpy node in the given graphExec.

Parameters

hGraphExec

- The executable graph in which to set the specified node

hNode

- Memcpy node from the graph which was used to instantiate graphExec

copyParams

- The updated parameters to set

ctx

- Context on which to run the node

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

Updates the work represented by hNode in hGraphExec as though hNode had contained copyParams at instantiation. hNode must remain in the graph which was used to instantiate hGraphExec. Changed edges to and from hNode are ignored.

The source and destination memory in copyParams 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 copyParams 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. `hNode` is also not modified by this call.

Returns `CUDA_ERROR_INVALID_VALUE` 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.

See also:

[cuGraphExecNodeSetParams](#), [cuGraphAddMemcpyNode](#),
[cuGraphMemcpyNodeSetParams](#), [cuGraphExecKernelNodeSetParams](#),
[cuGraphExecMemsetNodeSetParams](#), [cuGraphExecHostNodeSetParams](#),
[cuGraphExecChildGraphNodeSetParams](#), [cuGraphExecEventRecordNodeSetEvent](#),
[cuGraphExecEventWaitNodeSetEvent](#), [cuGraphExecExternalSemaphoresSignalNodeSetParams](#),
[cuGraphExecExternalSemaphoresWaitNodeSetParams](#), [cuGraphExecUpdate](#), [cuGraphInstantiate](#)

CUresult cuGraphExecMemsetNodeSetParams (CUgraphExec hGraphExec, CUgraphNode hNode, const CUDA_MEMSET_NODE_PARAMS *memsetParams, CUcontext ctx)

Sets the parameters for a memset node in the given graphExec.

Parameters

hGraphExec

- The executable graph in which to set the specified node

hNode

- Memset node from the graph which was used to instantiate graphExec

memsetParams

- The updated parameters to set

ctx

- Context on which to run the node

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

Updates the work represented by `hNode` in `hGraphExec` as though `hNode` had contained `memsetParams` at instantiation. `hNode` must remain in the graph which was used to instantiate `hGraphExec`. Changed edges to and from `hNode` are ignored.

Zero sized operations are not supported.

The new destination pointer in `memsetParams` 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. `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.

See also:

[cuGraphExecNodeSetParams](#), [cuGraphAddMemsetNode](#), [cuGraphMemsetNodeSetParams](#),
[cuGraphExecKernelNodeSetParams](#), [cuGraphExecMemcpyNodeSetParams](#),
[cuGraphExecHostNodeSetParams](#), [cuGraphExecChildGraphNodeSetParams](#),
[cuGraphExecEventRecordNodeSetEvent](#), [cuGraphExecEventWaitNodeSetEvent](#),
[cuGraphExecExternalSemaphoresSignalNodeSetParams](#),
[cuGraphExecExternalSemaphoresWaitNodeSetParams](#), [cuGraphExecUpdate](#), [cuGraphInstantiate](#)

CUresult cuGraphExecNodeSetParams (CUgraphExec hGraphExec, CUgraphNode hNode, CUgraphNodeParams *nodeParams)

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

Parameters

hGraphExec

- The executable graph in which to update the specified node

hNode

- Corresponding node from the graph from which graphExec was instantiated

nodeParams

- Updated Parameters to set

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_SUPPORTED

Description

Sets the parameters of a 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.

Allowed changes to parameters on executable graphs are as follows:

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

Node type	Allowed changes
batch memops	Addresses, values, and operation type for wait operations; see cuGraphExecBatchMemOpNodeSetParams



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphAddNode](#), [cuGraphNodeSetParams](#), [cuGraphExecUpdate](#), [cuGraphInstantiate](#)

CUresult cuGraphExecUpdate (CUgraphExec hGraphExec, CUgraph hGraph, CUgraphExecUpdateResultInfo *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

[CUDA_SUCCESS](#), [CUDA_ERROR_GRAPH_EXEC_UPDATE_FAILURE](#),

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 `CUDA_GRAPH_INSTANTIATE_FLAG_USE_NODE_PRIORITY`, 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 context 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.
- ▶ External semaphore wait nodes and record nodes:
 - ▶ Changing the number of semaphores 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.

`cuGraphExecUpdate` sets the result member of `resultInfo` to `CU_GRAPH_EXEC_UPDATE_ERROR_TOPOLOGY_CHANGED` 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.

`cuGraphExecUpdate` sets the result member of `resultInfo` to:

- ▶ `CU_GRAPH_EXEC_UPDATE_ERROR` if passed an invalid value.
- ▶ `CU_GRAPH_EXEC_UPDATE_ERROR_TOPOLOGY_CHANGED` if the graph topology changed
- ▶ `CU_GRAPH_EXEC_UPDATE_ERROR_NODE_TYPE_CHANGED` if the type of a node changed, in which case `hErrorNode_out` is set to the node from `hGraph`.
- ▶ `CU_GRAPH_EXEC_UPDATE_ERROR_UNSUPPORTED_FUNCTION_CHANGE` if the function changed in an unsupported way(see note above), in which case `hErrorNode_out` is set to the node from `hGraph`
- ▶ `CU_GRAPH_EXEC_UPDATE_ERROR_PARAMETERS_CHANGED` 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`.
- ▶ `CU_GRAPH_EXEC_UPDATE_ERROR_ATTRIBUTES_CHANGED` 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`.
- ▶ `CU_GRAPH_EXEC_UPDATE_ERROR_NOT_SUPPORTED` 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 `CU_GRAPH_EXEC_UPDATE_ERROR`. If the update succeeds, the result member will be set to `CU_GRAPH_EXEC_UPDATE_SUCCESS`.

`cuGraphExecUpdate` returns `CUDA_SUCCESS` when the updated was performed successfully. It returns `CUDA_ERROR_GRAPH_EXEC_UPDATE_FAILURE` 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.

See also:

[cuGraphInstantiate](#)

CUresult

cuGraphExternalSemaphoresSignalNodeGetParams
(CUgraphNode hNode,
CUDA_EXT_SEM_SIGNAL_NODE_PARAMS
*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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#)

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 [cuGraphExternalSemaphoresSignalNodeSetParams](#) 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.

See also:

[cuLaunchKernel](#), [cuGraphAddExternalSemaphoresSignalNode](#),
[cuGraphExternalSemaphoresSignalNodeSetParams](#), [cuGraphAddExternalSemaphoresWaitNode](#),
[cuSignalExternalSemaphoresAsync](#), [cuWaitExternalSemaphoresAsync](#)

CUresult

cuGraphExternalSemaphoresSignalNodeSetParams
(CUgraphNode hNode, const
CUDA_EXT_SEM_SIGNAL_NODE_PARAMS
*nodeParams)

Sets an external semaphore signal node's parameters.

Parameters

hNode

- Node to set the parameters for

nodeParams

- Parameters to copy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_OUT_OF_MEMORY](#)

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.

See also:

[cuGraphNodeSetParams](#), [cuGraphAddExternalSemaphoresSignalNode](#),
[cuGraphExternalSemaphoresSignalNodeSetParams](#), [cuGraphAddExternalSemaphoresWaitNode](#),
[cuSignalExternalSemaphoresAsync](#), [cuWaitExternalSemaphoresAsync](#)

CUresult
cuGraphExternalSemaphoresWaitNodeGetParams
 (CUgraphNode hNode,
 CUDA_EXT_SEM_WAIT_NODE_PARAMS
 *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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#)

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 [cuGraphExternalSemaphoresSignalNodeSetParams](#) 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.

See also:

[cuLaunchKernel](#), [cuGraphAddExternalSemaphoresWaitNode](#),
[cuGraphExternalSemaphoresWaitNodeSetParams](#), [cuGraphAddExternalSemaphoresWaitNode](#),
[cuSignalExternalSemaphoresAsync](#), [cuWaitExternalSemaphoresAsync](#)

CUresult cuGraphExternalSemaphoresWaitNodeSetParams (CUgraphNode hNode, const CUDA_EXT_SEM_WAIT_NODE_PARAMS *nodeParams)

Sets an external semaphore wait node's parameters.

Parameters

hNode

- Node to set the parameters for

nodeParams

- Parameters to copy

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_OUT_OF_MEMORY

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.

See also:

[cuGraphNodeSetParams](#), [cuGraphAddExternalSemaphoresWaitNode](#),
[cuGraphExternalSemaphoresWaitNodeSetParams](#), [cuGraphAddExternalSemaphoresWaitNode](#),
[cuSignalExternalSemaphoresAsync](#), [cuWaitExternalSemaphoresAsync](#)

CUresult cuGraphGetEdges (CUgraph hGraph, CUgraphNode *from, CUgraphNode *to, size_t *numEdges)

Returns a graph's dependency edges.

Parameters

hGraph

- Graph to get the edges from

from

- Location to return edge endpoints

to

- Location to return edge endpoints

numEdges

- See description

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns a list of `hGraph`'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.

See also:

[cuGraphGetNodes](#), [cuGraphGetRootNodes](#), [cuGraphAddDependencies](#), [cuGraphRemoveDependencies](#), [cuGraphNodeGetDependencies](#), [cuGraphNodeGetDependentNodes](#)

CUresult cuGraphGetEdges_v2 (CUgraph hGraph, CUgraphNode *from, CUgraphNode *to, CUgraphEdgeData *edgeData, size_t *numEdges)

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

Parameters

hGraph

- 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

CUDA_SUCCESS, CUDA_ERROR_LOSSY_QUERY, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE

Description

Returns a list of `hGraph`'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 lossy query via `NULL` `edgeData` will result in CUDA_ERROR_LOSSY_QUERY. 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.

See also:

[cuGraphGetNodes](#), [cuGraphGetRootNodes](#), [cuGraphAddDependencies](#),
[cuGraphRemoveDependencies](#), [cuGraphNodeGetDependencies](#), [cuGraphNodeGetDependentNodes](#)

CUresult cuGraphGetNodes (CUgraph hGraph, CUGraphNode *nodes, size_t *numNodes)

Returns a graph's nodes.

Parameters

hGraph

- Graph to query

nodes

- Pointer to return the nodes

numNodes

- See description

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#)

Description

Returns a list of `hGraph`'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.

See also:

[cuGraphCreate](#), [cuGraphGetRootNodes](#), [cuGraphGetEdges](#), [cuGraphNodeGetType](#),
[cuGraphNodeGetDependencies](#), [cuGraphNodeGetDependentNodes](#)

CUresult cuGraphGetRootNodes (CUgraph hGraph, CUgraphNode *rootNodes, size_t *numRootNodes)

Returns a graph's root nodes.

Parameters

hGraph

- Graph to query

rootNodes

- Pointer to return the root nodes

numRootNodes

- See description

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns a list of `hGraph`'s root nodes. `rootNodes` may be NULL, in which case this function will return the number of root nodes in `numRootNodes`. Otherwise, `numRootNodes` entries will be filled in. If `numRootNodes` is higher than the actual number of root nodes, the remaining entries in `rootNodes` will be set to NULL, and the number of nodes actually obtained will be returned in `numRootNodes`.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphCreate](#), [cuGraphGetNodes](#), [cuGraphGetEdges](#), [cuGraphNodeGetType](#), [cuGraphNodeGetDependencies](#), [cuGraphNodeGetDependentNodes](#)

CUresult cuGraphHostNodeGetParams (CUgraphNode hNode, CUDA_HOST_NODE_PARAMS *nodeParams)

Returns a host node's parameters.

Parameters

hNode

- Node to get the parameters for

nodeParams

- Pointer to return the parameters

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns the parameters of host node hNode in nodeParams.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuLaunchHostFunc](#), [cuGraphAddHostNode](#), [cuGraphHostNodeSetParams](#)

CUresult cuGraphHostNodeSetParams (CUgraphNode hNode, const CUDA_HOST_NODE_PARAMS *nodeParams)

Sets a host node's parameters.

Parameters

hNode

- Node to set the parameters for

nodeParams

- Parameters to copy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Sets the parameters of host 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.

See also:

[cuGraphNodeSetParams](#), [cuLaunchHostFunc](#), [cuGraphAddHostNode](#), [cuGraphHostNodeGetParams](#)

CUresult cuGraphInstantiate (CUgraphExec *phGraphExec, CUgraph hGraph, unsigned long long flags)

Creates an executable graph from a graph.

Parameters

phGraphExec

- Returns instantiated graph

hGraph

- Graph to instantiate

flags

- Flags to control instantiation. See [CUgraphInstantiate_flags](#).

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Instantiates `hGraph` 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 `phGraphExec`.

The `flags` parameter controls the behavior of instantiation and subsequent graph launches. Valid flags are:

- ▶ [CUDA_GRAPH_INSTANTIATE_FLAG_AUTO_FREE_ON_LAUNCH](#), which configures a graph containing memory allocation nodes to automatically free any unfreed memory allocations before the graph is relaunched.
- ▶ [CUDA_GRAPH_INSTANTIATE_FLAG_DEVICE_LAUNCH](#), 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 [CUDA_GRAPH_INSTANTIATE_FLAG_AUTO_FREE_ON_LAUNCH](#).
- ▶ [CUDA_GRAPH_INSTANTIATE_FLAG_USE_NODE_PRIORITY](#), 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 `hGraph` 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 [cuGraphExecDestroy](#) will result in an error. The same also applies if `hGraph` contains any device-updatable kernel nodes.

If `hGraph` contains kernels which call device-side [cudaGraphLaunch\(\)](#) from multiple contexts, 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 context.
- ▶ 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 context, and the current context must match the context 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.

See also:

[cuGraphInstantiate](#), [cuGraphCreate](#), [cuGraphUpload](#), [cuGraphLaunch](#), [cuGraphExecDestroy](#)

CUresult cuGraphInstantiateWithParams (CUgraphExec *phGraphExec, CUgraph hGraph, CUDA_GRAPH_INSTANTIATE_PARAMS *instantiateParams)

Creates an executable graph from a graph.

Parameters

phGraphExec

- Returns instantiated graph

hGraph

- Graph to instantiate

instantiateParams

- Instantiation parameters

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

Instantiates hGraph 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 `phGraphExec`.

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

[CUDA_GRAPH_INSTANTIATE_PARAMS](#) is defined as:

```
typedef struct {
    cuuint64_t flags;
    CUstream hUploadStream;
    CUgraphNode hErrNode_out;
    CUgraphInstantiateResult result_out;
} CUDA_GRAPH_INSTANTIATE_PARAMS;
```

The `flags` field controls the behavior of instantiation and subsequent graph launches. Valid flags are:

- ▶ [CUDA_GRAPH_INSTANTIATE_FLAG_AUTO_FREE_ON_LAUNCH](#), which configures a graph containing memory allocation nodes to automatically free any unfreed memory allocations before the graph is relaunched.
- ▶ [CUDA_GRAPH_INSTANTIATE_FLAG_UPLOAD](#), which will perform an upload of the graph into `hUploadStream` once the graph has been instantiated.
- ▶ [CUDA_GRAPH_INSTANTIATE_FLAG_DEVICE_LAUNCH](#), 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 [CUDA_GRAPH_INSTANTIATE_FLAG_AUTO_FREE_ON_LAUNCH](#).
- ▶ [CUDA_GRAPH_INSTANTIATE_FLAG_USE_NODE_PRIORITY](#), 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 `hGraph` 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 [cuGraphExecDestroy](#) will result in an error. The same also applies if `hGraph` contains any device-updatable kernel nodes.

If `hGraph` contains kernels which call device-side [cudaGraphLaunch\(\)](#) from multiple contexts, 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 context.
- ▶ 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 context, and the current context must match the context of other nodes in the graph.

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

- ▶ [CUDA_GRAPH_INSTANTIATE_ERROR](#), if passed an invalid value or if an unexpected error occurred which is described by the return value of the function. `hErrNode_out` will be set to `NULL`.
- ▶ [CUDA_GRAPH_INSTANTIATE_INVALID_STRUCTURE](#), if the graph structure is invalid. `hErrNode_out` will be set to one of the offending nodes.
- ▶ [CUDA_GRAPH_INSTANTIATE_NODE_OPERATION_NOT_SUPPORTED](#), 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. `hErrNode_out` will be set to this node.
- ▶ [CUDA_GRAPH_INSTANTIATE_MULTIPLE_CTXS_NOT_SUPPORTED](#), if the graph is instantiated for device launch but a node's context 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 contexts. `hErrNode_out` will be set to this node.

If instantiation is successful, `result_out` will be set to

[CUDA_GRAPH_INSTANTIATE_SUCCESS](#), 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.

See also:

[cuGraphCreate](#), [cuGraphInstantiate](#), [cuGraphExecDestroy](#)

CUresult cuGraphKernelNodeCopyAttributes (CUgraphNode dst, CUgraphNode src)

Copies attributes from source node to destination node.

Parameters

dst

Destination node

src

Source node For list of attributes see `CUkernelNodeAttrID`

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

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:

[CUaccessPolicyWindow](#)

CUresult cuGraphKernelNodeGetAttribute (CUgraphNode hNode, CUkernelNodeAttrID attr, CUkernelNodeAttrValue *value_out)

Queries node attribute.

Parameters

hNode

attr

value_out

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#)

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:

[CUaccessPolicyWindow](#)

CUresult cuGraphKernelNodeGetParams (CUgraphNode hNode, CUDA_KERNEL_NODE_PARAMS *nodeParams)

Returns a kernel node's parameters.

Parameters

hNode

- Node to get the parameters for

nodeParams

- Pointer to return the parameters

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns the parameters of kernel node `hNode` in `nodeParams`. The `kernelParams` or `extra` array returned in `nodeParams`, 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 [cuGraphKernelNodeSetParams](#) 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.

See also:

[cuLaunchKernel](#), [cuGraphAddKernelNode](#), [cuGraphKernelNodeSetParams](#)

CUresult cuGraphKernelNodeSetAttribute (CUgraphNode hNode, CUkernelNodeAttrID attr, const CUkernelNodeAttrValue *value)

Sets node attribute.

Parameters

hNode

attr

value

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#)

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:

[CUaccessPolicyWindow](#)

CUresult cuGraphKernelNodeSetParams (CUgraphNode hNode, const CUDA_KERNEL_NODE_PARAMS *nodeParams)

Sets a kernel node's parameters.

Parameters

hNode

- Node to set the parameters for

nodeParams

- Parameters to copy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Sets the parameters of kernel 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.

See also:

[cuGraphNodeSetParams](#), [cuLaunchKernel](#), [cuGraphAddKernelNode](#), [cuGraphKernelNodeGetParams](#)

CUresult cuGraphLaunch (CUgraphExec hGraphExec, CUstream hStream)

Launches an executable graph in a stream.

Parameters

hGraphExec

- Executable graph to launch

hStream

- Stream in which to launch the graph

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Executes `hGraphExec` in `hStream`. Only one instance of `hGraphExec` may be executing at a time. Each launch is ordered behind both any previous work in `hStream` and any previous launches of `hGraphExec`. To execute a graph concurrently, it must be instantiated multiple times into multiple executable graphs.

If any allocations created by `hGraphExec` remain unfreed (from a previous launch) and `hGraphExec` was not instantiated with [CUDA_GRAPH_INSTANTIATE_FLAG_AUTO_FREE_ON_LAUNCH](#), the launch will fail with [CUDA_ERROR_INVALID_VALUE](#).



Note:

- ▶ Graph objects are not threadsafe. [More here](#).

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

See also:

[cuGraphInstantiate](#), [cuGraphUpload](#), [cuGraphExecDestroy](#)

CUresult cuGraphMemAllocNodeGetParams
(CUgraphNode hNode,
CUDA_MEM_ALLOC_NODE_PARAMS *params_out)

Returns a memory alloc node's parameters.

Parameters

hNode

- Node to get the parameters for

params_out

- Pointer to return the parameters

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#)

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.

See also:

[cuGraphAddMemAllocNode](#), [cuGraphMemFreeNodeGetParams](#)

CUresult cuGraphMemcpyNodeGetParams (CUgraphNode hNode, CUDA_MEMCPY3D *nodeParams)

Returns a memcpy node's parameters.

Parameters

hNode

- Node to get the parameters for

nodeParams

- Pointer to return the parameters

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE

Description

Returns the parameters of memcpy node hNode in nodeParams.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuMemcpy3D](#), [cuGraphAddMemcpyNode](#), [cuGraphMemcpyNodeSetParams](#)

CUresult cuGraphMemcpyNodeSetParams (CUgraphNode hNode, const CUDA_MEMCPY3D *nodeParams)

Sets a memcpy node's parameters.

Parameters

hNode

- Node to set the parameters for

nodeParams

- Parameters to copy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE,

Description

Sets the parameters of memcpy 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.

See also:

[cuGraphNodeSetParams](#), [cuMemcpy3D](#), [cuGraphAddMemcpyNode](#), [cuGraphMemcpyNodeGetParams](#)

CUresult cuGraphMemFreeNodeGetParams (CUgraphNode hNode, CUdeviceptr *dptr_out)

Returns a memory free node's parameters.

Parameters

hNode

- Node to get the parameters for

dptr_out

- Pointer to return the device address

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

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.

See also:

[cuGraphAddMemFreeNode](#), [cuGraphMemAllocNodeGetParams](#)

CUresult cuGraphMemsetNodeGetParams (CUgraphNode hNode, CUDA_MEMSET_NODE_PARAMS *nodeParams)

Returns a memset node's parameters.

Parameters

hNode

- Node to get the parameters for

nodeParams

- Pointer to return the parameters

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns the parameters of memset node hNode in nodeParams.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuMemsetD2D32](#), [cuGraphAddMemsetNode](#), [cuGraphMemsetNodeSetParams](#)

CUresult cuGraphMemsetNodeSetParams (CUgraphNode hNode, const CUDA_MEMSET_NODE_PARAMS *nodeParams)

Sets a memset node's parameters.

Parameters

hNode

- Node to set the parameters for

nodeParams

- Parameters to copy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Sets the parameters of memset 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.

See also:

[cuGraphNodeSetParams](#), [cuMemsetD2D32](#), [cuGraphAddMemsetNode](#), [cuGraphMemsetNodeGetParams](#)

CUresult cuGraphNodeFindInClone (CUgraphNode *phNode, CUgraphNode hOriginalNode, CUgraph hClonedGraph)

Finds a cloned version of a node.

Parameters

phNode

- Returns handle to the cloned node

hOriginalNode

- Handle to the original node

hClonedGraph

- Cloned graph to query

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),

Description

This function returns the node in `hClonedGraph` corresponding to `hOriginalNode` in the original graph.

`hClonedGraph` must have been cloned from `hOriginalGraph` via [cuGraphClone](#).

`hOriginalNode` must have been in `hOriginalGraph` at the time of the call to [cuGraphClone](#),

and the corresponding cloned node in `hClonedGraph` must not have been removed. The cloned node is then returned via `phClonedNode`.



Note:

- ▶ Graph objects are not threadsafe. [More here.](#)
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphClone](#)

CUresult cuGraphNodeGetDependencies (CUgraphNode hNode, CUgraphNode *dependencies, size_t *numDependencies)

Returns a node's dependencies.

Parameters

hNode

- Node to query

dependencies

- Pointer to return the dependencies

numDependencies

- See description

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns a list of node's dependencies. `dependencies` may be NULL, in which case this function will return the number of dependencies in `numDependencies`. Otherwise, `numDependencies` entries will be filled in. If `numDependencies` is higher than the actual number of dependencies, the remaining entries in `dependencies` will be set to NULL, and the number of nodes actually obtained will be returned in `numDependencies`.



Note:

- ▶ Graph objects are not threadsafe. [More here.](#)

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

See also:

[cuGraphNodeGetDependentNodes](#), [cuGraphGetNodes](#), [cuGraphGetRootNodes](#), [cuGraphGetEdges](#), [cuGraphAddDependencies](#), [cuGraphRemoveDependencies](#)

CUresult cuGraphNodeGetDependencies_v2 (CUgraphNode hNode, CUgraphNode *dependencies, CUgraphEdgeData *edgeData, size_t *numDependencies)

Returns a node's dependencies (12.3+).

Parameters

hNode

- Node to query

dependencies

- Pointer to return the dependencies

edgeData

- Optional array to return edge data for each dependency

numDependencies

- See description

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_LOSSY_QUERY](#), [CUDA_ERROR_DEINITIALIZED](#),
[CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns a list of node's dependencies. `dependencies` may be NULL, in which case this function will return the number of dependencies in `numDependencies`. Otherwise, `numDependencies` entries will be filled in. If `numDependencies` is higher than the actual number of dependencies, the remaining entries in `dependencies` will be set to NULL, and the number of nodes actually obtained will be returned in `numDependencies`.

Note that if an edge has non-zero (non-default) edge data and `edgeData` is NULL, this API will return [CUDA_ERROR_LOSSY_QUERY](#). If `edgeData` is non-NULL, then `dependencies` 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.

See also:

[cuGraphNodeGetDependentNodes](#), [cuGraphGetNodes](#), [cuGraphGetRootNodes](#), [cuGraphGetEdges](#), [cuGraphAddDependencies](#), [cuGraphRemoveDependencies](#)

CUresult cuGraphNodeGetDependentNodes (CUgraphNode hNode, CUgraphNode *dependentNodes, size_t *numDependentNodes)

Returns a node's dependent nodes.

Parameters

hNode

- Node to query

dependentNodes

- Pointer to return the dependent nodes

numDependentNodes

- See description

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns a list of node's dependent nodes. `dependentNodes` may be NULL, in which case this function will return the number of dependent nodes in `numDependentNodes`. Otherwise, `numDependentNodes` entries will be filled in. If `numDependentNodes` is higher than the actual number of dependent nodes, the remaining entries in `dependentNodes` will be set to NULL, and the number of nodes actually obtained will be returned in `numDependentNodes`.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphNodeGetDependencies](#), [cuGraphGetNodes](#), [cuGraphGetRootNodes](#), [cuGraphGetEdges](#), [cuGraphAddDependencies](#), [cuGraphRemoveDependencies](#)

CUresult cuGraphNodeGetDependentNodes_v2
(CUgraphNode hNode, CUgraphNode
***dependentNodes, CUgraphEdgeData *edgeData, size_t**
***numDependentNodes)**

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

Parameters

hNode

- Node to query

dependentNodes

- Pointer to return the dependent nodes

edgeData

- Optional pointer to return edge data for dependent nodes

numDependentNodes

- See description

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_LOSSY_QUERY](#), [CUDA_ERROR_DEINITIALIZED](#),
[CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns a list of node's dependent nodes. `dependentNodes` may be NULL, in which case this function will return the number of dependent nodes in `numDependentNodes`. Otherwise, `numDependentNodes` entries will be filled in. If `numDependentNodes` is higher than the actual number of dependent nodes, the remaining entries in `dependentNodes` will be set to NULL, and the number of nodes actually obtained will be returned in `numDependentNodes`.

Note that if an edge has non-zero (non-default) edge data and `edgeData` is NULL, this API will return [CUDA_ERROR_LOSSY_QUERY](#). If `edgeData` is non-NULL, then `dependentNodes` 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.

See also:

[cuGraphNodeGetDependencies](#), [cuGraphGetNodes](#), [cuGraphGetRootNodes](#), [cuGraphGetEdges](#),
[cuGraphAddDependencies](#), [cuGraphRemoveDependencies](#)

CUresult cuGraphNodeGetEnabled (CUgraphExec hGraphExec, CUgraphNode 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#),

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.
- ▶ This function will not reflect device-side updates for device-updatable kernel nodes.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphNodeSetEnabled](#), [cuGraphExecUpdate](#), [cuGraphInstantiate](#) [cuGraphLaunch](#)

CUresult cuGraphNodeGetType (CUgraphNode hNode, CUgraphNodeType *type)

Returns a node's type.

Parameters

hNode

- Node to query

type

- Pointer to return the node type

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE

Description

Returns the node type of hNode in type.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphGetNodes](#), [cuGraphGetRootNodes](#), [cuGraphChildGraphNodeGetGraph](#), [cuGraphKernelNodeGetParams](#), [cuGraphKernelNodeSetParams](#), [cuGraphHostNodeGetParams](#), [cuGraphHostNodeSetParams](#), [cuGraphMemcpyNodeGetParams](#), [cuGraphMemcpyNodeSetParams](#), [cuGraphMemsetNodeGetParams](#), [cuGraphMemsetNodeSetParams](#)

CUresult cuGraphNodeSetEnabled (CUgraphExec hGraphExec, CUgraphNode 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

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE,

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.

If `hNode` 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:**

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.

See also:

[cuGraphNodeGetEnabled](#), [cuGraphExecUpdate](#), [cuGraphInstantiate](#) [cuGraphLaunch](#)

CUresult cuGraphNodeSetParams (CUgraphNode hNode, CUgraphNodeParams *nodeParams)

Update's a graph node's parameters.

Parameters

hNode

- Node to set the parameters for

nodeParams

- Parameters to copy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_SUPPORTED](#)

Description

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

Modifying parameters is not supported for node types CU_GRAPH_NODE_TYPE_MEM_ALLOC and CU_GRAPH_NODE_TYPE_MEM_FREE.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphAddNode](#), [cuGraphExecNodeSetParams](#)

CUresult cuGraphReleaseUserObject (CUgraph graph, CUUserObject 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

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:

[cuUserObjectCreate](#), [cuUserObjectRetain](#), [cuUserObjectRelease](#), [cuGraphRetainUserObject](#), [cuGraphCreate](#)

CUresult cuGraphRemoveDependencies (CUgraph hGraph, const CUgraphNode *from, const CUgraphNode *to, size_t numDependencies)

Removes dependency edges from a graph.

Parameters**hGraph**

- 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

The number of dependencies to be removed is defined by `numDependencies`. Elements in `from` and `to` at corresponding indices define a dependency. Each node in `from` and `to` must belong to `hGraph`.

If `numDependencies` is 0, elements in `from` and `to` will be ignored. Specifying a non-existing dependency will return an error.

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.

See also:

[cuGraphAddDependencies](#), [cuGraphGetEdges](#), [cuGraphNodeGetDependencies](#),
[cuGraphNodeGetDependentNodes](#)

CUresult cuGraphRemoveDependencies_v2 (CUgraph hGraph, const CUGraphNode *from, const CUGraphNode *to, const CUgraphEdgeData *edgeData, size_t numDependencies)

Removes dependency edges from a graph (12.3+).

Parameters

hGraph

- 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

The number of dependencies to be removed is defined by `numDependencies`. Elements in `from` and `to` at corresponding indices define a dependency. Each node in `from` and `to` must belong to `hGraph`.

If `numDependencies` is 0, elements in `from` and `to` 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.

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.

See also:

[cuGraphAddDependencies](#), [cuGraphGetEdges](#), [cuGraphNodeGetDependencies](#),
[cuGraphNodeGetDependentNodes](#)

CUresult cuGraphRetainUserObject (CUgraph graph, CUUserObject 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 [CU_GRAPH_USER_OBJECT_MOVE](#) transfers references from the calling thread, rather than create new references. Pass 0 to create new references.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

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:

[cuUserObjectCreate](#), [cuUserObjectRetain](#), [cuUserObjectRelease](#), [cuGraphReleaseUserObject](#), [cuGraphCreate](#)

CUresult cuGraphUpload (CUgraphExec hGraphExec, CUstream hStream)

Uploads an executable graph in a stream.

Parameters

hGraphExec

- Executable graph to upload

hStream

- Stream in which to upload the graph

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#)

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 `hGraphExec`.



Note:

- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphInstantiate](#), [cuGraphLaunch](#), [cuGraphExecDestroy](#)

CUresult cuUserObjectCreate (CUUserObject *object_out, void *ptr, CUhostFn 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 [CU_USER_OBJECT_NO_DESTRUCTOR_SYNC](#), 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

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:

[cuUserObjectRetain](#), [cuUserObjectRelease](#), [cuGraphRetainUserObject](#), [cuGraphReleaseUserObject](#), [cuGraphCreate](#)

CUresult cuUserObjectRelease (CUUserObject 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

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:

[cuUserObjectCreate](#), [cuUserObjectRetain](#), [cuGraphRetainUserObject](#), [cuGraphReleaseUserObject](#), [cuGraphCreate](#)

CUresult cuUserObjectRetain (CUUserObject 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#)

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:

[cuUserObjectCreate](#), [cuUserObjectRelease](#), [cuGraphRetainUserObject](#), [cuGraphReleaseUserObject](#), [cuGraphCreate](#)

6.25. Occupancy

This section describes the occupancy calculation functions of the low-level CUDA driver application programming interface.

CUresult cuOccupancyAvailableDynamicSMemPerBlock (size_t *dynamicSmemSize, CUfunction 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 blocks

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_UNKNOWN](#)

Description

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

Note that the API can also be used with context-less kernel [CUkernel](#) by querying the handle using [cuLibraryGetKernel\(\)](#) and then passing it to the API by casting to [CUfunction](#). Here, the context to use for calculations will be the current context.



Note:

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

CUresult cuOccupancyMaxActiveBlocksPerMultiprocessor(int *numBlocks, CUfunction func, int blockSize, size_t dynamicSMemSize)

Returns occupancy of a function.

Parameters

numBlocks

- Returned occupancy

func

- Kernel 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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_UNKNOWN](#)

Description

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

Note that the API can also be used with context-less kernel [CUkernel](#) by querying the handle using [cuLibraryGetKernel\(\)](#) and then passing it to the API by casting to [CUfunction](#). Here, the context to use for calculations will be the current context.



Note:

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

See also:

[cudaOccupancyMaxActiveBlocksPerMultiprocessor](#)

CUresult

cuOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
(int *numBlocks, CUfunction func, int blockSize, size_t dynamicSMemSize, unsigned int flags)

Returns occupancy of a function.

Parameters

numBlocks

- Returned occupancy

func

- Kernel 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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_UNKNOWN](#)

Description

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

The Flags parameter controls how special cases are handled. The valid flags are:

- ▶ [CU_OCCUPANCY_DEFAULT](#), which maintains the default behavior as [cuOccupancyMaxActiveBlocksPerMultiprocessor](#);
- ▶ [CU_OCCUPANCY_DISABLE_CACHING_OVERRIDE](#), which 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 [CU_OCCUPANCY_DISABLE_CACHING_OVERRIDE](#) 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 that the API can also be with launch context-less kernel [CUkernel](#) by querying the handle using [cuLibraryGetKernel\(\)](#) and then passing it to the API by casting to [CUfunction](#). Here, the context to use for calculations will be the current context.



Note:

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

See also:

[cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags](#)

CUresult cuOccupancyMaxActiveClusters (int *numClusters, CUfunction func, const CUlaunchConfig *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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_CLUSTER_SIZE](#), [CUDA_ERROR_UNKNOWN](#)

Description

If the function has required cluster size already set (see [cudaFuncGetAttributes](#) / [cuFuncGetAttribute](#)), 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 that the API can also be used with context-less kernel [CUkernel](#) by querying the handle using [cuLibraryGetKernel\(\)](#) and then passing it to the API by casting to [CUfunction](#). Here, the context to

use for calculations will either be taken from the specified stream `config->hStream` or the current context in case of NULL stream.



Note:

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

See also:

[cudaFuncGetAttributes](#), [cuFuncGetAttribute](#)

CUresult cuOccupancyMaxPotentialBlockSize (int *minGridSize, int *blockSize, CUfunction func, CUoccupancyB2DSize blockSizeToDynamicSMemSize, size_t dynamicSMemSize, int blockSizeLimit)

Suggest a launch configuration with reasonable occupancy.

Parameters

minGridSize

- Returned minimum grid size needed to achieve the maximum occupancy

blockSize

- Returned maximum block size that can achieve the maximum occupancy

func

- Kernel for which launch configuration is calculated

blockSizeToDynamicSMemSize

- A function that calculates how much per-block dynamic shared memory `func` uses based on the block size

dynamicSMemSize

- Dynamic shared memory usage intended, in bytes

blockSizeLimit

- The maximum block size `func` is designed to handle

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_UNKNOWN](#)

Description

Returns in `*blockSize` a reasonable block size that can achieve the maximum occupancy (or, the maximum number of active warps with the fewest blocks per multiprocessor), and in `*minGridSize` the minimum grid size to achieve the maximum occupancy.

If `blockSizeLimit` is 0, the configurator will use the maximum block size permitted by the device / function instead.

If per-block dynamic shared memory allocation is not needed, the user should leave both `blockSizeToDynamicSMemSize` and `dynamicSMemSize` as 0.

If per-block dynamic shared memory allocation is needed, then if the dynamic shared memory size is constant regardless of block size, the size should be passed through `dynamicSMemSize`, and `blockSizeToDynamicSMemSize` should be NULL.

Otherwise, if the per-block dynamic shared memory size varies with different block sizes, the user needs to provide a unary function through `blockSizeToDynamicSMemSize` that computes the dynamic shared memory needed by `func` for any given block size. `dynamicSMemSize` is ignored. An example signature is:

```
↑ // Take block size, returns dynamic shared memory needed
   size_t blockToSmem(int blockSize);
```

Note that the API can also be used with context-less kernel [CUkernel](#) by querying the handle using [cuLibraryGetKernel\(\)](#) and then passing it to the API by casting to [CUfunction](#). Here, the context to use for calculations will be the current context.



Note:

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

See also:

[cudaOccupancyMaxPotentialBlockSize](#)

CUresult cuOccupancyMaxPotentialBlockSizeWithFlags
 (int *minGridSize, int *blockSize, CUfunction func,
 CUoccupancyB2DSize blockSizeToDynamicSMemSize,
 size_t dynamicSMemSize, int blockSizeLimit, unsigned int
 flags)

Suggest a launch configuration with reasonable occupancy.

Parameters

minGridSize

- Returned minimum grid size needed to achieve the maximum occupancy

blockSize

- Returned maximum block size that can achieve the maximum occupancy

func

- Kernel for which launch configuration is calculated

blockSizeToDynamicSMemSize

- A function that calculates how much per-block dynamic shared memory `func` uses based on the block size

dynamicSMemSize

- Dynamic shared memory usage intended, in bytes

blockSizeLimit

- The maximum block size `func` is designed to handle

flags

- Options

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_UNKNOWN](#)

Description

An extended version of [cuOccupancyMaxPotentialBlockSize](#). In addition to arguments passed to [cuOccupancyMaxPotentialBlockSize](#), [cuOccupancyMaxPotentialBlockSizeWithFlags](#) also takes a `Flags` parameter.

The `Flags` parameter controls how special cases are handled. The valid flags are:

- [CU_OCCUPANCY_DEFAULT](#), which maintains the default behavior as [cuOccupancyMaxPotentialBlockSize](#);
- [CU_OCCUPANCY_DISABLE_CACHING_OVERRIDE](#), which suppresses the default behavior on platform where global caching affects occupancy. On such platforms, the launch

configurations that produces maximal occupancy might not support global caching. Setting [CU_OCCUPANCY_DISABLE_CACHING_OVERRIDE](#) guarantees that the produced launch configuration is global caching compatible at a potential cost of occupancy. More information can be found about this feature in the "Unified L1/Texture Cache" section of the Maxwell tuning guide.

Note that the API can also be used with context-less kernel [CUkernel](#) by querying the handle using [cuLibraryGetKernel\(\)](#) and then passing it to the API by casting to [CUfunction](#). Here, the context to use for calculations will be the current context.



Note:

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

See also:

[cudaOccupancyMaxPotentialBlockSizeWithFlags](#)

CUresult cuOccupancyMaxPotentialClusterSize (int *clusterSize, CUfunction func, const CUlaunchConfig *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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_UNKNOWN](#)

Description

The cluster dimensions in `config` are ignored. If `func` has a required cluster size set (see [cudaFuncGetAttributes](#) / [cuFuncGetAttribute](#)), `*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 that the API can also be used with context-less kernel [CUkernel](#) by querying the handle using [cuLibraryGetKernel\(\)](#) and then passing it to the API by casting to [CUfunction](#). Here, the context to use for calculations will either be taken from the specified stream `config->hStream` or the current context in case of NULL stream.



Note:

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

See also:

[cudaFuncGetAttributes](#), [cuFuncGetAttribute](#)

6.26. Texture Reference Management [DEPRECATED]

This section describes the deprecated texture reference management functions of the low-level CUDA driver application programming interface.

CUresult cuTexRefCreate (CUtexref *pTexRef)

Creates a texture reference.

Parameters

pTexRef

- Returned texture reference

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Creates a texture reference and returns its handle in `*pTexRef`. Once created, the application must call [cuTexRefSetArray\(\)](#) or [cuTexRefSetAddress\(\)](#) to associate the reference with allocated memory. Other texture reference functions are used to specify the format and interpretation (addressing, filtering, etc.) to be used when the memory is read through this texture reference.

See also:

[cuTexRefDestroy](#)

CUresult cuTexRefDestroy (CUtexref hTexRef)

Destroys a texture reference.

Parameters

hTexRef

- Texture reference to destroy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

[Deprecated](#)

Destroys the texture reference specified by hTexRef.

See also:

[cuTexRefCreate](#)

CUresult cuTexRefGetAddress (CUdeviceptr *pdptr, CUtexref hTexRef)

Gets the address associated with a texture reference.

Parameters

pdptr

- Returned device address

hTexRef

- Texture reference

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

[Deprecated](#)

Returns in `*pdptr` the base address bound to the texture reference `hTexRef`, or returns [`CUDA\_ERROR\_INVALID\_VALUE`](#) if the texture reference is not bound to any device memory range.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

CUresult cuTexRefGetAddressMode (CUaddress_mode *pam, CUtexref hTexRef, int dim)

Gets the addressing mode used by a texture reference.

Parameters

pam

- Returned addressing mode

hTexRef

- Texture reference

dim

- Dimension

Returns

[`CUDA\_SUCCESS`](#), [`CUDA\_ERROR\_DEINITIALIZED`](#), [`CUDA\_ERROR\_NOT\_INITIALIZED`](#), [`CUDA\_ERROR\_INVALID\_CONTEXT`](#), [`CUDA\_ERROR\_INVALID\_VALUE`](#)

Description

Deprecated

Returns in `*pam` the addressing mode corresponding to the dimension `dim` of the texture reference `hTexRef`. Currently, the only valid value for `dim` are 0 and 1.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

CUresult cuTexRefGetArray (CUarray *phArray, CUtexref hTexRef)

Gets the array bound to a texture reference.

Parameters

phArray

- Returned array

hTexRef

- Texture reference

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Returns in *phArray the CUDA array bound to the texture reference hTexRef, or returns [CUDA_ERROR_INVALID_VALUE](#) if the texture reference is not bound to any CUDA array.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

CUresult cuTexRefGetBorderColor (float *pBorderColor, CUtexref hTexRef)

Gets the border color used by a texture reference.

Parameters

pBorderColor

- Returned Type and Value of RGBA color

hTexRef

- Texture reference

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Returns in `pBorderColor`, values of the RGBA color used by the texture reference `hTexRef`. The color value is of type float and holds color components in the following sequence: `pBorderColor[0]` holds 'R' component `pBorderColor[1]` holds 'G' component `pBorderColor[2]` holds 'B' component `pBorderColor[3]` holds 'A' component

See also:

[`cuTexRefSetAddressMode`](#), [`cuTexRefSetAddressMode`](#), [`cuTexRefSetBorderColor`](#)

CUresult cuTexRefGetFilterMode (CUfilter_mode *pfm, CUtexref hTexRef)

Gets the filter-mode used by a texture reference.

Parameters

pfm

- Returned filtering mode

hTexRef

- Texture reference

Returns

[`CUDA\_SUCCESS`](#), [`CUDA\_ERROR\_DEINITIALIZED`](#), [`CUDA\_ERROR\_NOT\_INITIALIZED`](#), [`CUDA\_ERROR\_INVALID\_CONTEXT`](#), [`CUDA\_ERROR\_INVALID\_VALUE`](#)

Description

Deprecated

Returns in `*pfm` the filtering mode of the texture reference `hTexRef`.

See also:

[`cuTexRefSetAddress`](#), [`cuTexRefSetAddress2D`](#), [`cuTexRefSetAddressMode`](#), [`cuTexRefSetArray`](#), [`cuTexRefSetFilterMode`](#), [`cuTexRefSetFlags`](#), [`cuTexRefSetFormat`](#), [`cuTexRefGetAddress`](#), [`cuTexRefGetAddressMode`](#), [`cuTexRefGetArray`](#), [`cuTexRefGetFlags`](#), [`cuTexRefGetFormat`](#)

CUresult cuTexRefGetFlags (unsigned int *pFlags, CUtexref hTexRef)

Gets the flags used by a texture reference.

Parameters

pFlags

- Returned flags

hTexRef

- Texture reference

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Returns in *pFlags the flags of the texture reference hTexRef.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFormat](#)

CUresult cuTexRefGetFormat (CUarray_format *pFormat, int *pNumChannels, CUtexref hTexRef)

Gets the format used by a texture reference.

Parameters

pFormat

- Returned format

pNumChannels

- Returned number of components

hTexRef

- Texture reference

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Returns in `*pFormat` and `*pNumChannels` the format and number of components of the CUDA array bound to the texture reference `hTexRef`. If `pFormat` or `pNumChannels` is NULL, it will be ignored.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#)

CUresult cuTexRefGetMaxAnisotropy (int *pmaxAniso, CUtexref hTexRef)

Gets the maximum anisotropy for a texture reference.

Parameters

pmaxAniso

- Returned maximum anisotropy

hTexRef

- Texture reference

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Returns the maximum anisotropy in `pmaxAniso` that's used when reading memory through the texture reference `hTexRef`.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

CUresult cuTexRefGetMipmapFilterMode (CUfilter_mode *pfm, CUtexref hTexRef)

Gets the mipmap filtering mode for a texture reference.

Parameters

pfm

- Returned mipmap filtering mode

hTexRef

- Texture reference

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Returns the mipmap filtering mode in `pfm` that's used when reading memory through the texture reference `hTexRef`.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

CUresult cuTexRefGetMipmapLevelBias (float *pbias, CUtexref hTexRef)

Gets the mipmap level bias for a texture reference.

Parameters

pbias

- Returned mipmap level bias

hTexRef

- Texture reference

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Returns the mipmap level bias in `pBias` that's added to the specified mipmap level when reading memory through the texture reference `hTexRef`.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

CUresult cuTexRefGetMipmapLevelClamp (float *pminMipmapLevelClamp, float *pmaxMipmapLevelClamp, CUtexref hTexRef)

Gets the min/max mipmap level clamps for a texture reference.

Parameters

pminMipmapLevelClamp

- Returned mipmap min level clamp

pmaxMipmapLevelClamp

- Returned mipmap max level clamp

hTexRef

- Texture reference

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Returns the min/max mipmap level clamps in `pminMipmapLevelClamp` and `pmaxMipmapLevelClamp` that's used when reading memory through the texture reference `hTexRef`.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

CUresult cuTexRefGetMipmappedArray (CUmipmappedArray *phMipmappedArray, CUtexref hTexRef)

Gets the mipmapped array bound to a texture reference.

Parameters

phMipmappedArray

- Returned mipmapped array

hTexRef

- Texture reference

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Returns in *phMipmappedArray the CUDA mipmapped array bound to the texture reference hTexRef, or returns [CUDA_ERROR_INVALID_VALUE](#) if the texture reference is not bound to any CUDA mipmapped array.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#),
[cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#),
[cuTexRefGetAddressMode](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

CUresult cuTexRefSetAddress (size_t *ByteOffset, CUtexref hTexRef, CUdeviceptr dptr, size_t bytes)

Binds an address as a texture reference.

Parameters

ByteOffset

- Returned byte offset

hTexRef

- Texture reference to bind

dptr

- Device pointer to bind

bytes

- Size of memory to bind in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

DescriptionDeprecated

Binds a linear address range to the texture reference `hTexRef`. Any previous address or CUDA array state associated with the texture reference is superseded by this function. Any memory previously bound to `hTexRef` is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, [cuTexRefSetAddress\(\)](#) passes back a byte offset in `*ByteOffset` that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the `tex1Dfetch()` function.

If the device memory pointer was returned from [cuMemAlloc\(\)](#), the offset is guaranteed to be 0 and NULL may be passed as the `ByteOffset` parameter.

The total number of elements (or texels) in the linear address range cannot exceed [CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LINEAR_WIDTH](#). The number of elements is computed as $(bytes / bytesPerElement)$, where `bytesPerElement` is determined from the data format and number of components set using [cuTexRefSetFormat\(\)](#).

See also:

[cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#),
[cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#),
[cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

**CUresult cuTexRefSetAddress2D (CUtexref hTexRef,
const CUDA_ARRAY_DESCRIPTOR *desc, CUdeviceptr
dptr, size_t Pitch)**

Binds an address as a 2D texture reference.

Parameters**hTexRef**

- Texture reference to bind

desc

- Descriptor of CUDA array

dptr

- Device pointer to bind

Pitch

- Line pitch in bytes

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

DescriptionDeprecated

Binds a linear address range to the texture reference `hTexRef`. Any previous address or CUDA array state associated with the texture reference is superseded by this function. Any memory previously bound to `hTexRef` is unbound.

Using a `tex2D()` function inside a kernel requires a call to either [cuTexRefSetArray\(\)](#) to bind the corresponding texture reference to an array, or [cuTexRefSetAddress2D\(\)](#) to bind the texture reference to linear memory.

Function calls to [cuTexRefSetFormat\(\)](#) cannot follow calls to [cuTexRefSetAddress2D\(\)](#) for the same texture reference.

It is required that `dptr` be aligned to the appropriate hardware-specific texture alignment. You can query this value using the device attribute [CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT](#). If an unaligned `dptr` is supplied, [CUDA_ERROR_INVALID_VALUE](#) is returned.

`Pitch` has to be aligned to the hardware-specific texture pitch alignment. This value can be queried using the device attribute [CU_DEVICE_ATTRIBUTE_TEXTURE_PITCH_ALIGNMENT](#). If an unaligned `Pitch` is supplied, [CUDA_ERROR_INVALID_VALUE](#) is returned.

Width and Height, which are specified in elements (or texels), cannot exceed [CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_WIDTH](#) and [CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_HEIGHT](#) respectively. `Pitch`, which is specified in bytes, cannot exceed [CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_PITCH](#).

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#),
[cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#),
[cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

CUresult cuTexRefSetAddressMode (CUtexref hTexRef, int dim, CUaddress_mode am)

Sets the addressing mode for a texture reference.

Parameters

hTexRef

- Texture reference

dim

- Dimension

am

- Addressing mode to set

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Specifies the addressing mode [am](#) for the given dimension [dim](#) of the texture reference [hTexRef](#). If [dim](#) is zero, the addressing mode is applied to the first parameter of the functions used to fetch from the texture; if [dim](#) is 1, the second, and so on. [CUaddress_mode](#) is defined as:

```
typedef enum CUaddress_mode_enum {
    CU_TR_ADDRESS_MODE_WRAP = 0,
    CU_TR_ADDRESS_MODE_CLAMP = 1,
    CU_TR_ADDRESS_MODE_MIRROR = 2,
    CU_TR_ADDRESS_MODE_BORDER = 3
} CUaddress_mode;
```

Note that this call has no effect if [hTexRef](#) is bound to linear memory. Also, if the flag, [CU_TRSF_NORMALIZED_COORDINATES](#), is not set, the only supported address mode is [CU_TR_ADDRESS_MODE_CLAMP](#).

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

CUresult cuTexRefSetArray (CUtexref hTexRef, CUarray hArray, unsigned int Flags)

Binds an array as a texture reference.

Parameters

hTexRef

- Texture reference to bind

hArray

- Array to bind

Flags

- Options (must be [CU_TRSA_OVERRIDE_FORMAT](#))

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Binds the CUDA array `hArray` to the texture reference `hTexRef`. Any previous address or CUDA array state associated with the texture reference is superseded by this function. `Flags` must be set to [CU_TRSA_OVERRIDE_FORMAT](#). Any CUDA array previously bound to `hTexRef` is unbound.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

CUresult cuTexRefSetBorderColor (CUtexref hTexRef, float *pBorderColor)

Sets the border color for a texture reference.

Parameters

hTexRef

- Texture reference

pBorderColor

- RGBA color

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Specifies the value of the RGBA color via the `pBorderColor` to the texture reference `hTexRef`. The color value supports only float type and holds color components in the following sequence: `pBorderColor[0]` holds 'R' component `pBorderColor[1]` holds 'G' component `pBorderColor[2]` holds 'B' component `pBorderColor[3]` holds 'A' component

Note that the color values can be set only when the Address mode is set to `CU_TR_ADDRESS_MODE_BORDER` using [cuTexRefSetAddressMode](#). Applications using integer border color values have to "reinterpret_cast" their values to float.

See also:

[cuTexRefSetAddressMode](#), [cuTexRefGetAddressMode](#), [cuTexRefGetBorderColor](#)

CUresult cuTexRefSetFilterMode (CUtexref hTexRef, CUfilter_mode fm)

Sets the filtering mode for a texture reference.

Parameters

hTexRef

- Texture reference

fm

- Filtering mode to set

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Specifies the filtering mode `fm` to be used when reading memory through the texture reference `hTexRef`. `CUfilter_mode_enum` is defined as:

```
↑ typedef enum CUfilter_mode_enum {
    CU_TR_FILTER_MODE_POINT = 0,
    CU_TR_FILTER_MODE_LINEAR = 1
} CUfilter_mode;
```

Note that this call has no effect if `hTexRef` is bound to linear memory.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#),
[cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#),
[cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

CUresult cuTexRefSetFlags (CUtexref hTexRef, unsigned int Flags)

Sets the flags for a texture reference.

Parameters

hTexRef

- Texture reference

Flags

- Optional flags to set

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Specifies optional flags via `Flags` to specify the behavior of data returned through the texture reference `hTexRef`. The valid flags are:

- ▶ [CU_TRSF_READ_AS_INTEGER](#), which suppresses the default behavior of having the texture promote integer data to floating point data in the range [0, 1]. Note that texture with 32-bit integer format would not be promoted, regardless of whether or not this flag is specified;
- ▶ [CU_TRSF_NORMALIZED_COORDINATES](#), which suppresses the default behavior of having the texture coordinates range from [0, Dim) where Dim is the width or height of the CUDA array. Instead, the texture coordinates [0, 1.0) reference the entire breadth of the array dimension;
- ▶ [CU_TRSF_DISABLE_TRILINEAR_OPTIMIZATION](#), which disables any trilinear filtering optimizations. Trilinear optimizations improve texture filtering performance by allowing bilinear filtering on textures in scenarios where it can closely approximate the expected results.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#),
[cuTexRefSetFilterMode](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#),
[cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

CUresult cuTexRefSetFormat (CUtexref hTexRef, CUarray_format fmt, int NumPackedComponents)

Sets the format for a texture reference.

Parameters

hTexRef

- Texture reference

fmt

- Format to set

NumPackedComponents

- Number of components per array element

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Specifies the format of the data to be read by the texture reference `hTexRef`. `fmt` and `NumPackedComponents` are exactly analogous to the `Format` and `NumChannels` members of the `CUDA_ARRAY_DESCRIPTOR` structure: They specify the format of each component and the number of components per array element.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#), [cudaCreateChannelDesc](#)

CUresult cuTexRefSetMaxAnisotropy (CUtexref hTexRef, unsigned int maxAniso)

Sets the maximum anisotropy for a texture reference.

Parameters

hTexRef

- Texture reference

maxAniso

- Maximum anisotropy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

[Deprecated](#)

Specifies the maximum anisotropy `maxAniso` to be used when reading memory through the texture reference `hTexRef`.

Note that this call has no effect if `hTexRef` is bound to linear memory.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#),
[cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#),
[cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

CUresult cuTexRefSetMipmapFilterMode (CUtexref hTexRef, CUfilter_mode fm)

Sets the mipmap filtering mode for a texture reference.

Parameters

hTexRef

- Texture reference

fm

- Filtering mode to set

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

[Deprecated](#)

Specifies the mipmap filtering mode `fm` to be used when reading memory through the texture reference `hTexRef`. `CUfilter_mode_enum` is defined as:

```
typedef enum CUfilter_mode_enum {
    CU_TR_FILTER_MODE_POINT = 0,
    CU_TR_FILTER_MODE_LINEAR = 1
} CUfilter_mode;
```

Note that this call has no effect if `hTexRef` is not bound to a mipmapped array.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

CUresult cuTexRefSetMipmapLevelBias (CUtexref hTexRef, float bias)

Sets the mipmap level bias for a texture reference.

Parameters

hTexRef

- Texture reference

bias

- Mipmap level bias

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Specifies the mipmap level bias `bias` to be added to the specified mipmap level when reading memory through the texture reference `hTexRef`.

Note that this call has no effect if `hTexRef` is not bound to a mipmapped array.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

CUresult cuTexRefSetMipmapLevelClamp (CUtexref hTexRef, float minMipmapLevelClamp, float maxMipmapLevelClamp)

Sets the mipmap min/max mipmap level clamps for a texture reference.

Parameters

hTexRef

- Texture reference

minMipmapLevelClamp

- Mipmap min level clamp

maxMipmapLevelClamp

- Mipmap max level clamp

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

DescriptionDeprecated

Specifies the min/max mipmap level clamps, `minMipmapLevelClamp` and `maxMipmapLevelClamp` respectively, to be used when reading memory through the texture reference `hTexRef`.

Note that this call has no effect if `hTexRef` is not bound to a mipmapped array.

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#),
[cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#),
[cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

CUresult cuTexRefSetMipmappedArray (CUtexref hTexRef, CUmipmappedArray hMipmappedArray, unsigned int Flags)

Binds a mipmapped array to a texture reference.

Parameters**hTexRef**

- Texture reference to bind

hMipmappedArray

- Mipmapped array to bind

Flags

- Options (must be [CU_TRSA_OVERRIDE_FORMAT](#))

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Binds the CUDA mipmapped array `hMipmappedArray` to the texture reference `hTexRef`. Any previous address or CUDA array state associated with the texture reference is superseded by this function. `Flags` must be set to CUDA_TRSA_OVERRIDE_FORMAT. Any CUDA array previously bound to `hTexRef` is unbound.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

6.27. Surface Reference Management [DEPRECATED]

This section describes the surface reference management functions of the low-level CUDA driver application programming interface.

CUresult cuSurfRefGetArray (CUarray *phArray, CUsurfref hSurfRef)

Passes back the CUDA array bound to a surface reference.

Parameters

phArray

- Surface reference handle

hSurfRef

- Surface reference handle

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Deprecated

Returns in `*phArray` the CUDA array bound to the surface reference `hSurfRef`, or returns CUDA_ERROR_INVALID_VALUE if the surface reference is not bound to any CUDA array.

See also:

[cuModuleGetSurfRef](#), [cuSurfRefSetArray](#)

CUresult cuSurfRefSetArray (CUsurfref hSurfRef, CUarray hArray, unsigned int Flags)

Sets the CUDA array for a surface reference.

Parameters

hSurfRef

- Surface reference handle

hArray

- CUDA array handle

Flags

- set to 0

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated

Sets the CUDA array `hArray` to be read and written by the surface reference `hSurfRef`. Any previous CUDA array state associated with the surface reference is superseded by this function. `Flags` must be set to 0. The [CUDA_ARRAY3D_SURFACE_LDST](#) flag must have been set for the CUDA array. Any CUDA array previously bound to `hSurfRef` is unbound.

See also:

[cuModuleGetSurfRef](#), [cuSurfRefGetArray](#)

6.28. Texture Object Management

This section describes the texture object management functions of the low-level CUDA driver application programming interface. The texture object API is only supported on devices of compute capability 3.0 or higher.

```
CUresult cuTexObjectCreate (CUtexObject *pTexObject,
const CUDA_RESOURCE_DESC *pResDesc,
const CUDA_TEXTURE_DESC *pTexDesc, const
CUDA_RESOURCE_VIEW_DESC *pResViewDesc)
```

Creates a texture object.

Parameters

pTexObject

- Texture object to create

pResDesc

- Resource descriptor

pTexDesc

- Texture descriptor

pResViewDesc

- Resource view descriptor

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

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 `CUDA_RESOURCE_DESC` structure is defined as:

```
typedef struct CUDA_RESOURCE_DESC_st
{
    CUresourcetype resType;

    union {
        struct {
            CUarray hArray;
        } array;
        struct {
            CUmipmappedArray hMipmappedArray;
        } mipmap;
        struct {
            CUdeviceptr devPtr;
            CUarray_format format;
            unsigned int numChannels;
        }
    }
}
```

```

        size_t sizeInBytes;
    } linear;
    struct {
        CUdeviceptr devPtr;
        CUarray_format format;
        unsigned int numChannels;
        size_t width;
        size_t height;
        size_t pitchInBytes;
    } pitch2D;
} res;

unsigned int flags;
} CUDA_RESOURCE_DESC;

```

where:

- [CUDA_RESOURCE_DESC::resType](#) specifies the type of resource to texture from. CUresourceType is defined as:

```

typedef enum CUresourcetype_enum {
    CU_RESOURCE_TYPE_ARRAY = 0x00,
    CU_RESOURCE_TYPE_MIPMAPPED_ARRAY = 0x01,
    CU_RESOURCE_TYPE_LINEAR = 0x02,
    CU_RESOURCE_TYPE_PITCH2D = 0x03
} CUresourcetype;

```

If [CUDA_RESOURCE_DESC::resType](#) is set to [CU_RESOURCE_TYPE_ARRAY](#), [CUDA_RESOURCE_DESC::res::array::hArray](#) must be set to a valid CUDA array handle.

If [CUDA_RESOURCE_DESC::resType](#) is set to [CU_RESOURCE_TYPE_MIPMAPPED_ARRAY](#), [CUDA_RESOURCE_DESC::res::mipmap::hMipmappedArray](#) must be set to a valid CUDA mipmapped array handle.

If [CUDA_RESOURCE_DESC::resType](#) is set to [CU_RESOURCE_TYPE_LINEAR](#), [CUDA_RESOURCE_DESC::res::linear::devPtr](#) must be set to a valid device pointer, that is aligned to [CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT](#). [CUDA_RESOURCE_DESC::res::linear::format](#) and [CUDA_RESOURCE_DESC::res::linear::numChannels](#) describe the format of each component and the number of components per array element. [CUDA_RESOURCE_DESC::res::linear::sizeInBytes](#) specifies the size of the array in bytes. The total number of elements in the linear address range cannot exceed [CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LINEAR_WIDTH](#). The number of elements is computed as (sizeInBytes / (sizeof(format) * numChannels)).

If [CUDA_RESOURCE_DESC::resType](#) is set to [CU_RESOURCE_TYPE_PITCH2D](#), [CUDA_RESOURCE_DESC::res::pitch2D::devPtr](#) must be set to a valid device pointer, that is aligned to [CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT](#). [CUDA_RESOURCE_DESC::res::pitch2D::format](#) and [CUDA_RESOURCE_DESC::res::pitch2D::numChannels](#) describe the format of each component and the number of components per array element. [CUDA_RESOURCE_DESC::res::pitch2D::width](#) and [CUDA_RESOURCE_DESC::res::pitch2D::height](#) specify the width and height of the array in elements, and cannot exceed [CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_WIDTH](#) and [CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_HEIGHT](#) respectively.

`CUDA_RESOURCE_DESC::res::pitch2D::pitchInBytes` specifies the pitch between two rows in bytes and has to be aligned to `CU_DEVICE_ATTRIBUTE_TEXTURE_PITCH_ALIGNMENT`. Pitch cannot exceed `CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_PITCH`.

- flags must be set to zero.

The `CUDA_TEXTURE_DESC` struct is defined as

```
typedef struct CUDA_TEXTURE_DESC_st {
    CUaddress_mode addressMode[3];
    CUfilter_mode filterMode;
    unsigned int flags;
    unsigned int maxAnisotropy;
    CUfilter_mode mipmapFilterMode;
    float mipmapLevelBias;
    float minMipmapLevelClamp;
    float maxMipmapLevelClamp;
} CUDA_TEXTURE_DESC;
```

where

- `CUDA_TEXTURE_DESC::addressMode` specifies the addressing mode for each dimension of the texture data. `CUaddress_mode` is defined as:

```
typedef enum CUaddress_mode_enum {
    CU_TR_ADDRESS_MODE_WRAP = 0,
    CU_TR_ADDRESS_MODE_CLAMP = 1,
    CU_TR_ADDRESS_MODE_MIRROR = 2,
    CU_TR_ADDRESS_MODE_BORDER = 3
} CUaddress_mode;
```

This is ignored if `CUDA_RESOURCE_DESC::resType` is `CU_RESOURCE_TYPE_LINEAR`. Also, if the flag, `CU_TRSF_NORMALIZED_COORDINATES` is not set, the only supported address mode is `CU_TR_ADDRESS_MODE_CLAMP`.

- `CUDA_TEXTURE_DESC::filterMode` specifies the filtering mode to be used when fetching from the texture. `CUfilter_mode` is defined as:

```
typedef enum CUfilter_mode_enum {
    CU_TR_FILTER_MODE_POINT = 0,
    CU_TR_FILTER_MODE_LINEAR = 1
} CUfilter_mode;
```

This is ignored if `CUDA_RESOURCE_DESC::resType` is `CU_RESOURCE_TYPE_LINEAR`.

- `CUDA_TEXTURE_DESC::flags` can be any combination of the following:
 - `CU_TRSF_READ_AS_INTEGER`, which suppresses the default behavior of having the texture promote integer data to floating point data in the range [0, 1]. Note that texture with 32-bit integer format would not be promoted, regardless of whether or not this flag is specified.
 - `CU_TRSF_NORMALIZED_COORDINATES`, which suppresses the default behavior of having the texture coordinates range from [0, Dim) where Dim is the width or height of the CUDA array. Instead, the texture coordinates [0, 1.0) reference the entire breadth of the array dimension; Note that for CUDA mipmapped arrays, this flag has to be set.
 - `CU_TRSF_DISABLE_TRILINEAR_OPTIMIZATION`, which disables any trilinear filtering optimizations. Trilinear optimizations improve texture filtering performance by allowing bilinear filtering on textures in scenarios where it can closely approximate the expected results.
 - `CU_TRSF_SEAMLESS_CUBEMAP`, which enables seamless cube map filtering. 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 `CUDA_ARRAY3D_CUBEMAP`. When seamless cube map filtering is enabled, texture address modes specified by `CUDA_TEXTURE_DESC::addressMode` are ignored. Instead, if the `CUDA_TEXTURE_DESC::filterMode` is set to `CU_TR_FILTER_MODE_POINT` the address mode `CU_TR_ADDRESS_MODE_CLAMP` will be applied for all dimensions. If the `CUDA_TEXTURE_DESC::filterMode` is set to `CU_TR_FILTER_MODE_LINEAR` seamless cube map filtering will be performed when sampling along the cube face borders.

- ▶ `CUDA_TEXTURE_DESC::maxAnisotropy` specifies the maximum anisotropy ratio to be used when doing anisotropic filtering. This value will be clamped to the range [1,16].
- ▶ `CUDA_TEXTURE_DESC::mipmapFilterMode` specifies the filter mode when the calculated mipmap level lies between two defined mipmap levels.
- ▶ `CUDA_TEXTURE_DESC::mipmapLevelBias` specifies the offset to be applied to the calculated mipmap level.
- ▶ `CUDA_TEXTURE_DESC::minMipmapLevelClamp` specifies the lower end of the mipmap level range to clamp access to.
- ▶ `CUDA_TEXTURE_DESC::maxMipmapLevelClamp` specifies the upper end of the mipmap level range to clamp access to.

The `CUDA_RESOURCE_VIEW_DESC` struct is defined as

```
typedef struct CUDA_RESOURCE_VIEW_DESC_st
{
    CUresourceViewFormat format;
    size_t width;
    size_t height;
    size_t depth;
    unsigned int firstMipmapLevel;
    unsigned int lastMipmapLevel;
    unsigned int firstLayer;
    unsigned int lastLayer;
} CUDA_RESOURCE_VIEW_DESC;
```

where:

- ▶ `CUDA_RESOURCE_VIEW_DESC::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 base of format `CU_AD_FORMAT_UNSIGNED_INT32` with 2 or 4 channels, depending on the block compressed format. For ex., BC1 and BC4 require the underlying CUDA array to have a format of `CU_AD_FORMAT_UNSIGNED_INT32` with 2 channels. The other BC formats require the underlying resource to have the same base format but with 4 channels.
- ▶ `CUDA_RESOURCE_VIEW_DESC::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.

- ▶ [`CUDA\_RESOURCE\_VIEW\_DESC::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.
- ▶ [`CUDA\_RESOURCE\_VIEW\_DESC::depth`](#) specifies the new depth of the texture data. This value has to be equal to that of the original resource.
- ▶ [`CUDA\_RESOURCE\_VIEW\_DESC::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. [`CUDA\_TEXTURE\_DESC::minMipmapLevelClamp`](#) and [`CUDA\_TEXTURE\_DESC::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.
- ▶ [`CUDA\_RESOURCE\_VIEW\_DESC::lastMipmapLevel`](#) specifies the least detailed mipmap level. For non-mipmapped resources, this value has to be zero.
- ▶ [`CUDA\_RESOURCE\_VIEW\_DESC::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.
- ▶ [`CUDA\_RESOURCE\_VIEW\_DESC::lastLayer`](#) specifies the last layer index for layered textures. For non-layered resources, this value has to be zero.

See also:

[`cuTexObjectDestroy`](#), [`cudaCreateTextureObject`](#)

CUresult cuTexObjectDestroy (CUtexObject texObject)

Destroys a texture object.

Parameters

texObject

- Texture object to destroy

Returns

[`CUDA\_SUCCESS`](#), [`CUDA\_ERROR\_DEINITIALIZED`](#), [`CUDA\_ERROR\_NOT\_INITIALIZED`](#), [`CUDA\_ERROR\_INVALID\_CONTEXT`](#), [`CUDA\_ERROR\_INVALID\_VALUE`](#)

Description

Destroys the texture object specified by `texObject`.

See also:

[`cuTexObjectCreate`](#), [`cudaDestroyTextureObject`](#)

CUresult cuTexObjectGetResourceDesc (CUDA_RESOURCE_DESC *pResDesc, CUtexObject texObject)

Returns a texture object's resource descriptor.

Parameters

pResDesc

- Resource descriptor

texObject

- Texture object

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns the resource descriptor for the texture object specified by `texObject`.

See also:

[cuTexObjectCreate](#), [cudaGetTextureObjectResourceDesc](#),

CUresult cuTexObjectGetResourceViewDesc (CUDA_RESOURCE_VIEW_DESC *pResViewDesc, CUtexObject texObject)

Returns a texture object's resource view descriptor.

Parameters

pResViewDesc

- Resource view descriptor

texObject

- Texture object

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns the resource view descriptor for the texture object specified by `texObject`. If no resource view was set for `texObject`, the [CUDA_ERROR_INVALID_VALUE](#) is returned.

See also:

[cuTexObjectCreate](#), [cudaGetTextureObjectResourceViewDesc](#)

CUresult cuTexObjectGetTextureDesc (CUDA_TEXTURE_DESC *pTexDesc, CUtexObject texObject)

Returns a texture object's texture descriptor.

Parameters

pTexDesc

- Texture descriptor

texObject

- Texture object

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns the texture descriptor for the texture object specified by `texObject`.

See also:

[cuTexObjectCreate](#), [cudaGetTextureObjectTextureDesc](#)

6.29. Surface Object Management

This section describes the surface object management functions of the low-level CUDA driver application programming interface. The surface object API is only supported on devices of compute capability 3.0 or higher.

CUresult cuSurfObjectCreate (CUsurfObject *pSurfObject, const CUDA_RESOURCE_DESC *pResDesc)

Creates a surface object.

Parameters

pSurfObject

- Surface object to create

pResDesc

- Resource descriptor

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Creates a surface object and returns it in `pSurfObject`. `pResDesc` describes the data to perform surface load/stores on. [CUDA_RESOURCE_DESC::resType](#) must be [CU_RESOURCE_TYPE_ARRAY](#) and `CUDA_RESOURCE_DESC::res::array::hArray` must be set to a valid CUDA array handle. [CUDA_RESOURCE_DESC::flags](#) must be set to zero.

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.

See also:

[cuSurfObjectDestroy](#), [cudaCreateSurfaceObject](#)

CUresult cuSurfObjectDestroy (CUsurfObject surfObject)

Destroys a surface object.

Parameters

surfObject

- Surface object to destroy

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Destroys the surface object specified by `surfObject`.

See also:

[cuSurfObjectCreate](#), [cudaDestroySurfaceObject](#)

CUresult cuSurfObjectGetResourceDesc (CUDA_RESOURCE_DESC *pResDesc, CUsurfObject surfObject)

Returns a surface object's resource descriptor.

Parameters

pResDesc

- Resource descriptor

surfObject

- Surface object

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns the resource descriptor for the surface object specified by `surfObject`.

See also:

[cuSurfObjectCreate](#), [cudaGetSurfaceObjectResourceDesc](#)

6.30. Tensor Map Object Managment

This section describes the tensor map object management functions of the low-level CUDA driver application programming interface. The tensor core API is only supported on devices of compute capability 9.0 or higher.

```
CUresult cuTensorMapEncodeIm2col (CUtensorMap
*tensorMap, CUtensorMapDataType tensorDataType,
cuint32_t tensorRank, void *globalAddress,
const cuint64_t *globalDim, const cuint64_t
*globalStrides, const int *pixelBoxLowerCorner, const
int *pixelBoxUpperCorner, cuint32_t channelsPerPixel,
cuint32_t pixelsPerColumn, const cuint32_t
*elementStrides, CUtensorMapInterleave interleave,
CUtensorMapSwizzle swizzle, CUtensorMapL2promotion
l2Promotion, CUtensorMapFloatOOBfill oobFill)
```

Create a tensor map descriptor object representing im2col memory region.

Parameters

tensorMap

- Tensor map object to create

tensorDataType

- Tensor data type

tensorRank

- Dimensionality of tensor; must be at least 3

globalAddress

- Starting address of memory region described by tensor

globalDim

- Array containing tensor size (number of elements) along each of the `tensorRank` dimensions

globalStrides

- Array containing stride size (in bytes) along each of the `tensorRank - 1` dimensions

pixelBoxLowerCorner

- Array containing DHW dimensions of lower box corner

pixelBoxUpperCorner

- Array containing DHW dimensions of upper box corner

channelsPerPixel

- Number of channels per pixel

pixelsPerColumn

- Number of pixels per column

elementStrides

- Array containing traversal stride in each of the `tensorRank` dimensions

interleave

- Type of interleaved layout the tensor addresses

swizzle

- Bank swizzling pattern inside shared memory

l2Promotion

- L2 promotion size

oobFill

- Indicate whether zero or special NaN constant will be used to fill out-of-bound elements

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Creates a descriptor for Tensor Memory Access (TMA) object specified by the parameters describing a `im2col` memory layout and returns it in `tensorMap`.

Tensor map objects are only supported on devices of compute capability 9.0 or higher. Additionally, a tensor map object is an opaque value, and, as such, should only be accessed through CUDA APIs and PTX.

The parameters passed are bound to the following requirements:

- `tensorMap` address must be aligned to 64 bytes.
- `tensorDataType` has to be an enum from `CUtensorMapDataType` which is defined as:

```
typedef enum CUtensorMapDataType_enum {
    CU_TENSOR_MAP_DATA_TYPE_UINT8 = 0,           // 1 byte
    CU_TENSOR_MAP_DATA_TYPE_UINT16,              // 2 bytes
    CU_TENSOR_MAP_DATA_TYPE_UINT32,              // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_INT32,               // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_UINT64,              // 8 bytes
    CU_TENSOR_MAP_DATA_TYPE_INT64,               // 8 bytes
    CU_TENSOR_MAP_DATA_TYPE_FLOAT16,             // 2 bytes
    CU_TENSOR_MAP_DATA_TYPE_FLOAT32,             // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_FLOAT64,             // 8 bytes
    CU_TENSOR_MAP_DATA_TYPE_BFLOAT16,           // 2 bytes
    CU_TENSOR_MAP_DATA_TYPE_FLOAT32_FTZ,        // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_TFLOAT32,          // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_TFLOAT32_FTZ,      // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN8B,      // 4 bits
    CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B,     // 4 bits
    CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B,     // 6 bits
} CUtensorMapDataType;
```

`CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN8B` copies '16 x U4' packed values to memory aligned as 8 bytes. There are no gaps between packed values.

`CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B` copies '16 x U4' packed values to memory aligned as 16 bytes. There are 8 byte gaps between every 8 byte chunk of packed values.

`CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B` copies '16 x U6' packed values to memory aligned as 16 bytes. There are 4 byte gaps between every 12 byte chunk of packed values.

- `tensorRank`, which specifies the number of tensor dimensions, must be 3, 4, or 5.

- ▶ `globalAddress`, which specifies the starting address of the memory region described, must be 16 byte aligned. The following requirements need to also be met:
 - ▶ When `interleave` is `CU_TENSOR_MAP_INTERLEAVE_32B`, `globalAddress` must be 32 byte aligned.
 - ▶ When `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B` or `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, `globalAddress` must be 32 byte aligned.
- ▶ `globalDim` array, which specifies tensor size of each of the `tensorRank` dimensions, must be non-zero and less than or equal to 2^{32} . Additionally, the following requirements need to be met for the packed data types:
 - ▶ When `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B` or `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, `globalDim[0]` must be a multiple of 128.
 - ▶ When `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN8B`, `globalDim[0]` must be a multiple of 2.
 - ▶ Dimension for the packed data types must reflect the number of individual U# values.
- ▶ `globalStrides` array, which specifies tensor stride of each of the lower `tensorRank - 1` dimensions in bytes, must be a multiple of 16 and less than 2^{40} . Additionally, the following requirements need to be met:
 - ▶ When `interleave` is `CU_TENSOR_MAP_INTERLEAVE_32B`, the strides must be a multiple of 32.
 - ▶ When `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B` or `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, the strides must be a multiple of 32.

Each following dimension specified includes previous dimension stride:

```
globalStrides[0] = globalDim[0] * elementSizeInBytes(tensorDataType) + padding[0];
for (i = 1; i < tensorRank - 1; i++)
    globalStrides[i] = globalStrides[i - 1] * (globalDim[i] + padding[i]);
assert(globalStrides[i] >= globalDim[i]);
```
- ▶ `pixelBoxLowerCorner` array specifies the coordinate offsets {D, H, W} of the bounding box from top/left/front corner. The number of offsets and their precision depend on the tensor dimensionality:
 - ▶ When `tensorRank` is 3, one signed offset within range [-32768, 32767] is supported.
 - ▶ When `tensorRank` is 4, two signed offsets each within range [-128, 127] are supported.
 - ▶ When `tensorRank` is 5, three offsets each within range [-16, 15] are supported.
- ▶ `pixelBoxUpperCorner` array specifies the coordinate offsets {D, H, W} of the bounding box from bottom/right/back corner. The number of offsets and their precision depend on the tensor dimensionality:
 - ▶ When `tensorRank` is 3, one signed offset within range [-32768, 32767] is supported.

- ▶ When `tensorRank` is 4, two signed offsets each within range `[-128, 127]` are supported.
- ▶ When `tensorRank` is 5, three offsets each within range `[-16, 15]` are supported. The bounding box specified by `pixelBoxLowerCorner` and `pixelBoxUpperCorner` must have non-zero area.
- ▶ `channelsPerPixel`, which specifies the number of elements which must be accessed along C dimension, must be less than or equal to 256. Additionally, when `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B` or `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, `channelsPerPixel` must be 128.
- ▶ `pixelsPerColumn`, which specifies the number of elements that must be accessed along the {N, D, H, W} dimensions, must be less than or equal to 1024.
- ▶ `elementStrides` array, which specifies the iteration step along each of the `tensorRank` dimensions, must be non-zero and less than or equal to 8. Note that when `interleave` is `CU_TENSOR_MAP_INTERLEAVE_NONE`, the first element of this array is ignored since TMA doesn't support the stride for dimension zero. When all elements of the `elementStrides` array are one, `boxDim` specifies the number of elements to load. However, if `elementStrides[i]` is not equal to one for some `i`, then TMA loads `ceil( boxDim[i] / elementStrides[i] )` number of elements along `i`-th dimension. To load `N` elements along `i`-th dimension, `boxDim[i]` must be set to `N * elementStrides[i]`.

- ▶ `interleave` specifies the interleaved layout of type [CUtensorMapInterleave](#), which is defined as:

```
typedef enum CUtensorMapInterleave_enum {
    CU_TENSOR_MAP_INTERLEAVE_NONE = 0,
    CU_TENSOR_MAP_INTERLEAVE_16B,
    CU_TENSOR_MAP_INTERLEAVE_32B
} CUtensorMapInterleave;
```

TMA supports interleaved layouts like NC/8HWC8 where C8 utilizes 16 bytes in memory assuming 2 byte per channel or NC/16HWC16 where C16 uses 32 bytes. When `interleave` is `CU_TENSOR_MAP_INTERLEAVE_NONE` and `swizzle` is not `CU_TENSOR_MAP_SWIZZLE_NONE`, the bounding box inner dimension (computed as `channelsPerPixel` multiplied by element size in bytes derived from `tensorDataType`) must be less than or equal to the swizzle size.

- ▶ `CU_TENSOR_MAP_SWIZZLE_32B` requires the bounding box inner dimension to be `<= 32`.
- ▶ `CU_TENSOR_MAP_SWIZZLE_64B` requires the bounding box inner dimension to be `<= 64`.
- ▶ `CU_TENSOR_MAP_SWIZZLE_128B*` require the bounding box inner dimension to be `<= 128`. Additionally, `tensorDataType` of `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B` requires `interleave` to be `CU_TENSOR_MAP_INTERLEAVE_NONE`.
- ▶ `swizzle`, which specifies the shared memory bank swizzling pattern, has to be of type [CUtensorMapSwizzle](#) which is defined as:

```
typedef enum CUtensorMapSwizzle_enum {
    CU_TENSOR_MAP_SWIZZLE_NONE = 0,
    CU_TENSOR_MAP_SWIZZLE_32B, // Swizzle 16B chunks
    within 32B span
}
```

```

        CU_TENSOR_MAP_SWIZZLE_64B,                // Swizzle 16B chunks
within 64B span
        CU_TENSOR_MAP_SWIZZLE_128B,              // Swizzle 16B chunks
within 128B span
        CU_TENSOR_MAP_SWIZZLE_128B_ATOM_32B,      // Swizzle 32B chunks
within 128B span
        CU_TENSOR_MAP_SWIZZLE_128B_ATOM_32B_FLIP_8B, // Swizzle 32B chunks
within 128B span, additionally swap lower 8B with upper 8B within each 16B for
every alternate row
        CU_TENSOR_MAP_SWIZZLE_128B_ATOM_64B      // Swizzle 64B chunks
within 128B span
    } CUtensorMapSwizzle;

```

Data are organized in a specific order in global memory; however, this may not match the order in which the application accesses data in shared memory. This difference in data organization may cause bank conflicts when shared memory is accessed. In order to avoid this problem, data can be loaded to shared memory with shuffling across shared memory banks. When `interleave` is `CU_TENSOR_MAP_INTERLEAVE_32B`, `swizzle` must be `CU_TENSOR_MAP_SWIZZLE_32B`. Other `interleave` modes can have any swizzling pattern. When the `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B`, only the following swizzle modes are supported:

- ▶ `CU_TENSOR_MAP_SWIZZLE_NONE` (Load & Store)
- ▶ `CU_TENSOR_MAP_SWIZZLE_128B` (Load & Store)
- ▶ `CU_TENSOR_MAP_SWIZZLE_128B_ATOM_32B` (Load & Store)
- ▶ `CU_TENSOR_MAP_SWIZZLE_128B_ATOM_64B` (Store only) When the `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, only the following swizzle modes are supported:
 - ▶ `CU_TENSOR_MAP_SWIZZLE_NONE` (Load only)
 - ▶ `CU_TENSOR_MAP_SWIZZLE_128B` (Load only)
 - ▶ `CU_TENSOR_MAP_SWIZZLE_128B_ATOM_32B` (Load only)
- ▶ `l2Promotion` specifies L2 fetch size which indicates the byte granularity at which L2 requests are filled from DRAM. It must be of type [CUtensorMapL2promotion](#), which is defined as:

```

↑ typedef enum CUtensorMapL2promotion_enum {
    CU_TENSOR_MAP_L2_PROMOTION_NONE = 0,
    CU_TENSOR_MAP_L2_PROMOTION_L2_64B,
    CU_TENSOR_MAP_L2_PROMOTION_L2_128B,
    CU_TENSOR_MAP_L2_PROMOTION_L2_256B
} CUtensorMapL2promotion;

```

- ▶ `oobFill`, which indicates whether zero or a special NaN constant should be used to fill out-of-bound elements, must be of type [CUtensorMapFloatOOBfill](#) which is defined as:

```

↑ typedef enum CUtensorMapFloatOOBfill_enum {
    CU_TENSOR_MAP_FLOAT_OOB_FILL_NONE = 0,
    CU_TENSOR_MAP_FLOAT_OOB_FILL_NAN_REQUEST_ZERO_FMA
} CUtensorMapFloatOOBfill;

```

Note that `CU_TENSOR_MAP_FLOAT_OOB_FILL_NAN_REQUEST_ZERO_FMA` can only be used when `tensorDataType` represents a floating-point data type, and when `tensorDataType` is not `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN8B`, `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, and `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B`.

See also:

[cuTensorMapEncodeTiled](#), [cuTensorMapEncodeIm2colWide](#), [cuTensorMapReplaceAddress](#)

```
CUresult cuTensorMapEncodeIm2colWide (CUtensorMap
*tensorMap, CUtensorMapDataType tensorDataType,
cuint32_t tensorRank, void *globalAddress,
const cuuint64_t *globalDim, const cuuint64_t
*globalStrides, int pixelBoxLowerCornerWidth,
int pixelBoxUpperCornerWidth, cuuint32_t
channelsPerPixel, cuuint32_t pixelsPerColumn, const
cuint32_t *elementStrides, CUtensorMapInterleave
interleave, CUtensorMapIm2ColWideMode mode,
CUtensorMapSwizzle swizzle, CUtensorMapL2promotion
l2Promotion, CUtensorMapFloatOOBfill oobFill)
```

Create a tensor map descriptor object representing im2col memory region, but where the elements are exclusively loaded along the W dimension.

Parameters

tensorMap

- Tensor map object to create

tensorDataType

- Tensor data type

tensorRank

- Dimensionality of tensor; must be at least 3

globalAddress

- Starting address of memory region described by tensor

globalDim

- Array containing tensor size (number of elements) along each of the `tensorRank` dimensions

globalStrides

- Array containing stride size (in bytes) along each of the `tensorRank - 1` dimensions

pixelBoxLowerCornerWidth

- Width offset of left box corner

pixelBoxUpperCornerWidth

- Width offset of right box corner

channelsPerPixel

- Number of channels per pixel

pixelsPerColumn

- Number of pixels per column

elementStrides

- Array containing traversal stride in each of the `tensorRank` dimensions

interleave

- Type of interleaved layout the tensor addresses

mode

- W or W128 mode

swizzle

- Bank swizzling pattern inside shared memory

l2Promotion

- L2 promotion size

oobFill

- Indicate whether zero or special NaN constant will be used to fill out-of-bound elements

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Creates a descriptor for Tensor Memory Access (TMA) object specified by the parameters describing a `im2col` memory layout and where the row is always loaded along the W dimension and returns it in `tensorMap`. This assumes the tensor layout in memory is either NDHWC, NHWC, or NWC.

This API is only supported on devices of compute capability 10.0 or higher. Additionally, a tensor map object is an opaque value, and, as such, should only be accessed through CUDA APIs and PTX.

The parameters passed are bound to the following requirements:

- `tensorMap` address must be aligned to 64 bytes.
- `tensorDataType` has to be an enum from [CUtensorMapDataType](#) which is defined as:

```
typedef enum CUtensorMapDataType_enum {
    CU_TENSOR_MAP_DATA_TYPE_UINT8 = 0,           // 1 byte
    CU_TENSOR_MAP_DATA_TYPE_UINT16,              // 2 bytes
    CU_TENSOR_MAP_DATA_TYPE_UINT32,              // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_INT32,               // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_UINT64,              // 8 bytes
    CU_TENSOR_MAP_DATA_TYPE_INT64,               // 8 bytes
    CU_TENSOR_MAP_DATA_TYPE_FLOAT16,             // 2 bytes
    CU_TENSOR_MAP_DATA_TYPE_FLOAT32,             // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_FLOAT64,             // 8 bytes
    CU_TENSOR_MAP_DATA_TYPE_BFLOAT16,            // 2 bytes
    CU_TENSOR_MAP_DATA_TYPE_FLOAT32_FTZ,         // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_TFLOAT32,           // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_TFLOAT32_FTZ,       // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN8B,       // 4 bits
    CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B,      // 4 bits
    CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B       // 6 bits
} CUtensorMapDataType;
```

CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN8B copies '16 x U4' packed values to memory aligned as 8 bytes. There are no gaps between packed values.

CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B copies '16 x U4' packed values to memory aligned as 16 bytes. There are 8 byte gaps between every 8 byte chunk of packed values.

CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B copies '16 x U6' packed values to memory aligned as 16 bytes. There are 4 byte gaps between every 12 byte chunk of packed values.

- ▶ `tensorRank`, which specifies the number of tensor dimensions, must be 3, 4, or 5.
- ▶ `globalAddress`, which specifies the starting address of the memory region described, must be 16 byte aligned. The following requirements need to also be met:
 - ▶ When `interleave` is `CU_TENSOR_MAP_INTERLEAVE_32B`, `globalAddress` must be 32 byte aligned.
 - ▶ When `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B` or `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, `globalAddress` must be 32 byte aligned.
- ▶ `globalDim` array, which specifies tensor size of each of the `tensorRank` dimensions, must be non-zero and less than or equal to 2^{32} . Additionally, the following requirements need to be met for the packed data types:
 - ▶ When `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B` or `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, `globalDim[0]` must be a multiple of 128.
 - ▶ When `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN8B`, `globalDim[0]` must be a multiple of 2.
 - ▶ Dimension for the packed data types must reflect the number of individual U# values.
- ▶ `globalStrides` array, which specifies tensor stride of each of the lower `tensorRank - 1` dimensions in bytes, must be a multiple of 16 and less than 2^{40} . Additionally, the following requirements need to be met:
 - ▶ When `interleave` is `CU_TENSOR_MAP_INTERLEAVE_32B`, the strides must be a multiple of 32.
 - ▶ When `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B` or `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, the strides must be a multiple of 32.

Each following dimension specified includes previous dimension stride:

```

↑
globalStrides[0] = globalDim[0] * elementSizeInBytes(tensorDataType) +
padding[0];
    for (i = 1; i < tensorRank - 1; i++)
        globalStrides[i] = globalStrides[i - 1] * (globalDim[i] +
padding[i]);
        assert(globalStrides[i] >= globalDim[i]);

```
- ▶ `pixelBoxLowerCornerWidth` specifies the coordinate offset W of the bounding box from left corner. The offset must be within range `[-32768, 32767]`.

- ▶ `pixelBoxUpperCornerWidth` specifies the coordinate offset W of the bounding box from right corner. The offset must be within range [-32768, 32767].

The bounding box specified by `pixelBoxLowerCornerWidth` and `pixelBoxUpperCornerWidth` must have non-zero area. Note that the size of the box along D and H dimensions is always equal to one.

- ▶ `channelsPerPixel`, which specifies the number of elements which must be accessed along C dimension, must be less than or equal to 256. Additionally, when `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B` or `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, `channelsPerPixel` must be 128.
- ▶ `pixelsPerColumn`, which specifies the number of elements that must be accessed along the W dimension, must be less than or equal to 1024. This field is ignored when mode is `CU_TENSOR_MAP_IM2COL_WIDE_MODE_W128`.
- ▶ `elementStrides` array, which specifies the iteration step along each of the `tensorRank` dimensions, must be non-zero and less than or equal to 8. Note that when `interleave` is `CU_TENSOR_MAP_INTERLEAVE_NONE`, the first element of this array is ignored since TMA doesn't support the stride for dimension zero. When all elements of the `elementStrides` array are one, `boxDim` specifies the number of elements to load. However, if `elementStrides[i]` is not equal to one for some i, then TMA loads `ceil(boxDim[i] / elementStrides[i])` number of elements along i-th dimension. To load N elements along i-th dimension, `boxDim[i]` must be set to `N * elementStrides[i]`.
- ▶ `interleave` specifies the interleaved layout of type [CUtensorMapInterleave](#), which is defined as:

```
typedef enum CUtensorMapInterleave_enum {
    CU_TENSOR_MAP_INTERLEAVE_NONE = 0,
    CU_TENSOR_MAP_INTERLEAVE_16B,
    CU_TENSOR_MAP_INTERLEAVE_32B
} CUtensorMapInterleave;
```

TMA supports interleaved layouts like NC/8HWC8 where C8 utilizes 16 bytes in memory assuming 2 byte per channel or NC/16HWC16 where C16 uses 32 bytes. When `interleave` is `CU_TENSOR_MAP_INTERLEAVE_NONE`, the bounding box inner dimension (computed as `channelsPerPixel` multiplied by element size in bytes derived from `tensorDataType`) must be less than or equal to the swizzle size.

- ▶ `CU_TENSOR_MAP_SWIZZLE_64B` requires the bounding box inner dimension to be ≤ 64 .
- ▶ `CU_TENSOR_MAP_SWIZZLE_128B*` require the bounding box inner dimension to be ≤ 128 . Additionally, `tensorDataType` of `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B` requires `interleave` to be `CU_TENSOR_MAP_INTERLEAVE_NONE`.
- ▶ `mode`, which describes loading of elements loaded along the W dimension, has to be one of the following [CUtensorMapIm2ColWideMode](#) types:

```
CU_TENSOR_MAP_IM2COL_WIDE_MODE_W,
CU_TENSOR_MAP_IM2COL_WIDE_MODE_W128
```

CU_TENSOR_MAP_IM2COL_WIDE_MODE_W allows the number of elements loaded along the W dimension to be specified via the `pixelsPerColumn` field.

- `swizzle`, which specifies the shared memory bank swizzling pattern, must be one of the following [CUtensorMapSwizzle](#) modes (other swizzle modes are not supported):

```
↑ typedef enum CUtensorMapSwizzle_enum {
    CU_TENSOR_MAP_SWIZZLE_64B,           // Swizzle 16B chunks
    within 64B span
    CU_TENSOR_MAP_SWIZZLE_128B,          // Swizzle 16B chunks
    within 128B span
    CU_TENSOR_MAP_SWIZZLE_128B_ATOM_32B, // Swizzle 32B chunks
    within 128B span
} CUtensorMapSwizzle;
```

Data are organized in a specific order in global memory; however, this may not match the order in which the application accesses data in shared memory. This difference in data organization may cause bank conflicts when shared memory is accessed. In order to avoid this problem, data can be loaded to shared memory with shuffling across shared memory banks. When the `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B`, only the following swizzle modes are supported:

- `CU_TENSOR_MAP_SWIZZLE_128B` (Load & Store)
- `CU_TENSOR_MAP_SWIZZLE_128B_ATOM_32B` (Load & Store) When the `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, only the following swizzle modes are supported:
- `CU_TENSOR_MAP_SWIZZLE_128B` (Load only)
- `CU_TENSOR_MAP_SWIZZLE_128B_ATOM_32B` (Load only)
- `l2Promotion` specifies L2 fetch size which indicates the byte granularity at which L2 requests are filled from DRAM. It must be of type [CUtensorMapL2promotion](#), which is defined as:

```
↑ typedef enum CUtensorMapL2promotion_enum {
    CU_TENSOR_MAP_L2_PROMOTION_NONE = 0,
    CU_TENSOR_MAP_L2_PROMOTION_L2_64B,
    CU_TENSOR_MAP_L2_PROMOTION_L2_128B,
    CU_TENSOR_MAP_L2_PROMOTION_L2_256B
} CUtensorMapL2promotion;
```

- `oobFill`, which indicates whether zero or a special NaN constant should be used to fill out-of-bound elements, must be of type [CUtensorMapFloatOOBfill](#) which is defined as:

```
↑ typedef enum CUtensorMapFloatOOBfill_enum {
    CU_TENSOR_MAP_FLOAT_OOB_FILL_NONE = 0,
    CU_TENSOR_MAP_FLOAT_OOB_FILL_NAN_REQUEST_ZERO_FMA
} CUtensorMapFloatOOBfill;
```

Note that `CU_TENSOR_MAP_FLOAT_OOB_FILL_NAN_REQUEST_ZERO_FMA` can only be used when `tensorDataType` represents a floating-point data type, and when `tensorDataType` is not `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN8B`, `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, and `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B`.

See also:

[cuTensorMapEncodeTiled](#), [cuTensorMapEncodeIm2col](#), [cuTensorMapReplaceAddress](#)

CUresult cuTensorMapEncodeTiled (CUtensorMap *tensorMap, CUtensorMapDataType tensorDataType, cuuint32_t tensorRank, void *globalAddress, const cuuint64_t *globalDim, const cuuint64_t *globalStrides, const cuuint32_t *boxDim, const cuuint32_t *elementStrides, CUtensorMapInterleave interleave, CUtensorMapSwizzle swizzle, CUtensorMapL2promotion l2Promotion, CUtensorMapFloatOOBfill oobFill)

Create a tensor map descriptor object representing tiled memory region.

Parameters

tensorMap

- Tensor map object to create

tensorDataType

- Tensor data type

tensorRank

- Dimensionality of tensor

globalAddress

- Starting address of memory region described by tensor

globalDim

- Array containing tensor size (number of elements) along each of the `tensorRank` dimensions

globalStrides

- Array containing stride size (in bytes) along each of the `tensorRank - 1` dimensions

boxDim

- Array containing traversal box size (number of elements) along each of the `tensorRank` dimensions. Specifies how many elements to be traversed along each tensor dimension.

elementStrides

- Array containing traversal stride in each of the `tensorRank` dimensions

interleave

- Type of interleaved layout the tensor addresses

swizzle

- Bank swizzling pattern inside shared memory

l2Promotion

- L2 promotion size

oobFill

- Indicate whether zero or special NaN constant must be used to fill out-of-bound elements

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Creates a descriptor for Tensor Memory Access (TMA) object specified by the parameters describing a tiled region and returns it in `tensorMap`.

Tensor map objects are only supported on devices of compute capability 9.0 or higher. Additionally, a tensor map object is an opaque value, and, as such, should only be accessed through CUDA APIs and PTX.

The parameters passed are bound to the following requirements:

- ▶ `tensorMap` address must be aligned to 64 bytes.
- ▶ `tensorDataType` has to be an enum from [CUtensorMapDataType](#) which is defined as:

```
typedef enum CUtensorMapDataType_enum {
    CU_TENSOR_MAP_DATA_TYPE_UINT8 = 0,          // 1 byte
    CU_TENSOR_MAP_DATA_TYPE_UINT16,             // 2 bytes
    CU_TENSOR_MAP_DATA_TYPE_UINT32,             // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_INT32,              // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_UINT64,             // 8 bytes
    CU_TENSOR_MAP_DATA_TYPE_INT64,              // 8 bytes
    CU_TENSOR_MAP_DATA_TYPE_FLOAT16,            // 2 bytes
    CU_TENSOR_MAP_DATA_TYPE_FLOAT32,            // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_FLOAT64,            // 8 bytes
    CU_TENSOR_MAP_DATA_TYPE_BFLOAT16,           // 2 bytes
    CU_TENSOR_MAP_DATA_TYPE_FLOAT32_FTZ,        // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_TFLOAT32,           // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_TFLOAT32_FTZ,       // 4 bytes
    CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN8B,       // 4 bits
    CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B,       // 4 bits
    CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B       // 6 bits
} CUtensorMapDataType;
```

`CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN8B` copies '16 x U4' packed values to memory aligned as 8 bytes. There are no gaps between packed values.

`CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B` copies '16 x U4' packed values to memory aligned as 16 bytes. There are 8 byte gaps between every 8 byte chunk of packed values.

`CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B` copies '16 x U6' packed values to memory aligned as 16 bytes. There are 4 byte gaps between every 12 byte chunk of packed values.

- ▶ `tensorRank` must be non-zero and less than or equal to the maximum supported dimensionality of 5. If `interleave` is not `CU_TENSOR_MAP_INTERLEAVE_NONE`, then `tensorRank` must additionally be greater than or equal to 3.
- ▶ `globalAddress`, which specifies the starting address of the memory region described, must be 16 byte aligned. The following requirements need to also be met:
 - ▶ When `interleave` is `CU_TENSOR_MAP_INTERLEAVE_32B`, `globalAddress` must be 32 byte aligned.

- ▶ When `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B` or `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, `globalAddress` must be 32 byte aligned.
- ▶ `globalDim` array, which specifies tensor size of each of the `tensorRank` dimensions, must be non-zero and less than or equal to 2^{32} . Additionally, the following requirements need to be met for the packed data types:
 - ▶ When `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B` or `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, `globalDim[0]` must be a multiple of 128.
 - ▶ When `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN8B`, `globalDim[0]` must be a multiple of 2.
 - ▶ Dimension for the packed data types must reflect the number of individual U# values.
- ▶ `globalStrides` array, which specifies tensor stride of each of the lower `tensorRank - 1` dimensions in bytes, must be a multiple of 16 and less than 2^{40} . Additionally, the following requirements need to be met:
 - ▶ When `interleave` is `CU_TENSOR_MAP_INTERLEAVE_32B`, the strides must be a multiple of 32.
 - ▶ When `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B` or `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, the strides must be a multiple of 32. Each following dimension specified includes previous dimension stride:


```
↑
    globalStrides[0] = globalDim[0] * elementSizeInBytes(tensorDataType) +
padding[0];
    for (i = 1; i < tensorRank - 1; i++)
        globalStrides[i] = globalStrides[i - 1] * (globalDim[i] +
padding[i]);
    assert(globalStrides[i] >= globalDim[i]);
```
- ▶ `boxDim` array, which specifies number of elements to be traversed along each of the `tensorRank` dimensions, must be non-zero and less than or equal to 256. Additionally, the following requirements need to be met:
 - ▶ When `interleave` is `CU_TENSOR_MAP_INTERLEAVE_NONE`, `{ boxDim[0] * elementSizeInBytes( tensorDataType ) }` must be a multiple of 16 bytes.
 - ▶ When `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B` or `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, `boxDim[0]` must be 128.
- ▶ `elementStrides` array, which specifies the iteration step along each of the `tensorRank` dimensions, must be non-zero and less than or equal to 8. Note that when `interleave` is `CU_TENSOR_MAP_INTERLEAVE_NONE`, the first element of this array is ignored since TMA doesn't support the stride for dimension zero. When all elements of `elementStrides` array is one, `boxDim` specifies the number of elements to load. However, if the `elementStrides[i]` is not equal to one, then TMA loads `ceil( boxDim[i] / elementStrides[i] )` number of elements along i-th dimension. To load N elements along i-th dimension, `boxDim[i]` must be set to `N * elementStrides[i]`.

- `interleave` specifies the interleaved layout of type `CUtensorMapInterleave`, which is defined as:

```
typedef enum CUtensorMapInterleave_enum {
    CU_TENSOR_MAP_INTERLEAVE_NONE = 0,
    CU_TENSOR_MAP_INTERLEAVE_16B,
    CU_TENSOR_MAP_INTERLEAVE_32B
} CUtensorMapInterleave;
```

TMA supports interleaved layouts like NC/8HWC8 where C8 utilizes 16 bytes in memory assuming 2 byte per channel or NC/16HWC16 where C16 uses 32 bytes. When `interleave` is `CU_TENSOR_MAP_INTERLEAVE_NONE` and `swizzle` is not `CU_TENSOR_MAP_SWIZZLE_NONE`, the bounding box inner dimension (computed as `boxDim[0]` multiplied by element size derived from `tensorDataType`) must be less than or equal to the swizzle size.

- `CU_TENSOR_MAP_SWIZZLE_32B` requires the bounding box inner dimension to be ≤ 32 .
- `CU_TENSOR_MAP_SWIZZLE_64B` requires the bounding box inner dimension to be ≤ 64 .
- `CU_TENSOR_MAP_SWIZZLE_128B*` require the bounding box inner dimension to be ≤ 128 . Additionally, `tensorDataType` of `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B` requires `interleave` to be `CU_TENSOR_MAP_INTERLEAVE_NONE`.
- `swizzle`, which specifies the shared memory bank swizzling pattern, has to be of type `CUtensorMapSwizzle` which is defined as:

```
typedef enum CUtensorMapSwizzle_enum {
    CU_TENSOR_MAP_SWIZZLE_NONE = 0,
    CU_TENSOR_MAP_SWIZZLE_32B, // Swizzle 16B chunks
    within 32B span
    CU_TENSOR_MAP_SWIZZLE_64B, // Swizzle 16B chunks
    within 64B span
    CU_TENSOR_MAP_SWIZZLE_128B, // Swizzle 16B chunks
    within 128B span
    CU_TENSOR_MAP_SWIZZLE_128B_ATOM_32B, // Swizzle 32B chunks
    within 128B span
    CU_TENSOR_MAP_SWIZZLE_128B_ATOM_32B_FLIP_8B, // Swizzle 32B chunks
    within 128B span, additionally swap lower 8B with upper 8B within each 16B for
    every alternate row
    CU_TENSOR_MAP_SWIZZLE_128B_ATOM_64B // Swizzle 64B chunks
    within 128B span
} CUtensorMapSwizzle;
```

Data are organized in a specific order in global memory; however, this may not match the order in which the application accesses data in shared memory. This difference in data organization may cause bank conflicts when shared memory is accessed. In order to avoid this problem, data can be loaded to shared memory with shuffling across shared memory banks. When `interleave` is `CU_TENSOR_MAP_INTERLEAVE_32B`, `swizzle` must be `CU_TENSOR_MAP_SWIZZLE_32B`. Other `interleave` modes can have any swizzling pattern. When the `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B`, only the following swizzle modes are supported:

- `CU_TENSOR_MAP_SWIZZLE_NONE` (Load & Store)
- `CU_TENSOR_MAP_SWIZZLE_128B` (Load & Store)
- `CU_TENSOR_MAP_SWIZZLE_128B_ATOM_32B` (Load & Store)

- ▶ `CU_TENSOR_MAP_SWIZZLE_128B_ATOM_64B` (Store only) When the `tensorDataType` is `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, only the following swizzle modes are supported:
- ▶ `CU_TENSOR_MAP_SWIZZLE_NONE` (Load only)
- ▶ `CU_TENSOR_MAP_SWIZZLE_128B` (Load only)
- ▶ `CU_TENSOR_MAP_SWIZZLE_128B_ATOM_32B` (Load only)
- ▶ `l2Promotion` specifies L2 fetch size which indicates the byte granularity at which L2 requests is filled from DRAM. It must be of type [CUtensorMapL2promotion](#), which is defined as:

```
typedef enum CUtensorMapL2promotion_enum {
    CU_TENSOR_MAP_L2_PROMOTION_NONE = 0,
    CU_TENSOR_MAP_L2_PROMOTION_L2_64B,
    CU_TENSOR_MAP_L2_PROMOTION_L2_128B,
    CU_TENSOR_MAP_L2_PROMOTION_L2_256B
} CUtensorMapL2promotion;
```

- ▶ `oobFill`, which indicates whether zero or a special NaN constant should be used to fill out-of-bound elements, must be of type [CUtensorMapFloatOOBfill](#) which is defined as:

```
typedef enum CUtensorMapFloatOOBfill_enum {
    CU_TENSOR_MAP_FLOAT_OOB_FILL_NONE = 0,
    CU_TENSOR_MAP_FLOAT_OOB_FILL_NAN_REQUEST_ZERO_FMA
} CUtensorMapFloatOOBfill;
```

Note that `CU_TENSOR_MAP_FLOAT_OOB_FILL_NAN_REQUEST_ZERO_FMA` can only be used when `tensorDataType` represents a floating-point data type, and when `tensorDataType` is not `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN8B`, `CU_TENSOR_MAP_DATA_TYPE_16U4_ALIGN16B`, and `CU_TENSOR_MAP_DATA_TYPE_16U6_ALIGN16B`.

See also:

[cuTensorMapEncodeIm2col](#), [cuTensorMapEncodeIm2colWide](#), [cuTensorMapReplaceAddress](#)

CUresult cuTensorMapReplaceAddress (CUtensorMap *tensorMap, void *globalAddress)

Modify an existing tensor map descriptor with an updated global address.

Parameters

tensorMap

- Tensor map object to modify

globalAddress

- Starting address of memory region described by tensor, must follow previous alignment requirements

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Modifies the descriptor for Tensor Memory Access (TMA) object passed in `tensorMap` with an updated `globalAddress`.

Tensor map objects are only supported on devices of compute capability 9.0 or higher. Additionally, a tensor map object is an opaque value, and, as such, should only be accessed through CUDA API calls.

See also:

[cuTensorMapEncodeTiled](#), [cuTensorMapEncodeIm2col](#) [cuTensorMapEncodeIm2colWide](#)

6.31. Peer Context Memory Access

This section describes the direct peer context memory access functions of the low-level CUDA driver application programming interface.

CUresult cuCtxDisablePeerAccess (CUcontext peerContext)

Disables direct access to memory allocations in a peer context and unregisters any registered allocations.

Parameters

peerContext

- Peer context to disable direct access to

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_PEER_ACCESS_NOT_ENABLED](#), [CUDA_ERROR_INVALID_CONTEXT](#),

Description

Returns [CUDA_ERROR_PEER_ACCESS_NOT_ENABLED](#) if direct peer access has not yet been enabled from `peerContext` to the current context.

Returns [CUDA_ERROR_INVALID_CONTEXT](#) if there is no current context, or if `peerContext` is not a valid context.



Note:

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

See also:

[cuDeviceCanAccessPeer](#), [cuCtxEnablePeerAccess](#), [cudaDeviceDisablePeerAccess](#)

CUresult cuCtxEnablePeerAccess (CUcontext peerContext, unsigned int Flags)

Enables direct access to memory allocations in a peer context.

Parameters

peerContext

- Peer context to enable direct access to from the current context

Flags

- Reserved for future use and must be set to 0

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED](#), [CUDA_ERROR_TOO_MANY_PEERS](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_PEER_ACCESS_UNSUPPORTED](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

If both the current context and `peerContext` are on devices which support unified addressing (as may be queried using [CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING](#)) and same major compute capability, then on success all allocations from `peerContext` will immediately be accessible by the current context. See [Unified Addressing](#) for additional details.

Note that access granted by this call is unidirectional and that in order to access memory from the current context in `peerContext`, a separate symmetric call to [cuCtxEnablePeerAccess\(\)](#) 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 [CUDA_ERROR_PEER_ACCESS_UNSUPPORTED](#) if [cuDeviceCanAccessPeer\(\)](#) indicates that the [CUdevice](#) of the current context cannot directly access memory from the [CUdevice](#) of `peerContext`.

Returns [CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED](#) if direct access of `peerContext` from the current context has already been enabled.

Returns [CUDA_ERROR_TOO_MANY_PEERS](#) if direct peer access is not possible because hardware resources required for peer access have been exhausted.

Returns [CUDA_ERROR_INVALID_CONTEXT](#) if there is no current context, `peerContext` is not a valid context, or if the current context is `peerContext`.

Returns [CUDA_ERROR_INVALID_VALUE](#) if `Flags` is not 0.



Note:

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

See also:

[cuDeviceCanAccessPeer](#), [cuCtxDisablePeerAccess](#), [cudaDeviceEnablePeerAccess](#)

CUresult cuDeviceCanAccessPeer (int *canAccessPeer, CUdevice dev, CUdevice peerDev)

Queries if a device may directly access a peer device's memory.

Parameters

canAccessPeer

- Returned access capability

dev

- Device from which allocations on `peerDev` are to be directly accessed.

peerDev

- Device on which the allocations to be directly accessed by `dev` reside.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

Returns in `*canAccessPeer` a value of 1 if contexts on `dev` are capable of directly accessing memory from contexts on `peerDev` and 0 otherwise. If direct access of `peerDev` from `dev` is possible, then access may be enabled on two specific contexts by calling [cuCtxEnablePeerAccess\(\)](#).



Note:

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

See also:

[cuCtxEnablePeerAccess](#), [cuCtxDisablePeerAccess](#), [cudaDeviceCanAccessPeer](#)

CUresult cuDeviceGetP2PAttribute (int *value, CUdevice_P2PAttribute attrib, CUdevice srcDevice, CUdevice dstDevice)

Queries attributes of the link between two devices.

Parameters

value

- Returned value of the requested attribute

attrib

- The requested attribute of the link between `srcDevice` and `dstDevice`.

srcDevice

- The source device of the target link.

dstDevice

- The destination device of the target link.

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_DEVICE, CUDA_ERROR_INVALID_VALUE

Description

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

- ▶ CU_DEVICE_P2P_ATTRIBUTE_PERFORMANCE_RANK: A relative value indicating the performance of the link between two devices.
- ▶ CU_DEVICE_P2P_ATTRIBUTE_ACCESS_SUPPORTED P2P: 1 if P2P Access is enable.
- ▶ CU_DEVICE_P2P_ATTRIBUTE_NATIVE_ATOMIC_SUPPORTED: 1 if Atomic operations over the link are supported.
- ▶ CU_DEVICE_P2P_ATTRIBUTE_CUDA_ARRAY_ACCESS_SUPPORTED: 1 if `cudaArray` can be accessed over the link.

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

Returns CUDA_ERROR_INVALID_VALUE if `attrib` is not valid or if `value` is a null pointer.



Note:

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

See also:

[cuCtxEnablePeerAccess](#), [cuCtxDisablePeerAccess](#), [cuDeviceCanAccessPeer](#),
[cudaDeviceGetP2PAttribute](#)

6.32. Graphics Interoperability

This section describes the graphics interoperability functions of the low-level CUDA driver application programming interface.

CUresult cuGraphicsMapResources (unsigned int count, CUgraphicsResource *resources, CUstream hStream)

Map graphics resources for access by CUDA.

Parameters

count

- Number of resources to map

resources

- Resources to map for CUDA usage

hStream

- Stream with which to synchronize

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_ALREADY_MAPPED](#), [CUDA_ERROR_UNKNOWN](#)

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 [cuGraphicsMapResources\(\)](#) will complete before any subsequent CUDA work issued in `stream` begins.

If `resources` includes any duplicate entries then [CUDA_ERROR_INVALID_HANDLE](#) is returned. If any of `resources` are presently mapped for access by CUDA then [CUDA_ERROR_ALREADY_MAPPED](#) is returned.



Note:

- ▶ This function uses standard [default stream](#) semantics.
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphicsResourceGetMappedPointer](#), [cuGraphicsSubResourceGetMappedArray](#),
[cuGraphicsUnmapResources](#), [cudaGraphicsMapResources](#)

CUresult cuGraphicsResourceGetMappedMipmappedArray (CUmipmappedArray *pMipmappedArray, CUgraphicsResource resource)

Get a mipmapped array through which to access a mapped graphics resource.

Parameters

pMipmappedArray

- Returned mipmapped array through which `resource` may be accessed

resource

- Mapped resource to access

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_MAPPED](#),
[CUDA_ERROR_NOT_MAPPED_AS_ARRAY](#)

Description

Returns in `*pMipmappedArray` a mipmapped array through which the mapped graphics resource `resource`. The value set in `*pMipmappedArray` may change every time that `resource` is mapped.

If `resource` is not a texture then it cannot be accessed via a mipmapped array and [CUDA_ERROR_NOT_MAPPED_AS_ARRAY](#) is returned. If `resource` is not mapped then [CUDA_ERROR_NOT_MAPPED](#) is returned.



Note:

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

See also:

[cuGraphicsResourceGetMappedPointer](#), [cudaGraphicsResourceGetMappedMipmappedArray](#)

CUresult cuGraphicsResourceGetMappedPointer (CUdeviceptr *pDevPtr, size_t *pSize, CUgraphicsResource resource)

Get a device pointer through which to access a mapped graphics resource.

Parameters

pDevPtr

- Returned pointer through which `resource` may be accessed

pSize

- Returned size of the buffer accessible starting at `*pPointer`

resource

- Mapped resource to access

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_MAPPED](#),
[CUDA_ERROR_NOT_MAPPED_AS_POINTER](#)

Description

Returns in `*pDevPtr` a pointer through which the mapped graphics resource `resource` may be accessed. Returns in `pSize` the size of the memory in bytes which may be accessed from that pointer. The value set in `pPointer` may change every time that `resource` is mapped.

If `resource` is not a buffer then it cannot be accessed via a pointer and [CUDA_ERROR_NOT_MAPPED_AS_POINTER](#) is returned. If `resource` is not mapped then [CUDA_ERROR_NOT_MAPPED](#) is returned. *



Note:

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

See also:

[cuGraphicsMapResources](#), [cuGraphicsSubResourceGetMappedArray](#),
[cudaGraphicsResourceGetMappedPointer](#)

CUresult cuGraphicsResourceSetMapFlags (CUgraphicsResource 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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_ALREADY_MAPPED](#)

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:

- ▶ [CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE](#): 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.
- ▶ [CU_GRAPHICS_MAP_RESOURCE_FLAGS_READONLY](#): Specifies that CUDA kernels which access this resource will not write to this resource.
- ▶ [CU_GRAPHICS_MAP_RESOURCE_FLAGS_WRITEDISCARD](#): 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 `resource` is presently mapped for access by CUDA then [CUDA_ERROR_ALREADY_MAPPED](#) is returned. If `flags` is not one of the above values then [CUDA_ERROR_INVALID_VALUE](#) is returned.



Note:

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

See also:

[cuGraphicsMapResources](#), [cudaGraphicsResourceSetMapFlags](#)

CUresult cuGraphicsSubResourceGetMappedArray (CUarray *pArray, CUgraphicsResource resource, unsigned int arrayIndex, unsigned int mipLevel)

Get an array through which to access a subresource of a mapped graphics resource.

Parameters

pArray

- 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 [CUarray_cubemap_face](#) for cubemap textures for the subresource to access

mipLevel

- Mipmap level for the subresource to access

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_MAPPED](#),
[CUDA_ERROR_NOT_MAPPED_AS_ARRAY](#)

Description

Returns in `*pArray` 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 `*pArray` may change every time that `resource` is mapped.

If `resource` is not a texture then it cannot be accessed via an array and [CUDA_ERROR_NOT_MAPPED_AS_ARRAY](#) is returned. If `arrayIndex` is not a valid array index for `resource` then [CUDA_ERROR_INVALID_VALUE](#) is returned. If `mipLevel` is not a valid mipmap level for `resource` then [CUDA_ERROR_INVALID_VALUE](#) is returned. If `resource` is not mapped then [CUDA_ERROR_NOT_MAPPED](#) is returned.



Note:

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

See also:

[cuGraphicsResourceGetMappedPointer](#), [cudaGraphicsSubResourceGetMappedArray](#)

CUresult cuGraphicsUnmapResources (unsigned int count, CUgraphicsResource *resources, CUstream hStream)

Unmap graphics resources.

Parameters

count

- Number of resources to unmap

resources

- Resources to unmap

hStream

- Stream with which to synchronize

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_MAPPED](#), [CUDA_ERROR_UNKNOWN](#)

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 [cuGraphicsUnmapResources\(\)](#) will complete before any subsequently issued graphics work begins.

If `resources` includes any duplicate entries then [CUDA_ERROR_INVALID_HANDLE](#) is returned. If any of `resources` are not presently mapped for access by CUDA then [CUDA_ERROR_NOT_MAPPED](#) is returned.



Note:

- ▶ This function uses standard [default stream](#) semantics.
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cuGraphicsMapResources](#), [cudaGraphicsUnmapResources](#)

CUresult cuGraphicsUnregisterResource (CUgraphicsResource resource)

Unregisters a graphics resource for access by CUDA.

Parameters

resource

- Resource to unregister

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_UNKNOWN](#)

Description

Unregisters the graphics resource `resource` so it is not accessible by CUDA unless registered again.

If `resource` is invalid then [CUDA_ERROR_INVALID_HANDLE](#) is returned.



Note:

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

See also:

[cuGraphicsD3D9RegisterResource](#), [cuGraphicsD3D10RegisterResource](#),
[cuGraphicsD3D11RegisterResource](#), [cuGraphicsGLRegisterBuffer](#), [cuGraphicsGLRegisterImage](#),
[cudaGraphicsUnregisterResource](#)

6.33. Driver Entry Point Access

This section describes the driver entry point access functions of the low-level CUDA driver application programming interface.

**CUresult cuGetProcAddress (const char *symbol,
void **pfn, int cudaVersion, cuuint64_t flags,
CUdriverProcAddressQueryResult *symbolStatus)**

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` and `cudaVersion` would be the ABI compatible CUDA version for the `_v2` variant.

pfn

- 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.

symbolStatus

- Optional location to store the status of the search for `symbol` based on `cudaVersion`. See [CUdriverProcAddressQueryResult](#) for possible values.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_SUPPORTED](#)

Description

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

The CUDA version is specified as $(1000 * \text{major} + 10 * \text{minor})$, 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`.

The API will return [CUDA_SUCCESS](#) and set the returned `pfn` to NULL if the requested driver function is not supported on the platform, no ABI compatible driver function exists for the specified `cudaVersion` or if the driver symbol is invalid.

It will also set the optional `symbolStatus` to one of the values in [CUdriverProcAddressQueryResult](#) with the following meanings:

- ▶ [CU_GET_PROC_ADDRESS_SUCCESS](#) - The requested symbol was successfully found based on input arguments and `pfn` is valid
- ▶ [CU_GET_PROC_ADDRESS_SYMBOL_NOT_FOUND](#) - The requested symbol was not found
- ▶ [CU_GET_PROC_ADDRESS_VERSION_NOT_SUFFICIENT](#) - The requested symbol was found but is not supported by `cudaVersion` specified

The requested flags can be:

- ▶ [CU_GET_PROC_ADDRESS_DEFAULT](#): This is the default mode. This is equivalent to [CU_GET_PROC_ADDRESS_PER_THREAD_DEFAULT_STREAM](#) if the code is compiled with `--default-stream per-thread` compilation flag or the macro `CUDA_API_PER_THREAD_DEFAULT_STREAM` is defined; [CU_GET_PROC_ADDRESS_LEGACY_STREAM](#) otherwise.
- ▶ [CU_GET_PROC_ADDRESS_LEGACY_STREAM](#): This will enable the search for all driver symbols that match the requested driver symbol name except the corresponding per-thread versions.
- ▶ [CU_GET_PROC_ADDRESS_PER_THREAD_DEFAULT_STREAM](#): 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](#).

See also:

[cudaGetDriverEntryPoint](#)

6.34. Coredump Attributes Control API

This section describes the coredump attribute control functions of the low-level CUDA driver application programming interface.

enum CUCoredumpGenerationFlags

Flags for controlling coredump contents

Values

CU_COREDUMP_DEFAULT_FLAGS = 0

CU_COREDUMP_SKIP_NONRELOCATED_ELF_IMAGES = (1<<0)

```

CU_COREDUMP_SKIP_GLOBAL_MEMORY = (1<<1)
CU_COREDUMP_SKIP_SHARED_MEMORY = (1<<2)
CU_COREDUMP_SKIP_LOCAL_MEMORY = (1<<3)
CU_COREDUMP_SKIP_ABORT = (1<<4)
CU_COREDUMP_SKIP_CONSTBANK_MEMORY = (1<<5)
CU_COREDUMP_LIGHTWEIGHT_FLAGS =
CU_COREDUMP_SKIP_NONRELOCATED_ELF_IMAGES |
CU_COREDUMP_SKIP_GLOBAL_MEMORY |
CU_COREDUMP_SKIP_SHARED_MEMORY | CU_COREDUMP_SKIP_LOCAL_MEMORY |
CU_COREDUMP_SKIP_CONSTBANK_MEMORY

```

enum CUcoredumpSettings

Flags for choosing a coredump attribute to get/set

Values

```

CU_COREDUMP_ENABLE_ON_EXCEPTION = 1
CU_COREDUMP_TRIGGER_HOST
CU_COREDUMP_LIGHTWEIGHT
CU_COREDUMP_ENABLE_USER_TRIGGER
CU_COREDUMP_FILE
CU_COREDUMP_PIPE
CU_COREDUMP_GENERATION_FLAGS
CU_COREDUMP_MAX

```

CUresult cuCoredumpGetAttribute (CUcoredumpSettings attrib, void *value, size_t *size)

Allows caller to fetch a coredump attribute value for the current context.

Parameters

attrib

- The enum defining which value to fetch.

value

- void* containing the requested data.

size

- The size of the memory region `value` points to.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_PERMITTED,
CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_CONTEXT IS DESTROYED

Description

Returns in `*value` the requested value specified by `attrib`. It is up to the caller to ensure that the data type and size of `*value` matches the request.

If the caller calls this function with `*value` equal to `NULL`, the size of the memory region (in bytes) expected for `attrib` will be placed in `size`.

The supported attributes are:

- ▶ `CU_COREDUMP_ENABLE_ON_EXCEPTION`: Bool where true means that GPU exceptions from this context will create a coredump at the location specified by `CU_COREDUMP_FILE`. The default value is false unless set to true globally or locally, or the `CU_CTX_USER_COREDUMP_ENABLE` flag was set during context creation.
- ▶ `CU_COREDUMP_TRIGGER_HOST`: Bool where true means that the host CPU will also create a coredump. The default value is true unless set to false globally or or locally. This value is deprecated as of CUDA 12.5 - raise the `CU_COREDUMP_SKIP_ABORT` flag to disable host device abort() if needed.
- ▶ `CU_COREDUMP_LIGHTWEIGHT`: Bool where true means that any resulting coredumps will not have a dump of GPU memory or non-reloc ELF images. The default value is false unless set to true globally or locally. This attribute is deprecated as of CUDA 12.5, please use `CU_COREDUMP_GENERATION_FLAGS` instead.
- ▶ `CU_COREDUMP_ENABLE_USER_TRIGGER`: Bool where true means that a coredump can be created by writing to the system pipe specified by `CU_COREDUMP_PIPE`. The default value is false unless set to true globally or locally.
- ▶ `CU_COREDUMP_FILE`: String of up to 1023 characters that defines the location where any coredumps generated by this context will be written. The default value is `core.cuda.HOSTNAME.PID` where `HOSTNAME` is the host name of the machine running the CUDA applications and `PID` is the process ID of the CUDA application.
- ▶ `CU_COREDUMP_PIPE`: String of up to 1023 characters that defines the name of the pipe that will be monitored if user-triggered coredumps are enabled. The default value is `corepipe.cuda.HOSTNAME.PID` where `HOSTNAME` is the host name of the machine running the CUDA application and `PID` is the process ID of the CUDA application.
- ▶ `CU_COREDUMP_GENERATION_FLAGS`: An integer with values to allow granular control the data contained in a coredump specified as a bitwise OR combination of the following values:
 - + `CU_COREDUMP_DEFAULT_FLAGS` - if set by itself, coredump generation returns to its default settings of including all memory regions that it is able to access
 - + `CU_COREDUMP_SKIP_NONRELOCATED_ELF_IMAGES` - Coredump will not include the data from CUDA source modules that are not relocated at runtime.
 - + `CU_COREDUMP_SKIP_GLOBAL_MEMORY` - Coredump will not include device-side global data that does not belong to any context.
 - + `CU_COREDUMP_SKIP_SHARED_MEMORY` - Coredump will not include grid-scale shared memory for the warp that the dumped kernel belonged to.
 - + `CU_COREDUMP_SKIP_LOCAL_MEMORY` - Coredump will not include local

memory from the kernel. + `CU_COREDUMP_LIGHTWEIGHT_FLAGS` - Enables all of the above options. Equivalent to setting the `CU_COREDUMP_LIGHTWEIGHT` attribute to true. + `CU_COREDUMP_SKIP_ABORT` - If set, GPU exceptions will not raise an abort() in the host CPU process. Same functional goal as `CU_COREDUMP_TRIGGER_HOST` but better reflects the default behavior.

See also:

[`cuCoredumpGetAttributeGlobal`](#), [`cuCoredumpSetAttribute`](#), [`cuCoredumpSetAttributeGlobal`](#)

CUresult cuCoredumpGetAttributeGlobal (CUcoredumpSettings attrib, void *value, size_t *size)

Allows caller to fetch a coredump attribute value for the entire application.

Parameters

attrib

- The enum defining which value to fetch.

value

- void* containing the requested data.

size

- The size of the memory region `value` points to.

Returns

[`CUDA\_SUCCESS`](#), [`CUDA\_ERROR\_INVALID\_VALUE`](#)

Description

Returns in `*value` the requested value specified by `attrib`. It is up to the caller to ensure that the data type and size of `*value` matches the request.

If the caller calls this function with `*value` equal to `NULL`, the size of the memory region (in bytes) expected for `attrib` will be placed in `size`.

The supported attributes are:

- ▶ `CU_COREDUMP_ENABLE_ON_EXCEPTION`: Bool where true means that GPU exceptions from this context will create a coredump at the location specified by `CU_COREDUMP_FILE`. The default value is false.
- ▶ `CU_COREDUMP_TRIGGER_HOST`: Bool where true means that the host CPU will also create a coredump. The default value is true unless set to false globally or locally. This value is deprecated as of CUDA 12.5 - raise the `CU_COREDUMP_SKIP_ABORT` flag to disable host device abort() if needed.

- ▶ **CU_COREDUMP_LIGHTWEIGHT**: Bool where true means that any resulting coredumps will not have a dump of GPU memory or non-reloc ELF images. The default value is false. This attribute is deprecated as of CUDA 12.5, please use **CU_COREDUMP_GENERATION_FLAGS** instead.
- ▶ **CU_COREDUMP_ENABLE_USER_TRIGGER**: Bool where true means that a coredump can be created by writing to the system pipe specified by **CU_COREDUMP_PIPE**. The default value is false.
- ▶ **CU_COREDUMP_FILE**: String of up to 1023 characters that defines the location where any coredumps generated by this context will be written. The default value is `core.cuda.HOSTNAME.PID` where `HOSTNAME` is the host name of the machine running the CUDA applications and `PID` is the process ID of the CUDA application.
- ▶ **CU_COREDUMP_PIPE**: String of up to 1023 characters that defines the name of the pipe that will be monitored if user-triggered coredumps are enabled. The default value is `corepipe.cuda.HOSTNAME.PID` where `HOSTNAME` is the host name of the machine running the CUDA application and `PID` is the process ID of the CUDA application.
- ▶ **CU_COREDUMP_GENERATION_FLAGS**: An integer with values to allow granular control the data contained in a coredump specified as a bitwise OR combination of the following values:
 - + **CU_COREDUMP_DEFAULT_FLAGS** - if set by itself, coredump generation returns to its default settings of including all memory regions that it is able to access
 - + **CU_COREDUMP_SKIP_NONRELOCATED_ELF_IMAGES** - Coredump will not include the data from CUDA source modules that are not relocated at runtime.
 - + **CU_COREDUMP_SKIP_GLOBAL_MEMORY** - Coredump will not include device-side global data that does not belong to any context.
 - + **CU_COREDUMP_SKIP_SHARED_MEMORY** - Coredump will not include grid-scale shared memory for the warp that the dumped kernel belonged to.
 - + **CU_COREDUMP_SKIP_LOCAL_MEMORY** - Coredump will not include local memory from the kernel.
 - + **CU_COREDUMP_LIGHTWEIGHT_FLAGS** - Enables all of the above options. Equivalent to setting the **CU_COREDUMP_LIGHTWEIGHT** attribute to true.
 - + **CU_COREDUMP_SKIP_ABORT** - If set, GPU exceptions will not raise an `abort()` in the host CPU process. Same functional goal as **CU_COREDUMP_TRIGGER_HOST** but better reflects the default behavior.

See also:

[cuCoredumpGetAttribute](#), [cuCoredumpSetAttribute](#), [cuCoredumpSetAttributeGlobal](#)

CUresult cuCoredumpSetAttribute (CUcoredumpSettings attrib, void *value, size_t *size)

Allows caller to set a coredump attribute value for the current context.

Parameters

attrib

- The enum defining which value to set.

value

- void* containing the requested data.

size

- The size of the memory region `value` points to.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_PERMITTED,
CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_CONTEXT_IS_DESTROYED,
CUDA_ERROR_NOT_SUPPORTED

Description

This function should be considered an alternate interface to the CUDA-GDB environment variables defined in this document: <https://docs.nvidia.com/cuda/cuda-gdb/index.html#gpu-coredump>

An important design decision to note is that any coredump environment variable values set before CUDA initializes will take permanent precedence over any values set with this function. This decision was made to ensure no change in behavior for any users that may be currently using these variables to get coredumps.

*`value` shall contain the requested value specified by `set`. It is up to the caller to ensure that the data type and size of *`value` matches the request.

If the caller calls this function with *`value` equal to NULL, the size of the memory region (in bytes) expected for `set` will be placed in `size`.

/note This function will return CUDA_ERROR_NOT_SUPPORTED if the caller attempts to set `CU_COREDUMP_ENABLE_ON_EXCEPTION` on a GPU of with Compute Capability < 6.0. [cuCoredumpSetAttributeGlobal](#) works on those platforms as an alternative.

/note `CU_COREDUMP_ENABLE_USER_TRIGGER` and `CU_COREDUMP_PIPE` cannot be set on a per-context basis.

The supported attributes are:

- ▶ `CU_COREDUMP_ENABLE_ON_EXCEPTION`: Bool where true means that GPU exceptions from this context will create a coredump at the location specified by `CU_COREDUMP_FILE`. The default value is false.
- ▶ `CU_COREDUMP_TRIGGER_HOST`: Bool where true means that the host CPU will also create a coredump. The default value is true unless set to false globally or or locally. This value is deprecated as of CUDA 12.5 - raise the `CU_COREDUMP_SKIP_ABORT` flag to disable host device abort() if needed.
- ▶ `CU_COREDUMP_LIGHTWEIGHT`: Bool where true means that any resulting coredumps will not have a dump of GPU memory or non-reloc ELF images. The default value is false. This attribute is deprecated as of CUDA 12.5, please use `CU_COREDUMP_GENERATION_FLAGS` instead.

- ▶ **CU_COREDUMP_FILE**: String of up to 1023 characters that defines the location where any coredumps generated by this context will be written. The default value is `core.cuda.HOSTNAME.PID` where `HOSTNAME` is the host name of the machine running the CUDA applications and `PID` is the process ID of the CUDA application.
- ▶ **CU_COREDUMP_GENERATION_FLAGS**: An integer with values to allow granular control the data contained in a coredump specified as a bitwise OR combination of the following values:
 - + **CU_COREDUMP_DEFAULT_FLAGS** - if set by itself, coredump generation returns to its default settings of including all memory regions that it is able to access
 - + **CU_COREDUMP_SKIP_NONRELOCATED_ELF_IMAGES** - Coredump will not include the data from CUDA source modules that are not relocated at runtime.
 - + **CU_COREDUMP_SKIP_GLOBAL_MEMORY** - Coredump will not include device-side global data that does not belong to any context.
 - + **CU_COREDUMP_SKIP_SHARED_MEMORY** - Coredump will not include grid-scale shared memory for the warp that the dumped kernel belonged to.
 - + **CU_COREDUMP_SKIP_LOCAL_MEMORY** - Coredump will not include local memory from the kernel.
 - + **CU_COREDUMP_LIGHTWEIGHT_FLAGS** - Enables all of the above options. Equivalent to setting the **CU_COREDUMP_LIGHTWEIGHT** attribute to true.
 - + **CU_COREDUMP_SKIP_ABORT** - If set, GPU exceptions will not raise an `abort()` in the host CPU process. Same functional goal as **CU_COREDUMP_TRIGGER_HOST** but better reflects the default behavior.

See also:

[`cuCoredumpGetAttributeGlobal`](#), [`cuCoredumpGetAttribute`](#), [`cuCoredumpSetAttributeGlobal`](#)

CUresult cuCoredumpSetAttributeGlobal (CUcoredumpSettings attrib, void *value, size_t *size)

Allows caller to set a coredump attribute value globally.

Parameters

attrib

- The enum defining which value to set.

value

- void* containing the requested data.

size

- The size of the memory region `value` points to.

Returns

[`CUDA\_SUCCESS`](#), [`CUDA\_ERROR\_INVALID\_VALUE`](#), [`CUDA\_ERROR\_NOT\_PERMITTED`](#)

Description

This function should be considered an alternate interface to the CUDA-GDB environment variables defined in this document: <https://docs.nvidia.com/cuda/cuda-gdb/index.html#gpu-coredump>

An important design decision to note is that any coredump environment variable values set before CUDA initializes will take permanent precedence over any values set with this function. This decision was made to ensure no change in behavior for any users that may be currently using these variables to get coredumps.

*value shall contain the requested value specified by set. It is up to the caller to ensure that the data type and size of *value matches the request.

If the caller calls this function with *value equal to NULL, the size of the memory region (in bytes) expected for set will be placed in size.

The supported attributes are:

- ▶ **CU_COREDUMP_ENABLE_ON_EXCEPTION:** Bool where true means that GPU exceptions from this context will create a coredump at the location specified by CU_COREDUMP_FILE. The default value is false.
- ▶ **CU_COREDUMP_TRIGGER_HOST:** Bool where true means that the host CPU will also create a coredump. The default value is true unless set to false globally or locally. This value is deprecated as of CUDA 12.5 - raise the CU_COREDUMP_SKIP_ABORT flag to disable host device abort() if needed.
- ▶ **CU_COREDUMP_LIGHTWEIGHT:** Bool where true means that any resulting coredumps will not have a dump of GPU memory or non-reloc ELF images. The default value is false. This attribute is deprecated as of CUDA 12.5, please use CU_COREDUMP_GENERATION_FLAGS instead.
- ▶ **CU_COREDUMP_ENABLE_USER_TRIGGER:** Bool where true means that a coredump can be created by writing to the system pipe specified by CU_COREDUMP_PIPE. The default value is false.
- ▶ **CU_COREDUMP_FILE:** String of up to 1023 characters that defines the location where any coredumps generated by this context will be written. The default value is core.cuda.HOSTNAME.PID where HOSTNAME is the host name of the machine running the CUDA applications and PID is the process ID of the CUDA application.
- ▶ **CU_COREDUMP_PIPE:** String of up to 1023 characters that defines the name of the pipe that will be monitored if user-triggered coredumps are enabled. This value may not be changed after CU_COREDUMP_ENABLE_USER_TRIGGER is set to true. The default value is corepipe.cuda.HOSTNAME.PID where HOSTNAME is the host name of the machine running the CUDA application and PID is the process ID of the CUDA application.
- ▶ **CU_COREDUMP_GENERATION_FLAGS:** An integer with values to allow granular control the data contained in a coredump specified as a bitwise OR combination of the following values: + CU_COREDUMP_DEFAULT_FLAGS - if set by itself, coredump generation returns to its default settings of including all memory regions that it is able to access + CU_COREDUMP_SKIP_NONRELOCATED_ELF_IMAGES - Coredump

will not include the data from CUDA source modules that are not relocated at runtime. + `CU_COREDUMP_SKIP_GLOBAL_MEMORY` - Coredump will not include device-side global data that does not belong to any context. + `CU_COREDUMP_SKIP_SHARED_MEMORY` - Coredump will not include grid-scale shared memory for the warp that the dumped kernel belonged to. + `CU_COREDUMP_SKIP_LOCAL_MEMORY` - Coredump will not include local memory from the kernel. + `CU_COREDUMP_LIGHTWEIGHT_FLAGS` - Enables all of the above options. Equivalent to setting the `CU_COREDUMP_LIGHTWEIGHT` attribute to true. + `CU_COREDUMP_SKIP_ABORT` - If set, GPU exceptions will not raise an abort() in the host CPU process. Same functional goal as `CU_COREDUMP_TRIGGER_HOST` but better reflects the default behavior.

See also:

[cuCoredumpGetAttribute](#), [cuCoredumpGetAttributeGlobal](#), [cuCoredumpSetAttribute](#)

6.35. Green Contexts

This section describes the APIs for creation and manipulation of green contexts in the CUDA driver. Green contexts are a lightweight alternative to traditional contexts, with the ability to pass in a set of resources that they should be initialized with. This allows the developer to represent distinct spatial partitions of the GPU, provision resources for them, and target them via the same programming model that CUDA exposes (streams, kernel launches, etc.).

There are 4 main steps to using these new set of APIs.

- ▶ (1) Start with an initial set of resources, for example via [cuDeviceGetDevResource](#). Only SM type is supported today.
- ▶ (2) Partition this set of resources by providing them as input to a partition API, for example: [cuDevSmResourceSplitByCount](#).
- ▶ (3) Finalize the specification of resources by creating a descriptor via [cuDevResourceGenerateDesc](#).
- ▶ (4) Provision the resources and create a green context via [cuGreenCtxCreate](#).

For `CU_DEV_RESOURCE_TYPE_SM`, the partitions created have minimum SM count requirements, often rounding up and aligning the minCount provided to [cuDevSmResourceSplitByCount](#). The following is a guideline for each architecture and may be subject to change:

- ▶ On Compute Architecture 6.X: The minimum count is 1 SM.
- ▶ On Compute Architecture 7.X: The minimum count is 2 SMs and must be a multiple of 2.
- ▶ On Compute Architecture 8.X: The minimum count is 4 SMs and must be a multiple of 2.
- ▶ On Compute Architecture 9.0+: The minimum count is 8 SMs and must be a multiple of 8.

In the future, flags can be provided to tradeoff functional and performance characteristics versus finer grained SM partitions.

Even if the green contexts have disjoint SM partitions, it is not guaranteed that the kernels launched in them will run concurrently or have forward progress guarantees. This is due to other resources (like HW connections, see `CUDA_DEVICE_MAX_CONNECTIONS`) that could cause a dependency. Additionally, in certain scenarios, it is possible for the workload to run on more SMs than was provisioned (but never less). The following are two scenarios which can exhibit this behavior:

- ▶ On Volta+ MPS: When `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` is used, the set of SMs that are used for running kernels can be scaled up to the value of SMs used for the MPS client.
- ▶ On Compute Architecture 9.x: When a module with dynamic parallelism (CDP) is loaded, all future kernels running under green contexts may use and share an additional set of 2 SMs.

struct CUdevResource

struct CUdevSmResource

enum CUdevResourceType

Type of resource

Values

CU_DEV_RESOURCE_TYPE_INVALID = 0

CU_DEV_RESOURCE_TYPE_SM = 1

Streaming multiprocessors related information

typedef struct CUdevResourceDesc_st

*CUdevResourceDesc

An opaque descriptor handle. The descriptor encapsulates multiple created and configured resources.

Created via [cuDevResourceGenerateDesc](#)

CUresult cuCtxFromGreenCtx (CUcontext *pContext, CUgreenCtx hCtx)

Converts a green context into the primary context.

Parameters

pContext

Returned primary context with green context resources

hCtx

Green context to convert

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

The API converts a green context into the primary context returned in `pContext`. It is important to note that the converted context `pContext` is a normal primary context but with the resources of the specified green context `hCtx`. Once converted, it can then be used to set the context current with [cuCtxSetCurrent](#) or with any of the CUDA APIs that accept a `CUcontext` parameter.

Users are expected to call this API before calling any CUDA APIs that accept a `CUcontext`. Failing to do so will result in the APIs returning [CUDA_ERROR_INVALID_CONTEXT](#).

See also:

[cuGreenCtxCreate](#)

CUresult cuCtxGetDevResource (CUcontext hCtx, CUdevResource *resource, CUdevResourceType type)

Get context resources.

Parameters

hCtx

- Context to get resource for

resource

- Output pointer to a [CUdevResource](#) structure

type

- Type of resource to retrieve

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_RESOURCE_TYPE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_CONTEXT](#)

Description

Get the `type` resources available to the context represented by `hCtx` Note: The API is not supported on 32-bit platforms.

See also:

[cuDevResourceGenerateDesc](#)

CUresult cuDeviceGetDevResource (CUdevice device, CUdevResource *resource, CUdevResourceType type)

Get device resources.

Parameters

device

- Device to get resource for

resource

- Output pointer to a [CUdevResource](#) structure

type

- Type of resource to retrieve

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_RESOURCE_TYPE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_DEVICE](#)

Description

Get the `type` resources available to the `device`. This may often be the starting point for further partitioning or configuring of resources.

Note: The API is not supported on 32-bit platforms.

See also:

[cuDevResourceGenerateDesc](#)

CUresult cuDevResourceGenerateDesc (CUdevResourceDesc *phDesc, CUdevResource *resources, unsigned int nbResources)

Generate a resource descriptor.

Parameters

phDesc

- Output descriptor

resources

- Array of resources to be included in the descriptor

nbResources

- Number of resources passed in `resources`

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_RESOURCE_TYPE](#),
[CUDA_ERROR_INVALID_RESOURCE_CONFIGURATION](#)

Description

Generates a single resource descriptor with the set of resources specified in `resources`. The generated resource descriptor is necessary for the creation of green contexts via the [cuGreenCtxCreate](#) API. Resources of the same type can be passed in, provided they meet the requirements as noted below.

A successful API call must have:

- A valid output pointer for the `phDesc` descriptor as well as a valid array of `resources` pointers, with the array size passed in `nbResources`. If multiple resources are provided in `resources`, the device they came from must be the same, otherwise [CUDA_ERROR_INVALID_RESOURCE_CONFIGURATION](#) is returned. If multiple resources are provided in `resources` and they are of type [CU_DEV_RESOURCE_TYPE_SM](#), they must be outputs (whether `result` or `remaining`) from the same split API instance, otherwise [CUDA_ERROR_INVALID_RESOURCE_CONFIGURATION](#) is returned.

Note: The API is not supported on 32-bit platforms.

See also:

[cuDevSmResourceSplitByCount](#)

CUresult cuDevSmResourceSplitByCount (CUdevResource *result, unsigned int *nbGroups, const CUdevResource *input, CUdevResource *remaining, unsigned int useFlags, unsigned int minCount)

Splits [CU_DEV_RESOURCE_TYPE_SM](#) resources.

Parameters

result

- Output array of [CUdevResource](#) resources. Can be NULL to query the number of groups.

nbGroups

- This is a pointer, specifying the number of groups that would be or should be created as described below.

input

- Input SM resource to be split. Must be a valid [CU_DEV_RESOURCE_TYPE_SM](#) resource.

remaining

- If the input resource cannot be cleanly split among `nbGroups`, the remaining is placed in here. Can be omitted (NULL) if the user does not need the remaining set.

useFlags

- Flags specifying how these partitions are used or which constraints to abide by when splitting the input. Zero is valid for default behavior.

minCount

- Minimum number of SMs required

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_RESOURCE_TYPE](#), [CUDA_ERROR_INVALID_RESOURCE_CONFIGURATION](#)

Description

Splits `CU_DEV_RESOURCE_TYPE_SM` resources into `nbGroups`, adhering to the minimum SM count specified in `minCount` and the usage flags in `useFlags`. If `result` is NULL, the API simulates a split and provides the amount of groups that would be created in `nbGroups`. Otherwise, `nbGroups` must point to the amount of elements in `result` and on return, the API will overwrite `nbGroups` with the amount actually created. The groups are written to the array in `result`. `nbGroups` can be less than the total amount if a smaller number of groups is needed.

This API is used to spatially partition the input resource. The input resource needs to come from one of [cuDeviceGetDevResource](#), [cuCtxGetDevResource](#), or [cuGreenCtxGetDevResource](#). A limitation of the API is that the output results cannot be split again without first creating a descriptor and a green context with that descriptor.

When creating the groups, the API will take into account the performance and functional characteristics of the input resource, and guarantee a split that will create a disjoint set of symmetrical partitions. This may lead to fewer groups created than purely dividing the total SM count by the `minCount` due to cluster requirements or alignment and granularity requirements for the `minCount`.

The `remainder` set does not have the same functional or performance guarantees as the groups in `result`. Its use should be carefully planned and future partitions of the `remainder` set are discouraged.

The following flags are supported:

- ▶ `CU_DEV_SM_RESOURCE_SPLIT_IGNORE_SM_COSCHEDULING` : Lower the minimum SM count and alignment, and treat each SM independent of its hierarchy. This allows more fine grained partitions but at the cost of advanced features (such as large clusters on compute capability 9.0+).
- ▶ `CU_DEV_SM_RESOURCE_SPLIT_MAX_POTENTIAL_CLUSTER_SIZE` : Compute Capability 9.0+ only. Attempt to create groups that may allow for maximally sized thread clusters. This can be queried post green context creation using [cuOccupancyMaxPotentialClusterSize](#).

A successful API call must either have:

- ▶ A valid array of `result` pointers of size passed in `nbGroups`, with `input` of type `CU_DEV_RESOURCE_TYPE_SM`. Value of `minCount` must be between 0 and the SM count specified in `input`. `remaining` may be NULL.
- ▶ NULL passed in for `result`, with a valid integer pointer in `nbGroups` and `input` of type `CU_DEV_RESOURCE_TYPE_SM`. Value of `minCount` must be between 0 and the SM count specified in `input`. `remaining` may be NULL. This queries the number of groups that would be created by the API.

Note: The API is not supported on 32-bit platforms.

See also:

[cuGreenCtxGetDevResource](#), [cuCtxGetDevResource](#), [cuDeviceGetDevResource](#)

CUresult cuGreenCtxCreate (CUgreenCtx *phCtx, CUdevResourceDesc desc, CUdevice dev, unsigned int flags)

Creates a green context with a specified set of resources.

Parameters

phCtx

- Pointer for the output handle to the green context

desc

- Descriptor generated via [cuDevResourceGenerateDesc](#) which contains the set of resources to be used

dev

- Device on which to create the green context.

flags

- One of the supported green context creation flags. `CU_GREEN_CTX_DEFAULT_STREAM` is required.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_DEVICE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_SUPPORTED](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

This API creates a green context with the resources specified in the descriptor `desc` and returns it in the handle represented by `phCtx`. This API will retain the primary context on device `dev`, which will be released when the green context is destroyed. It is advised to have the primary context active before

calling this API to avoid the heavy cost of triggering primary context initialization and deinitialization multiple times.

The API does not set the green context current. In order to set it current, you need to explicitly set it current by first converting the green context to a CUcontext using [cuCtxFromGreenCtx](#) and subsequently calling [cuCtxSetCurrent](#) / [cuCtxPushCurrent](#). It should be noted that a green context can be current to only one thread at a time. There is no internal synchronization to make API calls accessing the same green context from multiple threads work.

Note: The API is not supported on 32-bit platforms.

The supported flags are:

- `CU_GREEN_CTX_DEFAULT_STREAM` : Creates a default stream to use inside the green context. Required.

See also:

[cuGreenCtxDestroy](#), [cuCtxFromGreenCtx](#), [cuCtxSetCurrent](#), [cuCtxPushCurrent](#), [cuDevResourceGenerateDesc](#), [cuDevicePrimaryCtxRetain](#), [cuCtxCreate](#), [cuCtxCreate_v3](#)

CUresult cuGreenCtxDestroy (CUgreenCtx hCtx)

Destroys a green context.

Parameters

hCtx

- Green context to be destroyed

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_CONTEXT_IS_DESTROYED](#)

Description

Destroys the green context, releasing the primary context of the device that this green context was created for. Any resources provisioned for this green context (that were initially available via the resource descriptor) are released as well.

See also:

[cuGreenCtxCreate](#), [cuCtxDestroy](#)

CUresult cuGreenCtxGetDevResource (CUgreenCtx hCtx, CUdevResource *resource, CUdevResourceType type)

Get green context resources.

Parameters

hCtx

- Green context to get resource for

resource

- Output pointer to a [CUdevResource](#) structure

type

- Type of resource to retrieve

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_RESOURCE_TYPE](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Get the type resources available to the green context represented by hCtx

See also:

[cuDevResourceGenerateDesc](#)

CUresult cuGreenCtxRecordEvent (CUgreenCtx hCtx, CUevent hEvent)

Records an event.

Parameters

hCtx

- Green context to record event for

hEvent

- Event to record

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_STREAM_CAPTURE_UNSUPPORTED](#)

Description

Captures in `hEvent` all the activities of the green context of `hCtx` at the time of this call. `hEvent` and `hCtx` must be from the same primary context otherwise [CUDA_ERROR_INVALID_HANDLE](#) is returned. Calls such as [cuEventQuery\(\)](#) or [cuGreenCtxWaitEvent\(\)](#) will then examine or wait for completion of the work that was captured. Uses of `hCtx` after this call do not modify `hEvent`.



Note:

The API will return [CUDA_ERROR_STREAM_CAPTURE_UNSUPPORTED](#) if the specified green context `hCtx` has a stream in the capture mode. In such a case, the call will invalidate all the conflicting captures.

See also:

[cuGreenCtxWaitEvent](#), [cuEventRecord](#), [cuCtxRecordEvent](#), [cuCtxWaitEvent](#)

CUresult cuGreenCtxStreamCreate (CUstream *phStream, CUgreenCtx greenCtx, unsigned int flags, int priority)

Create a stream for use in the green context.

Parameters

phStream

- Returned newly created stream

greenCtx

- Green context for which to create the stream for

flags

- Flags for stream creation. [CU_STREAM_NON_BLOCKING](#) must be specified.

priority

- Stream priority. Lower numbers represent higher priorities. See [cuCtxGetStreamPriorityRange](#) for more information about meaningful stream priorities that can be passed.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Creates a stream for use in the specified green context `greenCtx` and returns a handle in `phStream`. The stream can be destroyed by calling [cuStreamDestroy\(\)](#). Note that the API ignores the context that is current to the calling thread and creates a stream in the specified green context `greenCtx`.

The supported values for `flags` are:

- [`CU\_STREAM\_NON\_BLOCKING`](#): This must be specified. It indicates that work running in the created stream may run concurrently with work in the default stream, and that the created stream should perform no implicit synchronization with the default stream.

Specifying `priority` 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 [`cuCtxGetStreamPriorityRange`](#). If the specified priority is outside the numerical range returned by [`cuCtxGetStreamPriorityRange`](#), 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.
- 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:

[`cuStreamDestroy`](#), [`cuGreenCtxCreate`](#), [`cuStreamCreate`](#), [`cuStreamGetPriority`](#), [`cuCtxGetStreamPriorityRange`](#), [`cuStreamGetFlags`](#), [`cuStreamGetDevice`](#), [`cuStreamWaitEvent`](#), [`cuStreamQuery`](#), [`cuStreamSynchronize`](#), [`cuStreamAddCallback`](#), [`cudaStreamCreateWithPriority`](#)

CUresult cuGreenCtxWaitEvent (CUgreenCtx hCtx, CUevent hEvent)

Make a green context wait on an event.

Parameters

hCtx

- Green context to wait

hEvent

- Event to wait on

Returns

[`CUDA\_SUCCESS`](#), [`CUDA\_ERROR\_DEINITIALIZED`](#), [`CUDA\_ERROR\_NOT\_INITIALIZED`](#), [`CUDA\_ERROR\_INVALID\_CONTEXT`](#), [`CUDA\_ERROR\_INVALID\_HANDLE`](#), [`CUDA\_ERROR\_STREAM\_CAPTURE\_UNSUPPORTED`](#)

Description

Makes all future work submitted to green context `hCtx` wait for all work captured in `hEvent`. The synchronization will be performed on the device and will not block the calling CPU thread. See [cuGreenCtxRecordEvent\(\)](#) or [cuEventRecord\(\)](#), for details on what is captured by an event.



Note:

- ▶ `hEvent` may be from a different context or device than `hCtx`.
- ▶ The API will return [CUDA_ERROR_STREAM_CAPTURE_UNSUPPORTED](#) and invalidate the capture if the specified event `hEvent` is part of an ongoing capture sequence or if the specified green context `hCtx` has a stream in the capture mode.

See also:

[cuGreenCtxRecordEvent](#), [cuStreamWaitEvent](#) [cuCtxRecordEvent](#), [cuCtxWaitEvent](#)

CUresult cuStreamGetGreenCtx (CUstream hStream, CUgreenCtx *phCtx)

Query the green context associated with a stream.

Parameters

hStream

- Handle to the stream to be queried

phCtx

- Returned green context associated with the stream

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#),

Description

Returns the CUDA green context that the stream is associated with, or NULL if the stream is not associated with any green context.

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

- ▶ a stream created via any of the CUDA driver APIs such as [cuStreamCreate](#), [cuStreamCreateWithPriority](#) and [cuGreenCtxStreamCreate](#), or their runtime API equivalents such as [cudaStreamCreate](#), [cudaStreamCreateWithFlags](#) and [cudaStreamCreateWithPriority](#). If during stream creation the context that was active in the calling thread was obtained with

`cuCtxFromGreenCtx`, that green context is returned in `phCtx`. Otherwise, `*phCtx` is set to NULL instead.

- special stream such as the NULL stream or [CU_STREAM_LEGACY](#). In that case if context that is active in the calling thread was obtained with `cuCtxFromGreenCtx`, that green context is returned. Otherwise, `*phCtx` is set to NULL instead.

Passing an invalid handle will result in undefined behavior.



Note:

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

See also:

[cuStreamDestroy](#), [cuStreamCreate](#), [cuStreamCreateWithPriority](#), [cuStreamGetCtx_v2](#), [cuGreenCtxStreamCreate](#), [cuStreamGetPriority](#), [cuStreamGetFlags](#), [cuStreamGetDevice](#), [cuStreamWaitEvent](#), [cuStreamQuery](#), [cuStreamSynchronize](#), [cuStreamAddCallback](#), [cudaStreamCreate](#), [cudaStreamCreateWithFlags](#)

6.36. CUDA Checkpointing

CUDA API versioning support

This sections describes the checkpoint and restore functions of the low-level CUDA driver application programming interface.

The CUDA checkpoint and restore API's provide a way to save and restore GPU state for full process checkpoints when used with CPU side process checkpointing solutions. They can also be used to pause GPU work and suspend a CUDA process to allow other applications to make use of GPU resources.

Checkpoint and restore capabilities are currently restricted to Linux.

`CUresult cuCheckpointProcessCheckpoint (int pid, CUcheckpointCheckpointArgs *args)`

Checkpoint a CUDA process's GPU memory contents.

Parameters

pid

- The process ID of the CUDA process

args

- Optional checkpoint operation arguments

Returns

CUDA_SUCCESS CUDA_ERROR_INVALID_VALUE CUDA_ERROR_NOT_INITIALIZED
CUDA_ERROR_ILLEGAL_STATE CUDA_ERROR_NOT_SUPPORTED

Description

Checkpoints a CUDA process specified by `pid` that is in the LOCKED state. The GPU memory contents will be brought into host memory and all underlying references will be released. Process must be in the LOCKED state to checkpoint.

Upon successful return the process will be in the CHECKPOINTED state.

CUresult cuCheckpointProcessGetRestoreThreadId (int pid, int *tid)

Returns the restore thread ID for a CUDA process.

Parameters

pid

- The process ID of the CUDA process

tid

- Returned restore thread ID

Returns

CUDA_SUCCESS CUDA_ERROR_INVALID_VALUE CUDA_ERROR_NOT_INITIALIZED
CUDA_ERROR_NOT_SUPPORTED

Description

Returns in `*tid` the thread ID of the CUDA restore thread for the process specified by `pid`.

CUresult cuCheckpointProcessGetState (int pid, CUprocessState *state)

Returns the process state of a CUDA process.

Parameters

pid

- The process ID of the CUDA process

state

- Returned CUDA process state

Returns

CUDA_SUCCESS CUDA_ERROR_INVALID_VALUE CUDA_ERROR_NOT_INITIALIZED
CUDA_ERROR_NOT_SUPPORTED

Description

Returns in `*state` the current state of the CUDA process specified by `pid`.

CUresult cuCheckpointProcessLock (int pid, CUcheckpointLockArgs *args)

Lock a running CUDA process.

Parameters

pid

- The process ID of the CUDA process

args

- Optional lock operation arguments

Returns

CUDA_SUCCESS CUDA_ERROR_INVALID_VALUE CUDA_ERROR_NOT_INITIALIZED
CUDA_ERROR_ILLEGAL_STATE CUDA_ERROR_NOT_SUPPORTED
CUDA_ERROR_NOT_READY

Description

Lock the CUDA process specified by `pid` which will block further CUDA API calls. Process must be in the RUNNING state in order to lock.

Upon successful return the process will be in the LOCKED state.

If `timeoutMs` is specified and the timeout is reached the process will be left in the RUNNING state upon return.

CUresult cuCheckpointProcessRestore (int pid, CUcheckpointRestoreArgs *args)

Restore a CUDA process's GPU memory contents from its last checkpoint.

Parameters

pid

- The process ID of the CUDA process

args

- Optional restore operation arguments

Returns

CUDA_SUCCESS CUDA_ERROR_INVALID_VALUE CUDA_ERROR_NOT_INITIALIZED
CUDA_ERROR_ILLEGAL_STATE CUDA_ERROR_NOT_SUPPORTED

Description

Restores a CUDA process specified by `pid` from its last checkpoint. Process must be in the CHECKPOINTED state to restore.

Upon successful return the process will be in the LOCKED state.

CUDA process restore requires persistence mode to be enabled or [cuInit](#) to have been called before execution.

See also:

[cuInit](#)

CUresult cuCheckpointProcessUnlock (int pid, CUcheckpointUnlockArgs *args)

Unlock a CUDA process to allow CUDA API calls.

Parameters**pid**

- The process ID of the CUDA process

args

- Optional unlock operation arguments

Returns

CUDA_SUCCESS CUDA_ERROR_INVALID_VALUE CUDA_ERROR_NOT_INITIALIZED
CUDA_ERROR_ILLEGAL_STATE CUDA_ERROR_NOT_SUPPORTED

Description

Unlocks a process specified by `pid` allowing it to resume making CUDA API calls. Process must be in the LOCKED state.

Upon successful return the process will be in the RUNNING state.

6.37. Profiler Control [DEPRECATED]

This section describes the profiler control functions of the low-level CUDA driver application programming interface.

CUresult cuProfilerInitialize (const char *configFile, const char *outputFile, CUoutput_mode outputMode)

Initialize the profiling.

Parameters

configFile

- Name of the config file that lists the counters/options for profiling.

outputFile

- Name of the outputFile where the profiling results will be stored.

outputMode

- outputMode, can be CU_OUT_KEY_VALUE_PAIR or CU_OUT_CSV.

Returns

[CUDA_ERROR_NOT_SUPPORTED](#)

Description

[Deprecated](#)

Note that this function is deprecated and should not be used. Starting with CUDA 12.0, it always returns error code [CUDA_ERROR_NOT_SUPPORTED](#).

Using this API user can initialize the CUDA profiler by specifying the configuration file, output file and output file format. This API is generally used to profile different set of counters by looping the kernel launch. The `configFile` parameter can be used to select profiling options including profiler counters. Refer to the "Compute Command Line Profiler User Guide" for supported profiler options and counters.

Limitation: The CUDA profiler cannot be initialized with this API if another profiling tool is already active, as indicated by the [CUDA_ERROR_PROFILER_DISABLED](#) return code.

Typical usage of the profiling APIs is as follows:

```
for each set of counters/options { cuProfilerInitialize\(\); //Initialize profiling, set the counters or options
in the config file ... cuProfilerStart\(\); // code to be profiled cuProfilerStop\(\); ... cuProfilerStart\(\); // code
to be profiled cuProfilerStop\(\); ... }
```

**Note:**

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

See also:

[cuProfilerStart](#), [cuProfilerStop](#),

6.38. Profiler Control

This section describes the profiler control functions of the low-level CUDA driver application programming interface.

CUresult cuProfilerStart (void)

Enable profiling.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_CONTEXT](#)

Description

Enables profile collection by the active profiling tool for the current context. If profiling is already enabled, then [cuProfilerStart\(\)](#) has no effect.

[cuProfilerStart](#) and [cuProfilerStop](#) 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:

[cuProfilerInitialize](#), [cuProfilerStop](#), [cudaProfilerStart](#)

CUresult cuProfilerStop (void)

Disable profiling.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_CONTEXT](#)

Description

Disables profile collection by the active profiling tool for the current context. If profiling is already disabled, then [cuProfilerStop\(\)](#) has no effect.

[cuProfilerStart](#) and [cuProfilerStop](#) 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:

[cuProfilerInitialize](#), [cuProfilerStart](#), [cudaProfilerStop](#)

6.39. OpenGL Interoperability

This section describes the OpenGL interoperability functions of the low-level CUDA driver application programming interface. Note that mapping of OpenGL resources is performed with the graphics API agnostic, resource mapping interface described in [Graphics Interoperability](#).

OpenGL Interoperability [DEPRECATED]

enum CUGLDeviceList

CUDA devices corresponding to an OpenGL device

Values

CU_GL_DEVICE_LIST_ALL = 0x01

The CUDA devices for all GPUs used by the current OpenGL context

CU_GL_DEVICE_LIST_CURRENT_FRAME = 0x02

The CUDA devices for the GPUs used by the current OpenGL context in its currently rendering frame

CU_GL_DEVICE_LIST_NEXT_FRAME = 0x03

The CUDA devices for the GPUs to be used by the current OpenGL context in the next frame

CUresult cuGLGetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, CUGLDeviceList deviceList)

Gets the CUDA devices associated with the current OpenGL context.

Parameters

pCudaDeviceCount

- Returned number of CUDA devices.

pCudaDevices

- Returned CUDA devices.

cudaDeviceCount

- The size of the output device array pCudaDevices.

deviceList

- The set of devices to return.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_NO_DEVICE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_GRAPHICS_CONTEXT](#), [CUDA_ERROR_OPERATING_SYSTEM](#)

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 CUDA_ERROR_NO_DEVICE.

The deviceList argument may be any of the following:

- ▶ [CU_GL_DEVICE_LIST_ALL](#): Query all devices used by the current OpenGL context.
- ▶ [CU_GL_DEVICE_LIST_CURRENT_FRAME](#): Query the devices used by the current OpenGL context to render the current frame (in SLI).
- ▶ [CU_GL_DEVICE_LIST_NEXT_FRAME](#): Query the devices used by the current OpenGL context to render the next frame (in SLI). Note that this is a prediction, it can't be guaranteed that this is correct in all cases.



Note:

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

See also:

[cuWGLGetDevice](#), [cudaGLGetDevices](#)

CUresult cuGraphicsGLRegisterBuffer (CUgraphicsResource *pCudaResource, GLuint buffer, unsigned int Flags)

Registers an OpenGL buffer object.

Parameters

pCudaResource

- Pointer to the returned object handle

buffer

- name of buffer object to be registered

Flags

- Register flags

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_ALREADY_MAPPED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_OPERATING_SYSTEM](#)

Description

Registers the buffer object specified by `buffer` for access by CUDA. A handle to the registered object is returned as `pCudaResource`. The register flags `Flags` specify the intended usage, as follows:

- ▶ `CU_GRAPHICS_REGISTER_FLAGS_NONE`: 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.
- ▶ `CU_GRAPHICS_REGISTER_FLAGS_READ_ONLY`: Specifies that CUDA will not write to this resource.
- ▶ `CU_GRAPHICS_REGISTER_FLAGS_WRITE_DISCARD`: 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:

[cuGraphicsUnregisterResource](#), [cuGraphicsMapResources](#), [cuGraphicsResourceGetMappedPointer](#), [cudaGraphicsGLRegisterBuffer](#)

CUresult cuGraphicsGLRegisterImage (CUgraphicsResource *pCudaResource, GLuint image, GLenum target, unsigned int Flags)

Register an OpenGL texture or renderbuffer object.

Parameters

pCudaResource

- 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_ALREADY_MAPPED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_OPERATING_SYSTEM](#)

Description

Registers the texture or renderbuffer object specified by `image` for access by CUDA. A handle to the registered object is returned as `pCudaResource`.

`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:

- ▶ `CU_GRAPHICS_REGISTER_FLAGS_NONE`: 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.
- ▶ `CU_GRAPHICS_REGISTER_FLAGS_READ_ONLY`: Specifies that CUDA will not write to this resource.
- ▶ `CU_GRAPHICS_REGISTER_FLAGS_WRITE_DISCARD`: 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.
- ▶ `CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST`: Specifies that CUDA will bind this resource to a surface reference.

- ▶ `CU_GRAPHICS_REGISTER_FLAGS_TEXTURE_GATHER`: 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:

[cuGraphicsUnregisterResource](#), [cuGraphicsMapResources](#), [cuGraphicsSubResourceGetMappedArray](#), [cudaGraphicsGLRegisterImage](#)

CUresult cuWGLGetDevice (CUdevice *pDevice, HGPUNV hGpu)

Gets the CUDA device associated with hGpu.

Parameters

pDevice

- Device associated with hGpu

hGpu

- Handle to a GPU, as queried via `WGL_NV_gpu_affinity()`

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns in *pDevice 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:

[cuGLMapBufferObject](#), [cuGLRegisterBufferObject](#), [cuGLUnmapBufferObject](#),
[cuGLUnregisterBufferObject](#), [cuGLUnmapBufferObjectAsync](#), [cuGLSetBufferObjectMapFlags](#),
[cudaWGLGetDevice](#)

6.39.1. OpenGL Interoperability [DEPRECATED]

OpenGL Interoperability

This section describes deprecated OpenGL interoperability functionality.

enum CUGLmap_flags

Flags to map or unmap a resource

Values

CU_GL_MAP_RESOURCE_FLAGS_NONE = 0x00

CU_GL_MAP_RESOURCE_FLAGS_READ_ONLY = 0x01

CU_GL_MAP_RESOURCE_FLAGS_WRITE_DISCARD = 0x02

CUresult cuGLCtxCreate (CUcontext *pCtx, unsigned int Flags, CUdevice device)

Create a CUDA context for interoperability with OpenGL.

Parameters

pCtx

- Returned CUDA context

Flags

- Options for CUDA context creation

device

- Device on which to create the context

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_OUT_OF_MEMORY](#)

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 context with an OpenGL context in order to achieve maximum interoperability performance.



Note:

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

See also:

[cuCtxCreate](#), [cuGLInit](#), [cuGLMapBufferObject](#), [cuGLRegisterBufferObject](#),
[cuGLUnmapBufferObject](#), [cuGLUnregisterBufferObject](#), [cuGLMapBufferObjectAsync](#),
[cuGLUnmapBufferObjectAsync](#), [cuGLSetBufferObjectMapFlags](#), [cuWGLGetDevice](#)

CUresult cuGLInit (void)

Initializes OpenGL interoperability.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_UNKNOWN](#)

Description

Deprecated This function is deprecated as of Cuda 3.0.

Initializes OpenGL interoperability. This function is deprecated and calling it is no longer required. It may fail if the needed OpenGL driver facilities are not available.



Note:

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

See also:

[cuGLMapBufferObject](#), [cuGLRegisterBufferObject](#), [cuGLUnmapBufferObject](#),
[cuGLUnregisterBufferObject](#), [cuGLMapBufferObjectAsync](#), [cuGLUnmapBufferObjectAsync](#),
[cuGLSetBufferObjectMapFlags](#), [cuWGLGetDevice](#)

CUresult cuGLMapBufferObject (CUdeviceptr *dptr, size_t *size, GLuint buffer)

Maps an OpenGL buffer object.

Parameters

dptr

- Returned mapped base pointer

size

- Returned size of mapping

buffer

- The name of the buffer object to map

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_MAP_FAILED](#)

Description

[Deprecated](#) This function is deprecated as of Cuda 3.0.

Maps the buffer object specified by `buffer` into the address space of the current CUDA context and returns in `*dptr` and `*size` the base pointer and size of the resulting mapping.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

All streams in the current CUDA context are synchronized with the current GL context.



Note:

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

See also:

[cuGraphicsMapResources](#)

CUresult cuGLMapBufferObjectAsync (CUdeviceptr *dptr, size_t *size, GLuint buffer, CUstream hStream)

Maps an OpenGL buffer object.

Parameters

dptr

- Returned mapped base pointer

size

- Returned size of mapping

buffer

- The name of the buffer object to map

hStream

- Stream to synchronize

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_MAP_FAILED](#)

Description

[Deprecated](#) This function is deprecated as of Cuda 3.0.

Maps the buffer object specified by `buffer` into the address space of the current CUDA context and returns in `*dptr` and `*size` the base pointer and size of the resulting mapping.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

Stream `hStream` in the current CUDA context is synchronized with the current GL context.

**Note:**

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

See also:

[cuGraphicsMapResources](#)

CUresult cuGLRegisterBufferObject (GLuint buffer)

Registers an OpenGL buffer object.

Parameters

buffer

- The name of the buffer object to register.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_ALREADY_MAPPED](#)

Description

[Deprecated](#) This function is deprecated as of Cuda 3.0.

Registers the buffer object specified by `buffer` for access by CUDA. This function must be called before CUDA can map the buffer object. There must be a valid OpenGL context bound to the current thread when this function is called, and the buffer name is resolved by that context.



Note:

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

See also:

[cuGraphicsGLRegisterBuffer](#)

CUresult cuGLSetBufferObjectMapFlags (GLuint buffer, unsigned int Flags)

Set the map flags for an OpenGL buffer object.

Parameters

buffer

- Buffer object to unmap

Flags

- Map flags

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_ALREADY_MAPPED](#), [CUDA_ERROR_INVALID_CONTEXT](#),

Description

Deprecated This function is deprecated as of Cuda 3.0.

Sets the map flags for the buffer object specified by `buffer`.

Changes to `Flags` will take effect the next time `buffer` is mapped. The `Flags` argument may be any of the following:

- ▶ `CU_GL_MAP_RESOURCE_FLAGS_NONE`: 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.
- ▶ `CU_GL_MAP_RESOURCE_FLAGS_READ_ONLY`: Specifies that CUDA kernels which access this resource will not write to this resource.
- ▶ `CU_GL_MAP_RESOURCE_FLAGS_WRITE_DISCARD`: 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 `buffer` has not been registered for use with CUDA, then `CUDA_ERROR_INVALID_HANDLE` is returned. If `buffer` is presently mapped for access by CUDA, then `CUDA_ERROR_ALREADY_MAPPED` is returned.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same `shareGroup`, as the context that was bound when the buffer was registered.



Note:

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

See also:

`cuGraphicsResourceSetMapFlags`

CUresult cuGLUnmapBufferObject (GLuint buffer)

Unmaps an OpenGL buffer object.

Parameters

buffer

- Buffer object to unmap

Returns

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`,
`CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`

Description

Deprecated This function is deprecated as of Cuda 3.0.

Unmaps the buffer object specified by `buffer` for access by CUDA.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same `shareGroup`, as the context that was bound when the buffer was registered.

All streams in the current CUDA context are synchronized with the current GL context.



Note:

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

See also:

[cuGraphicsUnmapResources](#)

CUresult cuGLUnmapBufferObjectAsync (GLuint buffer, CUstream hStream)

Unmaps an OpenGL buffer object.

Parameters

buffer

- Name of the buffer object to unmap

hStream

- Stream to synchronize

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Deprecated This function is deprecated as of Cuda 3.0.

Unmaps the buffer object specified by `buffer` for access by CUDA.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same `shareGroup`, as the context that was bound when the buffer was registered.

Stream `hStream` in the current CUDA context is synchronized with the current GL context.



Note:

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

See also:

[cuGraphicsUnmapResources](#)

CUresult cuGLUnregisterBufferObject (GLuint buffer)

Unregister an OpenGL buffer object.

Parameters

buffer

- Name of the buffer object to unregister

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

[Deprecated](#) This function is deprecated as of Cuda 3.0.

Unregisters the buffer object specified by `buffer`. This releases any resources associated with the registered buffer. After this call, the buffer may no longer be mapped for access by CUDA.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.



Note:

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

See also:

[cuGraphicsUnregisterResource](#)

6.40. Direct3D 9 Interoperability

This section describes the Direct3D 9 interoperability functions of the low-level CUDA driver application programming interface. Note that mapping of Direct3D 9 resources is performed with the graphics API agnostic, resource mapping interface described in [Graphics Interoperability](#).

Direct3D 9 Interoperability [DEPRECATED]

enum CUd3d9DeviceList

CUDA devices corresponding to a D3D9 device

Values

CU_D3D9_DEVICE_LIST_ALL = 0x01

The CUDA devices for all GPUs used by a D3D9 device

CU_D3D9_DEVICE_LIST_CURRENT_FRAME = 0x02

The CUDA devices for the GPUs used by a D3D9 device in its currently rendering frame

CU_D3D9_DEVICE_LIST_NEXT_FRAME = 0x03

The CUDA devices for the GPUs to be used by a D3D9 device in the next frame

CUresult cuD3D9CtxCreate (CUcontext *pCtx, CUdevice *pCudaDevice, unsigned int Flags, IDirect3DDevice9 *pD3DDevice)

Create a CUDA context for interoperability with Direct3D 9.

Parameters

pCtx

- Returned newly created CUDA context

pCudaDevice

- Returned pointer to the device on which the context was created

Flags

- Context creation flags (see [cuCtxCreate\(\)](#) for details)

pD3DDevice

- Direct3D device to create interoperability context with

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_UNKNOWN](#)

Description

Creates a new CUDA context, enables interoperability for that context with the Direct3D device `pD3DDevice`, and associates the created CUDA context with the calling thread. The created [CUcontext](#) will be returned in `*pCtx`. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context. If `pCudaDevice` is non-NULL then the [CUdevice](#) on which this CUDA context was created will be returned in `*pCudaDevice`.

On success, this call will increase the internal reference count on `pD3DDevice`. This reference count will be decremented upon destruction of this context through [cuCtxDestroy\(\)](#). This context will cease to function if `pD3DDevice` is destroyed or encounters an error.

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 `pD3DDevice` is not an `IDirect3DDevice9Ex`. In all other circumstances, this function is not necessary.



Note:

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

See also:

[cuD3D9GetDevice](#), [cuGraphicsD3D9RegisterResource](#)

CUresult cuD3D9CtxCreateOnDevice (CUcontext *pCtx, unsigned int flags, IDirect3DDevice9 *pD3DDevice, CUdevice cudaDevice)

Create a CUDA context for interoperability with Direct3D 9.

Parameters

pCtx

- Returned newly created CUDA context

flags

- Context creation flags (see [cuCtxCreate\(\)](#) for details)

pD3DDevice

- Direct3D device to create interoperability context with

cudaDevice

- The CUDA device on which to create the context. This device must be among the devices returned when querying `CU_D3D9_DEVICES_ALL` from [cuD3D9GetDevices](#).

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_UNKNOWN](#)

Description

Creates a new CUDA context, enables interoperability for that context with the Direct3D device `pD3DDevice`, and associates the created CUDA context with the calling thread. The created

[CUcontext](#) will be returned in *pCtx. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context.

On success, this call will increase the internal reference count on pD3DDevice. This reference count will be decremented upon destruction of this context through [cuCtxDestroy\(\)](#). This context will cease to function if pD3DDevice is destroyed or encounters an error.

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 pD3DDevice is not an IDirect3DDevice9Ex. In all other circumstances, this function is not necessary.



Note:

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

See also:

[cuD3D9GetDevices](#), [cuGraphicsD3D9RegisterResource](#)

CUresult cuD3D9GetDevice (CUdevice *pCudaDevice, const char *pszAdapterName)

Gets the CUDA device corresponding to a display adapter.

Parameters

pCudaDevice

- Returned CUDA device corresponding to pszAdapterName

pszAdapterName

- Adapter name to query for device

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_FOUND](#), [CUDA_ERROR_UNKNOWN](#)

Description

Returns in *pCudaDevice 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:

[cuD3D9CtxCreate](#), [cudaD3D9GetDevice](#)

CUresult cuD3D9GetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, IDirect3DDevice9 *pD3D9Device, CUd3d9DeviceList 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 [CU_D3D9_DEVICE_LIST_ALL](#) for all devices, [CU_D3D9_DEVICE_LIST_CURRENT_FRAME](#) for the devices used to render the current frame (in SLI), or [CU_D3D9_DEVICE_LIST_NEXT_FRAME](#) for the devices used to render the next frame (in SLI).

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_NO_DEVICE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_FOUND](#), [CUDA_ERROR_UNKNOWN](#)

Description

Returns in *pCudaDeviceCount the number of CUDA-compatible device corresponding to the Direct3D 9 device pD3D9Device. Also returns in *pCudaDevices at most cudaDeviceCount of 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 [CUDA_ERROR_NO_DEVICE](#).

**Note:**

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

See also:

[cuD3D9CtxCreate](#), [cudaD3D9GetDevices](#)

CUresult cuD3D9GetDirect3DDevice (IDirect3DDevice9 **ppD3DDevice)

Get the Direct3D 9 device against which the current CUDA context was created.

Parameters

ppD3DDevice

- Returned Direct3D device corresponding to CUDA context

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#) [CUDA_ERROR_INVALID_GRAPHICS_CONTEXT](#)

Description

Returns in `*ppD3DDevice` the Direct3D device against which this CUDA context was created in [cuD3D9CtxCreate\(\)](#).

**Note:**

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

See also:

[cuD3D9GetDevice](#), [cudaD3D9GetDirect3DDevice](#)

CUresult cuGraphicsD3D9RegisterResource (CUgraphicsResource *pCudaResource, IDirect3DResource9 *pD3DResource, unsigned int Flags)

Register a Direct3D 9 resource for access by CUDA.

Parameters

pCudaResource

- Returned graphics resource handle

pD3DResource

- Direct3D resource to register

Flags

- Parameters for resource registration

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE,
CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY,
CUDA_ERROR_UNKNOWN

Description

Registers the Direct3D 9 resource `pD3DResource` for access by CUDA and returns a CUDA handle to `pD3DResource` in `pCudaResource`. The handle returned in `pCudaResource` may be used to map and unmap this resource until it is unregistered. On success this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through [cuGraphicsUnregisterResource\(\)](#).

This call is potentially 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

- ▶ `CU_GRAPHICS_REGISTER_FLAGS_NONE`: Specifies no hints about how this resource will be used.
- ▶ `CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST`: Specifies that CUDA will bind this resource to a surface reference.
- ▶ `CU_GRAPHICS_REGISTER_FLAGS_TEXTURE_GATHER`: 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 Direct3D interoperability is not initialized for this context using `cuD3D9CtxCreate` then `CUDA_ERROR_INVALID_CONTEXT` is returned. If `pD3DResource` is of incorrect type or is already registered then `CUDA_ERROR_INVALID_HANDLE` is returned. If `pD3DResource` cannot be registered then `CUDA_ERROR_UNKNOWN` is returned. If `Flags` is not one of the above specified value then `CUDA_ERROR_INVALID_VALUE` is returned.

**Note:**

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

See also:

[cuD3D9CtxCreate](#), [cuGraphicsUnregisterResource](#), [cuGraphicsMapResources](#),
[cuGraphicsSubResourceGetMappedArray](#), [cuGraphicsResourceGetMappedPointer](#),
[cudaGraphicsD3D9RegisterResource](#)

6.40.1. Direct3D 9 Interoperability [DEPRECATED]

Direct3D 9 Interoperability

This section describes deprecated Direct3D 9 interoperability functionality.

enum CUd3d9map_flags

Flags to map or unmap a resource

Values

CU_D3D9_MAPRESOURCE_FLAGS_NONE = 0x00

CU_D3D9_MAPRESOURCE_FLAGS_READONLY = 0x01

CU_D3D9_MAPRESOURCE_FLAGS_WRITEDISCARD = 0x02

enum CUd3d9register_flags

Flags to register a resource

Values

CU_D3D9_REGISTER_FLAGS_NONE = 0x00

CU_D3D9_REGISTER_FLAGS_ARRAY = 0x01

CUresult cuD3D9MapResources (unsigned int count, IDirect3DResource9 **ppResource)

Map Direct3D resources for access by CUDA.

Parameters

count

- Number of resources in ppResource

ppResource

- Resources to map for CUDA usage

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_ALREADY_MAPPED](#), [CUDA_ERROR_UNKNOWN](#)

Description

Deprecated This function is deprecated as of CUDA 3.0.

Maps the `count` Direct3D resources in `ppResource` for access by CUDA.

The resources in `ppResource` 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 [cuD3D9MapResources\(\)](#) will complete before any CUDA kernels issued after [cuD3D9MapResources\(\)](#) begin.

If any of `ppResource` have not been registered for use with CUDA or if `ppResource` contains any duplicate entries, then [CUDA_ERROR_INVALID_HANDLE](#) is returned. If any of `ppResource` are presently mapped for access by CUDA, then [CUDA_ERROR_ALREADY_MAPPED](#) is returned.



Note:

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

See also:

[cuGraphicsMapResources](#)

CUresult cuD3D9RegisterResource (IDirect3DResource9 *pResource, unsigned int Flags)

Register a Direct3D resource for access by CUDA.

Parameters

pResource

- Resource to register for CUDA access

Flags

- Flags for resource registration

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),

CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

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 [cuD3D9UnregisterResource\(\)](#). 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 [cuD3D9UnregisterResource\(\)](#).

This call is potentially high-overhead and should not be called every frame in interactive applications.

The type of `pResource` must be one of the following.

- ▶ `IDirect3DVertexBuffer9`: Cannot be used with `Flags` set to `CU_D3D9_REGISTER_FLAGS_ARRAY`.
- ▶ `IDirect3DIndexBuffer9`: Cannot be used with `Flags` set to `CU_D3D9_REGISTER_FLAGS_ARRAY`.
- ▶ `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. For restrictions on the `Flags` parameter, see type `IDirect3DBaseTexture9`.
- ▶ `IDirect3DBaseTexture9`: When a texture is registered, all surfaces associated with the 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 values are allowed.

- ▶ `CU_D3D9_REGISTER_FLAGS_NONE`: Specifies that CUDA will access this resource through a [CUdeviceptr](#). The pointer, size, and (for textures), pitch for each subresource of this allocation may be queried through [cuD3D9ResourceGetMappedPointer\(\)](#), [cuD3D9ResourceGetMappedSize\(\)](#), and [cuD3D9ResourceGetMappedPitch\(\)](#) respectively. This option is valid for all resource types.
- ▶ `CU_D3D9_REGISTER_FLAGS_ARRAY`: Specifies that CUDA will access this resource through a [CUarray](#) queried on a sub-resource basis through [cuD3D9ResourceGetMappedArray\(\)](#). This option is only valid for resources of type `IDirect3DSurface9` and subtypes of `IDirect3DBaseTexture9`.

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

[CUDA_ERROR_INVALID_CONTEXT](#) is returned. If `pResource` is of incorrect type (e.g. is a non-stand-alone `IDirect3DSurface9`) or is already registered, then [CUDA_ERROR_INVALID_HANDLE](#) is returned. If `pResource` cannot be registered then [CUDA_ERROR_UNKNOWN](#) is returned.



Note:

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

See also:

[cuGraphicsD3D9RegisterResource](#)

CUresult cuD3D9ResourceGetMappedArray (CUarray *pArray, 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

pArray

- Returned array corresponding to subresource

pResource

- Mapped resource to access

Face

- Face of resource to access

Level

- Level of resource to access

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_MAPPED](#)

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 [CUDA_ERROR_INVALID_HANDLE](#) is returned. If `pResource` was not registered with usage flags `CU_D3D9_REGISTER_FLAGS_ARRAY` then [CUDA_ERROR_INVALID_HANDLE](#) is returned. If `pResource` is not mapped then [CUDA_ERROR_NOT_MAPPED](#) is returned.

For usage requirements of `Face` and `Level` parameters, see [cuD3D9ResourceGetMappedPointer\(\)](#).



Note:

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

See also:

[cuGraphicsSubResourceGetMappedArray](#)

CUresult cuD3D9ResourceGetMappedPitch (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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_MAPPED](#)

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:

$$y * \text{pitch} + (\text{bytes per pixel}) * x$$

For a 3D surface, the byte offset of the sample at position `x`, `y`, `z` from the base pointer of the surface is:

$$z * \text{slicePitch} + y * \text{pitch} + (\text{bytes per pixel}) * x$$

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 `CU_D3D9_REGISTER_FLAGS_NONE`, then [`CUDA\_ERROR\_INVALID\_HANDLE`](#) is returned. If `pResource` is not mapped for access by CUDA then [`CUDA\_ERROR\_NOT\_MAPPED`](#) is returned.

For usage requirements of `Face` and `Level` parameters, see [`cuD3D9ResourceGetMappedPointer\(\)`](#).



Note:

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

See also:

[`cuGraphicsSubResourceGetMappedArray`](#)

CUresult cuD3D9ResourceGetMappedPointer (CUdeviceptr *pDevPtr, IDirect3DResource9 *pResource, unsigned int Face, unsigned int Level)

Get the pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

Parameters

pDevPtr

- Returned pointer corresponding to subresource

pResource

- Mapped resource to access

Face

- Face of resource to access

Level

- Level of resource to access

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_MAPPED](#)

Description

[Deprecated](#) This function is deprecated as of CUDA 3.0.

Returns in `*pDevPtr` the base pointer of the subresource of the mapped Direct3D resource `pResource`, which corresponds to `Face` and `Level`. The value set in `pDevPtr` may change every time that `pResource` is mapped.

If `pResource` is not registered, then [CUDA_ERROR_INVALID_HANDLE](#) is returned. If `pResource` was not registered with usage flags `CU_D3D9_REGISTER_FLAGS_NONE`, then [CUDA_ERROR_INVALID_HANDLE](#) is returned. If `pResource` is not mapped, then [CUDA_ERROR_NOT_MAPPED](#) 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 [CUDA_ERROR_INVALID_VALUE](#) is returned.

If `pResource` is of type `IDirect3DBaseTexture9`, then `Level` must correspond to a valid mipmap level. At present only mipmap level 0 is supported. For all other types `Level` must be 0. If `Level` is invalid, then [CUDA_ERROR_INVALID_VALUE](#) is returned.

**Note:**

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

See also:

[cuGraphicsResourceGetMappedPointer](#)

**CUresult cuD3D9ResourceGetMappedSize (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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_MAPPED](#)

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

[CUDA_ERROR_INVALID_HANDLE](#) is returned. If pResource was not registered with usage flags CU_D3D9_REGISTER_FLAGS_NONE, then [CUDA_ERROR_INVALID_HANDLE](#) is returned. If pResource is not mapped for access by CUDA, then [CUDA_ERROR_NOT_MAPPED](#) is returned.

For usage requirements of Face and Level parameters, see [cuD3D9ResourceGetMappedPointer](#).

**Note:**

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

See also:

[cuGraphicsResourceGetMappedPointer](#)

CUresult cuD3D9ResourceGetSurfaceDimensions (size_t *pWidth, size_t *pHeight, size_t *pDepth, IDirect3DResource9 *pResource, unsigned int Face, unsigned int Level)

Get the dimensions of a registered 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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#)

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`.

Because 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 [CUDA_ERROR_INVALID_HANDLE](#) is returned.

For usage requirements of `Face` and `Level` parameters, see [cuD3D9ResourceGetMappedPointer\(\)](#).

**Note:**

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

See also:

[cuGraphicsSubResourceGetMappedArray](#)

CUresult cuD3D9ResourceSetMapFlags (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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_ALREADY_MAPPED](#)

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:

- ▶ `CU_D3D9_MAPRESOURCE_FLAGS_NONE`: 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.
- ▶ `CU_D3D9_MAPRESOURCE_FLAGS_READONLY`: Specifies that CUDA kernels which access this resource will not write to this resource.
- ▶ `CU_D3D9_MAPRESOURCE_FLAGS_WRITEDISCARD`: 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

[CUDA_ERROR_INVALID_HANDLE](#) is returned. If `pResource` is presently mapped for access by CUDA, then [CUDA_ERROR_ALREADY_MAPPED](#) is returned.



Note:

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

See also:

[cuGraphicsResourceSetMapFlags](#)

CUresult cuD3D9UnmapResources (unsigned int count, IDirect3DResource9 **ppResource)

Unmaps Direct3D resources.

Parameters

count

- Number of resources to unmap for CUDA

ppResource

- Resources to unmap for CUDA

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_MAPPED](#), [CUDA_ERROR_UNKNOWN](#)

Description

[Deprecated](#) This function is deprecated as of CUDA 3.0.

Unmaps the `count` Direct3D resources in `ppResource`.

This function provides the synchronization guarantee that any CUDA kernels issued before [cuD3D9UnmapResources\(\)](#) will complete before any Direct3D calls issued after [cuD3D9UnmapResources\(\)](#) begin.

If any of `ppResource` have not been registered for use with CUDA or if `ppResource` contains any duplicate entries, then [CUDA_ERROR_INVALID_HANDLE](#) is returned. If any of `ppResource` are not presently mapped for access by CUDA, then [CUDA_ERROR_NOT_MAPPED](#) is returned.



Note:

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

See also:

[cuGraphicsUnmapResources](#)

CUresult cuD3D9UnregisterResource (IDirect3DResource9 *pResource)

Unregister a Direct3D resource.

Parameters

pResource

- Resource to unregister

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_UNKNOWN](#)

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 [CUDA_ERROR_INVALID_HANDLE](#) is returned.



Note:

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

See also:

[cuGraphicsUnregisterResource](#)

6.41. Direct3D 10 Interoperability

This section describes the Direct3D 10 interoperability functions of the low-level CUDA driver application programming interface. Note that mapping of Direct3D 10 resources is performed with the graphics API agnostic, resource mapping interface described in [Graphics Interoperability](#).

Direct3D 10 Interoperability [DEPRECATED]

enum CUd3d10DeviceList

CUDA devices corresponding to a D3D10 device

Values

CU_D3D10_DEVICE_LIST_ALL = 0x01

The CUDA devices for all GPUs used by a D3D10 device

CU_D3D10_DEVICE_LIST_CURRENT_FRAME = 0x02

The CUDA devices for the GPUs used by a D3D10 device in its currently rendering frame

CU_D3D10_DEVICE_LIST_NEXT_FRAME = 0x03

The CUDA devices for the GPUs to be used by a D3D10 device in the next frame

CUresult cuD3D10GetDevice (CUdevice *pCudaDevice, IDXGIAdapter *pAdapter)

Gets the CUDA device corresponding to a display adapter.

Parameters

pCudaDevice

- Returned CUDA device corresponding to pAdapter

pAdapter

- Adapter to query for CUDA device

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_FOUND](#), [CUDA_ERROR_UNKNOWN](#)

Description

Returns in *pCudaDevice the CUDA-compatible device corresponding to the adapter pAdapter obtained from IDXGIFactory::EnumAdapters.

If no device on pAdapter is CUDA-compatible then the call will fail.



Note:

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

See also:

[cuD3D10GetDevices](#), [cudaD3D10GetDevice](#)

CUresult cuD3D10GetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, ID3D10Device *pD3D10Device, CUd3d10DeviceList 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 [CU_D3D10_DEVICE_LIST_ALL](#) for all devices, [CU_D3D10_DEVICE_LIST_CURRENT_FRAME](#) for the devices used to render the current frame (in SLI), or [CU_D3D10_DEVICE_LIST_NEXT_FRAME](#) for the devices used to render the next frame (in SLI).

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_NO_DEVICE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_FOUND](#), [CUDA_ERROR_UNKNOWN](#)

Description

Returns in *pCudaDeviceCount the number of CUDA-compatible device corresponding to the Direct3D 10 device pD3D10Device. Also returns in *pCudaDevices at most cudaDeviceCount of 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 [CUDA_ERROR_NO_DEVICE](#).



Note:

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

See also:

[cuD3D10GetDevice](#), [cudaD3D10GetDevices](#)

CUresult cuGraphicsD3D10RegisterResource (CUgraphicsResource *pCudaResource, ID3D10Resource *pD3DResource, unsigned int Flags)

Register a Direct3D 10 resource for access by CUDA.

Parameters

pCudaResource

- Returned graphics resource handle

pD3DResource

- Direct3D resource to register

Flags

- Parameters for resource registration

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_OUT_OF_MEMORY](#),
[CUDA_ERROR_UNKNOWN](#)

Description

Registers the Direct3D 10 resource `pD3DResource` for access by CUDA and returns a CUDA handle to `pD3DResource` in `pCudaResource`. The handle returned in `pCudaResource` may be used to map and unmap this resource until it is unregistered. On success this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through [cuGraphicsUnregisterResource\(\)](#).

This call is potentially 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 through 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

- ▶ `CU_GRAPHICS_REGISTER_FLAGS_NONE`: Specifies no hints about how this resource will be used.

- ▶ `CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST`: Specifies that CUDA will bind this resource to a surface reference.
- ▶ `CU_GRAPHICS_REGISTER_FLAGS_TEXTURE_GATHER`: 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

[`CUDA\_ERROR\_INVALID\_HANDLE`](#) is returned. If `pD3DResource` cannot be registered then [`CUDA\_ERROR\_UNKNOWN`](#) is returned. If `Flags` is not one of the above specified value then [`CUDA\_ERROR\_INVALID\_VALUE`](#) is returned.



Note:

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

See also:

[cuGraphicsUnregisterResource](#), [cuGraphicsMapResources](#), [cuGraphicsSubResourceGetMappedArray](#), [cuGraphicsResourceGetMappedPointer](#), [cudaGraphicsD3D10RegisterResource](#)

6.41.1. Direct3D 10 Interoperability [DEPRECATED]

Direct3D 10 Interoperability

This section describes deprecated Direct3D 10 interoperability functionality.

enum CUD3D10map_flags

Flags to map or unmap a resource

Values

CU_D3D10_MAPRESOURCE_FLAGS_NONE = 0x00

CU_D3D10_MAPRESOURCE_FLAGS_READONLY = 0x01

CU_D3D10_MAPRESOURCE_FLAGS_WRITEDISCARD = 0x02

enum CUD3D10register_flags

Flags to register a resource

Values

CU_D3D10_REGISTER_FLAGS_NONE = 0x00

CU_D3D10_REGISTER_FLAGS_ARRAY = 0x01

CUresult cuD3D10CtxCreate (CUcontext *pCtx, CUdevice *pCudaDevice, unsigned int Flags, ID3D10Device *pD3DDevice)

Create a CUDA context for interoperability with Direct3D 10.

Parameters

pCtx

- Returned newly created CUDA context

pCudaDevice

- Returned pointer to the device on which the context was created

Flags

- Context creation flags (see [cuCtxCreate\(\)](#) for details)

pD3DDevice

- Direct3D device to create interoperability context with

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_UNKNOWN](#)

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 context 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:

[cuD3D10GetDevice](#), [cuGraphicsD3D10RegisterResource](#)

CUresult cuD3D10CtxCreateOnDevice (CUcontext *pCtx, unsigned int flags, ID3D10Device *pD3DDevice, CUdevice cudaDevice)

Create a CUDA context for interoperability with Direct3D 10.

Parameters

pCtx

- Returned newly created CUDA context

flags

- Context creation flags (see [cuCtxCreate\(\)](#) for details)

pD3DDevice

- Direct3D device to create interoperability context with

cudaDevice

- The CUDA device on which to create the context. This device must be among the devices returned when querying CU_D3D10_DEVICES_ALL from [cuD3D10GetDevices](#).

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_UNKNOWN](#)

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 context 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:

[cuD3D10GetDevices](#), [cuGraphicsD3D10RegisterResource](#)

CUresult cuD3D10GetDirect3DDevice (ID3D10Device **ppD3DDevice)

Get the Direct3D 10 device against which the current CUDA context was created.

Parameters

ppD3DDevice

- Returned Direct3D device corresponding to CUDA context

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#)

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 context 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:

[cuD3D10GetDevice](#)

CUresult cuD3D10MapResources (unsigned int count, ID3D10Resource **ppResources)

Map Direct3D resources for access by CUDA.

Parameters

count

- Number of resources to map for CUDA

ppResources

- Resources to map for CUDA

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_ALREADY_MAPPED](#), [CUDA_ERROR_UNKNOWN](#)

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 [cuD3D10MapResources\(\)](#) will complete before any CUDA kernels issued after [cuD3D10MapResources\(\)](#) begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries, then [CUDA_ERROR_INVALID_HANDLE](#) is returned. If any of `ppResources` are presently mapped for access by CUDA, then [CUDA_ERROR_ALREADY_MAPPED](#) is returned.



Note:

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

See also:

[cuGraphicsMapResources](#)

CUresult cuD3D10RegisterResource (ID3D10Resource *pResource, unsigned int Flags)

Register a Direct3D resource for access by CUDA.

Parameters

pResource

- Resource to register

Flags

- Parameters for resource registration

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_UNKNOWN](#)

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 [cuD3D10UnregisterResource\(\)](#). 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 [cuD3D10UnregisterResource\(\)](#).

This call is potentially 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 `CU_D3D10_REGISTER_FLAGS_ARRAY`.
- ▶ 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.

- ▶ `CU_D3D10_REGISTER_FLAGS_NONE`: Specifies that CUDA will access this resource through a [CUdeviceptr](#). The pointer, size, and (for textures), pitch for each subresource of this allocation may be queried through [cuD3D10ResourceGetMappedPointer\(\)](#), [cuD3D10ResourceGetMappedSize\(\)](#), and [cuD3D10ResourceGetMappedPitch\(\)](#) respectively. This option is valid for all resource types.
- ▶ `CU_D3D10_REGISTER_FLAGS_ARRAY`: Specifies that CUDA will access this resource through a [CUarray](#) queried on a sub-resource basis through [cuD3D10ResourceGetMappedArray\(\)](#).

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 [`CUDA\_ERROR\_INVALID\_CONTEXT`](#) is returned. If `pResource` is of incorrect type or is already registered, then [`CUDA\_ERROR\_INVALID\_HANDLE`](#) is returned. If `pResource` cannot be registered, then [`CUDA\_ERROR\_UNKNOWN`](#) is returned.



Note:

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

See also:

[`cuGraphicsD3D10RegisterResource`](#)

CUresult cuD3D10ResourceGetMappedArray (CUarray *pArray, ID3D10Resource *pResource, unsigned int SubResource)

Get an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

Parameters

pArray

- Returned array corresponding to subresource

pResource

- Mapped resource to access

SubResource

- Subresource of `pResource` to access

Returns

[`CUDA\_SUCCESS`](#), [`CUDA\_ERROR\_DEINITIALIZED`](#), [`CUDA\_ERROR\_NOT\_INITIALIZED`](#), [`CUDA\_ERROR\_INVALID\_CONTEXT`](#), [`CUDA\_ERROR\_INVALID\_VALUE`](#), [`CUDA\_ERROR\_INVALID\_HANDLE`](#), [`CUDA\_ERROR\_NOT\_MAPPED`](#)

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 `SubResource` may be accessed. The value set in `pArray` may change every time that `pResource` is mapped.

If `pResource` is not registered, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` was not registered with usage flags `CU_D3D10_REGISTER_FLAGS_ARRAY`, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` is not mapped, then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of the `SubResource` parameter, see [cuD3D10ResourceGetMappedPointer\(\)](#).



Note:

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

See also:

[cuGraphicsSubResourceGetMappedArray](#)

CUresult cuD3D10ResourceGetMappedPitch (size_t *pPitch, size_t *pPitchSlice, ID3D10Resource *pResource, unsigned int SubResource)

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

SubResource

- Subresource of `pResource` to access

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

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:

$$y * \text{pitch} + (\text{bytes per pixel}) * x$$

For a 3D surface, the byte offset of the sample at position `x`, `y`, `z` from the base pointer of the surface is:

$$z * \text{slicePitch} + y * \text{pitch} + (\text{bytes per pixel}) * x$$

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

If `pResource` is not of type `IDirect3DBaseTexture10` or one of its sub-types or if `pResource` has not been registered for use with CUDA, then [CUDA_ERROR_INVALID_HANDLE](#) is returned. If `pResource` was not registered with usage flags `CU_D3D10_REGISTER_FLAGS_NONE`, then [CUDA_ERROR_INVALID_HANDLE](#) is returned. If `pResource` is not mapped for access by CUDA, then [CUDA_ERROR_NOT_MAPPED](#) is returned.

For usage requirements of the `SubResource` parameter, see [cuD3D10ResourceGetMappedPointer\(\)](#).



Note:

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

See also:

[cuGraphicsSubResourceGetMappedArray](#)

CUresult cuD3D10ResourceGetMappedPointer (CUdeviceptr *pDevPtr, ID3D10Resource *pResource, unsigned int SubResource)

Get a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

Parameters

pDevPtr

- Returned pointer corresponding to subresource

pResource

- Mapped resource to access

SubResource

- Subresource of pResource to access

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_MAPPED](#)

Description

Deprecated This function is deprecated as of CUDA 3.0.

Returns in *pDevPtr the base pointer of the subresource of the mapped Direct3D resource pResource, which corresponds to SubResource. The value set in pDevPtr may change every time that pResource is mapped.

If pResource is not registered, then [CUDA_ERROR_INVALID_HANDLE](#) is returned. If pResource was not registered with usage flags CU_D3D10_REGISTER_FLAGS_NONE, then [CUDA_ERROR_INVALID_HANDLE](#) is returned. If pResource is not mapped, then [CUDA_ERROR_NOT_MAPPED](#) 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:

[cuGraphicsResourceGetMappedPointer](#)

CUresult cuD3D10ResourceGetMappedSize (size_t *pSize, ID3D10Resource *pResource, unsigned int SubResource)

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

SubResource

- Subresource of pResource to access

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_MAPPED](#)

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

[CUDA_ERROR_INVALID_HANDLE](#) is returned. If pResource was not registered with usage flags CU_D3D10_REGISTER_FLAGS_NONE, then [CUDA_ERROR_INVALID_HANDLE](#) is returned. If pResource is not mapped for access by CUDA, then [CUDA_ERROR_NOT_MAPPED](#) is returned.

For usage requirements of the SubResource parameter, see [cuD3D10ResourceGetMappedPointer\(\)](#).

**Note:**

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

See also:

[cuGraphicsResourceGetMappedPointer](#)

CUresult cuD3D10ResourceGetSurfaceDimensions (size_t *pWidth, size_t *pHeight, size_t *pDepth, ID3D10Resource *pResource, unsigned int SubResource)

Get the dimensions of a registered 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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#)

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.

Because 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 IDirect3DBaseTexture10 or IDirect3DSurface10 or if pResource has not been registered for use with CUDA, then [CUDA_ERROR_INVALID_HANDLE](#) is returned.

For usage requirements of the SubResource parameter, see [cuD3D10ResourceGetMappedPointer\(\)](#).

**Note:**

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

See also:

[cuGraphicsSubResourceGetMappedArray](#)

CUresult cuD3D10ResourceSetMapFlags (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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_ALREADY_MAPPED](#)

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.

- ▶ `CU_D3D10_MAPRESOURCE_FLAGS_NONE`: 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.
- ▶ `CU_D3D10_MAPRESOURCE_FLAGS_READONLY`: Specifies that CUDA kernels which access this resource will not write to this resource.
- ▶ `CU_D3D10_MAPRESOURCE_FLAGS_WRITEDISCARD`: 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

[CUDA_ERROR_INVALID_HANDLE](#) is returned. If `pResource` is presently mapped for access by CUDA then [CUDA_ERROR_ALREADY_MAPPED](#) is returned.



Note:

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

See also:

[cuGraphicsResourceSetMapFlags](#)

CUresult cuD3D10UnmapResources (unsigned int count, ID3D10Resource **ppResources)

Unmap Direct3D resources.

Parameters

count

- Number of resources to unmap for CUDA

ppResources

- Resources to unmap for CUDA

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_MAPPED](#),
[CUDA_ERROR_UNKNOWN](#)

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 [cuD3D10UnmapResources\(\)](#) will complete before any Direct3D calls issued after [cuD3D10UnmapResources\(\)](#) begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries, then [CUDA_ERROR_INVALID_HANDLE](#) is returned.

If any of `ppResources` are not presently mapped for access by CUDA, then [CUDA_ERROR_NOT_MAPPED](#) is returned.



Note:

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

See also:

[cuGraphicsUnmapResources](#)

CUresult cuD3D10UnregisterResource (ID3D10Resource *pResource)

Unregister a Direct3D resource.

Parameters

pResource

- Resources to unregister

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#),
[CUDA_ERROR_UNKNOWN](#)

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 CUDA_ERROR_INVALID_HANDLE is returned.



Note:

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

See also:

[cuGraphicsUnregisterResource](#)

6.42. Direct3D 11 Interoperability

This section describes the Direct3D 11 interoperability functions of the low-level CUDA driver application programming interface. Note that mapping of Direct3D 11 resources is performed with the graphics API agnostic, resource mapping interface described in [Graphics Interoperability](#).

Direct3D 11 Interoperability [DEPRECATED]

enum CUd3d11DeviceList

CUDA devices corresponding to a D3D11 device

Values

CU_D3D11_DEVICE_LIST_ALL = 0x01

The CUDA devices for all GPUs used by a D3D11 device

CU_D3D11_DEVICE_LIST_CURRENT_FRAME = 0x02

The CUDA devices for the GPUs used by a D3D11 device in its currently rendering frame

CU_D3D11_DEVICE_LIST_NEXT_FRAME = 0x03

The CUDA devices for the GPUs to be used by a D3D11 device in the next frame

CUresult cuD3D11GetDevice (CUdevice *pCudaDevice, IDXGIAdapter *pAdapter)

Gets the CUDA device corresponding to a display adapter.

Parameters

pCudaDevice

- Returned CUDA device corresponding to pAdapter

pAdapter

- Adapter to query for CUDA device

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_NO_DEVICE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_FOUND](#), [CUDA_ERROR_UNKNOWN](#)

Description

Returns in *pCudaDevice the CUDA-compatible device corresponding to the adapter pAdapter obtained from IDXGIFactory::EnumAdapters.

If no device on pAdapter is CUDA-compatible the call will return [CUDA_ERROR_NO_DEVICE](#).



Note:

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

See also:

[cuD3D11GetDevices](#), [cudaD3D11GetDevice](#)

CUresult cuD3D11GetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, IDXGIAdapter *pD3D11Device, CUd3d11DeviceList 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 [CU_D3D11_DEVICE_LIST_ALL](#) for all devices, [CU_D3D11_DEVICE_LIST_CURRENT_FRAME](#) for the devices used to render the current frame (in SLI), or [CU_D3D11_DEVICE_LIST_NEXT_FRAME](#) for the devices used to render the next frame (in SLI).

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_NO_DEVICE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_NOT_FOUND](#), [CUDA_ERROR_UNKNOWN](#)

Description

Returns in *pCudaDeviceCount the number of CUDA-compatible device corresponding to the Direct3D 11 device pD3D11Device. Also returns in *pCudaDevices at most cudaDeviceCount of 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 [CUDA_ERROR_NO_DEVICE](#).

**Note:**

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

See also:

[cuD3D11GetDevice](#), [cudaD3D11GetDevices](#)

CUresult cuGraphicsD3D11RegisterResource (CUgraphicsResource *pCudaResource, ID3D11Resource *pD3DResource, unsigned int Flags)

Register a Direct3D 11 resource for access by CUDA.

Parameters

pCudaResource

- Returned graphics resource handle

pD3DResource

- Direct3D resource to register

Flags

- Parameters for resource registration

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_OUT_OF_MEMORY](#),
[CUDA_ERROR_UNKNOWN](#)

Description

Registers the Direct3D 11 resource `pD3DResource` for access by CUDA and returns a CUDA handle to `pD3DResource` in `pCudaResource`. The handle returned in `pCudaResource` may be used to map and unmap this resource until it is unregistered. On success this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through [cuGraphicsUnregisterResource\(\)](#).

This call is potentially 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 through 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

- ▶ `CU_GRAPHICS_REGISTER_FLAGS_NONE`: Specifies no hints about how this resource will be used.
- ▶ `CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST`: Specifies that CUDA will bind this resource to a surface reference.

- ▶ `CU_GRAPHICS_REGISTER_FLAGS_TEXTURE_GATHER`: 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 [`CUDA\_ERROR\_INVALID\_HANDLE`](#) is returned. If `pD3DResource` cannot be registered then [`CUDA\_ERROR\_UNKNOWN`](#) is returned. If `Flags` is not one of the above specified value then [`CUDA\_ERROR\_INVALID\_VALUE`](#) is returned.



Note:

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

See also:

[cuGraphicsUnregisterResource](#), [cuGraphicsMapResources](#), [cuGraphicsSubResourceGetMappedArray](#), [cuGraphicsResourceGetMappedPointer](#), [cudaGraphicsD3D11RegisterResource](#)

6.42.1. Direct3D 11 Interoperability [DEPRECATED]

Direct3D 11 Interoperability

This section describes deprecated Direct3D 11 interoperability functionality.

CUresult cuD3D11CtxCreate (CUcontext *pCtx, CUdevice *pCudaDevice, unsigned int Flags, ID3D11Device *pD3DDevice)

Create a CUDA context for interoperability with Direct3D 11.

Parameters

pCtx

- Returned newly created CUDA context

pCudaDevice

- Returned pointer to the device on which the context was created

Flags

- Context creation flags (see [cuCtxCreate\(\)](#) for details)

pD3DDevice

- Direct3D device to create interoperability context with

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_UNKNOWN](#)

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 context 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:

[cuD3D11GetDevice](#), [cuGraphicsD3D11RegisterResource](#)

CUresult cuD3D11CtxCreateOnDevice (CUcontext *pCtx, unsigned int flags, ID3D11Device *pD3DDevice, CUdevice cudaDevice)

Create a CUDA context for interoperability with Direct3D 11.

Parameters

pCtx

- Returned newly created CUDA context

flags

- Context creation flags (see [cuCtxCreate\(\)](#) for details)

pD3DDevice

- Direct3D device to create interoperability context with

cudaDevice

- The CUDA device on which to create the context. This device must be among the devices returned when querying CU_D3D11_DEVICES_ALL from [cuD3D11GetDevices](#).

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_OUT_OF_MEMORY](#), [CUDA_ERROR_UNKNOWN](#)

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 context 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:

[cuD3D11GetDevices](#), [cuGraphicsD3D11RegisterResource](#)

CUresult cuD3D11GetDirect3DDevice (ID3D11Device **ppD3DDevice)

Get the Direct3D 11 device against which the current CUDA context was created.

Parameters

ppD3DDevice

- Returned Direct3D device corresponding to CUDA context

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#)

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 context 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:

[cuD3D11GetDevice](#)

6.43. VDPAU Interoperability

This section describes the VDPAU interoperability functions of the low-level CUDA driver application programming interface.

CUresult cuGraphicsVDPAURegisterOutputSurface (CUgraphicsResource *pCudaResource, VdpOutputSurface vdpSurface, unsigned int flags)

Registers a VDPAU VdpOutputSurface object.

Parameters

pCudaResource

- Pointer to the returned object handle

vdpSurface

- The VdpOutputSurface to be registered

flags

- Map flags

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_ALREADY_MAPPED](#), [CUDA_ERROR_INVALID_CONTEXT](#),

Description

Registers the VdpOutputSurface specified by `vdpSurface` for access by CUDA. A handle to the registered object is returned as `pCudaResource`. The surface's intended usage is specified using `flags`, as follows:

- ▶ `CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE`: 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.
- ▶ `CU_GRAPHICS_MAP_RESOURCE_FLAGS_READ_ONLY`: Specifies that CUDA will not write to this resource.
- ▶ `CU_GRAPHICS_MAP_RESOURCE_FLAGS_WRITE_DISCARD`: 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 VdpOutputSurface is presented as an array of subresources that may be accessed using pointers returned by [cuGraphicsSubResourceGetMappedArray](#). The exact number of valid `arrayIndex` values depends on the VDPAU surface format. The mapping is shown in the table below. `mipLevel` must be 0.

**Note:**

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

See also:

[cuCtxCreate](#), [cuVDPAUCtxCreate](#), [cuGraphicsVDPAURegisterVideoSurface](#), [cuGraphicsUnregisterResource](#), [cuGraphicsResourceSetMapFlags](#), [cuGraphicsMapResources](#), [cuGraphicsUnmapResources](#), [cuGraphicsSubResourceGetMappedArray](#), [cuVDPAUGetDevice](#), [cudaGraphicsVDPAURegisterOutputSurface](#)

CUresult cuGraphicsVDPAURegisterVideoSurface (CUgraphicsResource *pCudaResource, VdpVideoSurface vdpSurface, unsigned int flags)

Registers a VDPAU VdpVideoSurface object.

Parameters

pCudaResource

- Pointer to the returned object handle

vdpSurface

- The VdpVideoSurface to be registered

flags

- Map flags

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_ALREADY_MAPPED](#),
[CUDA_ERROR_INVALID_CONTEXT](#),

Description

Registers the VdpVideoSurface specified by vdpSurface for access by CUDA. A handle to the registered object is returned as pCudaResource. The surface's intended usage is specified using flags, as follows:

- ▶ [CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE](#): 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.
- ▶ [CU_GRAPHICS_MAP_RESOURCE_FLAGS_READ_ONLY](#): Specifies that CUDA will not write to this resource.
- ▶ [CU_GRAPHICS_MAP_RESOURCE_FLAGS_WRITE_DISCARD](#): 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 VdpVideoSurface is presented as an array of subresources that may be accessed using pointers returned by [cuGraphicsSubResourceGetMappedArray](#). The exact number of valid arrayIndex values depends on the VDPAU surface format. The mapping is shown in the table below. mipLevel must be 0.



Note:

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

See also:

[cuCtxCreate](#), [cuVDPAUCtxCreate](#), [cuGraphicsVDPAURegisterOutputSurface](#),
[cuGraphicsUnregisterResource](#), [cuGraphicsResourceSetMapFlags](#), [cuGraphicsMapResources](#),
[cuGraphicsUnmapResources](#), [cuGraphicsSubResourceGetMappedArray](#), [cuVDPAUGetDevice](#),
[cudaGraphicsVDPAURegisterVideoSurface](#)

CUresult cuVDPAUCtxCreate (CUcontext *pCtx, unsigned int flags, CUdevice device, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)

Create a CUDA context for interoperability with VDPAU.

Parameters

pCtx

- Returned CUDA context

flags

- Options for CUDA context creation

device

- Device on which to create the context

vdpDevice

- The VdpDevice to interop with

vdpGetProcAddress

- VDPAU's VdpGetProcAddress function pointer

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Creates a new CUDA context, initializes VDPAU interoperability, and associates the CUDA context with the calling thread. It must be called before performing any other VDPAU interoperability operations. It may fail if the needed VDPAU driver facilities are not available. For usage of the `flags` parameter, see [cuCtxCreate\(\)](#).



Note:

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

See also:

[cuCtxCreate](#), [cuGraphicsVDPAURegisterVideoSurface](#), [cuGraphicsVDPAURegisterOutputSurface](#), [cuGraphicsUnregisterResource](#), [cuGraphicsResourceSetMapFlags](#), [cuGraphicsMapResources](#), [cuGraphicsUnmapResources](#), [cuGraphicsSubResourceGetMappedArray](#), [cuVDPAUGetDevice](#)

CUresult cuVDPAUGetDevice (CUdevice *pDevice, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)

Gets the CUDA device associated with a VDPAU device.

Parameters

pDevice

- Device associated with vdpDevice

vdpDevice

- A VdpDevice handle

vdpGetProcAddress

- VDPAU's VdpGetProcAddress function pointer

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#)

Description

Returns in *pDevice 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:

[cuCtxCreate](#), [cuVDPAUCtxCreate](#), [cuGraphicsVDPAURegisterVideoSurface](#), [cuGraphicsVDPAURegisterOutputSurface](#), [cuGraphicsUnregisterResource](#), [cuGraphicsResourceSetMapFlags](#), [cuGraphicsMapResources](#), [cuGraphicsUnmapResources](#), [cuGraphicsSubResourceGetMappedArray](#), [cuVDPAUGetDevice](#)

6.44. EGL Interoperability

This section describes the EGL interoperability functions of the low-level CUDA driver application programming interface.

CUresult cuEGLStreamConsumerAcquireFrame (CUeglStreamConnection *conn, CUgraphicsResource *pCudaResource, CUstream *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 stream frame will be mapped for use.

pStream

- CUDA stream for synchronization and any data migrations implied by [CUeglResourceLocationFlags](#).

timeout

- Desired timeout in usec for a new frame to be acquired. If set as [CUDA_EGL_INFINITE_TIMEOUT](#), acquire waits infinitely. After timeout occurs CUDA consumer tries to acquire an old frame if available and EGL_SUPPORT_REUSE_NV flag is set.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_LAUNCH_TIMEOUT](#),

Description

Acquire an image frame from EGLStreamKHR. This API can also acquire an old frame presented by the producer unless explicitly disabled by setting EGL_SUPPORT_REUSE_NV flag to EGL_FALSE during stream initialization. By default, EGLStream is created with this flag set to EGL_TRUE.

[cuGraphicsResourceGetMappedEglFrame](#) can be called on pCudaResource to get CUeglFrame.

See also:

[cuEGLStreamConsumerConnect](#), [cuEGLStreamConsumerDisconnect](#),
[cuEGLStreamConsumerAcquireFrame](#), [cuEGLStreamConsumerReleaseFrame](#),
[cudaEGLStreamConsumerAcquireFrame](#)

CUresult cuEGLStreamConsumerConnect (CUeglStreamConnection *conn, EGLStreamKHR stream)

Connect CUDA to EGLStream as a consumer.

Parameters

conn

- Pointer to the returned connection handle

stream

- EGLStreamKHR handle

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_CONTEXT](#),

Description

Connect CUDA as a consumer 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:

[cuEGLStreamConsumerConnect](#), [cuEGLStreamConsumerDisconnect](#),
[cuEGLStreamConsumerAcquireFrame](#), [cuEGLStreamConsumerReleaseFrame](#),
[cudaEGLStreamConsumerConnect](#)

CUresult cuEGLStreamConsumerConnectWithFlags (CUeglStreamConnection *conn, EGLStreamKHR stream, unsigned int flags)

Connect CUDA to EGLStream as a consumer with given flags.

Parameters

conn

- Pointer to the returned connection handle

stream

- EGLStreamKHR handle

flags

- Flags denote intended location - system or video.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_CONTEXT](#),

Description

Connect CUDA as a consumer to EGLStreamKHR specified by `stream` with specified `flags` defined by `CUeglResourceLocationFlags`.

The flags specify whether the consumer wants to access frames from system memory or video memory. Default is [CU_EGL_RESOURCE_LOCATION_VIDMEM](#).

See also:

[cuEGLStreamConsumerConnect](#), [cuEGLStreamConsumerDisconnect](#),
[cuEGLStreamConsumerAcquireFrame](#), [cuEGLStreamConsumerReleaseFrame](#),
[cudaEGLStreamConsumerConnectWithFlags](#)

CUresult cuEGLStreamConsumerDisconnect (CUeglStreamConnection *conn)

Disconnect CUDA as a consumer to EGLStream .

Parameters

conn

- Conection to disconnect.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_CONTEXT](#),

Description

Disconnect CUDA as a consumer to EGLStreamKHR.

See also:

[cuEGLStreamConsumerConnect](#), [cuEGLStreamConsumerDisconnect](#),
[cuEGLStreamConsumerAcquireFrame](#), [cuEGLStreamConsumerReleaseFrame](#),
[cudaEGLStreamConsumerDisconnect](#)

CUresult cuEGLStreamConsumerReleaseFrame (CUeglStreamConnection *conn, CUgraphicsResource pCudaResource, CUstream *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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#),

Description

Release the acquired image frame specified by pCudaResource to EGLStreamKHR. If EGL_SUPPORT_REUSE_NV flag is set to EGL_TRUE, at the time of EGL creation this API doesn't release the last frame acquired on the EGLStream. By default, EGLStream is created with this flag set to EGL_TRUE.

See also:

[cuEGLStreamConsumerConnect](#), [cuEGLStreamConsumerDisconnect](#),
[cuEGLStreamConsumerAcquireFrame](#), [cuEGLStreamConsumerReleaseFrame](#),
[cudaEGLStreamConsumerReleaseFrame](#)

CUresult cuEGLStreamProducerConnect (CUeglStreamConnection *conn, EGLStreamKHR stream, EGLint width, EGLint height)

Connect CUDA to EGLStream as a producer.

Parameters

conn

- Pointer to the returned connection handle

stream

- 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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_CONTEXT](#),

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:

[cuEGLStreamProducerConnect](#), [cuEGLStreamProducerDisconnect](#),
[cuEGLStreamProducerPresentFrame](#), [cudaEGLStreamProducerConnect](#)

CUresult cuEGLStreamProducerDisconnect (CUeglStreamConnection *conn)

Disconnect CUDA as a producer to EGLStream .

Parameters**conn**

- Connection to disconnect.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_CONTEXT](#),

Description

Disconnect CUDA as a producer to EGLStreamKHR.

See also:

[cuEGLStreamProducerConnect](#), [cuEGLStreamProducerDisconnect](#),
[cuEGLStreamProducerPresentFrame](#), [cudaEGLStreamProducerDisconnect](#)

CUresult cuEGLStreamProducerPresentFrame (CUeglStreamConnection *conn, CUeglFrame eglframe, CUstream *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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#),

Description

When a frame is presented by the producer, it gets associated with the EGLStream and thus it is illegal to free the frame before the producer is disconnected. If a frame is freed and reused it may lead to undefined behavior.

If producer and consumer are on different GPUs (iGPU and dGPU) then frametype [CU_EGL_FRAME_TYPE_ARRAY](#) is not supported. [CU_EGL_FRAME_TYPE_PITCH](#) can be used for such cross-device applications.

The CUeglFrame is defined as:

```
typedef struct CUeglFrame_st {
    union {
        CUarray pArray[MAX_PLANES];
        void* pPitch[MAX_PLANES];
    } frame;
    unsigned int width;
    unsigned int height;
    unsigned int depth;
    unsigned int pitch;
    unsigned int planeCount;
    unsigned int numChannels;
    CUeglFrameType frameType;
    CUeglColorFormat eglColorFormat;
    CUarray_format cuFormat;
} CUeglFrame;
```

For CUeglFrame of type [CU_EGL_FRAME_TYPE_PITCH](#), the application may present sub-region of a memory allocation. In that case, the pitched pointer will specify the start address of the sub-region in the allocation and corresponding CUeglFrame fields will specify the dimensions of the sub-region.

See also:

[cuEGLStreamProducerConnect](#), [cuEGLStreamProducerDisconnect](#),
[cuEGLStreamProducerReturnFrame](#), [cudaEGLStreamProducerPresentFrame](#)

CUresult cuEGLStreamProducerReturnFrame (CUeglStreamConnection *conn, CUeglFrame *eglframe, CUstream *pStream)

Return the CUDA eglFrame to the EGLStream released by the consumer.

Parameters

conn

- Connection on which to return

eglframe

- CUDA Eglstream Proucer Frame handle returned from the consumer over EglStream.

pStream

- CUDA stream on which to return the frame.

Returns

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_LAUNCH_TIMEOUT](#)

Description

This API can potentially return CUDA_ERROR_LAUNCH_TIMEOUT if the consumer has not returned a frame to EGL stream. If timeout is returned the application can retry.

See also:

[cuEGLStreamProducerConnect](#), [cuEGLStreamProducerDisconnect](#),
[cuEGLStreamProducerPresentFrame](#), [cudaEGLStreamProducerReturnFrame](#)

CUresult cuEventCreateFromEGLSync (CUevent *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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_OUT_OF_MEMORY](#)

Description

Creates an event `*phEvent` from an `EGLSyncKHR` `eglSync` with the flags specified via `flags`. Valid flags include:

- ▶ [CU_EVENT_DEFAULT](#): Default event creation flag.
- ▶ [CU_EVENT_BLOCKING_SYNC](#): Specifies that the created event should use blocking synchronization. A CPU thread that uses [cuEventSynchronize\(\)](#) to wait on an event created with this flag will block until the event has actually been completed.

Once the `eglSync` gets destroyed, [cuEventDestroy](#) is the only API that can be invoked on the event.

[cuEventRecord](#) 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:

[cuEventQuery](#), [cuEventSynchronize](#), [cuEventDestroy](#)

CUresult cuGraphicsEGLRegisterImage (CUgraphicsResource *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

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_ALREADY_MAPPED](#),
[CUDA_ERROR_INVALID_CONTEXT](#),

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 [cuGraphicsResourceGetMappedEglFrame](#) 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 GL contexts (for OpenGL and likewise for other APIs). The application will be also responsible for ensuring that any pending operation on the registered CUDA resource has completed prior to executing subsequent commands in other APIs accessing the same memory objects. This can be accomplished by calling `cuCtxSynchronize` or `cuEventSynchronize` (preferably).

The surface's intended usage is specified using `flags`, as follows:

- ▶ `CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE`: 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.
- ▶ `CU_GRAPHICS_MAP_RESOURCE_FLAGS_READ_ONLY`: Specifies that CUDA will not write to this resource.
- ▶ `CU_GRAPHICS_MAP_RESOURCE_FLAGS_WRITE_DISCARD`: 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:

[cuGraphicsEGLRegisterImage](#), [cuGraphicsUnregisterResource](#), [cuGraphicsResourceSetMapFlags](#), [cuGraphicsMapResources](#), [cuGraphicsUnmapResources](#), [cudaGraphicsEGLRegisterImage](#)

CUresult cuGraphicsResourceGetMappedEglFrame (CUeglFrame *eglFrame, CUgraphicsResource 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

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#),
[CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_VALUE](#),
[CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_NOT_MAPPED](#)

Description

Returns in *eglFrame an eglFrame pointer through which the registered graphics resource resource may be accessed. This API can only be called for registered EGL graphics resources.

The CUeglFrame is defined as:

```
typedef struct CUeglFrame_st {
    union {
        CUarray pArray[MAX_PLANES];
        void* pPitch[MAX_PLANES];
    } frame;
    unsigned int width;
    unsigned int height;
    unsigned int depth;
    unsigned int pitch;
    unsigned int planeCount;
    unsigned int numChannels;
    CUeglFrameType frameType;
    CUeglColorFormat eglColorFormat;
    CUarray_format cuFormat;
} CUeglFrame;
```

If resource is not registered then [CUDA_ERROR_NOT_MAPPED](#) is returned. *

See also:

[cuGraphicsMapResources](#), [cuGraphicsSubResourceGetMappedArray](#),
[cuGraphicsResourceGetMappedPointer](#), [cudaGraphicsResourceGetMappedEglFrame](#)

Chapter 7. Data Structures

Here are the data structures with brief descriptions:

[CUaccessPolicyWindow_v1](#)

[CUarrayMapInfo_v1](#)

[CUasyncNotificationInfo](#)

[CUcheckpointCheckpointArgs](#)

[CUcheckpointLockArgs](#)

[CUcheckpointRestoreArgs](#)

[CUcheckpointUnlockArgs](#)

[CUctxCigParam](#)

[CUctxCreateParams](#)

[CUDA_ARRAY3D_DESCRIPTOR_v2](#)

[CUDA_ARRAY_DESCRIPTOR_v2](#)

[CUDA_ARRAY_MEMORY_REQUIREMENTS_v1](#)

[CUDA_ARRAY_SPARSE_PROPERTIES_v1](#)

[CUDA_BATCH_MEM_OP_NODE_PARAMS_v2](#)

[CUDA_CHILD_GRAPH_NODE_PARAMS](#)

[CUDA_CONDITIONAL_NODE_PARAMS](#)

[CUDA_EVENT_RECORD_NODE_PARAMS](#)

[CUDA_EVENT_WAIT_NODE_PARAMS](#)

[CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v1](#)

[CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v2](#)

[CUDA_EXT_SEM_WAIT_NODE_PARAMS_v1](#)

[CUDA_EXT_SEM_WAIT_NODE_PARAMS_v2](#)

[CUDA_EXTERNAL_MEMORY_BUFFER_DESC_v1](#)

[CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1](#)

[CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC_v1](#)

[CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_v1](#)

[CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1](#)

[CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1](#)

[CUDA_GRAPH_INSTANTIATE_PARAMS](#)

[CUDA_HOST_NODE_PARAMS_v1](#)

[CUDA_HOST_NODE_PARAMS_v2](#)

[CUDA_KERNEL_NODE_PARAMS_v1](#)

[CUDA_KERNEL_NODE_PARAMS_v2](#)
[CUDA_KERNEL_NODE_PARAMS_v3](#)
[CUDA_LAUNCH_PARAMS_v1](#)
[CUDA_MEM_ALLOC_NODE_PARAMS_v1](#)
[CUDA_MEM_ALLOC_NODE_PARAMS_v2](#)
[CUDA_MEM_FREE_NODE_PARAMS](#)
[CUDA_MEMCPY2D_v2](#)
[CUDA_MEMCPY3D_PEER_v1](#)
[CUDA_MEMCPY3D_v2](#)
[CUDA_MEMCPY_NODE_PARAMS](#)
[CUDA_MEMSET_NODE_PARAMS_v1](#)
[CUDA_MEMSET_NODE_PARAMS_v2](#)
[CUDA_POINTER_ATTRIBUTE_P2P_TOKENS_v1](#)
[CUDA_RESOURCE_DESC_v1](#)
[CUDA_RESOURCE_VIEW_DESC_v1](#)
[CUDA_TEXTURE_DESC_v1](#)
[CUdevprop_v1](#)
[CUdevResource](#)
[CUdevSmResource](#)
[CUeglFrame_v1](#)
[CUexecAffinityParam_v1](#)
[CUexecAffinitySmCount_v1](#)
[CUextent3D_v1](#)
[CUgraphEdgeData](#)
[CUgraphExecUpdateResultInfo_v1](#)
[CUgraphNodeParams](#)
[CUipcEventHandle_v1](#)
[CUipcMemHandle_v1](#)
[CUlaunchAttribute](#)
[CUlaunchAttributeValue](#)
[CUlaunchConfig](#)
[CUlaunchMemSyncDomainMap](#)
[CUmемAccessDesc_v1](#)
[CUmемAllocationProp_v1](#)
[CUmемcpy3DOperand_v1](#)
[CUmемcpyAttributes_v1](#)
[CUmемDecompressParams](#)

Structure describing the parameters that compose a single decompression operation

[CUmемFabricHandle_v1](#)
[CUmемLocation_v1](#)
[CUmемPoolProps_v1](#)
[CUmемPoolPtrExportData_v1](#)
[CUmulticastObjectProp_v1](#)

[CUoffset3D_v1](#)[CUstreamBatchMemOpParams_v1](#)[CUtensorMap](#)

7.1. CUaccessPolicyWindow_v1 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`. `num_bytes` is limited by `CU_DEVICE_ATTRIBUTE_MAX_ACCESS_POLICY_WINDOW_SIZE`. Partition into many segments and assign segments such that: $\text{sum of "hit segments"} / \text{window} \approx \text{ratio}$. $\text{sum of "miss segments"} / \text{window} \approx 1 - \text{ratio}$. 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 *CUaccessPolicyWindow_v1::base_ptr`

Starting address of the access policy window. CUDA driver may align it.

`CUaccessProperty CUaccessPolicyWindow_v1::hitProp`

[CUaccessProperty](#) set for hit.

`float CUaccessPolicyWindow_v1::hitRatio`

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

`CUaccessProperty CUaccessPolicyWindow_v1::missProp`

[CUaccessProperty](#) set for miss. Must be either `NORMAL` or `STREAMING`

`size_t CUaccessPolicyWindow_v1::num_bytes`

Size in bytes of the window policy. CUDA driver may restrict the maximum size and alignment.

7.2. CUarrayMapInfo_v1 Struct Reference

Specifies the CUDA array or CUDA mipmapped array memory mapping information

unsigned int CUarrayMapInfo_v1::deviceBitMask

Device ordinal bit mask

unsigned int CUarrayMapInfo_v1::extentDepth

Depth in elements

unsigned int CUarrayMapInfo_v1::extentHeight

Height in elements

unsigned int CUarrayMapInfo_v1::extentWidth

Width in elements

unsigned int CUarrayMapInfo_v1::flags

flags for future use, must be zero now.

unsigned int CUarrayMapInfo_v1::layer

For CUDA layered arrays must be a valid layer index. Otherwise, must be zero

unsigned int CUarrayMapInfo_v1::level

For CUDA mipmapped arrays must a valid mipmap level. For CUDA arrays must be zero

CUmemHandleType

CUarrayMapInfo_v1::memHandleType

Memory handle type

CUmemOperationType

CUarrayMapInfo_v1::memOperationType

Memory operation type

unsigned long long CUarrayMapInfo_v1::offset

Offset within mip tail

Offset within the memory

unsigned int CUarrayMapInfo_v1::offsetX

Starting X offset in elements

unsigned int CUarrayMapInfo_v1::offsetY

Starting Y offset in elements

unsigned int CUarrayMapInfo_v1::offsetZ

Starting Z offset in elements

unsigned int CUarrayMapInfo_v1::reserved

Reserved for future use, must be zero now.

CUresourcetype CUarrayMapInfo_v1::resourceType

Resource type

unsigned long long CUarrayMapInfo_v1::size

Extent in bytes

CUarraySparseSubresourceType

CUarrayMapInfo_v1::subresourceType

Sparse subresource type

7.3. CUasyncNotificationInfo Struct Reference

Information passed to the user via the async notification callback

7.4. CUcheckpointCheckpointArgs Struct Reference

CUDA checkpoint optional checkpoint arguments

`cuint64_t CUcheckpointCheckpointArgs::reserved`

Reserved for future use, must be zeroed

7.5. CUcheckpointLockArgs Struct Reference

CUDA checkpoint optional lock arguments

`unsigned int CUcheckpointLockArgs::reserved0`

Reserved for future use, must be zero

`cuint64_t CUcheckpointLockArgs::reserved1`

Reserved for future use, must be zeroed

`unsigned int CUcheckpointLockArgs::timeoutMs`

Timeout in milliseconds to attempt to lock the process, 0 indicates no timeout

7.6. CUcheckpointRestoreArgs Struct Reference

CUDA checkpoint optional restore arguments

`cuint64_t CUcheckpointRestoreArgs::reserved`

Reserved for future use, must be zeroed

7.7. CUcheckpointUnlockArgs Struct Reference

CUDA checkpoint optional unlock arguments

`cuint64_t CUcheckpointUnlockArgs::reserved`

Reserved for future use, must be zeroed

7.8. CUctxCigParam Struct Reference

CIG Context Create Params

7.9. CUctxCreateParams Struct Reference

Params for creating CUDA context Exactly one of `execAffinityParams` and `cigParams` must be non-NULL.

7.10. CUDA_ARRAY3D_DESCRIPTOR_v2 Struct Reference

3D array descriptor

`size_t` `CUDA_ARRAY3D_DESCRIPTOR_v2::Depth`

Depth of 3D array

`unsigned int` `CUDA_ARRAY3D_DESCRIPTOR_v2::Flags`

Flags

`CUarray_format`

`CUDA_ARRAY3D_DESCRIPTOR_v2::Format`

Array format

`size_t` `CUDA_ARRAY3D_DESCRIPTOR_v2::Height`

Height of 3D array

`unsigned int`

`CUDA_ARRAY3D_DESCRIPTOR_v2::NumChannels`

Channels per array element

`size_t CUDA_ARRAY3D_DESCRIPTOR_v2::Width`

Width of 3D array

7.11. `CUDA_ARRAY_DESCRIPTOR_v2` Struct Reference

Array descriptor

`CUarray_format`

`CUDA_ARRAY_DESCRIPTOR_v2::Format`

Array format

`size_t CUDA_ARRAY_DESCRIPTOR_v2::Height`

Height of array

`unsigned int`

`CUDA_ARRAY_DESCRIPTOR_v2::NumChannels`

Channels per array element

`size_t CUDA_ARRAY_DESCRIPTOR_v2::Width`

Width of array

7.12. `CUDA_ARRAY_MEMORY_REQUIREMENTS` Struct Reference

CUDA array memory requirements

`size_t`

`CUDA_ARRAY_MEMORY_REQUIREMENTS_v1::alignment`

alignment requirement

size_t

CUDA_ARRAY_MEMORY_REQUIREMENTS_v1::size

Total required memory size

7.13. CUDA_ARRAY_SPARSE_PROPERTIES_v1 Struct Reference

CUDA array sparse properties

unsigned int

CUDA_ARRAY_SPARSE_PROPERTIES_v1::depth

Depth of sparse tile in elements

unsigned int

CUDA_ARRAY_SPARSE_PROPERTIES_v1::flags

Flags will either be zero or [CU_ARRAY_SPARSE_PROPERTIES_SINGLE_MIPTAIL](#)

unsigned int

CUDA_ARRAY_SPARSE_PROPERTIES_v1::height

Height of sparse tile in elements

unsigned int

CUDA_ARRAY_SPARSE_PROPERTIES_v1::mipTailFirstLevel

First mip level at which the mip tail begins.

unsigned long long

CUDA_ARRAY_SPARSE_PROPERTIES_v1::mipTailSize

Total size of the mip tail.

unsigned int

CUDA_ARRAY_SPARSE_PROPERTIES_v1::width

Width of sparse tile in elements

7.15. CUDA_CHILD_GRAPH_NODE_PARAMS Struct Reference

Child graph node parameters

CUgraph

CUDA_CHILD_GRAPH_NODE_PARAMS::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

7.16. CUDA_CONDITIONAL_NODE_PARAMS Struct Reference

Conditional node parameters

CUcontext

CUDA_CONDITIONAL_NODE_PARAMS::ctx

Context on which to run the node. Must match context used to create the handle and all body nodes.

CUgraphConditionalHandle

CUDA_CONDITIONAL_NODE_PARAMS::handle

Conditional node handle. Handles must be created in advance of creating the node using [cuGraphConditionalHandleCreate](#).

CUgraph

*CUDA_CONDITIONAL_NODE_PARAMS::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, memcpyes, 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 [cuStreamBeginCaptureToGraph](#).

CU_GRAPH_COND_TYPE_IF: 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.

CU_GRAPH_COND_TYPE_WHILE: phGraph_out[0] is executed as long as the condition is non-zero. CU_GRAPH_COND_TYPE_SWITCH: phGraph_out[n] is executed when the condition is equal to n. If the condition >= size, no body graph is executed.

unsigned int

CUDA_CONDITIONAL_NODE_PARAMS::size

Size of graph output array. Allowed values are 1 for CU_GRAPH_COND_TYPE_WHILE, 1 or 2 for CU_GRAPH_COND_TYPE_IF, or any value greater than zero for CU_GRAPH_COND_TYPE_SWITCH.

CUgraphConditionalNodeType

CUDA_CONDITIONAL_NODE_PARAMS::type

Type of conditional node.

7.17. CUDA_EVENT_RECORD_NODE_PARAMS Struct Reference

Event record node parameters

CUevent

CUDA_EVENT_RECORD_NODE_PARAMS::event

The event to record when the node executes

7.18. CUDA_EVENT_WAIT_NODE_PARAMS Struct Reference

Event wait node parameters

CUevent

CUDA_EVENT_WAIT_NODE_PARAMS::event

The event to wait on from the node

7.19. CUDA_EXT_SEM_SIGNAL_NODE_PARAMS Struct Reference

Semaphore signal node parameters

CUexternalSemaphore

*CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v1::extSemArray

Array of external semaphore handles.

unsigned int

CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v1::numExtSems

Number of handles and parameters supplied in extSemArray and paramsArray.

const

CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS

*CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v1::paramsArray

Array of external semaphore signal parameters.

7.20. CUDA_EXT_SEM_SIGNAL_NODE_PARAMS Struct Reference

Semaphore signal node parameters

CUexternalSemaphore

***CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v2::extSemArray**

Array of external semaphore handles.

unsigned int

CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v2::numExtSems

Number of handles and parameters supplied in extSemArray and paramsArray.

const

CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS

***CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v2::paramsArray**

Array of external semaphore signal parameters.

7.21. CUDA_EXT_SEM_WAIT_NODE_PARAMS_v Struct Reference

Semaphore wait node parameters

CUexternalSemaphore

***CUDA_EXT_SEM_WAIT_NODE_PARAMS_v1::extSemArray**

Array of external semaphore handles.

unsigned int

CUDA_EXT_SEM_WAIT_NODE_PARAMS_v1::numExtSems

Number of handles and parameters supplied in extSemArray and paramsArray.

```
const
CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS
*CUDA_EXT_SEM_WAIT_NODE_PARAMS_v1::paramsArray
```

Array of external semaphore wait parameters.

7.22. CUDA_EXT_SEM_WAIT_NODE_PARAMS_v1 Struct Reference

Semaphore wait node parameters

```
CUexternalSemaphore
*CUDA_EXT_SEM_WAIT_NODE_PARAMS_v2::extSemArray
```

Array of external semaphore handles.

```
unsigned int
CUDA_EXT_SEM_WAIT_NODE_PARAMS_v2::numExtSems
```

Number of handles and parameters supplied in extSemArray and paramsArray.

```
const
CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS
*CUDA_EXT_SEM_WAIT_NODE_PARAMS_v2::paramsArray
```

Array of external semaphore wait parameters.

7.23. CUDA_EXTERNAL_MEMORY_BUFFER_DESCRIPTOR Struct Reference

External memory buffer descriptor

unsigned int

CUDA_EXTERNAL_MEMORY_BUFFER_DESC_v1::flags

Flags reserved for future use. Must be zero.

unsigned long long

CUDA_EXTERNAL_MEMORY_BUFFER_DESC_v1::offset

Offset into the memory object where the buffer's base is

unsigned long long

CUDA_EXTERNAL_MEMORY_BUFFER_DESC_v1::size

Size of the buffer

7.24. CUDA_EXTERNAL_MEMORY_HANDLE_DESC Struct Reference

External memory handle descriptor

int

CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1::fd

File descriptor referencing the memory object. Valid when type is [CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD](#)

unsigned int

CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1::flags

Flags must either be zero or [CUDA_EXTERNAL_MEMORY_DEDICATED](#)

void

*CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1::handle

Valid NT handle. Must be NULL if 'name' is non-NULL

const void

*CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1::name

Name of a valid memory object. Must be NULL if 'handle' is non-NULL.

const void

*CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1::nvSciBuf

A handle representing an NvSciBuf Object. Valid when type is

[CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF](#)

unsigned long long

CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1::size

Size of the memory allocation

CUexternalMemoryHandleType

CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1::type

Type of the handle

CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1::@19::@20

CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1::win32

Win32 handle referencing the semaphore object. Valid when type is one of the following:

- ▶ [CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32](#)
- ▶ [CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT](#)
- ▶ [CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP](#)
- ▶ [CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE](#)
- ▶ [CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE](#)
- ▶ [CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE_KMT](#)

Exactly one of 'handle' and 'name' must be non-NULL. If type is one of the following:

[CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT](#)

[CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE_KMT](#) then 'name' must be NULL.

7.25. CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESCRIPTOR Struct Reference

External memory mipmap descriptor

```
struct CUDA_ARRAY3D_DESCRIPTOR
CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESCRIPTOR_v1
```

Format, dimension and type of base level of the mipmap chain

```
unsigned int
CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESCRIPTOR_v1::format
```

Total number of levels in the mipmap chain

```
unsigned long long
CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESCRIPTOR_v1::num_levels
```

Offset into the memory object where the base level of the mipmap chain is.

7.26. CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESCRIPTOR Struct Reference

External semaphore handle descriptor

```
int
CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESCRIPTOR_v1::fd
```

File descriptor referencing the semaphore object. Valid when type is one of the following:

- ▶ [CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD](#)
- ▶ [CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_TIMELINE_SEMAPHORE_FD](#)

```
unsigned int
CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESCRIPTOR_v1::flags
```

Flags reserved for the future. Must be zero.

void

*CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_v1::handle

Valid NT handle. Must be NULL if 'name' is non-NULL

const void

*CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_v1::name

Name of a valid synchronization primitive. Must be NULL if 'handle' is non-NULL.

const void

*CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_v1::nvSciS

Valid NvSciSyncObj. Must be non NULL

CUexternalSemaphoreHandleType

CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_v1::type

Type of the handle

CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_v1::@21::@

CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_v1::win32

Win32 handle referencing the semaphore object. Valid when type is one of the following:

- ▶ CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32
- ▶ CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT
- ▶ CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE
- ▶ CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE
- ▶ CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX
- ▶ CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_TIMELINE_SEMAPHORE_WIN32 Exactly one of 'handle' and 'name' must be non-NULL. If type is one of the following:
- ▶ CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT
- ▶ CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX_KMT then 'name' must be NULL.

7.27. CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1 Struct Reference

External semaphore signal parameters

void

*CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1::fence

Pointer to NvSciSyncFence. Valid if [CUexternalSemaphoreHandleType](#) is of type [CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC](#).

CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1::@23

CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1::fence

Parameters for fence objects

unsigned int

CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1::flags

Only when [CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS](#) is used to signal a [CUexternalSemaphore](#) of type [CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC](#), the valid flag is [CUDA_EXTERNAL_SEMAPHORE_SIGNAL_SKIP_NVSCIBUF_MEMSYNC](#) which indicates that while signaling the [CUexternalSemaphore](#), no memory synchronization operations should be performed for any external memory object imported as [CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF](#). For all other types of [CUexternalSemaphore](#), flags must be zero.

unsigned long long

CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1::key

Value of key to release the mutex with

CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1::@23

CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1::key

Parameters for keyed mutex objects

unsigned long long

CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1::value

Value of fence to be signaled

7.28. CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS Struct Reference

External semaphore wait parameters

CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1::@27::@

CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1::fence

Parameters for fence objects

unsigned int

CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1::flags

Only when CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS is used to wait on a [CUexternalSemaphore](#) of type [CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC](#), the valid flag is [CUDA_EXTERNAL_SEMAPHORE_WAIT_SKIP_NVSCIBUF_MEMSYNC](#) which indicates that while waiting for the [CUexternalSemaphore](#), no memory synchronization operations should be performed for any external memory object imported as [CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF](#). For all other types of [CUexternalSemaphore](#), flags must be zero.

unsigned long long

CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1::key

Value of key to acquire the mutex with

CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1::@27::@

CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1::keyedM

Parameters for keyed mutex objects

`CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1::@27::@`
`CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1::nvSciSyncFence`

Pointer to `NvSciSyncFence`. Valid if `CUexternalSemaphoreHandleType` is of type `CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC`.

unsigned int

`CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1::timeout`

Timeout in milliseconds to wait to acquire the mutex

unsigned long long

`CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1::value`

Value of fence to be waited on

7.29. `CUDA_GRAPH_INSTANTIATE_PARAMS` Struct Reference

Graph instantiation parameters

`cuuint64_t`

`CUDA_GRAPH_INSTANTIATE_PARAMS::flags`

Instantiation flags

`CUgraphNode`

`CUDA_GRAPH_INSTANTIATE_PARAMS::hErrNode_out`

The node which caused instantiation to fail, if any

`CUstream`

`CUDA_GRAPH_INSTANTIATE_PARAMS::hUploadStream`

Upload stream

`CUgraphInstantiateResult`

`CUDA_GRAPH_INSTANTIATE_PARAMS::result_out`

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

7.30. `CUDA_HOST_NODE_PARAMS_v1` Struct Reference

Host node parameters

`CUhostFn CUDA_HOST_NODE_PARAMS_v1::fn`

The function to call when the node executes

`void *CUDA_HOST_NODE_PARAMS_v1::userData`

Argument to pass to the function

7.31. `CUDA_HOST_NODE_PARAMS_v2` Struct Reference

Host node parameters

`CUhostFn CUDA_HOST_NODE_PARAMS_v2::fn`

The function to call when the node executes

`void *CUDA_HOST_NODE_PARAMS_v2::userData`

Argument to pass to the function

7.32. `CUDA_KERNEL_NODE_PARAMS_v1` Struct Reference

GPU kernel node parameters

unsigned int

CUDA_KERNEL_NODE_PARAMS_v1::blockDimX

X dimension of each thread block

unsigned int

CUDA_KERNEL_NODE_PARAMS_v1::blockDimY

Y dimension of each thread block

unsigned int

CUDA_KERNEL_NODE_PARAMS_v1::blockDimZ

Z dimension of each thread block

**CUDA_KERNEL_NODE_PARAMS_v1::extra

Extra options

CUfunction CUDA_KERNEL_NODE_PARAMS_v1::func

Kernel to launch

unsigned int

CUDA_KERNEL_NODE_PARAMS_v1::gridDimX

Width of grid in blocks

unsigned int

CUDA_KERNEL_NODE_PARAMS_v1::gridDimY

Height of grid in blocks

unsigned int

CUDA_KERNEL_NODE_PARAMS_v1::gridDimZ

Depth of grid in blocks

****CUDA_KERNEL_NODE_PARAMS_v1::kernelParams**

Array of pointers to kernel parameters

unsigned int

CUDA_KERNEL_NODE_PARAMS_v1::sharedMemBytes

Dynamic shared-memory size per thread block in bytes

7.33. CUDA_KERNEL_NODE_PARAMS_v2 Struct Reference

GPU kernel node parameters

unsigned int

CUDA_KERNEL_NODE_PARAMS_v2::blockDimX

X dimension of each thread block

unsigned int

CUDA_KERNEL_NODE_PARAMS_v2::blockDimY

Y dimension of each thread block

unsigned int

CUDA_KERNEL_NODE_PARAMS_v2::blockDimZ

Z dimension of each thread block

CUcontext CUDA_KERNEL_NODE_PARAMS_v2::ctx

Context for the kernel task to run in. The value NULL will indicate the current context should be used by the api. This field is ignored if func is set.

****CUDA_KERNEL_NODE_PARAMS_v2::extra**

Extra options

CUfunction CUDA_KERNEL_NODE_PARAMS_v2::func

Kernel to launch

unsigned int

CUDA_KERNEL_NODE_PARAMS_v2::gridDimX

Width of grid in blocks

unsigned int

CUDA_KERNEL_NODE_PARAMS_v2::gridDimY

Height of grid in blocks

unsigned int

CUDA_KERNEL_NODE_PARAMS_v2::gridDimZ

Depth of grid in blocks

CUkernel CUDA_KERNEL_NODE_PARAMS_v2::kern

Kernel to launch, will only be referenced if func is NULL

****CUDA_KERNEL_NODE_PARAMS_v2::kernelParams**

Array of pointers to kernel parameters

unsigned int

CUDA_KERNEL_NODE_PARAMS_v2::sharedMemBytes

Dynamic shared-memory size per thread block in bytes

7.34. CUDA_KERNEL_NODE_PARAMS_v3 Struct Reference

GPU kernel node parameters

unsigned int

CUDA_KERNEL_NODE_PARAMS_v3::blockDimX

X dimension of each thread block

unsigned int

CUDA_KERNEL_NODE_PARAMS_v3::blockDimY

Y dimension of each thread block

unsigned int

CUDA_KERNEL_NODE_PARAMS_v3::blockDimZ

Z dimension of each thread block

CUcontext CUDA_KERNEL_NODE_PARAMS_v3::ctx

Context for the kernel task to run in. The value NULL will indicate the current context should be used by the api. This field is ignored if func is set.

**CUDA_KERNEL_NODE_PARAMS_v3::extra

Extra options

CUfunction CUDA_KERNEL_NODE_PARAMS_v3::func

Kernel to launch

unsigned int

CUDA_KERNEL_NODE_PARAMS_v3::gridDimX

Width of grid in blocks

unsigned int

CUDA_KERNEL_NODE_PARAMS_v3::gridDimY

Height of grid in blocks

unsigned int

CUDA_KERNEL_NODE_PARAMS_v3::gridDimZ

Depth of grid in blocks

CUkernel CUDA_KERNEL_NODE_PARAMS_v3::kern

Kernel to launch, will only be referenced if func is NULL

**CUDA_KERNEL_NODE_PARAMS_v3::kernelParams

Array of pointers to kernel parameters

unsigned int

CUDA_KERNEL_NODE_PARAMS_v3::sharedMemBytes

Dynamic shared-memory size per thread block in bytes

7.35. CUDA_LAUNCH_PARAMS_v1 Struct Reference

Kernel launch parameters

unsigned int CUDA_LAUNCH_PARAMS_v1::blockDimX

X dimension of each thread block

unsigned int CUDA_LAUNCH_PARAMS_v1::blockDimY

Y dimension of each thread block

unsigned int CUDA_LAUNCH_PARAMS_v1::blockDimZ

Z dimension of each thread block

CUfunction CUDA_LAUNCH_PARAMS_v1::function

Kernel to launch

`unsigned int CUDA_LAUNCH_PARAMS_v1::gridDimX`

Width of grid in blocks

`unsigned int CUDA_LAUNCH_PARAMS_v1::gridDimY`

Height of grid in blocks

`unsigned int CUDA_LAUNCH_PARAMS_v1::gridDimZ`

Depth of grid in blocks

`CUstream CUDA_LAUNCH_PARAMS_v1::hStream`

Stream identifier

`**CUDA_LAUNCH_PARAMS_v1::kernelParams`

Array of pointers to kernel parameters

`unsigned int`

`CUDA_LAUNCH_PARAMS_v1::sharedMemBytes`

Dynamic shared-memory size per thread block in bytes

7.36. `CUDA_MEM_ALLOC_NODE_PARAMS_v1` Struct Reference

Memory allocation node parameters

`size_t`

`CUDA_MEM_ALLOC_NODE_PARAMS_v1::accessDescCount`

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

`const CUmemAccessDesc`

`*CUDA_MEM_ALLOC_NODE_PARAMS_v1::accessDescs`

in: array of memory access descriptors. Used to describe peer GPU access

`size_t`

`CUDA_MEM_ALLOC_NODE_PARAMS_v1::bytesize`

in: size in bytes of the requested allocation

`CUdeviceptr`

`CUDA_MEM_ALLOC_NODE_PARAMS_v1::dptr`

out: address of the allocation returned by CUDA

`struct CUmemPoolProps`

`CUDA_MEM_ALLOC_NODE_PARAMS_v1::poolProps`

in: location where the allocation should reside (specified in location). handleTypes must be `CU_MEM_HANDLE_TYPE_NONE`. IPC is not supported.

7.37. `CUDA_MEM_ALLOC_NODE_PARAMS_v2` Struct Reference

Memory allocation node parameters

`size_t`

`CUDA_MEM_ALLOC_NODE_PARAMS_v2::accessDescCount`

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

`const CUmemAccessDesc`

`*CUDA_MEM_ALLOC_NODE_PARAMS_v2::accessDescs`

in: array of memory access descriptors. Used to describe peer GPU access

`size_t`

`CUDA_MEM_ALLOC_NODE_PARAMS_v2::bytesize`

in: size in bytes of the requested allocation

CUdeviceptr

CUDA_MEM_ALLOC_NODE_PARAMS_v2::dptr

out: address of the allocation returned by CUDA

struct CUmemPoolProps

CUDA_MEM_ALLOC_NODE_PARAMS_v2::poolProps

in: location where the allocation should reside (specified in location). handleTypes must be [CU_MEM_HANDLE_TYPE_NONE](#). IPC is not supported.

7.38. CUDA_MEM_FREE_NODE_PARAMS Struct Reference

Memory free node parameters

CUdeviceptr

CUDA_MEM_FREE_NODE_PARAMS::dptr

in: the pointer to free

7.39. CUDA_MEMCPY2D_v2 Struct Reference

2D memory copy parameters

CUarray CUDA_MEMCPY2D_v2::dstArray

Destination array reference

CUdeviceptr CUDA_MEMCPY2D_v2::dstDevice

Destination device pointer

void *CUDA_MEMCPY2D_v2::dstHost

Destination host pointer

CUmemorytype

CUDA_MEMCPY2D_v2::dstMemoryType

Destination memory type (host, device, array)

size_t CUDA_MEMCPY2D_v2::dstPitch

Destination pitch (ignored when dst is array)

size_t CUDA_MEMCPY2D_v2::dstXInBytes

Destination X in bytes

size_t CUDA_MEMCPY2D_v2::dstY

Destination Y

size_t CUDA_MEMCPY2D_v2::Height

Height of 2D memory copy

CUarray CUDA_MEMCPY2D_v2::srcArray

Source array reference

CUdeviceptr CUDA_MEMCPY2D_v2::srcDevice

Source device pointer

const void *CUDA_MEMCPY2D_v2::srcHost

Source host pointer

CUmemorytype

CUDA_MEMCPY2D_v2::srcMemoryType

Source memory type (host, device, array)

size_t CUDA_MEMCPY2D_v2::srcPitch

Source pitch (ignored when src is array)

`size_t CUDA_MEMCPY2D_v2::srcXInBytes`

Source X in bytes

`size_t CUDA_MEMCPY2D_v2::srcY`

Source Y

`size_t CUDA_MEMCPY2D_v2::WidthInBytes`

Width of 2D memory copy in bytes

7.40. `CUDA_MEMCPY3D_PEER_v1` Struct Reference

3D memory cross-context copy parameters

`size_t CUDA_MEMCPY3D_PEER_v1::Depth`

Depth of 3D memory copy

`CUarray CUDA_MEMCPY3D_PEER_v1::dstArray`

Destination array reference

`CUcontext CUDA_MEMCPY3D_PEER_v1::dstContext`

Destination context (ignored with `dstMemoryType` is `CU_MEMORYTYPE_ARRAY`)

`CUdeviceptr CUDA_MEMCPY3D_PEER_v1::dstDevice`

Destination device pointer

`size_t CUDA_MEMCPY3D_PEER_v1::dstHeight`

Destination height (ignored when `dst` is array; may be 0 if `Depth==1`)

`void *CUDA_MEMCPY3D_PEER_v1::dstHost`

Destination host pointer

`size_t CUDA_MEMCPY3D_PEER_v1::dstLOD`

Destination LOD

`CUmemorytype`

`CUDA_MEMCPY3D_PEER_v1::dstMemoryType`

Destination memory type (host, device, array)

`size_t CUDA_MEMCPY3D_PEER_v1::dstPitch`

Destination pitch (ignored when dst is array)

`size_t CUDA_MEMCPY3D_PEER_v1::dstXInBytes`

Destination X in bytes

`size_t CUDA_MEMCPY3D_PEER_v1::dstY`

Destination Y

`size_t CUDA_MEMCPY3D_PEER_v1::dstZ`

Destination Z

`size_t CUDA_MEMCPY3D_PEER_v1::Height`

Height of 3D memory copy

`CUarray CUDA_MEMCPY3D_PEER_v1::srcArray`

Source array reference

`CUcontext CUDA_MEMCPY3D_PEER_v1::srcContext`

Source context (ignored with srcMemoryType is [CU_MEMORYTYPE_ARRAY](#))

`CUdeviceptr CUDA_MEMCPY3D_PEER_v1::srcDevice`

Source device pointer

`size_t CUDA_MEMCPY3D_PEER_v1::srcHeight`

Source height (ignored when src is array; may be 0 if Depth==1)

`const void *CUDA_MEMCPY3D_PEER_v1::srcHost`

Source host pointer

`size_t CUDA_MEMCPY3D_PEER_v1::srcLOD`

Source LOD

`CUmemorytype`

`CUDA_MEMCPY3D_PEER_v1::srcMemoryType`

Source memory type (host, device, array)

`size_t CUDA_MEMCPY3D_PEER_v1::srcPitch`

Source pitch (ignored when src is array)

`size_t CUDA_MEMCPY3D_PEER_v1::srcXInBytes`

Source X in bytes

`size_t CUDA_MEMCPY3D_PEER_v1::srcY`

Source Y

`size_t CUDA_MEMCPY3D_PEER_v1::srcZ`

Source Z

`size_t CUDA_MEMCPY3D_PEER_v1::WidthInBytes`

Width of 3D memory copy in bytes

7.41. CUDA_MEMCPY3D_v2 Struct Reference

3D memory copy parameters

`size_t CUDA_MEMCPY3D_v2::Depth`

Depth of 3D memory copy

`CUarray CUDA_MEMCPY3D_v2::dstArray`

Destination array reference

`CUdeviceptr CUDA_MEMCPY3D_v2::dstDevice`

Destination device pointer

`size_t CUDA_MEMCPY3D_v2::dstHeight`

Destination height (ignored when dst is array; may be 0 if Depth==1)

`void *CUDA_MEMCPY3D_v2::dstHost`

Destination host pointer

`size_t CUDA_MEMCPY3D_v2::dstLOD`

Destination LOD

`CUmemorytype`

`CUDA_MEMCPY3D_v2::dstMemoryType`

Destination memory type (host, device, array)

`size_t CUDA_MEMCPY3D_v2::dstPitch`

Destination pitch (ignored when dst is array)

`size_t CUDA_MEMCPY3D_v2::dstXInBytes`

Destination X in bytes

`size_t CUDA_MEMCPY3D_v2::dstY`

Destination Y

`size_t CUDA_MEMCPY3D_v2::dstZ`

Destination Z

`size_t CUDA_MEMCPY3D_v2::Height`

Height of 3D memory copy

`void *CUDA_MEMCPY3D_v2::reserved0`

Must be NULL

`void *CUDA_MEMCPY3D_v2::reserved1`

Must be NULL

`CUarray CUDA_MEMCPY3D_v2::srcArray`

Source array reference

`CUdeviceptr CUDA_MEMCPY3D_v2::srcDevice`

Source device pointer

`size_t CUDA_MEMCPY3D_v2::srcHeight`

Source height (ignored when src is array; may be 0 if Depth==1)

`const void *CUDA_MEMCPY3D_v2::srcHost`

Source host pointer

`size_t CUDA_MEMCPY3D_v2::srcLOD`

Source LOD

`CUmemorytype`

`CUDA_MEMCPY3D_v2::srcMemoryType`

Source memory type (host, device, array)

`size_t CUDA_MEMCPY3D_v2::srcPitch`

Source pitch (ignored when src is array)

`size_t CUDA_MEMCPY3D_v2::srcXInBytes`

Source X in bytes

`size_t CUDA_MEMCPY3D_v2::srcY`

Source Y

`size_t CUDA_MEMCPY3D_v2::srcZ`

Source Z

`size_t CUDA_MEMCPY3D_v2::WidthInBytes`

Width of 3D memory copy in bytes

7.42. `CUDA_MEMCPY_NODE_PARAMS` Struct Reference

Memcpy node parameters

`CUcontext`

`CUDA_MEMCPY_NODE_PARAMS::copyCtx`

Context on which to run the node

`struct CUDA_MEMCPY3D`

`CUDA_MEMCPY_NODE_PARAMS::copyParams`

Parameters for the memory copy

`int CUDA_MEMCPY_NODE_PARAMS::flags`

Must be zero

`int CUDA_MEMCPY_NODE_PARAMS::reserved`

Must be zero

7.43. `CUDA_MEMSET_NODE_PARAMS_v1` Struct Reference

Memset node parameters

`CUdeviceptr CUDA_MEMSET_NODE_PARAMS_v1::dst`

Destination device pointer

`unsigned int`

`CUDA_MEMSET_NODE_PARAMS_v1::elementSize`

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

`size_t CUDA_MEMSET_NODE_PARAMS_v1::height`

Number of rows

`size_t CUDA_MEMSET_NODE_PARAMS_v1::pitch`

Pitch of destination device pointer. Unused if height is 1

`unsigned int`

`CUDA_MEMSET_NODE_PARAMS_v1::value`

Value to be set

`size_t CUDA_MEMSET_NODE_PARAMS_v1::width`

Width of the row in elements

7.44. CUDA_MEMSET_NODE_PARAMS_v2 Struct Reference

Memset node parameters

`CUcontext CUDA_MEMSET_NODE_PARAMS_v2::ctx`

Context on which to run the node

`CUdeviceptr CUDA_MEMSET_NODE_PARAMS_v2::dst`

Destination device pointer

`unsigned int`

`CUDA_MEMSET_NODE_PARAMS_v2::elementSize`

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

`size_t CUDA_MEMSET_NODE_PARAMS_v2::height`

Number of rows

`size_t CUDA_MEMSET_NODE_PARAMS_v2::pitch`

Pitch of destination device pointer. Unused if height is 1

`unsigned int`

`CUDA_MEMSET_NODE_PARAMS_v2::value`

Value to be set

`size_t CUDA_MEMSET_NODE_PARAMS_v2::width`

Width of the row in elements

7.45. `CUDA_POINTER_ATTRIBUTE_P2P_TOKENS` Struct Reference

GPU Direct v3 tokens

7.46. `CUDA_RESOURCE_DESC_v1` Struct Reference

CUDA Resource descriptor

`CUdeviceptr` `CUDA_RESOURCE_DESC_v1::devPtr`

Device pointer

`unsigned int` `CUDA_RESOURCE_DESC_v1::flags`

Flags (must be zero)

`CUarray_format` `CUDA_RESOURCE_DESC_v1::format`

Array format

`CUarray` `CUDA_RESOURCE_DESC_v1::hArray`

CUDA array

`size_t` `CUDA_RESOURCE_DESC_v1::height`

Height of the array in elements

`CUmipmappedArray`

`CUDA_RESOURCE_DESC_v1::hMipmappedArray`

CUDA mipmapped array

unsigned int

CUDA_RESOURCE_DESC_v1::numChannels

Channels per array element

size_t CUDA_RESOURCE_DESC_v1::pitchInBytes

Pitch between two rows in bytes

CUresourcetype CUDA_RESOURCE_DESC_v1::resType

Resource type

size_t CUDA_RESOURCE_DESC_v1::sizeInBytes

Size in bytes

size_t CUDA_RESOURCE_DESC_v1::width

Width of the array in elements

7.47. CUDA_RESOURCE_VIEW_DESC_v1 Struct Reference

Resource view descriptor

size_t CUDA_RESOURCE_VIEW_DESC_v1::depth

Depth of the resource view

unsigned int

CUDA_RESOURCE_VIEW_DESC_v1::firstLayer

First layer index

unsigned int

CUDA_RESOURCE_VIEW_DESC_v1::firstMipmapLevel

First defined mipmap level

CUresourceViewFormat

CUDA_RESOURCE_VIEW_DESC_v1::format

Resource view format

size_t CUDA_RESOURCE_VIEW_DESC_v1::height

Height of the resource view

unsigned int

CUDA_RESOURCE_VIEW_DESC_v1::lastLayer

Last layer index

unsigned int

CUDA_RESOURCE_VIEW_DESC_v1::lastMipmapLevel

Last defined mipmap level

size_t CUDA_RESOURCE_VIEW_DESC_v1::width

Width of the resource view

7.48. CUDA_TEXTURE_DESC_v1 Struct Reference

Texture descriptor

CUaddress_mode

CUDA_TEXTURE_DESC_v1::addressMode

Address modes

float CUDA_TEXTURE_DESC_v1::borderColor

Border Color

`CUfilter_mode CUDA_TEXTURE_DESC_v1::filterMode`

Filter mode

`unsigned int CUDA_TEXTURE_DESC_v1::flags`

Flags

`unsigned int
CUDA_TEXTURE_DESC_v1::maxAnisotropy`

Maximum anisotropy ratio

`float
CUDA_TEXTURE_DESC_v1::maxMipmapLevelClamp`

Mipmap maximum level clamp

`float
CUDA_TEXTURE_DESC_v1::minMipmapLevelClamp`

Mipmap minimum level clamp

`CUfilter_mode
CUDA_TEXTURE_DESC_v1::mipmapFilterMode`

Mipmap filter mode

`float CUDA_TEXTURE_DESC_v1::mipmapLevelBias`

Mipmap level bias

7.49. CUdevprop_v1 Struct Reference

Legacy device properties

`int CUdevprop_v1::clockRate`

Clock frequency in kilohertz

int CUdevprop_v1::maxGridSize

Maximum size of each dimension of a grid

int CUdevprop_v1::maxThreadsDim

Maximum size of each dimension of a block

int CUdevprop_v1::maxThreadsPerBlock

Maximum number of threads per block

int CUdevprop_v1::memPitch

Maximum pitch in bytes allowed by memory copies

int CUdevprop_v1::regsPerBlock

32-bit registers available per block

int CUdevprop_v1::sharedMemPerBlock

Shared memory available per block in bytes

int CUdevprop_v1::SIMDWidth

Warp size in threads

int CUdevprop_v1::textureAlign

Alignment requirement for textures

int CUdevprop_v1::totalConstantMemory

Constant memory available on device in bytes

7.50. CUdevResource Struct Reference

A tagged union describing different resources identified by the type field. This structure should not be directly modified outside of the API that created it.

```

struct {
    CUdevResourceType type;
    union {

```

```

        CUdevSmResource sm;
    };
};

```

- ▶ If `type` is `CU_DEV_RESOURCE_TYPE_INVALID`, this resource is not valid and cannot be further accessed.
- ▶ If `type` is `CU_DEV_RESOURCE_TYPE_SM`, the `CUdevSmResource` structure `sm` is filled in. For example, `sm.smCount` will reflect the amount of streaming multiprocessors available in this resource.

7.51. CUdevSmResource Struct Reference

Data for SM-related resources

unsigned int CUdevSmResource::smCount

The amount of streaming multiprocessors available in this resource. This is an output parameter only, do not write to this field.

7.52. CUeglFrame_v1 Struct Reference

CUDA EGLFrame structure Descriptor - structure defining one frame of EGL.

Each frame may contain one or more planes depending on whether the surface * is Multiplanar or not.

CUarray_format CUeglFrame_v1::cuFormat

CUDA Array Format

unsigned int CUeglFrame_v1::depth

Depth of first plane

CUeglColorFormat CUeglFrame_v1::eglColorFormat

CUDA EGL Color Format

CUeglFrameType CUeglFrame_v1::frameType

Array or Pitch

unsigned int CUeglFrame_v1::height

Height of first plane

unsigned int CUeglFrame_v1::numChannels

Number of channels for the plane

CUarray CUeglFrame_v1::pArray

Array of CUarray corresponding to each plane

unsigned int CUeglFrame_v1::pitch

Pitch of first plane

unsigned int CUeglFrame_v1::planeCount

Number of planes

void *CUeglFrame_v1::pPitch

Array of Pointers corresponding to each plane

unsigned int CUeglFrame_v1::width

Width of first plane

7.53. CUexecAffinityParam_v1 Struct Reference

Execution Affinity Parameters

7.54. CUexecAffinitySmCount_v1 Struct Reference

Value for [CU_EXEC_AFFINITY_TYPE_SM_COUNT](#)

unsigned int CUexecAffinitySmCount_v1::val

The number of SMs the context is limited to use.

7.55. CUextent3D_v1 Struct Reference

Struct representing width/height/depth of a CUarray in elements

7.56. CUgraphEdgeData 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 CUgraphEdgeData::from_port

This indicates when the dependency is triggered from the upstream node on the edge. The meaning is specific 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: [CU_GRAPH_KERNEL_NODE_PORT_DEFAULT](#), [CU_GRAPH_KERNEL_NODE_PORT_PROGRAMMATIC](#), or [CU_GRAPH_KERNEL_NODE_PORT_LAUNCH_ORDER](#).

unsigned char CUgraphEdgeData::reserved

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

unsigned char CUgraphEdgeData::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 CUgraphEdgeData::type

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

7.57. CUgraphExecUpdateResultInfo_v1 Struct Reference

Result information returned by cuGraphExecUpdate

CUgraphNode

CUgraphExecUpdateResultInfo_v1::errorFromNode

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

CUgraphNode

CUgraphExecUpdateResultInfo_v1::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.

CUgraphExecUpdateResult

CUgraphExecUpdateResultInfo_v1::result

Gives more specific detail when a cuda graph update fails.

7.58. CUgraphNodeParams Struct Reference

Graph node parameters. See [cuGraphAddNode](#).

struct CUDA_MEM_ALLOC_NODE_PARAMS_v2

CUgraphNodeParams::alloc

Memory allocation node parameters.

struct CUDA_CONDITIONAL_NODE_PARAMS

CUgraphNodeParams::conditional

Conditional node parameters.

```
struct CUDA_EVENT_RECORD_NODE_PARAMS  
CUgraphNodeParams::eventRecord
```

Event record node parameters.

```
struct CUDA_EVENT_WAIT_NODE_PARAMS  
CUgraphNodeParams::eventWait
```

Event wait node parameters.

```
struct CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v2  
CUgraphNodeParams::extSemSignal
```

External semaphore signal node parameters.

```
struct CUDA_EXT_SEM_WAIT_NODE_PARAMS_v2  
CUgraphNodeParams::extSemWait
```

External semaphore wait node parameters.

```
struct CUDA_MEM_FREE_NODE_PARAMS  
CUgraphNodeParams::free
```

Memory free node parameters.

```
struct CUDA_CHILD_GRAPH_NODE_PARAMS  
CUgraphNodeParams::graph
```

Child graph node parameters.

```
struct CUDA_HOST_NODE_PARAMS_v2  
CUgraphNodeParams::host
```

Host node parameters.

```
struct CUDA_KERNEL_NODE_PARAMS_v3  
CUgraphNodeParams::kernel
```

Kernel node parameters.

```
struct CUDA_MEMCPY_NODE_PARAMS
CUgraphNodeParams::memcpy
```

Memcpy node parameters.

```
struct CUDA_BATCH_MEM_OP_NODE_PARAMS_v2
CUgraphNodeParams::memOp
```

MemOp node parameters.

```
struct CUDA_MEMSET_NODE_PARAMS_v2
CUgraphNodeParams::memset
```

Memset node parameters.

```
int CUgraphNodeParams::reserved0
```

Reserved. Must be zero.

```
long long CUgraphNodeParams::reserved1
```

Padding. Unused bytes must be zero.

```
long long CUgraphNodeParams::reserved2
```

Reserved bytes. Must be zero.

```
CUgraphNodeType CUgraphNodeParams::type
```

Type of the node

7.59. CUipcEventHandle_v1 Struct Reference

CUDA IPC event handle

7.60. CUipcMemHandle_v1 Struct Reference

CUDA IPC mem handle

7.61. CUlaunchAttribute Struct Reference

Launch attribute

CUlaunchAttributeID CUlaunchAttribute::id

Attribute to set

CUlaunchAttribute::value

Value of the attribute

7.62. CUlaunchAttributeValue Union Reference

Launch attributes union; used as value field of [CUlaunchAttribute](#)

struct CUaccessPolicyWindow

CUlaunchAttributeValue::accessPolicyWindow

Value of launch attribute [CU_LAUNCH_ATTRIBUTE_ACCESS_POLICY_WINDOW](#).

CUlaunchAttributeValue::@6

CUlaunchAttributeValue::clusterDim

Value of launch attribute [CU_LAUNCH_ATTRIBUTE_CLUSTER_DIMENSION](#) 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.

CUclusterSchedulingPolicy

CUlaunchAttributeValue::clusterSchedulingPolicyPreference

Value of launch attribute

[CU_LAUNCH_ATTRIBUTE_CLUSTER_SCHEDULING_POLICY_PREFERENCE](#). Cluster scheduling policy preference for the kernel.

int CUlaunchAttributeValue::cooperative

Value of launch attribute [CU_LAUNCH_ATTRIBUTE_COOPERATIVE](#). Nonzero indicates a cooperative kernel (see [cuLaunchCooperativeKernel](#)).

CUlaunchAttributeValue::@10

CUlaunchAttributeValue::deviceUpdatableKernelNode

Value of launch attribute [CU_LAUNCH_ATTRIBUTE_DEVICE_UPDATABLE_KERNEL_NODE](#) with the following fields:

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

CUlaunchAttributeValue::@8

CUlaunchAttributeValue::launchCompletionEvent

Value of launch attribute [CU_LAUNCH_ATTRIBUTE_LAUNCH_COMPLETION_EVENT](#) with the following fields:

- ▶ `CUevent event` - Event to fire when the last block launches
- ▶ `int flags` - Event record flags, see [cuEventRecordWithFlags](#). Does not accept [CU_EVENT_RECORD_EXTERNAL](#).

CUlaunchMemSyncDomain

CUlaunchAttributeValue::memSyncDomain

Value of launch attribute [CU_LAUNCH_ATTRIBUTE_MEM_SYNC_DOMAIN](#). See: [CUlaunchMemSyncDomain](#)

struct CUlaunchMemSyncDomainMap

CUlaunchAttributeValue::memSyncDomainMap

Value of launch attribute [CU_LAUNCH_ATTRIBUTE_MEM_SYNC_DOMAIN_MAP](#). See [CUlaunchMemSyncDomainMap](#).

CUlaunchAttributeValue::@9

CUlaunchAttributeValue::preferredClusterDim

Value of launch attribute [CU_LAUNCH_ATTRIBUTE_PREFERRED_CLUSTER_DIMENSION](#) 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 [CUlaunchAttributeValue::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 [CUlaunchAttributeValue::clusterDim](#).
- ▶ `z` - The Z dimension of the preferred cluster, in blocks. Must be equal to the `z` field of [CUlaunchAttributeValue::clusterDim](#).

int CUlaunchAttributeValue::priority

Value of launch attribute [CU_LAUNCH_ATTRIBUTE_PRIORITY](#). Execution priority of the kernel.

CUlaunchAttributeValue::@7

CUlaunchAttributeValue::programmaticEvent

Value of launch attribute [CU_LAUNCH_ATTRIBUTE_PROGRAMMATIC_EVENT](#) with the following fields:

- ▶ `CUevent event` - Event to fire when all blocks trigger it.
- ▶ `Event record flags`, see [cuEventRecordWithFlags](#). Does not accept `:CU_EVENT_RECORD_EXTERNAL`.
- ▶ `triggerAtBlockStart` - If this is set to non-0, each block launch will automatically trigger the event.

int

CUlaunchAttributeValue::programmaticStreamSerializationAllowed

Value of launch attribute [CU_LAUNCH_ATTRIBUTE_PROGRAMMATIC_STREAM_SERIALIZATION](#).

unsigned int

CUlaunchAttributeValue::sharedMemCarveout

Value of launch attribute [CU_LAUNCH_ATTRIBUTE_PREFERRED_SHARED_MEMORY_CARVEOUT](#).

CUsynchronizationPolicy

CUlaunchAttributeValue::syncPolicy

Value of launch attribute [CU_LAUNCH_ATTRIBUTE_SYNCHRONIZATION_POLICY](#).

CUsynchronizationPolicy for work queued up in this stream

7.63. CUlaunchConfig Struct Reference

CUDA extensible launch configuration

CUlaunchAttribute *CUlaunchConfig::attrs

List of attributes; nullable if [CUlaunchConfig::numAttrs](#) == 0

unsigned int CUlaunchConfig::blockDimX

X dimension of each thread block

unsigned int CUlaunchConfig::blockDimY

Y dimension of each thread block

unsigned int CUlaunchConfig::blockDimZ

Z dimension of each thread block

unsigned int CUlaunchConfig::gridDimX

Width of grid in blocks

unsigned int CUlaunchConfig::gridDimY

Height of grid in blocks

unsigned int CUlaunchConfig::gridDimZ

Depth of grid in blocks

CUstream CUlaunchConfig::hStream

Stream identifier

unsigned int CUlaunchConfig::numAttrs

Number of attributes populated in [CUlaunchConfig::attrs](#)

unsigned int CUlaunchConfig::sharedMemBytes

Dynamic shared-memory size per thread block in bytes

7.64. CUlaunchMemSyncDomainMap Struct Reference

Memory Synchronization Domain map

See [cudaLaunchMemSyncDomain](#).

By default, kernels are launched in domain 0. Kernel launched with [CU_LAUNCH_MEM_SYNC_DOMAIN_REMOTE](#) will have a different domain ID. User may also alter the domain ID with [CUlaunchMemSyncDomainMap](#) for a specific stream / graph node / kernel launch. See [CU_LAUNCH_ATTRIBUTE_MEM_SYNC_DOMAIN_MAP](#).

Domain ID range is available through [CU_DEVICE_ATTRIBUTE_MEM_SYNC_DOMAIN_COUNT](#).

unsigned char CUlaunchMemSyncDomainMap::default_

The default domain ID to use for designated kernels

unsigned char CUlaunchMemSyncDomainMap::remote

The remote domain ID to use for designated kernels

7.65. CUmemAccessDesc_v1 Struct Reference

Memory access descriptor

CUmemAccess_flags CUmemAccessDesc_v1::flags

CUmemProt accessibility flags to set on the request

struct CUmemLocation CUmemAccessDesc_v1::location

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

7.66. CUmemAllocationProp_v1 Struct Reference

Specifies the allocation properties for a allocation.

unsigned char

CUmemAllocationProp_v1::compressionType

Allocation hint for requesting compressible memory. On devices that support Compute Data Compression, compressible memory can be used to accelerate accesses to data with unstructured sparsity and other compressible data patterns. Applications are expected to query allocation property of the handle obtained with [cuMemCreate](#) using [cuMemGetAllocationPropertiesFromHandle](#) to validate if the obtained allocation is compressible or not. Note that compressed memory may not be mappable on all devices.

struct CUmemLocation

CUmemAllocationProp_v1::location

Location of allocation

CUmemAllocationHandleType

CUmemAllocationProp_v1::requestedHandleTypes

requested [CUmemAllocationHandleType](#)

CUmemAllocationType CUmemAllocationProp_v1::type

Allocation type

unsigned short CUmemAllocationProp_v1::usage

Bitmask indicating intended usage for this allocation

void *CUmemAllocationProp_v1::win32HandleMetaData

Windows-specific `OBJECT_ATTRIBUTES` required when `CU_MEM_HANDLE_TYPE_WIN32` is specified. This object attributes structure includes security attributes that define the scope of which exported allocations may be transferred to other processes. In all other cases, this field is required to be zero.

7.67. CUmemcpy3DOperand_v1 Struct Reference

Struct representing an operand for copy with [cuMemcpy3DBatchAsync](#)

[CUmemcpy3DOperand_v1::@39::@41](#)
[CUmemcpy3DOperand_v1::array](#)

Struct representing an operand when CUmemcpy3DOperand::type is [CU_MEMCPY_OPERAND_TYPE_ARRAY](#)

[size_t CUmemcpy3DOperand_v1::layerHeight](#)

Height of each layer in elements.

[struct CUmemLocation](#)
[CUmemcpy3DOperand_v1::locHint](#)

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

[CUmemcpy3DOperand_v1::@39::@40](#)
[CUmemcpy3DOperand_v1::ptr](#)

Struct representing an operand when CUmemcpy3DOperand::type is [CU_MEMCPY_OPERAND_TYPE_POINTER](#)

[size_t CUmemcpy3DOperand_v1::rowLength](#)

Length of each row in elements.

7.68. CUmemcpyAttributes_v1 Struct Reference

Attributes specific to copies within a batch. For more details on usage see [cuMemcpyBatchAsync](#).

struct CUmemLocation CUmemcpyAttributes_v1::dstLocHint

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

unsigned int CUmemcpyAttributes_v1::flags

Additional flags for copies with this attribute. See [CUmemcpyFlags](#)

CUmemcpySrcAccessOrder CUmemcpyAttributes_v1::srcAccessOrder

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

struct CUmemLocation CUmemcpyAttributes_v1::srcLocHint

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

7.69. CUmemDecompressParams Struct Reference

Structure describing the parameters that compose a single decompression operation.

CUmemDecompressAlgorithm CUmemDecompressParams::algo

The decompression algorithm to use.

void *CUmemDecompressParams::dst

Pointer to a buffer where the decompressed data will be written. The number of bytes written to this location will be recorded in the memory pointed to by CUmemDecompressParams_st.dstActBytes

cuint32_t *CUMemDecompressParams::dstActBytes

After the decompression operation has completed, the actual number of bytes written to [CUMemDecompressParams.dst](#) will be recorded as a 32-bit unsigned integer in the memory at this address.

size_t CUMemDecompressParams::dstNumBytes

The number of bytes that the decompression operation will be expected to write to CUMemDecompressParams_st.dst. This value is optional; if present, it may be used by the CUDA driver as a heuristic for scheduling the individual decompression operations.

const void *CUMemDecompressParams::src

Pointer to a buffer of at least CUMemDecompressParams_st.srcNumBytes compressed bytes.

size_t CUMemDecompressParams::srcNumBytes

The number of bytes to be read and decompressed from CUMemDecompressParams_st.src.

7.70. CUMemFabricHandle_v1 Struct Reference

Fabric handle - An opaque handle representing a memory allocation that can be exported to processes in same or different nodes. For IPC between processes on different nodes they must be connected via the NVSwitch fabric.

7.71. CUMemLocation_v1 Struct Reference

Specifies a memory location.

int CUMemLocation_v1::id

identifier for a given this location's [CUMemLocationType](#).

CUMemLocationType CUMemLocation_v1::type

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

7.72. CUmempoolProps_v1 Struct Reference

Specifies the properties of allocations made from the pool.

CUmempoolAllocationType CUmempoolProps_v1::allocType

Allocation type. Currently must be specified as `CU_MEM_ALLOCATION_TYPE_PINNED`

**CUmempoolAllocationHandleType
CUmempoolProps_v1::handleTypes**

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

struct CUmempoolLocation CUmempoolProps_v1::location

Location where allocations should reside.

size_t CUmempoolProps_v1::maxSize

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

unsigned char CUmempoolProps_v1::reserved

reserved for future use, must be 0

unsigned short CUmempoolProps_v1::usage

Bitmask indicating intended usage for the pool.

void *CUmempoolProps_v1::win32SecurityAttributes

Windows-specific `LPSECURITY_ATTRIBUTES` required when `CU_MEM_HANDLE_TYPE_WIN32` is specified. This security attribute defines the scope of which exported allocations may be transferred to other processes. In all other cases, this field is required to be zero.

7.73. CUmempoolPtrExportData_v1 Struct Reference

Opaque data for exporting a pool allocation

7.74. CUmulticastObjectProp_v1 Struct Reference

Specifies the properties for a multicast object.

`unsigned long long CUmulticastObjectProp_v1::flags`

Flags for future use, must be zero now

`unsigned long long
CUmulticastObjectProp_v1::handleTypes`

Bitmask of exportable handle types (see [CUmemAllocationHandleType](#)) for this object

`unsigned int CUmulticastObjectProp_v1::numDevices`

The number of devices in the multicast team that will bind memory to this object

`size_t CUmulticastObjectProp_v1::size`

The maximum amount of memory that can be bound to this multicast object per device

7.75. CUoffset3D_v1 Struct Reference

Struct representing offset into a CUarray in elements

7.76. CUstreamBatchMemOpParams_v1 Union Reference

Per-operation parameters for [cuStreamBatchMemOp](#)

7.77. CUtensorMap Struct Reference

Tensor map descriptor. Requires compiler support for aligning to 64 bytes.

7.14. 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 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

[CUDA_MEM_ALLOC_NODE_PARAMS_v1](#)

[CUDA_MEM_ALLOC_NODE_PARAMS_v2](#)

accessDescs

[CUDA_MEM_ALLOC_NODE_PARAMS_v2](#)

[CUDA_MEM_ALLOC_NODE_PARAMS_v1](#)

accessPolicyWindow

[CUlaunchAttributeValue](#)

addressMode

[CUDA_TEXTURE_DESC_v1](#)

algo

[CUmemDecompressParams](#)

alignment

[CUDA_ARRAY_MEMORY_REQUIREMENTS_v1](#)

alloc

[CUgraphNodeParams](#)

allocType

[CUmemPoolProps_v1](#)

array

[CUmemcpy3DOperand_v1](#)

arrayDesc

[CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC_v1](#)

attrs

[CUlaunchConfig](#)

B

base_ptr

[CUaccessPolicyWindow_v1](#)

blockDimX

[CUDA_KERNEL_NODE_PARAMS_v1](#)
[CUDA_KERNEL_NODE_PARAMS_v3](#)
[CUlaunchConfig](#)
[CUDA_KERNEL_NODE_PARAMS_v2](#)
[CUDA_LAUNCH_PARAMS_v1](#)

blockDimY

[CUDA_KERNEL_NODE_PARAMS_v2](#)
[CUDA_KERNEL_NODE_PARAMS_v3](#)
[CUlaunchConfig](#)
[CUDA_LAUNCH_PARAMS_v1](#)
[CUDA_KERNEL_NODE_PARAMS_v1](#)

blockDimZ

[CUDA_KERNEL_NODE_PARAMS_v3](#)
[CUDA_LAUNCH_PARAMS_v1](#)
[CUlaunchConfig](#)
[CUDA_KERNEL_NODE_PARAMS_v2](#)
[CUDA_KERNEL_NODE_PARAMS_v1](#)

borderColor

[CUDA_TEXTURE_DESC_v1](#)

bytesize

[CUDA_MEM_ALLOC_NODE_PARAMS_v2](#)
[CUDA_MEM_ALLOC_NODE_PARAMS_v1](#)

C**clockRate**

[CUdevprop_v1](#)

clusterDim

[CUlaunchAttributeValue](#)

clusterSchedulingPolicyPreference

[CUlaunchAttributeValue](#)

compressionType

[CUmemAllocationProp_v1](#)

conditional

[CUgraphNodeParams](#)

cooperative

[CUlaunchAttributeValue](#)

copyCtx

[CUDA_MEMCPY_NODE_PARAMS](#)

copyParams

[CUDA_MEMCPY_NODE_PARAMS](#)

count

[CUDA_BATCH_MEM_OP_NODE_PARAMS_v2](#)

ctx

[CUDA_KERNEL_NODE_PARAMS_v2](#)
[CUDA_CONDITIONAL_NODE_PARAMS](#)
[CUDA_MEMSET_NODE_PARAMS_v2](#)
[CUDA_KERNEL_NODE_PARAMS_v3](#)
[CUDA_BATCH_MEM_OP_NODE_PARAMS_v2](#)

cuFormat

[CUeglFrame_v1](#)

D**default_**

[CUlaunchMemSyncDomainMap](#)

Depth

[CUDA_MEMCPY3D_v2](#)
[CUDA_ARRAY3D_DESCRIPTOR_v2](#)

depth

[CUDA_ARRAY_SPARSE_PROPERTIES_v1](#)

Depth

[CUDA_MEMCPY3D_PEER_v1](#)

depth

[CUDA_RESOURCE_VIEW_DESC_v1](#)
[CUeglFrame_v1](#)

deviceBitMask

[CUarrayMapInfo_v1](#)

deviceUpdatableKernelNode

[CUlaunchAttributeValue](#)

devPtr

[CUDA_RESOURCE_DESC_v1](#)

dptr

[CUDA_MEM_ALLOC_NODE_PARAMS_v1](#)
[CUDA_MEM_ALLOC_NODE_PARAMS_v2](#)
[CUDA_MEM_FREE_NODE_PARAMS](#)

dst

[CUmemDecompressParams](#)
[CUDA_MEMSET_NODE_PARAMS_v1](#)
[CUDA_MEMSET_NODE_PARAMS_v2](#)

dstActBytes

[CUmemDecompressParams](#)

dstArray

[CUDA_MEMCPY2D_v2](#)
[CUDA_MEMCPY3D_v2](#)
[CUDA_MEMCPY3D_PEER_v1](#)

dstContext[CUDA_MEMCPY3D_PEER_v1](#)**dstDevice**[CUDA_MEMCPY2D_v2](#)[CUDA_MEMCPY3D_v2](#)[CUDA_MEMCPY3D_PEER_v1](#)**dstHeight**[CUDA_MEMCPY3D_v2](#)[CUDA_MEMCPY3D_PEER_v1](#)**dstHost**[CUDA_MEMCPY2D_v2](#)[CUDA_MEMCPY3D_v2](#)[CUDA_MEMCPY3D_PEER_v1](#)**dstLocHint**[CUmecpyAttributes_v1](#)**dstLOD**[CUDA_MEMCPY3D_v2](#)[CUDA_MEMCPY3D_PEER_v1](#)**dstMemoryType**[CUDA_MEMCPY2D_v2](#)[CUDA_MEMCPY3D_v2](#)[CUDA_MEMCPY3D_PEER_v1](#)**dstNumBytes**[CUmecpyDecompressParams](#)**dstPitch**[CUDA_MEMCPY3D_PEER_v1](#)[CUDA_MEMCPY3D_v2](#)[CUDA_MEMCPY2D_v2](#)**dstXInBytes**[CUDA_MEMCPY3D_PEER_v1](#)[CUDA_MEMCPY3D_v2](#)[CUDA_MEMCPY2D_v2](#)**dstY**[CUDA_MEMCPY3D_PEER_v1](#)[CUDA_MEMCPY2D_v2](#)[CUDA_MEMCPY3D_v2](#)**dstZ**[CUDA_MEMCPY3D_PEER_v1](#)[CUDA_MEMCPY3D_v2](#)**E****eglColorFormat**[CUeglFrame_v1](#)

elementSize[CUDA_MEMSET_NODE_PARAMS_v1](#)[CUDA_MEMSET_NODE_PARAMS_v2](#)**errorFromNode**[CUgraphExecUpdateResultInfo_v1](#)**errorNode**[CUgraphExecUpdateResultInfo_v1](#)**event**[CUDA_EVENT_RECORD_NODE_PARAMS](#)[CUDA_EVENT_WAIT_NODE_PARAMS](#)**eventRecord**[CUgraphNodeParams](#)**eventWait**[CUgraphNodeParams](#)**extentDepth**[CUarrayMapInfo_v1](#)**extentHeight**[CUarrayMapInfo_v1](#)**extentWidth**[CUarrayMapInfo_v1](#)**extra**[CUDA_KERNEL_NODE_PARAMS_v3](#)[CUDA_KERNEL_NODE_PARAMS_v1](#)[CUDA_KERNEL_NODE_PARAMS_v2](#)**extSemArray**[CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v1](#)[CUDA_EXT_SEM_WAIT_NODE_PARAMS_v1](#)[CUDA_EXT_SEM_WAIT_NODE_PARAMS_v2](#)[CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v2](#)**extSemSignal**[CUgraphNodeParams](#)**extSemWait**[CUgraphNodeParams](#)**F****fd**[CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1](#)[CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_v1](#)**fence**[CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1](#)[CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1](#)[CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1](#)

filterMode[CUDA_TEXTURE_DESC_v1](#)**firstLayer**[CUDA_RESOURCE_VIEW_DESC_v1](#)**firstMipmapLevel**[CUDA_RESOURCE_VIEW_DESC_v1](#)**flags**[CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1](#)[CUDA_EXTERNAL_MEMORY_BUFFER_DESC_v1](#)**Flags**[CUDA_ARRAY3D_DESCRIPTOR_v2](#)**flags**[CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_v1](#)[CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1](#)[CUDA_BATCH_MEM_OP_NODE_PARAMS_v2](#)[CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1](#)[CUmemAccessDesc_v1](#)[CUarrayMapInfo_v1](#)[CUDA_GRAPH_INSTANTIATE_PARAMS](#)[CUmulticastObjectProp_v1](#)[CUDA_MEMCPY_NODE_PARAMS](#)[CUmemcpyAttributes_v1](#)[CUDA_ARRAY_SPARSE_PROPERTIES_v1](#)[CUDA_RESOURCE_DESC_v1](#)[CUDA_TEXTURE_DESC_v1](#)**fn**[CUDA_HOST_NODE_PARAMS_v2](#)[CUDA_HOST_NODE_PARAMS_v1](#)**format**[CUDA_RESOURCE_DESC_v1](#)**Format**[CUDA_ARRAY3D_DESCRIPTOR_v2](#)[CUDA_ARRAY_DESCRIPTOR_v2](#)**format**[CUDA_RESOURCE_VIEW_DESC_v1](#)**frameType**[CUeglFrame_v1](#)**free**[CUgraphNodeParams](#)**from_port**[CUgraphEdgeData](#)**func**[CUDA_KERNEL_NODE_PARAMS_v3](#)

[CUDA_KERNEL_NODE_PARAMS_v2](#)

[CUDA_KERNEL_NODE_PARAMS_v1](#)

function

[CUDA_LAUNCH_PARAMS_v1](#)

G

graph

[CUDA_CHILD_GRAPH_NODE_PARAMS](#)

[CUgraphNodeParams](#)

gridDimX

[CUDA_KERNEL_NODE_PARAMS_v2](#)

[CUDA_KERNEL_NODE_PARAMS_v3](#)

[CUDA_KERNEL_NODE_PARAMS_v1](#)

[CUlaunchConfig](#)

[CUDA_LAUNCH_PARAMS_v1](#)

gridDimY

[CUlaunchConfig](#)

[CUDA_KERNEL_NODE_PARAMS_v1](#)

[CUDA_KERNEL_NODE_PARAMS_v2](#)

[CUDA_KERNEL_NODE_PARAMS_v3](#)

[CUDA_LAUNCH_PARAMS_v1](#)

gridDimZ

[CUDA_KERNEL_NODE_PARAMS_v2](#)

[CUDA_KERNEL_NODE_PARAMS_v3](#)

[CUlaunchConfig](#)

[CUDA_KERNEL_NODE_PARAMS_v1](#)

[CUDA_LAUNCH_PARAMS_v1](#)

H

handle

[CUDA_CONDITIONAL_NODE_PARAMS](#)

[CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1](#)

[CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_v1](#)

handleTypes

[CUmulticastObjectProp_v1](#)

[CUmemPoolProps_v1](#)

hArray

[CUDA_RESOURCE_DESC_v1](#)

Height

[CUDA_MEMCPY3D_v2](#)

height

[CUDA_RESOURCE_VIEW_DESC_v1](#)

[CUeglFrame_v1](#)

Height[CUDA_MEMCPY3D_PEER_v1](#)[CUDA_ARRAY_DESCRIPTOR_v2](#)**height**[CUDA_RESOURCE_DESC_v1](#)[CUDA_ARRAY_SPARSE_PROPERTIES_v1](#)**Height**[CUDA_ARRAY3D_DESCRIPTOR_v2](#)**height**[CUDA_MEMSET_NODE_PARAMS_v1](#)[CUDA_MEMSET_NODE_PARAMS_v2](#)**Height**[CUDA_MEMCPY2D_v2](#)**hErrNode_out**[CUDA_GRAPH_INSTANTIATE_PARAMS](#)**hitProp**[CUaccessPolicyWindow_v1](#)**hitRatio**[CUaccessPolicyWindow_v1](#)**hMipmappedArray**[CUDA_RESOURCE_DESC_v1](#)**host**[CUgraphNodeParams](#)**hStream**[CUlaunchConfig](#)[CUDA_LAUNCH_PARAMS_v1](#)**hUploadStream**[CUDA_GRAPH_INSTANTIATE_PARAMS](#)**I****id**[CUlaunchAttribute](#)[CUmemLocation_v1](#)**K****kern**[CUDA_KERNEL_NODE_PARAMS_v2](#)[CUDA_KERNEL_NODE_PARAMS_v3](#)**kernel**[CUgraphNodeParams](#)**kernelParams**[CUDA_KERNEL_NODE_PARAMS_v1](#)[CUDA_KERNEL_NODE_PARAMS_v2](#)

[CUDA_LAUNCH_PARAMS_v1](#)

[CUDA_KERNEL_NODE_PARAMS_v3](#)

key

[CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1](#)

[CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1](#)

keyedMutex

[CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1](#)

[CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1](#)

L

lastLayer

[CUDA_RESOURCE_VIEW_DESC_v1](#)

lastMipmapLevel

[CUDA_RESOURCE_VIEW_DESC_v1](#)

launchCompletionEvent

[CUlaunchAttributeValue](#)

layer

[CUarrayMapInfo_v1](#)

layerHeight

[CUmecpy3DOperand_v1](#)

level

[CUarrayMapInfo_v1](#)

location

[CUmecAllocationProp_v1](#)

[CUmecAccessDesc_v1](#)

[CUmecPoolProps_v1](#)

locHint

[CUmecpy3DOperand_v1](#)

M

maxAnisotropy

[CUDA_TEXTURE_DESC_v1](#)

maxGridSize

[CUdevprop_v1](#)

maxMipmapLevelClamp

[CUDA_TEXTURE_DESC_v1](#)

maxSize

[CUmecPoolProps_v1](#)

maxThreadsDim

[CUdevprop_v1](#)

maxThreadsPerBlock

[CUdevprop_v1](#)

memcpy[CUgraphNodeParams](#)**memHandleType**[CUarrayMapInfo_v1](#)**memOp**[CUgraphNodeParams](#)**memOperationType**[CUarrayMapInfo_v1](#)**memPitch**[CUdevprop_v1](#)**memset**[CUgraphNodeParams](#)**memSyncDomain**[CUlaunchAttributeValue](#)**memSyncDomainMap**[CUlaunchAttributeValue](#)**minMipmapLevelClamp**[CUDA_TEXTURE_DESC_v1](#)**mipmapFilterMode**[CUDA_TEXTURE_DESC_v1](#)**mipmapLevelBias**[CUDA_TEXTURE_DESC_v1](#)**miptailFirstLevel**[CUDA_ARRAY_SPARSE_PROPERTIES_v1](#)**miptailSize**[CUDA_ARRAY_SPARSE_PROPERTIES_v1](#)**missProp**[CUaccessPolicyWindow_v1](#)**N****name**[CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1](#)[CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_v1](#)**num_bytes**[CUaccessPolicyWindow_v1](#)**numAttrs**[CUlaunchConfig](#)**numChannels**[CUeglFrame_v1](#)**NumChannels**[CUDA_ARRAY_DESCRIPTOR_v2](#)[CUDA_ARRAY3D_DESCRIPTOR_v2](#)

numChannels[CUDA_RESOURCE_DESC_v1](#)**numDevices**[CUmulticastObjectProp_v1](#)**numExtSems**[CUDA_EXT_SEM_WAIT_NODE_PARAMS_v2](#)[CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v1](#)[CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v2](#)[CUDA_EXT_SEM_WAIT_NODE_PARAMS_v1](#)**numLevels**[CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC_v1](#)**nvSciBufObject**[CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1](#)**nvSciSync**[CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1](#)**nvSciSyncObj**[CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_v1](#)**O****offset**[CUDA_EXTERNAL_MEMORY_BUFFER_DESC_v1](#)[CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC_v1](#)[CUarrayMapInfo_v1](#)**offsetX**[CUarrayMapInfo_v1](#)**offsetY**[CUarrayMapInfo_v1](#)**offsetZ**[CUarrayMapInfo_v1](#)**P****paramArray**[CUDA_BATCH_MEM_OP_NODE_PARAMS_v2](#)**paramsArray**[CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v1](#)[CUDA_EXT_SEM_WAIT_NODE_PARAMS_v1](#)[CUDA_EXT_SEM_WAIT_NODE_PARAMS_v2](#)[CUDA_EXT_SEM_SIGNAL_NODE_PARAMS_v2](#)**pArray**[CUeglFrame_v1](#)**phGraph_out**[CUDA_CONDITIONAL_NODE_PARAMS](#)

pitch[CUDA_MEMSET_NODE_PARAMS_v1](#)[CUDA_MEMSET_NODE_PARAMS_v2](#)[CUeglFrame_v1](#)**pitchInBytes**[CUDA_RESOURCE_DESC_v1](#)**planeCount**[CUeglFrame_v1](#)**poolProps**[CUDA_MEM_ALLOC_NODE_PARAMS_v1](#)[CUDA_MEM_ALLOC_NODE_PARAMS_v2](#)**pPitch**[CUeglFrame_v1](#)**preferredClusterDim**[CUlaunchAttributeValue](#)**priority**[CUlaunchAttributeValue](#)**programmaticEvent**[CUlaunchAttributeValue](#)**programmaticStreamSerializationAllowed**[CUlaunchAttributeValue](#)**ptr**[CUmecpy3DOperand_v1](#)**R****regsPerBlock**[CUdevprop_v1](#)**remote**[CUlaunchMemSyncDomainMap](#)**requestedHandleTypes**[CUmecpy3DOperand_v1](#)**reserved**[CUgraphEdgeData](#)[CUcheckpointCheckpointArgs](#)[CUcheckpointRestoreArgs](#)[CUDA_MEMCPY_NODE_PARAMS](#)[CUcheckpointUnlockArgs](#)[CUarrayMapInfo_v1](#)[CUmecpy3DOperand_v1](#)**reserved0**[CUDA_MEMCPY3D_v2](#)[CUgraphNodeParams](#)[CUcheckpointLockArgs](#)

reserved1[CUDA_MEMCPY3D_v2](#)[CUcheckpointLockArgs](#)[CUgraphNodeParams](#)**reserved2**[CUgraphNodeParams](#)**resourceType**[CUarrayMapInfo_v1](#)**resType**[CUDA_RESOURCE_DESC_v1](#)**result**[CUgraphExecUpdateResultInfo_v1](#)**result_out**[CUDA_GRAPH_INSTANTIATE_PARAMS](#)**rowLength**[CUmecpy3DOperand_v1](#)**S****sharedMemBytes**[CUDA_KERNEL_NODE_PARAMS_v1](#)[CUDA_KERNEL_NODE_PARAMS_v2](#)[CUlaunchConfig](#)[CUDA_LAUNCH_PARAMS_v1](#)[CUDA_KERNEL_NODE_PARAMS_v3](#)**sharedMemCarveout**[CUlaunchAttributeValue](#)**sharedMemPerBlock**[CUdevprop_v1](#)**SIMDWidth**[CUdevprop_v1](#)**size**[CUDA_CONDITIONAL_NODE_PARAMS](#)[CUDA_ARRAY_MEMORY_REQUIREMENTS_v1](#)[CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1](#)[CUDA_EXTERNAL_MEMORY_BUFFER_DESC_v1](#)[CUarrayMapInfo_v1](#)[CUmulticastObjectProp_v1](#)**sizeInBytes**[CUDA_RESOURCE_DESC_v1](#)**smCount**[CUdevSmResource](#)**src**[CUmecpy3DOperand_v1](#)

srcAccessOrder[CUmemcpyAttributes_v1](#)**srcArray**[CUDA_MEMCPY2D_v2](#)[CUDA_MEMCPY3D_v2](#)[CUDA_MEMCPY3D_PEER_v1](#)**srcContext**[CUDA_MEMCPY3D_PEER_v1](#)**srcDevice**[CUDA_MEMCPY2D_v2](#)[CUDA_MEMCPY3D_v2](#)[CUDA_MEMCPY3D_PEER_v1](#)**srcHeight**[CUDA_MEMCPY3D_v2](#)[CUDA_MEMCPY3D_PEER_v1](#)**srcHost**[CUDA_MEMCPY2D_v2](#)[CUDA_MEMCPY3D_v2](#)[CUDA_MEMCPY3D_PEER_v1](#)**srcLocHint**[CUmemcpyAttributes_v1](#)**srcLOD**[CUDA_MEMCPY3D_v2](#)[CUDA_MEMCPY3D_PEER_v1](#)**srcMemoryType**[CUDA_MEMCPY2D_v2](#)[CUDA_MEMCPY3D_v2](#)[CUDA_MEMCPY3D_PEER_v1](#)**srcNumBytes**[CUmemDecompressParams](#)**srcPitch**[CUDA_MEMCPY2D_v2](#)[CUDA_MEMCPY3D_PEER_v1](#)[CUDA_MEMCPY3D_v2](#)**srcXInBytes**[CUDA_MEMCPY3D_v2](#)[CUDA_MEMCPY2D_v2](#)[CUDA_MEMCPY3D_PEER_v1](#)**srcY**[CUDA_MEMCPY2D_v2](#)[CUDA_MEMCPY3D_PEER_v1](#)[CUDA_MEMCPY3D_v2](#)

srcZ[CUDA_MEMCPY3D_v2](#)[CUDA_MEMCPY3D_PEER_v1](#)**subresourceType**[CUarrayMapInfo_v1](#)**syncPolicy**[CUlaunchAttributeValue](#)**T****textureAlign**[CUdevprop_v1](#)**timeoutMs**[CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1](#)[CUcheckpointLockArgs](#)**to_port**[CUgraphEdgeData](#)**totalConstantMemory**[CUdevprop_v1](#)**type**[CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1](#)[CUgraphEdgeData](#)[CUDA_CONDITIONAL_NODE_PARAMS](#)[CUmemLocation_v1](#)[CUgraphNodeParams](#)[CUmemAllocationProp_v1](#)[CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_v1](#)**U****usage**[CUmemAllocationProp_v1](#)[CUmemPoolProps_v1](#)**userData**[CUDA_HOST_NODE_PARAMS_v1](#)[CUDA_HOST_NODE_PARAMS_v2](#)**V****val**[CUexecAffinitySmCount_v1](#)**value**[CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS_v1](#)[CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS_v1](#)[CUlaunchAttribute](#)[CUDA_MEMSET_NODE_PARAMS_v2](#)

[CUDA_MEMSET_NODE_PARAMS_v1](#)

W

Width

[CUDA_ARRAY_DESCRIPTOR_v2](#)

[CUDA_ARRAY3D_DESCRIPTOR_v2](#)

width

[CUDA_MEMSET_NODE_PARAMS_v2](#)

[CUeglFrame_v1](#)

[CUDA_ARRAY_SPARSE_PROPERTIES_v1](#)

[CUDA_MEMSET_NODE_PARAMS_v1](#)

[CUDA_RESOURCE_DESC_v1](#)

[CUDA_RESOURCE_VIEW_DESC_v1](#)

WidthInBytes

[CUDA_MEMCPY2D_v2](#)

[CUDA_MEMCPY3D_v2](#)

[CUDA_MEMCPY3D_PEER_v1](#)

win32

[CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_v1](#)

[CUDA_EXTERNAL_MEMORY_HANDLE_DESC_v1](#)

win32HandleMetaData

[CUmemAllocationProp_v1](#)

win32SecurityAttributes

[CUmemPoolProps_v1](#)

Chapter 9. Deprecated List

Global CU_CTX_BLOCKING_SYNC

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

Global CU_CTX_MAP_HOST

This flag was deprecated as of CUDA 11.0 and it no longer has any effect. All contexts as of CUDA 3.2 behave as though the flag is enabled.

Global CU_DEVICE_P2P_ATTRIBUTE_ACCESS_ACCESS_SUPPORTED

use CU_DEVICE_P2P_ATTRIBUTE_CUDA_ARRAY_ACCESS_SUPPORTED instead

Global CU_JIT_NEW_SM3X_OPT

This jit option is deprecated and should not be used.

Global CU_JIT_LTO

Enable link-time optimization (-dlto) for device code (Disabled by default).

This option is not supported on 32-bit platforms.

Option type: int

Applies to: compiler and linker

Global CU_JIT_FTZ

Control single-precision denormals (-ftz) support (0: false, default). 1 : flushes denormal values to zero 0 : preserves denormal values Option type: int

Applies to: link-time optimization specified with CU_JIT_LTO

Global CU_JIT_PREC_DIV

Control single-precision floating-point division and reciprocals (-prec-div) support (1: true, default).
 1 : Enables the IEEE round-to-nearest mode 0 : Enables the fast approximation mode Option type: int

Applies to: link-time optimization specified with CU_JIT_LTO

Global CU_JIT_PREC_SQRT

Control single-precision floating-point square root (-prec-sqrt) support (1: true, default). 1 : Enables the IEEE round-to-nearest mode 0 : Enables the fast approximation mode Option type: int

Applies to: link-time optimization specified with CU_JIT_LTO

Global CU_JIT_FMA

Enable/Disable the contraction of floating-point multiplies and adds/subtracts into floating-point multiply-add (-fma) operations (1: Enable, default; 0: Disable). Option type: int

Applies to: link-time optimization specified with CU_JIT_LTO

Global CU_JIT_REFERENCED_KERNEL_NAMES

Array of kernel names that should be preserved at link time while others can be removed.

Must contain CU_JIT_REFERENCED_KERNEL_COUNT entries.

Note that kernel names can be mangled by the compiler in which case the mangled name needs to be specified.

Wildcard "*" can be used to represent zero or more characters instead of specifying the full or mangled name.

It is important to note that the wildcard "*" is also added implicitly. For example, specifying "foo" will match "foobaz", "barfoo", "barfoobaz" and thus preserve all kernels with those names. This can be avoided by providing a more specific name like "barfoobaz".

Option type: const char **

Applies to: dynamic linker only

Global CU_JIT_REFERENCED_KERNEL_COUNT

Number of entries in CU_JIT_REFERENCED_KERNEL_NAMES array.

Option type: unsigned int

Applies to: dynamic linker only

Global CU_JIT_REFERENCED_VARIABLE_NAMES

Array of variable names (`__device__` and/or `__constant__`) that should be preserved at link time while others can be removed.

Must contain `CU_JIT_REFERENCED_VARIABLE_COUNT` entries.

Note that variable names can be mangled by the compiler in which case the mangled name needs to be specified.

Wildcard "*" can be used to represent zero or more characters instead of specifying the full or mangled name.

It is important to note that the wildcard "*" is also added implicitly. For example, specifying "foo" will match "foobaz", "barfoo", "barfoobaz" and thus preserve all variables with those names. This can be avoided by providing a more specific name like "barfoobaz".

Option type: `const char **`

Applies to: link-time optimization specified with `CU_JIT_LTO`

Global CU_JIT_REFERENCED_VARIABLE_COUNT

Number of entries in `CU_JIT_REFERENCED_VARIABLE_NAMES` array.

Option type: unsigned int

Applies to: link-time optimization specified with `CU_JIT_LTO`

Global CU_JIT_OPTIMIZE_UNUSED_DEVICE_VARIABLES

This option serves as a hint to enable the JIT compiler/linker to remove constant (`__constant__`) and device (`__device__`) variables unreferenced in device code (Disabled by default).

Note that host references to constant and device variables using APIs like `cuModuleGetGlobal()` with this option specified may result in undefined behavior unless the variables are explicitly specified using `CU_JIT_REFERENCED_VARIABLE_NAMES`.

Option type: int

Applies to: link-time optimization specified with `CU_JIT_LTO`

Global CU_JIT_INPUT_NVVM

High-level intermediate code for link-time optimization

Applicable options: NVVM compiler options, PTX compiler options

Global CUDA_ERROR_PROFILER_NOT_INITIALIZED

This error return is deprecated as of CUDA 5.0. It is no longer an error to attempt to enable/disable the profiling via cuProfilerStart or cuProfilerStop without initialization.

Global CUDA_ERROR_PROFILER_ALREADY_STARTED

This error return is deprecated as of CUDA 5.0. It is no longer an error to call cuProfilerStart() when profiling is already enabled.

Global CUDA_ERROR_PROFILER_ALREADY_STOPPED

This error return is deprecated as of CUDA 5.0. It is no longer an error to call cuProfilerStop() when profiling is already disabled.

Global CUDA_ERROR_CONTEXT_ALREADY_CURRENT

This error return is deprecated as of CUDA 3.2. It is no longer an error to attempt to push the active context via cuCtxPushCurrent().

Global CUsharedconfig**Global cuDeviceComputeCapability****Global cuDeviceGetProperties****Global cuCtxAttach****Global cuCtxDetach****Global cuCtxGetSharedMemConfig****Global cuCtxSetSharedMemConfig****Global cuModuleGetSurfRef**

Global cuModuleGetTexRef

Global cuLaunchCooperativeKernelMultiDevice

This function is deprecated as of CUDA 11.3.

Global cuFuncSetBlockShape

Global cuFuncSetSharedMemConfig

Global cuFuncSetSharedSize

Global cuLaunch

Global cuLaunchGrid

Global cuLaunchGridAsync

Global cuParamSetf

Global cuParamSeti

Global cuParamSetSize

Global cuParamSetTexRef

Global cuParamSetv

Global cuTexRefCreate

Global cuTexRefDestroy

Global cuTexRefGetAddress

Global cuTexRefGetAddressMode

Global cuTexRefGetArray

Global cuTexRefGetBorderColor

Global cuTexRefGetFilterMode

Global cuTexRefGetFlags

Global cuTexRefGetFormat

Global cuTexRefGetMaxAnisotropy

Global cuTexRefGetMipmapFilterMode

Global cuTexRefGetMipmapLevelBias

Global cuTexRefGetMipmapLevelClamp

Global cuTexRefGetMipmappedArray

Global cuTexRefSetAddress

Global cuTexRefSetAddress2D

Global cuTexRefSetAddressMode

Global cuTexRefSetArray

Global cuTexRefSetBorderColor

Global cuTexRefSetFilterMode

Global cuTexRefSetFlags

Global cuTexRefSetFormat

Global cuTexRefSetMaxAnisotropy

Global cuTexRefSetMipmapFilterMode

Global cuTexRefSetMipmapLevelBias

Global cuTexRefSetMipmapLevelClamp

Global cuTexRefSetMipmappedArray

Global cuSurfRefGetArray**Global cuSurfRefSetArray****Global cuProfilerInitialize****Global cuGLCtxCreate**

This function is deprecated as of Cuda 5.0.

Global cuGLInit

This function is deprecated as of Cuda 3.0.

Global cuGLMapBufferObject

This function is deprecated as of Cuda 3.0.

Global cuGLMapBufferObjectAsync

This function is deprecated as of Cuda 3.0.

Global cuGLRegisterBufferObject

This function is deprecated as of Cuda 3.0.

Global cuGLSetBufferObjectMapFlags

This function is deprecated as of Cuda 3.0.

Global cuGLUnmapBufferObject

This function is deprecated as of Cuda 3.0.

Global cuGLUnmapBufferObjectAsync

This function is deprecated as of Cuda 3.0.

Global cuGLUnregisterBufferObject

This function is deprecated as of Cuda 3.0.

Global cuD3D9MapResources

This function is deprecated as of CUDA 3.0.

Global cuD3D9RegisterResource

This function is deprecated as of CUDA 3.0.

Global cuD3D9ResourceGetMappedArray

This function is deprecated as of CUDA 3.0.

Global cuD3D9ResourceGetMappedPitch

This function is deprecated as of CUDA 3.0.

Global cuD3D9ResourceGetMappedPointer

This function is deprecated as of CUDA 3.0.

Global cuD3D9ResourceGetMappedSize

This function is deprecated as of CUDA 3.0.

Global cuD3D9ResourceGetSurfaceDimensions

This function is deprecated as of CUDA 3.0.

Global cuD3D9ResourceSetMapFlags

This function is deprecated as of Cuda 3.0.

Global cuD3D9UnmapResources

This function is deprecated as of CUDA 3.0.

Global cuD3D9UnregisterResource

This function is deprecated as of CUDA 3.0.

Global cuD3D10CtxCreate

This function is deprecated as of CUDA 5.0.

Global cuD3D10CtxCreateOnDevice

This function is deprecated as of CUDA 5.0.

Global cuD3D10GetDirect3DDevice

This function is deprecated as of CUDA 5.0.

Global cuD3D10MapResources

This function is deprecated as of CUDA 3.0.

Global cuD3D10RegisterResource

This function is deprecated as of CUDA 3.0.

Global cuD3D10ResourceGetMappedArray

This function is deprecated as of CUDA 3.0.

Global cuD3D10ResourceGetMappedPitch

This function is deprecated as of CUDA 3.0.

Global cuD3D10ResourceGetMappedPointer

This function is deprecated as of CUDA 3.0.

Global cuD3D10ResourceGetMappedSize

This function is deprecated as of CUDA 3.0.

Global cuD3D10ResourceGetSurfaceDimensions

This function is deprecated as of CUDA 3.0.

Global cuD3D10ResourceSetMapFlags

This function is deprecated as of CUDA 3.0.

Global cuD3D10UnmapResources

This function is deprecated as of CUDA 3.0.

Global cuD3D10UnregisterResource

This function is deprecated as of CUDA 3.0.

Global cuD3D11CtxCreate

This function is deprecated as of CUDA 5.0.

Global cuD3D11CtxCreateOnDevice

This function is deprecated as of CUDA 5.0.

Global cuD3D11GetDirect3DDevice

This function is deprecated as of CUDA 5.0.

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2007-2025 NVIDIA Corporation & affiliates. All rights reserved.