



# CUDA RUNTIME API

vRelease Version | July 2019

## 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. Modules.....	8
5.1. Device Management.....	9
cudaChooseDevice.....	9
cudaDeviceGetAttribute.....	10
cudaDeviceGetByPCIBusId.....	15
cudaDeviceGetCacheConfig.....	16
cudaDeviceGetLimit.....	17
cudaDeviceGetNvSciSyncAttributes.....	18
cudaDeviceGetP2PAttribute.....	19
cudaDeviceGetPCIBusId.....	20
cudaDeviceGetSharedMemConfig.....	21
cudaDeviceGetStreamPriorityRange.....	22
cudaDeviceReset.....	23
cudaDeviceSetCacheConfig.....	24
cudaDeviceSetLimit.....	25
cudaDeviceSetSharedMemConfig.....	27
cudaDeviceSynchronize.....	28
cudaGetDevice.....	29
cudaGetDeviceCount.....	29
cudaGetDeviceFlags.....	30
cudaGetDeviceProperties.....	31
cudalpcCloseMemHandle.....	36
cudalpcGetEventHandle.....	37
cudalpcGetMemHandle.....	38
cudalpcOpenEventHandle.....	39
cudalpcOpenMemHandle.....	40
cudaSetDevice.....	42
cudaSetDeviceFlags.....	43
cudaSetValidDevices.....	44
5.2. Thread Management [DEPRECATED].....	45
cudaThreadExit.....	45
cudaThreadGetCacheConfig.....	46
cudaThreadGetLimit.....	47
cudaThreadSetCacheConfig.....	48
cudaThreadSetLimit.....	49
cudaThreadSynchronize.....	51

5.3. Error Handling.....	51
cudaGetErrorName.....	52
cudaGetErrorString.....	52
cudaGetLastError.....	53
cudaPeekAtLastError.....	54
5.4. Stream Management.....	55
cudaStreamCallback_t.....	55
cudaStreamAddCallback.....	55
cudaStreamAttachMemAsync.....	57
cudaStreamBeginCapture.....	59
cudaStreamCreate.....	60
cudaStreamCreateWithFlags.....	61
cudaStreamCreateWithPriority.....	62
cudaStreamDestroy.....	63
cudaStreamEndCapture.....	64
cudaStreamGetCaptureInfo.....	65
cudaStreamGetFlags.....	66
cudaStreamGetPriority.....	67
cudaStreamIsCapturing.....	68
cudaStreamQuery.....	69
cudaStreamSynchronize.....	70
cudaStreamWaitEvent.....	71
cudaThreadExchangeStreamCaptureMode.....	72
5.5. Event Management.....	73
cudaEventCreate.....	73
cudaEventCreateWithFlags.....	74
cudaEventDestroy.....	75
cudaEventElapsedTime.....	76
cudaEventQuery.....	77
cudaEventRecord.....	78
cudaEventSynchronize.....	79
5.6. External Resource Interoperability.....	80
cudaDestroyExternalMemory.....	80
cudaDestroyExternalSemaphore.....	81
cudaExternalMemoryGetMappedBuffer.....	82
cudaExternalMemoryGetMappedMipmappedArray.....	83
cudaImportExternalMemory.....	85
cudaImportExternalSemaphore.....	88
cudaSignalExternalSemaphoresAsync.....	91
cudaWaitExternalSemaphoresAsync.....	92
5.7. Execution Control.....	94
cudaFuncGetAttributes.....	95
cudaFuncSetAttribute.....	96

cudaFuncSetCacheConfig.....	97
cudaFuncSetSharedMemConfig.....	98
cudaGetParameterBuffer.....	100
cudaGetParameterBufferV2.....	100
cudaLaunchCooperativeKernel.....	101
cudaLaunchCooperativeKernelMultiDevice.....	103
cudaLaunchHostFunc.....	105
cudaLaunchKernel.....	107
cudaSetDoubleForDevice.....	108
cudaSetDoubleForHost.....	109
5.8. Occupancy.....	110
cudaOccupancyMaxActiveBlocksPerMultiprocessor.....	110
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags.....	111
5.9. Memory Management.....	113
cudaArrayGetInfo.....	113
cudaFree.....	114
cudaFreeArray.....	115
cudaFreeHost.....	115
cudaFreeMipmappedArray.....	116
cudaGetMipmappedArrayLevel.....	117
cudaGetSymbolAddress.....	118
cudaGetSymbolSize.....	119
cudaHostAlloc.....	120
cudaHostGetDevicePointer.....	121
cudaHostGetFlags.....	123
cudaHostRegister.....	123
cudaHostUnregister.....	125
cudaMalloc.....	126
cudaMalloc3D.....	127
cudaMalloc3DArray.....	128
cudaMallocArray.....	131
cudaMallocHost.....	132
cudaMallocManaged.....	133
cudaMallocMipmappedArray.....	136
cudaMallocPitch.....	139
cudaMemAdvise.....	140
cudaMemcpy.....	143
cudaMemcpy2D.....	144
cudaMemcpy2DArrayToArray.....	146
cudaMemcpy2DAsync.....	147
cudaMemcpy2DFromArray.....	149
cudaMemcpy2DFromArrayAsync.....	150
cudaMemcpy2DToArray.....	152

cudaMemcpy2DToArrayAsync.....	154
cudaMemcpy3D.....	155
cudaMemcpy3DAsync.....	158
cudaMemcpy3DPeer.....	160
cudaMemcpy3DPeerAsync.....	161
cudaMemcpyAsync.....	162
cudaMemcpyFromSymbol.....	164
cudaMemcpyFromSymbolAsync.....	165
cudaMemcpyPeer.....	166
cudaMemcpyPeerAsync.....	168
cudaMemcpyToSymbol.....	169
cudaMemcpyToSymbolAsync.....	170
cudaMemGetInfo.....	172
cudaMemPrefetchAsync.....	172
cudaMemRangeGetAttribute.....	174
cudaMemRangeGetAttributes.....	176
cudaMemset.....	178
cudaMemset2D.....	179
cudaMemset2DAsync.....	180
cudaMemset3D.....	181
cudaMemset3DAsync.....	182
cudaMemsetAsync.....	184
make_cudaExtent.....	185
make_cudaPitchedPtr.....	185
make_cudaPos.....	186
5.10. Memory Management [DEPRECATED].....	186
cudaMemcpyArrayToArray.....	187
cudaMemcpyFromArray.....	188
cudaMemcpyFromArrayAsync.....	189
cudaMemcpyToArray.....	191
cudaMemcpyToArrayAsync.....	192
5.11. Unified Addressing.....	194
cudaPointerGetAttributes.....	195
5.12. Peer Device Memory Access.....	197
cudaDeviceCanAccessPeer.....	197
cudaDeviceDisablePeerAccess.....	198
cudaDeviceEnablePeerAccess.....	199
5.13. OpenGL Interoperability.....	200
cudaGLDeviceList.....	200
cudaGLGetDevices.....	200
cudaGraphicsGLRegisterBuffer.....	201
cudaGraphicsGLRegisterImage.....	202
cudaWGLGetDevice.....	204

5.14. OpenGL Interoperability [DEPRECATED].....	204
cudaGLMapFlags.....	204
cudaGLMapBufferObject.....	205
cudaGLMapBufferObjectAsync.....	206
cudaGLRegisterBufferObject.....	207
cudaGLSetBufferObjectMapFlags.....	207
cudaGLSetGLDevice.....	208
cudaGLUnmapBufferObject.....	209
cudaGLUnmapBufferObjectAsync.....	210
cudaGLUnregisterBufferObject.....	210
5.15. Direct3D 9 Interoperability.....	211
cudaD3D9DeviceList.....	211
cudaD3D9GetDevice.....	212
cudaD3D9GetDevices.....	212
cudaD3D9GetDirect3DDevice.....	213
cudaD3D9SetDirect3DDevice.....	214
cudaGraphicsD3D9RegisterResource.....	215
5.16. Direct3D 9 Interoperability [DEPRECATED].....	217
cudaD3D9MapFlags.....	217
cudaD3D9RegisterFlags.....	217
cudaD3D9MapResources.....	218
cudaD3D9RegisterResource.....	219
cudaD3D9ResourceGetMappedArray.....	220
cudaD3D9ResourceGetMappedPitch.....	221
cudaD3D9ResourceGetMappedPointer.....	223
cudaD3D9ResourceGetMappedSize.....	224
cudaD3D9ResourceGetSurfaceDimensions.....	225
cudaD3D9ResourceSetMapFlags.....	226
cudaD3D9UnmapResources.....	227
cudaD3D9UnregisterResource.....	228
5.17. Direct3D 10 Interoperability.....	229
cudaD3D10DeviceList.....	229
cudaD3D10GetDevice.....	229
cudaD3D10GetDevices.....	230
cudaGraphicsD3D10RegisterResource.....	231
5.18. Direct3D 10 Interoperability [DEPRECATED].....	233
cudaD3D10MapFlags.....	233
cudaD3D10RegisterFlags.....	233
cudaD3D10GetDirect3DDevice.....	234
cudaD3D10MapResources.....	234
cudaD3D10RegisterResource.....	235
cudaD3D10ResourceGetMappedArray.....	237
cudaD3D10ResourceGetMappedPitch.....	238

cudaD3D10ResourceGetMappedPointer.....	239
cudaD3D10ResourceGetMappedSize.....	240
cudaD3D10ResourceGetSurfaceDimensions.....	241
cudaD3D10ResourceSetMapFlags.....	242
cudaD3D10SetDirect3DDevice.....	243
cudaD3D10UnmapResources.....	244
cudaD3D10UnregisterResource.....	245
5.19. Direct3D 11 Interoperability.....	246
cudaD3D11DeviceList.....	246
cudaD3D11GetDevice.....	246
cudaD3D11GetDevices.....	247
cudaGraphicsD3D11RegisterResource.....	248
5.20. Direct3D 11 Interoperability [DEPRECATED].....	250
cudaD3D11GetDirect3DDevice.....	250
cudaD3D11SetDirect3DDevice.....	251
5.21. VDPAU Interoperability.....	251
cudaGraphicsVDPAURegisterOutputSurface.....	251
cudaGraphicsVDPAURegisterVideoSurface.....	252
cudaVDPAUGetDevice.....	254
cudaVDPAUSetVDPAUDevice.....	254
5.22. EGL Interoperability.....	255
cudaEGLStreamConsumerAcquireFrame.....	255
cudaEGLStreamConsumerConnect.....	256
cudaEGLStreamConsumerConnectWithFlags.....	257
cudaEGLStreamConsumerDisconnect.....	258
cudaEGLStreamConsumerReleaseFrame.....	258
cudaEGLStreamProducerConnect.....	259
cudaEGLStreamProducerDisconnect.....	260
cudaEGLStreamProducerPresentFrame.....	260
cudaEGLStreamProducerReturnFrame.....	261
cudaEventCreateFromEGLSync.....	262
cudaGraphicsEGLRegisterImage.....	263
cudaGraphicsResourceGetMappedEglFrame.....	264
5.23. Graphics Interoperability.....	265
cudaGraphicsMapResources.....	265
cudaGraphicsResourceGetMappedMipmappedArray.....	266
cudaGraphicsResourceGetMappedPointer.....	267
cudaGraphicsResourceSetMapFlags.....	268
cudaGraphicsSubResourceGetMappedArray.....	269
cudaGraphicsUnmapResources.....	270
cudaGraphicsUnregisterResource.....	272
5.24. Texture Reference Management [DEPRECATED].....	272
cudaBindTexture.....	273

cudaBindTexture2D.....	274
cudaBindTextureToArray.....	276
cudaBindTextureToMipmappedArray.....	277
cudaGetTextureAlignmentOffset.....	278
cudaGetTextureReference.....	279
cudaUnbindTexture.....	280
5.25. Surface Reference Management [DEPRECATED].....	280
cudaBindSurfaceToArray.....	281
cudaGetSurfaceReference.....	282
5.26. Texture Object Management.....	282
cudaCreateChannelDesc.....	283
cudaCreateTextureObject.....	283
cudaDestroyTextureObject.....	289
cudaGetChannelDesc.....	289
cudaGetTextureObjectResourceDesc.....	290
cudaGetTextureObjectResourceViewDesc.....	291
cudaGetTextureObjectTextureDesc.....	292
5.27. Surface Object Management.....	292
cudaCreateSurfaceObject.....	293
cudaDestroySurfaceObject.....	294
cudaGetSurfaceObjectResourceDesc.....	294
5.28. Version Management.....	295
cudaDriverGetVersion.....	295
cudaRuntimeGetVersion.....	296
5.29. Graph Management.....	297
cudaGraphAddChildGraphNode.....	297
cudaGraphAddDependencies.....	298
cudaGraphAddEmptyNode.....	299
cudaGraphAddHostNode.....	300
cudaGraphAddKernelNode.....	301
cudaGraphAddMemcpyNode.....	304
cudaGraphAddMemsetNode.....	305
cudaGraphChildGraphNodeGetGraph.....	306
cudaGraphClone.....	307
cudaGraphCreate.....	308
cudaGraphDestroy.....	309
cudaGraphDestroyNode.....	310
cudaGraphExecDestroy.....	310
cudaGraphExecHostNodeSetParams.....	311
cudaGraphExecKernelNodeSetParams.....	312
cudaGraphExecMemcpyNodeSetParams.....	313
cudaGraphExecMemsetNodeSetParams.....	314
cudaGraphExecUpdate.....	316



cudaGraphGetEdges.....	318
cudaGraphGetNodes.....	319
cudaGraphGetRootNodes.....	320
cudaGraphHostNodeGetParams.....	321
cudaGraphHostNodeSetParams.....	322
cudaGraphInstantiate.....	323
cudaGraphKernelNodeGetParams.....	324
cudaGraphKernelNodeSetParams.....	325
cudaGraphLaunch.....	326
cudaGraphMemcpyNodeGetParams.....	327
cudaGraphMemcpyNodeSetParams.....	328
cudaGraphMemsetNodeGetParams.....	329
cudaGraphMemsetNodeSetParams.....	330
cudaGraphNodeFindInClone.....	331
cudaGraphNodeGetDependencies.....	332
cudaGraphNodeGetDependentNodes.....	333
cudaGraphNodeGetType.....	334
cudaGraphRemoveDependencies.....	335
5.30. C++ API Routines.....	336
__cudaOccupancyB2DHelper.....	336
cudaBindSurfaceToArray.....	336
cudaBindSurfaceToArray.....	337
cudaBindTexture.....	338
cudaBindTexture.....	339
cudaBindTexture2D.....	340
cudaBindTexture2D.....	341
cudaBindTextureToArray.....	343
cudaBindTextureToArray.....	343
cudaBindTextureToMipmappedArray.....	345
cudaBindTextureToMipmappedArray.....	346
cudaCreateChannelDesc.....	347
cudaEventCreate.....	347
cudaFuncGetAttributes.....	348
cudaFuncSetAttribute.....	349
cudaFuncSetCacheConfig.....	351
cudaGetSymbolAddress.....	352
cudaGetSymbolSize.....	353
cudaGetTextureAlignmentOffset.....	353
cudaLaunchCooperativeKernel.....	354
cudaLaunchKernel.....	356
cudaMallocHost.....	357
cudaMallocManaged.....	359
cudaMemcpyFromSymbol.....	362

cudaMemcpyFromSymbolAsync.....	363
cudaMemcpyToSymbol.....	364
cudaMemcpyToSymbolAsync.....	365
cudaOccupancyMaxActiveBlocksPerMultiprocessor.....	367
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags.....	368
cudaOccupancyMaxPotentialBlockSize.....	369
cudaOccupancyMaxPotentialBlockSizeVariableSMem.....	370
cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags.....	372
cudaOccupancyMaxPotentialBlockSizeWithFlags.....	373
cudaStreamAttachMemAsync.....	375
cudaUnbindTexture.....	377
5.31. Interactions with the CUDA Driver API.....	378
5.32. Profiler Control.....	380
cudaProfilerInitialize.....	380
cudaProfilerStart.....	381
cudaProfilerStop.....	382
5.33. Data types used by CUDA Runtime.....	382
cudaChannelFormatDesc.....	383
cudaDeviceProp.....	383
cudaEglFrame.....	383
cudaEglPlaneDesc.....	383
cudaExtent.....	383
cudaExternalMemoryBufferDesc.....	383
cudaExternalMemoryHandleDesc.....	383
cudaExternalMemoryMipmappedArrayDesc.....	383
cudaExternalSemaphoreHandleDesc.....	383
cudaExternalSemaphoreSignalParams.....	383
cudaExternalSemaphoreWaitParams.....	383
cudaFuncAttributes.....	383
cudaHostNodeParams.....	383
cudaIpcEventHandle_t.....	383
cudaIpcMemHandle_t.....	383
cudaKernelNodeParams.....	383
cudaLaunchParams.....	383
cudaMemcpy3DParms.....	383
cudaMemcpy3DPeerParms.....	384
cudaMemsetParams.....	384
cudaPitchedPtr.....	384
cudaPointerAttributes.....	384
cudaPos.....	384
cudaResourceDesc.....	384
cudaResourceViewDesc.....	384
cudaTextureDesc.....	384

CUuid_st.....	384
surfaceReference.....	384
textureReference.....	384
cudaCGScope.....	384
cudaChannelFormatKind.....	384
cudaComputeMode.....	385
cudaDeviceAttr.....	385
cudaDeviceP2PAttr.....	390
cudaEglColorFormat.....	390
cudaEglFrameType.....	395
cudaEglResourceLocationFlags.....	395
cudaError.....	396
cudaExternalMemoryHandleType.....	405
cudaExternalSemaphoreHandleType.....	406
cudaFuncAttribute.....	406
cudaFuncCache.....	406
cudaGraphExecUpdateResult.....	407
cudaGraphicsCubeFace.....	407
cudaGraphicsMapFlags.....	408
cudaGraphicsRegisterFlags.....	408
cudaGraphNodeType.....	408
cudaLimit.....	409
cudaMemcpyKind.....	409
cudaMemoryAdvise.....	410
cudaMemoryType.....	410
cudaMemRangeAttribute.....	410
cudaOutputMode.....	411
cudaResourceType.....	411
cudaResourceViewFormat.....	411
cudaSharedCarveout.....	413
cudaSharedMemConfig.....	413
cudaStreamCaptureMode.....	414
cudaStreamCaptureStatus.....	414
cudaSurfaceBoundaryMode.....	414
cudaSurfaceFormatMode.....	414
cudaTextureAddressMode.....	415
cudaTextureFilterMode.....	415
cudaTextureReadMode.....	415
cudaArray_const_t.....	415
cudaArray_t.....	415
cudaEglStreamConnection.....	416
cudaError_t.....	416
cudaEvent_t.....	416

cudaExternalMemory_t.....	416
cudaExternalSemaphore_t.....	416
cudaGraph_t.....	416
cudaGraphExec_t.....	416
cudaGraphicsResource_t.....	416
cudaGraphNode_t.....	416
cudaHostFn_t.....	416
cudaMipmappedArray_const_t.....	417
cudaMipmappedArray_t.....	417
cudaOutputMode_t.....	417
cudaStream_t.....	417
cudaSurfaceObject_t.....	417
cudaTextureObject_t.....	417
CUDA_EGL_MAX_PLANES.....	417
CUDA_IPC_HANDLE_SIZE.....	417
cudaArrayColorAttachment.....	417
cudaArrayCubemap.....	417
cudaArrayDefault.....	418
cudaArrayLayered.....	418
cudaArraySurfaceLoadStore.....	418
cudaArrayTextureGather.....	418
cudaCooperativeLaunchMultiDeviceNoPostSync.....	418
cudaCooperativeLaunchMultiDeviceNoPreSync.....	418
cudaCpuDeviceId.....	418
cudaDeviceBlockingSync.....	418
cudaDeviceLmemResizeToMax.....	419
cudaDeviceMapHost.....	419
cudaDeviceMask.....	419
cudaDevicePropDontCare.....	419
cudaDeviceScheduleAuto.....	419
cudaDeviceScheduleBlockingSync.....	419
cudaDeviceScheduleMask.....	419
cudaDeviceScheduleSpin.....	419
cudaDeviceScheduleYield.....	419
cudaEventBlockingSync.....	419
cudaEventDefault.....	419
cudaEventDisableTiming.....	420
cudaEventInterprocess.....	420
cudaExternalMemoryDedicated.....	420
cudaExternalSemaphoreSignalSkipNvSciBufMemSync.....	420
cudaExternalSemaphoreWaitSkipNvSciBufMemSync.....	420
cudaHostAllocDefault.....	420
cudaHostAllocMapped.....	420

cudaHostAllocPortable.....	421
cudaHostAllocWriteCombined.....	421
cudaHostRegisterDefault.....	421
cudaHostRegisterIoMemory.....	421
cudaHostRegisterMapped.....	421
cudaHostRegisterPortable.....	421
cudaInvalidDeviceId.....	421
cudaIpcMemLazyEnablePeerAccess.....	421
cudaMemAttachGlobal.....	421
cudaMemAttachHost.....	421
cudaMemAttachSingle.....	421
cudaNvSciSyncAttrSignal.....	422
cudaNvSciSyncAttrWait.....	422
cudaOccupancyDefault.....	422
cudaOccupancyDisableCachingOverride.....	422
cudaPeerAccessDefault.....	422
cudaStreamDefault.....	422
cudaStreamLegacy.....	422
cudaStreamNonBlocking.....	422
cudaStreamPerThread.....	422
<b>Chapter 6. Data Structures.....</b>	<b>424</b>
__cudaOccupancyB2DHelper.....	425
cudaChannelFormatDesc.....	425
f.....	425
w.....	425
x.....	425
y.....	425
z.....	425
cudaDeviceProp.....	425
asyncEngineCount.....	425
canMapHostMemory.....	426
canUseHostPointerForRegisteredMem.....	426
clockRate.....	426
computeMode.....	426
computePreemptionSupported.....	426
concurrentKernels.....	426
concurrentManagedAccess.....	426
cooperativeLaunch.....	426
cooperativeMultiDeviceLaunch.....	426
deviceOverlap.....	426
directManagedMemAccessFromHost.....	427
ECCEnabled.....	427
globalL1CacheSupported.....	427

hostNativeAtomicSupported.....	427
integrated.....	427
isMultiGpuBoard.....	427
kernelExecTimeoutEnabled.....	427
l2CacheSize.....	427
localL1CacheSupported.....	427
luid.....	427
luidDeviceNodeMask.....	427
major.....	428
managedMemory.....	428
maxGridSize.....	428
maxSurface1D.....	428
maxSurface1DLayered.....	428
maxSurface2D.....	428
maxSurface2DLayered.....	428
maxSurface3D.....	428
maxSurfaceCubemap.....	428
maxSurfaceCubemapLayered.....	428
maxTexture1D.....	428
maxTexture1DLayered.....	429
maxTexture1DLinear.....	429
maxTexture1DMipmap.....	429
maxTexture2D.....	429
maxTexture2DGather.....	429
maxTexture2DLayered.....	429
maxTexture2DLinear.....	429
maxTexture2DMipmap.....	429
maxTexture3D.....	429
maxTexture3DAlt.....	429
maxTextureCubemap.....	429
maxTextureCubemapLayered.....	430
maxThreadsDim.....	430
maxThreadsPerBlock.....	430
maxThreadsPerMultiProcessor.....	430
memoryBusWidth.....	430
memoryClockRate.....	430
memPitch.....	430
minor.....	430
multiGpuBoardGroupID.....	430
multiProcessorCount.....	430
name.....	430
pageableMemoryAccess.....	431
pageableMemoryAccessUsesHostPageTables.....	431

pciBusID.....	431
pciDeviceID.....	431
pciDomainID.....	431
regsPerBlock.....	431
regsPerMultiprocessor.....	431
sharedMemPerBlock.....	431
sharedMemPerBlockOptin.....	431
sharedMemPerMultiprocessor.....	431
singleToDoublePrecisionPerfRatio.....	432
streamPrioritiesSupported.....	432
surfaceAlignment.....	432
tccDriver.....	432
textureAlignment.....	432
texturePitchAlignment.....	432
totalConstMem.....	432
totalGlobalMem.....	432
unifiedAddressing.....	432
uuid.....	432
warpSize.....	432
cudaEglFrame.....	433
eglColorFormat.....	433
frameType.....	433
pArray.....	433
planeCount.....	433
planeDesc.....	433
pPitch.....	433
cudaEglPlaneDesc.....	433
channelDesc.....	434
depth.....	434
height.....	434
numChannels.....	434
pitch.....	434
reserved.....	434
width.....	434
cudaExtent.....	434
depth.....	434
height.....	434
width.....	435
cudaExternalMemoryBufferDesc.....	435
flags.....	435
offset.....	435
size.....	435
cudaExternalMemoryHandleDesc.....	435

fd.....	435
flags.....	435
handle.....	435
name.....	436
nvSciBufObject.....	436
size.....	436
type.....	436
win32.....	436
cudaExternalMemoryMipmappedArrayDesc.....	436
extent.....	437
flags.....	437
formatDesc.....	437
numLevels.....	437
offset.....	437
cudaExternalSemaphoreHandleDesc.....	437
fd.....	437
flags.....	437
handle.....	438
name.....	438
nvSciSyncObj.....	438
type.....	438
win32.....	438
cudaExternalSemaphoreSignalParams.....	438
fence.....	438
fence.....	439
flags.....	439
keyedMutex.....	439
value.....	439
cudaExternalSemaphoreWaitParams.....	439
fence.....	439
fence.....	439
flags.....	440
key.....	440
keyedMutex.....	440
timeoutMs.....	440
value.....	440
cudaFuncAttributes.....	440
binaryVersion.....	440
cacheModeCA.....	441
constSizeBytes.....	441
localSizeBytes.....	441
maxDynamicSharedSizeBytes.....	441
maxThreadsPerBlock.....	441



numRegs.....	441
preferredShmemCarveout.....	441
ptxVersion.....	441
sharedSizeBytes.....	442
cudaHostNodeParams.....	442
fn.....	442
userData.....	442
cudaIpcEventHandle_t.....	442
cudaIpcMemHandle_t.....	442
cudaKernelNodeParams.....	442
blockDim.....	442
extra.....	442
func.....	443
gridDim.....	443
kernelParams.....	443
sharedMemBytes.....	443
cudaLaunchParams.....	443
args.....	443
blockDim.....	443
func.....	443
gridDim.....	443
sharedMem.....	443
stream.....	443
cudaMemcpy3DParms.....	444
dstArray.....	444
dstPos.....	444
dstPtr.....	444
extent.....	444
kind.....	444
srcArray.....	444
srcPos.....	444
srcPtr.....	444
cudaMemcpy3DPeerParms.....	444
dstArray.....	445
dstDevice.....	445
dstPos.....	445
dstPtr.....	445
extent.....	445
srcArray.....	445
srcDevice.....	445
srcPos.....	445
srcPtr.....	445
cudaMemsetParams.....	445

dst.....	445
elementSize.....	446
height.....	446
pitch.....	446
value.....	446
width.....	446
cudaPitchedPtr.....	446
pitch.....	446
ptr.....	446
xsize.....	446
ysize.....	446
cudaPointerAttributes.....	447
device.....	447
devicePointer.....	447
hostPointer.....	447
isManaged.....	447
memoryType.....	447
type.....	448
cudaPos.....	448
x.....	448
y.....	448
z.....	448
cudaResourceDesc.....	448
array.....	448
desc.....	448
devPtr.....	448
height.....	449
mipmap.....	449
pitchInBytes.....	449
resType.....	449
sizeInBytes.....	449
width.....	449
cudaResourceViewDesc.....	449
depth.....	449
firstLayer.....	449
firstMipmapLevel.....	449
format.....	450
height.....	450
lastLayer.....	450
lastMipmapLevel.....	450
width.....	450
cudaTextureDesc.....	450
addressMode.....	450

borderColor.....	450
filterMode.....	450
maxAnisotropy.....	450
maxMipmapLevelClamp.....	451
minMipmapLevelClamp.....	451
mipmapFilterMode.....	451
mipmapLevelBias.....	451
normalizedCoords.....	451
readMode.....	451
sRGB.....	451
CUuuid_st.....	451
bytes.....	451
surfaceReference.....	451
channelDesc.....	452
textureReference.....	452
addressMode.....	452
channelDesc.....	452
filterMode.....	452
maxAnisotropy.....	452
maxMipmapLevelClamp.....	452
minMipmapLevelClamp.....	452
mipmapFilterMode.....	452
mipmapLevelBias.....	453
normalized.....	453
sRGB.....	453
<b>Chapter 7. Data Fields.....</b>	<b>454</b>
<b>Chapter 8. Deprecated List.....</b>	<b>466</b>



# Chapter 1.

## DIFFERENCE BETWEEN THE DRIVER AND RUNTIME APIS

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

### **Complexity vs. control**

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

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

### **Context management**

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

context or primary context, can be synchronized with `cudaDeviceSynchronize()`, and destroyed with `cudaDeviceReset()`.

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

# Chapter 2.

## API SYNCHRONIZATION BEHAVIOR

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

### **Memcpy**

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

#### **Synchronous**

1. All transfers involving Unified Memory regions are fully synchronous with respect to the host.
2. For transfers from pageable host memory to device memory, a stream sync is performed before the copy is initiated. The function will return once the pageable buffer has been copied to the staging memory for DMA transfer to device memory, but the DMA to final destination may not have completed.
3. For transfers from pinned host memory to device memory, the function is synchronous with respect to the host.
4. For transfers from device to either pageable or pinned host memory, the function returns only once the copy has completed.
5. For transfers from device memory to device memory, no host-side synchronization is performed.
6. For transfers from any host memory to any host memory, the function is fully synchronous with respect to the host.

#### **Asynchronous**

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

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

### **Memset**

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

### **Kernel Launches**

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



# Chapter 3.

## STREAM SYNCHRONIZATION BEHAVIOR

### Default stream

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

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

### Legacy default stream

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

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

```
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.

## MODULES

Here is a list of all modules:

- ▶ Device Management
- ▶ Thread Management [DEPRECATED]
- ▶ Error Handling
- ▶ Stream Management
- ▶ Event Management
- ▶ External Resource Interoperability
- ▶ Execution Control
- ▶ Occupancy
- ▶ Memory Management
- ▶ Memory Management [DEPRECATED]
- ▶ Unified Addressing
- ▶ Peer Device Memory Access
- ▶ OpenGL Interoperability
- ▶ OpenGL Interoperability [DEPRECATED]
- ▶ Direct3D 9 Interoperability
- ▶ Direct3D 9 Interoperability [DEPRECATED]
- ▶ Direct3D 10 Interoperability
- ▶ Direct3D 10 Interoperability [DEPRECATED]
- ▶ Direct3D 11 Interoperability
- ▶ Direct3D 11 Interoperability [DEPRECATED]
- ▶ VDPAU Interoperability
- ▶ EGL Interoperability
- ▶ Graphics Interoperability
- ▶ Texture Reference Management [DEPRECATED]
- ▶ Surface Reference Management [DEPRECATED]
- ▶ Texture Object Management

- ▶ Surface Object Management
- ▶ Version Management
- ▶ Graph Management
- ▶ C++ API Routines
- ▶ Interactions with the CUDA Driver API
- ▶ Profiler Control
- ▶ Data types used by CUDA Runtime

## 5.1. Device Management

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

**`__host__ cudaError_t cudaChooseDevice (int *device, const cudaDeviceProp *prop)`**

Select compute-device which best matches criteria.

### Parameters

#### **device**

- Device with best match

#### **prop**

- Desired device properties

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

### Description

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



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

**See also:**

[cudaGetDeviceCount](#), [cudaGetDevice](#), [cudaSetDevice](#), [cudaGetDeviceProperties](#)

## **\_\_host\_\_ \_\_device\_\_ cudaError\_t cudaDeviceGetAttribute (int \*value, cudaDeviceAttr attr, int device)**

Returns information about the device.

**Parameters****value**

- Returned device attribute value

**attr**

- Device attribute to query

**device**

- Device number to query

**Returns**

[cudaSuccess](#), [cudaErrorInvalidDevice](#), [cudaErrorInvalidValue](#)

**Description**

Returns in \*value the integer value of the attribute attr on device device. The supported attributes are:

- ▶ [cudaDevAttrMaxThreadsPerBlock](#): Maximum number of threads per block;
- ▶ [cudaDevAttrMaxBlockDimX](#): Maximum x-dimension of a block;
- ▶ [cudaDevAttrMaxBlockDimY](#): Maximum y-dimension of a block;
- ▶ [cudaDevAttrMaxBlockDimZ](#): Maximum z-dimension of a block;
- ▶ [cudaDevAttrMaxGridDimX](#): Maximum x-dimension of a grid;
- ▶ [cudaDevAttrMaxGridDimY](#): Maximum y-dimension of a grid;
- ▶ [cudaDevAttrMaxGridDimZ](#): Maximum z-dimension of a grid;
- ▶ [cudaDevAttrMaxSharedMemoryPerBlock](#): Maximum amount of shared memory available to a thread block in bytes;
- ▶ [cudaDevAttrTotalConstantMemory](#): Memory available on device for \_\_constant\_\_ variables in a CUDA C kernel in bytes;
- ▶ [cudaDevAttrWarpSize](#): Warp size in threads;
- ▶ [cudaDevAttrMaxPitch](#): Maximum pitch in bytes allowed by the memory copy functions that involve memory regions allocated through [cudaMallocPitch\(\)](#);
- ▶ [cudaDevAttrMaxTexture1DWidth](#): Maximum 1D texture width;
- ▶ [cudaDevAttrMaxTexture1DLinearWidth](#): Maximum width for a 1D texture bound to linear memory;

- ▶ `cudaDevAttrMaxTexture1DMipmappedWidth`: Maximum mipmapped 1D texture width;
- ▶ `cudaDevAttrMaxTexture2DWidth`: Maximum 2D texture width;
- ▶ `cudaDevAttrMaxTexture2DHeight`: Maximum 2D texture height;
- ▶ `cudaDevAttrMaxTexture2DLinearWidth`: Maximum width for a 2D texture bound to linear memory;
- ▶ `cudaDevAttrMaxTexture2DLinearHeight`: Maximum height for a 2D texture bound to linear memory;
- ▶ `cudaDevAttrMaxTexture2DLinearPitch`: Maximum pitch in bytes for a 2D texture bound to linear memory;
- ▶ `cudaDevAttrMaxTexture2DMipmappedWidth`: Maximum mipmapped 2D texture width;
- ▶ `cudaDevAttrMaxTexture2DMipmappedHeight`: Maximum mipmapped 2D texture height;
- ▶ `cudaDevAttrMaxTexture3DWidth`: Maximum 3D texture width;
- ▶ `cudaDevAttrMaxTexture3DHeight`: Maximum 3D texture height;
- ▶ `cudaDevAttrMaxTexture3DDepth`: Maximum 3D texture depth;
- ▶ `cudaDevAttrMaxTexture3DWidthAlt`: Alternate maximum 3D texture width, 0 if no alternate maximum 3D texture size is supported;
- ▶ `cudaDevAttrMaxTexture3DHeightAlt`: Alternate maximum 3D texture height, 0 if no alternate maximum 3D texture size is supported;
- ▶ `cudaDevAttrMaxTexture3DDepthAlt`: Alternate maximum 3D texture depth, 0 if no alternate maximum 3D texture size is supported;
- ▶ `cudaDevAttrMaxTextureCubemapWidth`: Maximum cubemap texture width or height;
- ▶ `cudaDevAttrMaxTexture1DLayeredWidth`: Maximum 1D layered texture width;
- ▶ `cudaDevAttrMaxTexture1DLayeredLayers`: Maximum layers in a 1D layered texture;
- ▶ `cudaDevAttrMaxTexture2DLayeredWidth`: Maximum 2D layered texture width;
- ▶ `cudaDevAttrMaxTexture2DLayeredHeight`: Maximum 2D layered texture height;
- ▶ `cudaDevAttrMaxTexture2DLayeredLayers`: Maximum layers in a 2D layered texture;
- ▶ `cudaDevAttrMaxTextureCubemapLayeredWidth`: Maximum cubemap layered texture width or height;
- ▶ `cudaDevAttrMaxTextureCubemapLayeredLayers`: Maximum layers in a cubemap layered texture;
- ▶ `cudaDevAttrMaxSurface1DWidth`: Maximum 1D surface width;
- ▶ `cudaDevAttrMaxSurface2DWidth`: Maximum 2D surface width;
- ▶ `cudaDevAttrMaxSurface2DHeight`: Maximum 2D surface height;
- ▶ `cudaDevAttrMaxSurface3DWidth`: Maximum 3D surface width;
- ▶ `cudaDevAttrMaxSurface3DHeight`: Maximum 3D surface height;

- ▶ `cudaDevAttrMaxSurface3DDepth`: Maximum 3D surface depth;
- ▶ `cudaDevAttrMaxSurface1DLayeredWidth`: Maximum 1D layered surface width;
- ▶ `cudaDevAttrMaxSurface1DLayeredLayers`: Maximum layers in a 1D layered surface;
- ▶ `cudaDevAttrMaxSurface2DLayeredWidth`: Maximum 2D layered surface width;
- ▶ `cudaDevAttrMaxSurface2DLayeredHeight`: Maximum 2D layered surface height;
- ▶ `cudaDevAttrMaxSurface2DLayeredLayers`: Maximum layers in a 2D layered surface;
- ▶ `cudaDevAttrMaxSurfaceCubemapWidth`: Maximum cubemap surface width;
- ▶ `cudaDevAttrMaxSurfaceCubemapLayeredWidth`: Maximum cubemap layered surface width;
- ▶ `cudaDevAttrMaxSurfaceCubemapLayeredLayers`: Maximum layers in a cubemap layered surface;
- ▶ `cudaDevAttrMaxRegistersPerBlock`: Maximum number of 32-bit registers available to a thread block;
- ▶ `cudaDevAttrClockRate`: Peak clock frequency in kilohertz;
- ▶ `cudaDevAttrTextureAlignment`: Alignment requirement; texture base addresses aligned to `textureAlign` bytes do not need an offset applied to texture fetches;
- ▶ `cudaDevAttrTexturePitchAlignment`: Pitch alignment requirement for 2D texture references bound to pitched memory;
- ▶ `cudaDevAttrGpuOverlap`: 1 if the device can concurrently copy memory between host and device while executing a kernel, or 0 if not;
- ▶ `cudaDevAttrMultiProcessorCount`: Number of multiprocessors on the device;
- ▶ `cudaDevAttrKernelExecTimeout`: 1 if there is a run time limit for kernels executed on the device, or 0 if not;
- ▶ `cudaDevAttrIntegrated`: 1 if the device is integrated with the memory subsystem, or 0 if not;
- ▶ `cudaDevAttrCanMapHostMemory`: 1 if the device can map host memory into the CUDA address space, or 0 if not;
- ▶ `cudaDevAttrComputeMode`: Compute mode is the compute mode that the device is currently in. Available modes are as follows:
  - ▶ `cudaComputeModeDefault`: Default mode - Device is not restricted and multiple threads can use `cudaSetDevice()` with this device.
  - ▶ `cudaComputeModeExclusive`: Compute-exclusive mode - Only one thread will be able to use `cudaSetDevice()` with this device.
  - ▶ `cudaComputeModeProhibited`: Compute-prohibited mode - No threads can use `cudaSetDevice()` with this device.
  - ▶ `cudaComputeModeExclusiveProcess`: Compute-exclusive-process mode - Many threads in one process will be able to use `cudaSetDevice()` with this device.
- ▶ `cudaDevAttrConcurrentKernels`: 1 if the device supports executing multiple kernels within the same context simultaneously, or 0 if not. It is not guaranteed that multiple



kernels will be resident on the device concurrently so this feature should not be relied upon for correctness;

- ▶ `cudaDevAttrEccEnabled`: 1 if error correction is enabled on the device, 0 if error correction is disabled or not supported by the device;
- ▶ `cudaDevAttrPciBusId`: PCI bus identifier of the device;
- ▶ `cudaDevAttrPciDeviceId`: PCI device (also known as slot) identifier of the device;
- ▶ `cudaDevAttrTccDriver`: 1 if the device is using a TCC driver. TCC is only available on Tesla hardware running Windows Vista or later;
- ▶ `cudaDevAttrMemoryClockRate`: Peak memory clock frequency in kilohertz;
- ▶ `cudaDevAttrGlobalMemoryBusWidth`: Global memory bus width in bits;
- ▶ `cudaDevAttrL2CacheSize`: Size of L2 cache in bytes. 0 if the device doesn't have L2 cache;
- ▶ `cudaDevAttrMaxThreadsPerMultiProcessor`: Maximum resident threads per multiprocessor;
- ▶ `cudaDevAttrUnifiedAddressing`: 1 if the device shares a unified address space with the host, or 0 if not;
- ▶ `cudaDevAttrComputeCapabilityMajor`: Major compute capability version number;
- ▶ `cudaDevAttrComputeCapabilityMinor`: Minor compute capability version number;
- ▶ `cudaDevAttrStreamPrioritiesSupported`: 1 if the device supports stream priorities, or 0 if not;
- ▶ `cudaDevAttrGlobalL1CacheSupported`: 1 if device supports caching globals in L1 cache, 0 if not;
- ▶ `cudaDevAttrLocalL1CacheSupported`: 1 if device supports caching locals in L1 cache, 0 if not;
- ▶ `cudaDevAttrMaxSharedMemoryPerMultiprocessor`: Maximum amount of shared memory available to a multiprocessor in bytes; this amount is shared by all thread blocks simultaneously resident on a multiprocessor;
- ▶ `cudaDevAttrMaxRegistersPerMultiprocessor`: Maximum number of 32-bit registers available to a multiprocessor; this number is shared by all thread blocks simultaneously resident on a multiprocessor;
- ▶ `cudaDevAttrManagedMemory`: 1 if device supports allocating managed memory, 0 if not;
- ▶ `cudaDevAttrIsMultiGpuBoard`: 1 if device is on a multi-GPU board, 0 if not;
- ▶ `cudaDevAttrMultiGpuBoardGroupID`: Unique identifier for a group of devices on the same multi-GPU board;
- ▶ `cudaDevAttrHostNativeAtomicSupported`: 1 if the link between the device and the host supports native atomic operations;
- ▶ `cudaDevAttrSingleToDoublePrecisionPerfRatio`: Ratio of single precision performance (in floating-point operations per second) to double precision performance;

- ▶ `cudaDevAttrPageableMemoryAccess`: 1 if the device supports coherently accessing pageable memory without calling `cudaHostRegister` on it, and 0 otherwise.
- ▶ `cudaDevAttrConcurrentManagedAccess`: 1 if the device can coherently access managed memory concurrently with the CPU, and 0 otherwise.
- ▶ `cudaDevAttrComputePreemptionSupported`: 1 if the device supports Compute Preemption, 0 if not.
- ▶ `cudaDevAttrCanUseHostPointerForRegisteredMem`: 1 if the device can access host registered memory at the same virtual address as the CPU, and 0 otherwise.
- ▶ `cudaDevAttrCooperativeLaunch`: 1 if the device supports launching cooperative kernels via `cudaLaunchCooperativeKernel`, and 0 otherwise.
- ▶ `cudaDevAttrCooperativeMultiDeviceLaunch`: 1 if the device supports launching cooperative kernels via `cudaLaunchCooperativeKernelMultiDevice`, and 0 otherwise.
- ▶ `cudaDevAttrCanFlushRemoteWrites`: 1 if the device supports flushing of outstanding remote writes, and 0 otherwise.
- ▶ `cudaDevAttrHostRegisterSupported`: 1 if the device supports host memory registration via `cudaHostRegister`, and 0 otherwise.
- ▶ `cudaDevAttrPageableMemoryAccessUsesHostPageTables`: 1 if the device accesses pageable memory via the host's page tables, and 0 otherwise.
- ▶ `cudaDevAttrDirectManagedMemAccessFromHost`: 1 if the host can directly access managed memory on the device without migration, and 0 otherwise.
- ▶ `cudaDevAttrMaxSharedMemoryPerBlockOptin`: Maximum per block shared memory size on the device. This value can be opted into when using `cudaFuncSetAttribute`



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

#### See also:

`cudaGetDeviceCount`, `cudaGetDevice`, `cudaSetDevice`, `cudaChooseDevice`, `cudaGetDeviceProperties`, `cuDeviceGetAttribute`

## `__host__ cudaError_t cudaDeviceGetByPCIBusId (int *device, const char *pciBusId)`

Returns a handle to a compute device.

### Parameters

#### **device**

- Returned device ordinal

#### **pciBusId**

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

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`

### Description

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



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

### See also:

`cudaDeviceGetPCIBusId`, `cuDeviceGetByPCIBusId`

## \_\_host\_\_ \_\_device\_\_ cudaError\_t cudaDeviceGetCacheConfig (cudaFuncCache \*pCacheConfig)

Returns the preferred cache configuration for the current device.

### Parameters

#### pCacheConfig

- Returned cache configuration

### Returns

cudaSuccess

### Description

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

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

The supported cache configurations are:

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



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

### See also:

[cudaDeviceSetCacheConfig](#), [cudaFuncSetCacheConfig](#) ( C API),  
[cudaFuncSetCacheConfig](#) ( C++ API), [cuCtxGetCacheConfig](#)

## `__host__ __device__ cudaError_t cudaDeviceGetLimit(size_t *pValue, cudaLimit limit)`

Returns resource limits.

### Parameters

#### **pValue**

- Returned size of the limit

#### **limit**

- Limit to query

### Returns

`cudaSuccess`, `cudaErrorUnsupportedLimit`, `cudaErrorInvalidValue`

### Description

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

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



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

### See also:

`cudaDeviceSetLimit`, `cuCtxGetLimit`

## `__host__ cudaError_t cudaDeviceGetNvSciSyncAttributes (void *nvSciSyncAttrList, int device, int flags)`

Return NvSciSync attributes that this device can support.

### Parameters

#### **nvSciSyncAttrList**

- Return NvSciSync attributes supported.

#### **device**

- Valid Cuda Device to get NvSciSync attributes for.

#### **flags**

- flags describing NvSciSync usage.

### Description

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

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

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

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

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

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

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

### See also:

`cudaImportExternalSemaphore`, `cudaDestroyExternalSemaphore`, `cudaSignalExternalSemaphoresAsync`, `cudaWaitExternalSemaphoresAsync`

## `__host__ cudaError_t cudaDeviceGetP2PAttribute (int *value, cudaDeviceP2PAttr attr, int srcDevice, int dstDevice)`

Queries attributes of the link between two devices.

### Parameters

#### **value**

- Returned value of the requested attribute

#### **attr**

#### **srcDevice**

- The source device of the target link.

#### **dstDevice**

- The destination device of the target link.

### Returns

`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorInvalidValue`

### Description

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

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

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

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



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

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

See also:

[cudaCtxEnablePeerAccess](#), [cudaCtxDisablePeerAccess](#), [cudaCtxCanAccessPeer](#), [cuDeviceGetP2PAttribute](#)

## **\_\_host\_\_ cudaError\_t cudaDeviceGetPCIBusId (char \*pciBusId, int len, int device)**

Returns a PCI Bus Id string for the device.

### Parameters

#### **pciBusId**

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

#### **len**

- Maximum length of string to store in name

#### **device**

- Device to get identifier string for

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidDevice](#)

### Description

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



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



See also:

[cudaDeviceGetByPCIBusId](#), [cuDeviceGetPCIBusId](#)

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

Returns the shared memory configuration for the current device.

### Parameters

#### `pConfig`

- Returned cache configuration

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

### Description

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

The returned bank configurations can be either:

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



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

See also:

[cudaDeviceSetCacheConfig](#), [cudaDeviceGetCacheConfig](#),  
[cudaDeviceSetSharedMemConfig](#), [cudaFuncSetCacheConfig](#),  
[cuCtxGetSharedMemConfig](#)

**\_\_host\_\_ cudaError\_t**

**cudaDeviceGetStreamPriorityRange (int \*leastPriority,  
int \*greatestPriority)**

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

### Parameters

#### **leastPriority**

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

#### **greatestPriority**

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

### Returns

[cudaSuccess](#)

### Description

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

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



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

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

#### See also:

[cudaStreamCreateWithPriority](#), [cudaStreamGetPriority](#), [cuCtxGetStreamPriorityRange](#)

## `__host__ cudaError_t cudaDeviceReset (void)`

Destroy all allocations and reset all state on the current device in the current process.

#### Returns

[cudaSuccess](#)

#### Description

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

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



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

#### See also:

[cudaDeviceSynchronize](#)

## `__host__ cudaError_t cudaDeviceSetCacheConfig(cudaFuncCache cacheConfig)`

Sets the preferred cache configuration for the current device.

### Parameters

#### **cacheConfig**

- Requested cache configuration

### Returns

`cudaSuccess`

### Description

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

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

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

The supported cache configurations are:

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



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

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

See also:

`cudaDeviceGetCacheConfig`, `cudaFuncSetCacheConfig` ( C API),  
`cudaFuncSetCacheConfig` ( C++ API), `cuCtxSetCacheConfig`

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

Set resource limits.

### Parameters

#### **limit**

- Limit to set

#### **value**

- Size of limit

### Returns

`cudaSuccess`, `cudaErrorUnsupportedLimit`, `cudaErrorInvalidValue`,  
`cudaErrorMemoryAllocation`

### Description

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

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

- `cudaLimitStackSize` controls the stack size in bytes of each GPU thread. Note that the CUDA driver will set the `limit` to the maximum of `value` and what the kernel function requires.
- `cudaLimitPrintfFifoSize` controls the size in bytes of the shared FIFO used by the `printf()` device system call. Setting `cudaLimitPrintfFifoSize` must not be performed after launching any kernel that uses the `printf()` device system call - in such case `cudaErrorInvalidValue` will be returned.
- `cudaLimitMallocHeapSize` controls the size in bytes of the heap used by the `malloc()` and `free()` device system calls. Setting `cudaLimitMallocHeapSize` must not be

performed after launching any kernel that uses the `malloc()` or `free()` device system calls - in such case `cudaErrorInvalidValue` will be returned.

- ▶ `cudaLimitDevRuntimeSyncDepth` controls the maximum nesting depth of a grid at which a thread can safely call `cudaDeviceSynchronize()`. Setting this limit must be performed before any launch of a kernel that uses the device runtime and calls `cudaDeviceSynchronize()` above the default sync depth, two levels of grids. Calls to `cudaDeviceSynchronize()` will fail with error code `cudaErrorSyncDepthExceeded` if the limitation is violated. This limit can be set smaller than the default or up the maximum launch depth of 24. When setting this limit, keep in mind that additional levels of sync depth require the runtime to reserve large amounts of device memory which can no longer be used for user allocations. If these reservations of device memory fail, `cudaDeviceSetLimit` will return `cudaErrorMemoryAllocation`, and the limit can be reset to a lower value. This limit is only applicable to devices of compute capability 3.5 and higher. Attempting to set this limit on devices of compute capability less than 3.5 will result in the error `cudaErrorUnsupportedLimit` being returned.
- ▶ `cudaLimitDevRuntimePendingLaunchCount` controls the maximum number of outstanding device runtime launches that can be made from the current device. A grid is outstanding from the point of launch up until the grid is known to have been completed. Device runtime launches which violate this limitation fail and return `cudaErrorLaunchPendingCountExceeded` when `cudaGetLastError()` is called after launch. If more pending launches than the default (2048 launches) are needed for a module using the device runtime, this limit can be increased. Keep in mind that being able to sustain additional pending launches will require the runtime to reserve larger amounts of device memory upfront which can no longer be used for allocations. If these reservations fail, `cudaDeviceSetLimit` will return `cudaErrorMemoryAllocation`, and the limit can be reset to a lower value. This limit is only applicable to devices of compute capability 3.5 and higher. Attempting to set this limit on devices of compute capability less than 3.5 will result in the error `cudaErrorUnsupportedLimit` being returned.
- ▶ `cudaLimitMaxL2FetchGranularity` controls the L2 cache fetch granularity. Values can range from 0B to 128B. This is purely a performance hint and it can be ignored or clamped depending on the platform.



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

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

See also:

[cudaDeviceGetLimit](#), [cuCtxSetLimit](#)

## `__host__ cudaError_t cudaDeviceSetSharedMemConfig(cudaSharedMemConfig config)`

Sets the shared memory configuration for the current device.

### Parameters

**config**

- Requested cache configuration

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

### Description

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

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

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

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

The supported bank configurations are:

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



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

#### See also:

`cudaDeviceSetCacheConfig`, `cudaDeviceGetCacheConfig`,  
`cudaDeviceGetSharedMemConfig`, `cudaFuncSetCacheConfig`,  
`cuCtxSetSharedMemConfig`

## `__host__ __device__ cudaError_t cudaDeviceSynchronize(void)`

Wait for compute device to finish.

#### Returns

`cudaSuccess`

#### Description

Blocks until the device has completed all preceding requested tasks.

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



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

#### See also:

`cudaDeviceReset`, `cuCtxSynchronize`



## `__host__ __device__ cudaError_t cudaGetDevice (int *device)`

Returns which device is currently being used.

### Parameters

#### `device`

- Returns the device on which the active host thread executes the device code.

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

### Description

Returns in `*device` the current device for the calling host thread.



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

### See also:

`cudaGetDeviceCount`, `cudaSetDevice`, `cudaGetDeviceProperties`, `cudaChooseDevice`, `cuCtxGetCurrent`

## `__host__ __device__ cudaError_t cudaGetDeviceCount (int *count)`

Returns the number of compute-capable devices.

### Parameters

#### `count`

- Returns the number of devices with compute capability greater or equal to 2.0

### Returns

`cudaErrorInvalidValue` (if a NULL device pointer is assigned), `cudaSuccess`

## Description

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



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

## See also:

`cudaGetDevice`, `cudaSetDevice`, `cudaGetDeviceProperties`, `cudaChooseDevice`, `cuDeviceGetCount`

## `__host__ cudaError_t cudaGetDeviceFlags (unsigned int *flags)`

Gets the flags for the current device.

## Parameters

**flags**

- Pointer to store the device flags

## Returns

`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorInvalidValue`

## Description

Returns in `flags` the flags for the current device. If there is a current device for the calling thread, and the device has been initialized or flags have been set on that device specifically, the flags for the device are returned. If there is no current device, but flags have been set for the thread with `cudaSetDeviceFlags`, the thread flags are returned. Finally, if there is no current device and no thread flags, the flags for the first device are returned, which may be the default flags. Compare to the behavior of `cudaSetDeviceFlags`.

Typically, the flags returned should match the behavior that will be seen if the calling thread uses a device after this call, without any change to the flags or current device

inbetween by this or another thread. Note that if the device is not initialized, it is possible for another thread to change the flags for the current device before it is initialized. Additionally, when using exclusive mode, if this thread has not requested a specific device, it may use a device other than the first device, contrary to the assumption made by this function.

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

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



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

#### See also:

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

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

Returns information about the compute-device.

### Parameters

#### **prop**

- Properties for the specified device

#### **device**

- Device number to get properties for

### Returns

`cudaSuccess`, `cudaErrorInvalidDevice`

## Description

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

```

struct cudaDeviceProp {
    char name[256];
    cudaUUID_t uuid;
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int regsPerBlock;
    int warpSize;
    size_t memPitch;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    int clockRate;
    size_t totalConstMem;
    int major;
    int minor;
    size_t textureAlignment;
    size_t texturePitchAlignment;
    int deviceOverlap;
    int multiProcessorCount;
    int kernelExecTimeoutEnabled;
    int integrated;
    int canMapHostMemory;
    int computeMode;
    int maxTexture1D;
    int maxTexture1DMipmap;
    int maxTexture1DLinear;
    int maxTexture2D[2];
    int maxTexture2DMipmap[2];
    int maxTexture2DLinear[3];
    int maxTexture2DGather[2];
    int maxTexture3D[3];
    int maxTexture3DAlt[3];
    int maxTextureCubemap;
    int maxTexture1DLayered[2];
    int maxTexture2DLayered[3];
    int maxTextureCubemapLayered[2];
    int maxSurface1D;
    int maxSurface2D[2];
    int maxSurface3D[3];
    int maxSurface1DLayered[2];
    int maxSurface2DLayered[3];
    int maxSurfaceCubemap;
    int maxSurfaceCubemapLayered[2];
    size_t surfaceAlignment;
    int concurrentKernels;
    int ECCEnabled;
    int pciBusID;
    int pciDeviceID;
    int pciDomainID;
    int tccDriver;
    int asyncEngineCount;
    int unifiedAddressing;
    int memoryClockRate;
    int memoryBusWidth;
    int l2CacheSize;
    int maxThreadsPerMultiProcessor;
    int streamPrioritiesSupported;
    int globalL1CacheSupported;
    int localL1CacheSupported;
    size_t sharedMemPerMultiprocessor;
    int regsPerMultiprocessor;
    int managedMemory;
    int isMultiGpuBoard;
    int multiGpuBoardGroupID;
    int singleToDoublePrecisionPerfRatio;
    int pageableMemoryAccess;
    int concurrentManagedAccess;
    int computePreemptionSupported;
    int canUseHostPointerForRegisteredMem;
    int cooperativeLaunch;

```

where:

- ▶ `name[256]` is an ASCII string identifying the device;
- ▶ `uuid` is a 16-byte unique identifier.
- ▶ `totalGlobalMem` is the total amount of global memory available on the device in bytes;
- ▶ `sharedMemPerBlock` is the maximum amount of shared memory available to a thread block in bytes;
- ▶ `regsPerBlock` is the maximum number of 32-bit registers available to a thread block;
- ▶ `warpSize` is the warp size in threads;
- ▶ `memPitch` is the maximum pitch in bytes allowed by the memory copy functions that involve memory regions allocated through `cudaMallocPitch()`;
- ▶ `maxThreadsPerBlock` is the maximum number of threads per block;
- ▶ `maxThreadsDim[3]` contains the maximum size of each dimension of a block;
- ▶ `maxGridSize[3]` contains the maximum size of each dimension of a grid;
- ▶ `clockRate` is the clock frequency in kilohertz;
- ▶ `totalConstMem` is the total amount of constant memory available on the device in bytes;
- ▶ `major`, `minor` are the major and minor revision numbers defining the device's compute capability;
- ▶ `textureAlignment` is the alignment requirement; texture base addresses that are aligned to `textureAlignment` bytes do not need an offset applied to texture fetches;
- ▶ `texturePitchAlignment` is the pitch alignment requirement for 2D texture references that are bound to pitched memory;
- ▶ `deviceOverlap` is 1 if the device can concurrently copy memory between host and device while executing a kernel, or 0 if not. Deprecated, use instead `asyncEngineCount`.
- ▶ `multiProcessorCount` is the number of multiprocessors on the device;
- ▶ `kernelExecTimeoutEnabled` is 1 if there is a run time limit for kernels executed on the device, or 0 if not.
- ▶ `integrated` is 1 if the device is an integrated (motherboard) GPU and 0 if it is a discrete (card) component.
- ▶ `canMapHostMemory` is 1 if the device can map host memory into the CUDA address space for use with `cudaHostAlloc()/cudaHostGetDevicePointer()`, or 0 if not;
- ▶ `computeMode` is the compute mode that the device is currently in. Available modes are as follows:
  - ▶ `cudaComputeModeDefault`: Default mode - Device is not restricted and multiple threads can use `cudaSetDevice()` with this device.
  - ▶ `cudaComputeModeExclusive`: Compute-exclusive mode - Only one thread will be able to use `cudaSetDevice()` with this device.

- ▶ `cudaComputeModeProhibited`: Compute-prohibited mode - No threads can use `cudaSetDevice()` with this device.
- ▶ `cudaComputeModeExclusiveProcess`: Compute-exclusive-process mode - Many threads in one process will be able to use `cudaSetDevice()` with this device.

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

- ▶ `maxTexture1D` is the maximum 1D texture size.
- ▶ `maxTexture1DMipmap` is the maximum 1D mipmapped texture texture size.
- ▶ `maxTexture1DLinear` is the maximum 1D texture size for textures bound to linear memory.
- ▶ `maxTexture2D[2]` contains the maximum 2D texture dimensions.
- ▶ `maxTexture2DMipmap[2]` contains the maximum 2D mipmapped texture dimensions.
- ▶ `maxTexture2DLinear[3]` contains the maximum 2D texture dimensions for 2D textures bound to pitch linear memory.
- ▶ `maxTexture2DGather[2]` contains the maximum 2D texture dimensions if texture gather operations have to be performed.
- ▶ `maxTexture3D[3]` contains the maximum 3D texture dimensions.
- ▶ `maxTexture3DAlt[3]` contains the maximum alternate 3D texture dimensions.
- ▶ `maxTextureCubemap` is the maximum cubemap texture width or height.
- ▶ `maxTexture1DLayered[2]` contains the maximum 1D layered texture dimensions.
- ▶ `maxTexture2DLayered[3]` contains the maximum 2D layered texture dimensions.
- ▶ `maxTextureCubemapLayered[2]` contains the maximum cubemap layered texture dimensions.
- ▶ `maxSurface1D` is the maximum 1D surface size.
- ▶ `maxSurface2D[2]` contains the maximum 2D surface dimensions.
- ▶ `maxSurface3D[3]` contains the maximum 3D surface dimensions.
- ▶ `maxSurface1DLayered[2]` contains the maximum 1D layered surface dimensions.
- ▶ `maxSurface2DLayered[3]` contains the maximum 2D layered surface dimensions.
- ▶ `maxSurfaceCubemap` is the maximum cubemap surface width or height.
- ▶ `maxSurfaceCubemapLayered[2]` contains the maximum cubemap layered surface dimensions.
- ▶ `surfaceAlignment` specifies the alignment requirements for surfaces.
- ▶ `concurrentKernels` is 1 if the device supports executing multiple kernels within the same context simultaneously, or 0 if not. It is not guaranteed that multiple kernels will be resident on the device concurrently so this feature should not be relied upon for correctness;

- ▶ `ECCEnabled` is 1 if the device has ECC support turned on, or 0 if not.
- ▶ `pciBusID` is the PCI bus identifier of the device.
- ▶ `pciDeviceID` is the PCI device (sometimes called slot) identifier of the device.
- ▶ `pciDomainID` is the PCI domain identifier of the device.
- ▶ `tccDriver` is 1 if the device is using a TCC driver or 0 if not.
- ▶ `asyncEngineCount` is 1 when the device can concurrently copy memory between host and device while executing a kernel. It is 2 when the device can concurrently copy memory between host and device in both directions and execute a kernel at the same time. It is 0 if neither of these is supported.
- ▶ `unifiedAddressing` is 1 if the device shares a unified address space with the host and 0 otherwise.
- ▶ `memoryClockRate` is the peak memory clock frequency in kilohertz.
- ▶ `memoryBusWidth` is the memory bus width in bits.
- ▶ `l2CacheSize` is L2 cache size in bytes.
- ▶ `maxThreadsPerMultiProcessor` is the number of maximum resident threads per multiprocessor.
- ▶ `streamPrioritiesSupported` is 1 if the device supports stream priorities, or 0 if it is not supported.
- ▶ `globalL1CacheSupported` is 1 if the device supports caching of globals in L1 cache, or 0 if it is not supported.
- ▶ `localL1CacheSupported` is 1 if the device supports caching of locals in L1 cache, or 0 if it is not supported.
- ▶ `sharedMemPerMultiprocessor` is the maximum amount of shared memory available to a multiprocessor in bytes; this amount is shared by all thread blocks simultaneously resident on a multiprocessor;
- ▶ `regsPerMultiprocessor` is the maximum number of 32-bit registers available to a multiprocessor; this number is shared by all thread blocks simultaneously resident on a multiprocessor;
- ▶ `managedMemory` is 1 if the device supports allocating managed memory on this system, or 0 if it is not supported.
- ▶ `isMultiGpuBoard` is 1 if the device is on a multi-GPU board (e.g. Gemini cards), and 0 if not;
- ▶ `multiGpuBoardGroupID` is a unique identifier for a group of devices associated with the same board. Devices on the same multi-GPU board will share the same identifier;
- ▶ `singleToDoublePrecisionPerfRatio` is the ratio of single precision performance (in floating-point operations per second) to double precision performance.
- ▶ `pageableMemoryAccess` is 1 if the device supports coherently accessing pageable memory without calling `cudaHostRegister` on it, and 0 otherwise.
- ▶ `concurrentManagedAccess` is 1 if the device can coherently access managed memory concurrently with the CPU, and 0 otherwise.

- ▶ `computePreemptionSupported` is 1 if the device supports Compute Preemption, and 0 otherwise.
- ▶ `canUseHostPointerForRegisteredMem` is 1 if the device can access host registered memory at the same virtual address as the CPU, and 0 otherwise.
- ▶ `cooperativeLaunch` is 1 if the device supports launching cooperative kernels via `cudaLaunchCooperativeKernel`, and 0 otherwise.
- ▶ `cooperativeMultiDeviceLaunch` is 1 if the device supports launching cooperative kernels via `cudaLaunchCooperativeKernelMultiDevice`, and 0 otherwise.
- ▶ `pageableMemoryAccessUsesHostPageTables` is 1 if the device accesses pageable memory via the host's page tables, and 0 otherwise.
- ▶ `directManagedMemAccessFromHost` is 1 if the host can directly access managed memory on the device without migration, and 0 otherwise.



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

#### See also:

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

## `__host__ cudaError_t cudalpcCloseMemHandle (void *devPtr)`

Close memory mapped with `cudaIpcOpenMemHandle`.

#### Parameters

##### `devPtr`

- Device pointer returned by `cudaIpcOpenMemHandle`

#### Returns

`cudaSuccess`, `cudaErrorMapBufferObjectFailed`, `cudaErrorInvalidResourceHandle`,  
`cudaErrorNotSupported`



## Description

Unmaps memory returned by `cudaIpcOpenMemHandle`. The original allocation in the exporting process as well as imported mappings in other processes will be unaffected.

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

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



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

## See also:

`cudaMalloc`, `cudaFree`, `cudaIpcGetEventHandle`, `cudaIpcOpenEventHandle`, `cudaIpcGetMemHandle`, `cudaIpcOpenMemHandle`, `cuIpcCloseMemHandle`

## `__host__ cudaError_t cudalpcGetEventHandle(cudaIpcEventHandle_t *handle, cudaEvent_t event)`

Gets an interprocess handle for a previously allocated event.

## Parameters

### handle

- Pointer to a user allocated `cudaIpcEventHandle` in which to return the opaque event handle

### event

- Event allocated with `cudaEventInterprocess` and `cudaEventDisableTiming` flags.

## Returns

`cudaSuccess`, `cudaErrorInvalidResourceHandle`, `cudaErrorMemoryAllocation`, `cudaErrorMapBufferObjectFailed`, `cudaErrorNotSupported`

## Description

Takes as input a previously allocated event. This event must have been created with the `cudaEventInterprocess` and `cudaEventDisableTiming` flags set. This opaque handle

may be copied into other processes and opened with `cudaIpcOpenEventHandle` to allow efficient hardware synchronization between GPU work in different processes.

After the event has been opened in the importing process, `cudaEventRecord`, `cudaEventSynchronize`, `cudaStreamWaitEvent` and `cudaEventQuery` may be used in either process. Performing operations on the imported event after the exported event has been freed with `cudaEventDestroy` will result in undefined behavior.

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



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

See also:

`cudaEventCreate`, `cudaEventDestroy`, `cudaEventSynchronize`, `cudaEventQuery`, `cudaStreamWaitEvent`, `cudaIpcOpenEventHandle`, `cudaIpcGetMemHandle`, `cudaIpcOpenMemHandle`, `cudaIpcCloseMemHandle`, `cuIpcGetEventHandle`

## `__host__ cudaError_t cudalpcGetMemHandle (cudalpcMemHandle_t *handle, void *devPtr)`

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

### Parameters

#### **handle**

- Pointer to user allocated `cudalpcMemHandle` to return the handle in.

#### **devPtr**

- Base pointer to previously allocated device memory

### Returns

`cudaSuccess`, `cudaErrorInvalidResourceHandle`, `cudaErrorMemoryAllocation`, `cudaErrorMapBufferObjectFailed`, `cudaErrorNotSupported`

## Description

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

If a region of memory is freed with `cudaFree` and a subsequent call to `cudaMalloc` returns memory with the same device address, `cudaIpcGetMemHandle` will return a unique handle for the new memory.

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



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

## See also:

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

## `__host__ cudaError_t cudalpcOpenEventHandle(cudaEvent_t *event, cudalpcEventHandle_t handle)`

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

## Parameters

### **event**

- Returns the imported event

### **handle**

- Interprocess handle to open

## Returns

`cudaSuccess`, `cudaErrorMapBufferObjectFailed`, `cudaErrorInvalidResourceHandle`, `cudaErrorNotSupported`

## Description

Opens an interprocess event handle exported from another process with `cudaIpcGetEventHandle`. This function returns a `cudaEvent_t` that behaves like a locally

created event with the `cudaEventDisableTiming` flag specified. This event must be freed with `cudaEventDestroy`.

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

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



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

See also:

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

## `__host__ cudaError_t cudalpcOpenMemHandle (void **devPtr, cudalpcMemHandle_t handle, unsigned int flags)`

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

### Parameters

#### **devPtr**

- Returned device pointer

#### **handle**

- `cudalpcMemHandle` to open

#### **flags**

- Flags for this operation. Must be specified as `cudaIpcMemLazyEnablePeerAccess`

### Returns

`cudaSuccess`, `cudaErrorMapBufferObjectFailed`, `cudaErrorInvalidResourceHandle`, `cudaErrorTooManyPeers`, `cudaErrorNotSupported`

## Description

Maps memory exported from another process with `cudaIpcGetMemHandle` into the current device address space. For contexts on different devices `cudaIpcOpenMemHandle` can attempt to enable peer access between the devices as if the user called `cudaDeviceEnablePeerAccess`. This behavior is controlled by the `cudaIpcMemLazyEnablePeerAccess` flag. `cudaDeviceCanAccessPeer` can determine if a mapping is possible.

`cudaIpcOpenMemHandle` can open handles to devices that may not be visible in the process calling the API.

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

Memory returned from `cudaIpcOpenMemHandle` must be freed with `cudaIpcCloseMemHandle`.

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

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



- ▶ Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- ▶ Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.
- ▶ No guarantees are made about the address returned in `*devPtr`. In particular, multiple processes may not receive the same address for the same `handle`.

## See also:

`cudaMalloc`, `cudaFree`, `cudaIpcGetEventHandle`, `cudaIpcOpenEventHandle`, `cudaIpcGetMemHandle`, `cudaIpcCloseMemHandle`, `cudaDeviceEnablePeerAccess`, `cudaDeviceCanAccessPeer`, `cuIpcOpenMemHandle`

## `__host__ cudaError_t cudaSetDevice (int device)`

Set device to be used for GPU executions.

### Parameters

#### **device**

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

### Returns

`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorDeviceAlreadyInUse`

### Description

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

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

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



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

### See also:

`cudaGetDeviceCount`, `cudaGetDevice`, `cudaGetDeviceProperties`, `cudaChooseDevice`, `cuCtxSetCurrent`

## `__host__ cudaError_t cudaSetDeviceFlags (unsigned int flags)`

Sets flags to be used for device executions.

### Parameters

#### `flags`

- Parameters for device operation

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorSetOnActiveProcess`

### Description

Records `flags` as the flags to use when initializing the current device. If no device has been made current to the calling thread, then `flags` will be applied to the initialization of any device initialized by the calling host thread, unless that device has had its initialization flags set explicitly by this or any host thread.

If the current device has been set and that device has already been initialized then this call will fail with the error `cudaErrorSetOnActiveProcess`. In this case it is necessary to reset device using `cudaDeviceReset()` before the device's initialization flags may be set.

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

- ▶ `cudaDeviceScheduleAuto`: The default value if the `flags` parameter is zero, uses a heuristic based on the number of active CUDA contexts in the process `C` and the number of logical processors in the system `P`. If  $C > P$ , then CUDA will yield to other OS threads when waiting for the device, otherwise CUDA will not yield while waiting for results and actively spin on the processor. Additionally, on Tegra devices, `cudaDeviceScheduleAuto` uses a heuristic based on the power profile of the platform and may choose `cudaDeviceScheduleBlockingSync` for low-powered devices.
- ▶ `cudaDeviceScheduleSpin`: Instruct CUDA to actively spin when waiting for results from the device. This can decrease latency when waiting for the device, but may lower the performance of CPU threads if they are performing work in parallel with the CUDA thread.
- ▶ `cudaDeviceScheduleYield`: Instruct CUDA to yield its thread when waiting for results from the device. This can increase latency when waiting for the device, but can increase the performance of CPU threads performing work in parallel with the device.
- ▶ `cudaDeviceScheduleBlockingSync`: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the device to finish work.

- ▶ `cudaDeviceBlockingSync`: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the device to finish work.  
**Deprecated:** This flag was deprecated as of CUDA 4.0 and replaced with `cudaDeviceScheduleBlockingSync`.
- ▶ `cudaDeviceMapHost`: This flag enables allocating pinned host memory that is accessible to the device. It is implicit for the runtime but may be absent if a context is created using the driver API. If this flag is not set, `cudaHostGetDevicePointer()` will always return a failure code.
- ▶ `cudaDeviceLmemResizeToMax`: Instruct CUDA to not reduce local memory after resizing local memory for a kernel. This can prevent thrashing by local memory allocations when launching many kernels with high local memory usage at the cost of potentially increased memory usage.



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

#### See also:

`cudaGetDeviceFlags`, `cudaGetDeviceCount`, `cudaGetDevice`, `cudaGetDeviceProperties`, `cudaSetDevice`, `cudaSetValidDevices`, `cudaChooseDevice`, `cuDevicePrimaryCtxSetFlags`

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

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

#### Parameters

##### `device_arr`

- List of devices to try

##### `len`

- Number of devices in specified list

#### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`



## Description

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



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

## See also:

`cudaGetDeviceCount`, `cudaSetDevice`, `cudaGetDeviceProperties`, `cudaSetDeviceFlags`, `cudaChooseDevice`

## 5.2. Thread Management [DEPRECATED]

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

### `__host__ cudaError_t cudaThreadExit (void)`

Exit and clean up from CUDA launches.

## Returns

`cudaSuccess`

## Description

Deprecated

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

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

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



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

See also:

`cudaDeviceReset`

## `__host__ cudaError_t cudaThreadGetCacheConfig(cudaFuncCache *pCacheConfig)`

Returns the preferred cache configuration for the current device.

### Parameters

**pCacheConfig**

- Returned cache configuration

### Returns

`cudaSuccess`

### Description

Deprecated

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

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

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

The supported cache configurations are:

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



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

See also:

`cudaDeviceGetCacheConfig`

## `__host__ cudaError_t cudaThreadGetLimit (size_t *pValue, cudaLimit limit)`

Returns resource limits.

### Parameters

#### **pValue**

- Returned size in bytes of limit

#### **limit**

- Limit to query

### Returns

`cudaSuccess`, `cudaErrorUnsupportedLimit`, `cudaErrorInvalidValue`

### Description

Deprecated

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

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

- ▶ `cudaLimitStackSize`: stack size of each GPU thread;
- ▶ `cudaLimitPrintfFifoSize`: size of the shared FIFO used by the `printf()` device system call.
- ▶ `cudaLimitMallocHeapSize`: size of the heap used by the `malloc()` and `free()` device system calls;



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

See also:

`cudaDeviceGetLimit`

## `__host__ cudaError_t cudaThreadSetCacheConfig(cudaFuncCache cacheConfig)`

Sets the preferred cache configuration for the current device.

### Parameters

#### `cacheConfig`

- Requested cache configuration

### Returns

`cudaSuccess`

### Description

#### Deprecated

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

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

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

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

The supported cache configurations are:

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



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

See also:

`cudaDeviceSetCacheConfig`

## `__host__ cudaError_t cudaThreadSetLimit (cudaLimit limit, size_t value)`

Set resource limits.

### Parameters

#### **limit**

- Limit to set

**value**

- Size in bytes of limit

**Returns**

[cudaSuccess](#), [cudaErrorUnsupportedLimit](#), [cudaErrorInvalidValue](#)

**Description****Deprecated**

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

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

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

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



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

**See also:**

[cudaDeviceSetLimit](#)

## `__host__ cudaError_t cudaThreadSynchronize (void)`

Wait for compute device to finish.

### Returns

`cudaSuccess`

### Description

#### Deprecated

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

Blocks until the device has completed all preceding requested tasks.

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



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

### See also:

`cudaDeviceSynchronize`

## 5.3. Error Handling

This section describes the error handling functions of the CUDA runtime application programming interface.

## `__host__ __device__ const char *cudaGetErrorName(cudaError_t error)`

Returns the string representation of an error code enum name.

### Parameters

#### **error**

- Error code to convert to string

### Returns

`char*` pointer to a NULL-terminated string

### Description

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

### See also:

[cudaGetErrorString](#), [cudaGetLastError](#), [cudaPeekAtLastError](#), [cudaError](#), [cuGetErrorName](#)

## `__host__ __device__ const char *cudaGetErrorString(cudaError_t error)`

Returns the description string for an error code.

### Parameters

#### **error**

- Error code to convert to string

### Returns

`char*` pointer to a NULL-terminated string

### Description

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

### See also:

[cudaGetErrorName](#), [cudaGetLastError](#), [cudaPeekAtLastError](#), [cudaError](#), [cuGetErrorString](#)



## `__host__ __device__ cudaError_t cudaGetLastError (void)`

Returns the last error from a runtime call.

### Returns

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

### Description

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



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

### See also:

`cudaPeekAtLastError`, `cudaGetErrorName`, `cudaGetErrorString`, `cudaError`

## `__host____device__cudaError_t cudaPeekAtLastError (void)`

Returns the last error from a runtime call.

### Returns

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

### Description

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



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

### See also:

`cudaGetLastError`, `cudaGetErrorName`, `cudaGetErrorString`, `cudaError`

## 5.4. Stream Management

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

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

Type of stream callback functions.

```
__host__ cudaError_t cudaStreamAddCallback
(cudaStream_t stream, cudaStreamCallback_t callback,
void *userData, unsigned int flags)
```

Add a callback to a compute stream.

### Parameters

#### **stream**

- Stream to add callback to

#### **callback**

- The function to call once preceding stream operations are complete

#### **userData**

- User specified data to be passed to the callback function

#### **flags**

- Reserved for future use, must be 0

### Returns

[cudaSuccess](#), [cudaErrorInvalidResourceHandle](#), [cudaErrorInvalidValue](#),  
[cudaErrorNotSupported](#)

### Description



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

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

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

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

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

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



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

See also:

`cudaStreamCreate`, `cudaStreamCreateWithFlags`, `cudaStreamQuery`,  
`cudaStreamSynchronize`, `cudaStreamWaitEvent`, `cudaStreamDestroy`,

`cudaMallocManaged`, `cudaStreamAttachMemAsync`, `cudaLaunchHostFunc`,  
`cuStreamAddCallback`

**`__host__ cudaError_t cudaStreamAttachMemAsync`**  
**`(cudaStream_t stream, void *devPtr, size_t length,`**  
**`unsigned int flags)`**

Attach memory to a stream asynchronously.

### Parameters

#### **stream**

- Stream in which to enqueue the attach operation

#### **devPtr**

- Pointer to memory (must be a pointer to managed memory or to a valid host-accessible region of system-allocated memory)

#### **length**

- Length of memory (defaults to zero)

#### **flags**

- Must be one of `cudaMemAttachGlobal`, `cudaMemAttachHost` or `cudaMemAttachSingle` (defaults to `cudaMemAttachSingle`)

### Returns

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

### Description

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

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

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

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

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

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

When memory is associated with a single stream, the Unified Memory system will allow CPU access to this memory region so long as all operations in `stream` have completed, regardless of whether other streams are active. In effect, this constrains exclusive ownership of the managed memory region by an active GPU to per-stream activity instead of whole-GPU activity.

Accessing memory on the device from streams that are not associated with it will produce undefined results. No error checking is performed by the Unified Memory system to ensure that kernels launched into other streams do not access this region.

It is a program's responsibility to order calls to `cudaStreamAttachMemAsync` via events, synchronization or other means to ensure legal access to memory at all times. Data visibility and coherency will be changed appropriately for all kernels which follow a stream-association change.

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



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

**See also:**

[cudaStreamCreate](#), [cudaStreamCreateWithFlags](#), [cudaStreamWaitEvent](#),  
[cudaStreamSynchronize](#), [cudaStreamAddCallback](#), [cudaStreamDestroy](#),  
[cudaMallocManaged](#), [cuStreamAttachMemAsync](#)

## **\_\_host\_\_ cudaError\_t cudaStreamBeginCapture (cudaStream\_t stream, cudaStreamCaptureMode mode)**

Begins graph capture on a stream.

**Parameters****stream**

- Stream in which to initiate capture

**mode**

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

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#)

**Description**

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

If `mode` is not [cudaStreamCaptureModeRelaxed](#), [cudaStreamEndCapture](#) must be called on this stream from the same thread.



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

**See also:**

[cudaStreamCreate](#), [cudaStreamIsCapturing](#), [cudaStreamEndCapture](#),  
[cudaThreadExchangeStreamCaptureMode](#)

## `__host__ cudaError_t cudaStreamCreate (cudaStream_t *pStream)`

Create an asynchronous stream.

### Parameters

#### `pStream`

- Pointer to new stream identifier

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

### Description

Creates a new asynchronous stream.



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

### See also:

[cudaStreamCreateWithPriority](#), [cudaStreamCreateWithFlags](#), [cudaStreamGetPriority](#),  
[cudaStreamGetFlags](#), [cudaStreamQuery](#), [cudaStreamSynchronize](#),  
[cudaStreamWaitEvent](#), [cudaStreamAddCallback](#), [cudaStreamDestroy](#), [cuStreamCreate](#)



## `__host__ __device__ cudaError_t cudaStreamCreateWithFlags (cudaStream_t *pStream, unsigned int flags)`

Create an asynchronous stream.

### Parameters

#### `pStream`

- Pointer to new stream identifier

#### `flags`

- Parameters for stream creation

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

### Description

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

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



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

### See also:

`cudaStreamCreate`, `cudaStreamCreateWithPriority`, `cudaStreamGetFlags`,  
`cudaStreamQuery`, `cudaStreamSynchronize`, `cudaStreamWaitEvent`,  
`cudaStreamAddCallback`, `cudaStreamDestroy`, `cuStreamCreate`

## `__host__ cudaError_t cudaStreamCreateWithPriority(cudaStream_t *pStream, unsigned int flags, int priority)`

Create an asynchronous stream with the specified priority.

### Parameters

#### **pStream**

- Pointer to new stream identifier

#### **flags**

- Flags for stream creation. See [cudaStreamCreateWithFlags](#) for a list of valid flags that can be passed

#### **priority**

- Priority of the stream. Lower numbers represent higher priorities. See [cudaDeviceGetStreamPriorityRange](#) for more information about the meaningful stream priorities that can be passed.

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

### Description

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

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



- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ Note that this function may also return [cudaErrorInitializationError](#), [cudaErrorInsufficientDriver](#) or [cudaErrorNoDevice](#) if this call tries to initialize internal CUDA RT state.
- ▶ Note that as specified by [cudaStreamAddCallback](#) no CUDA function may be called from callback. [cudaErrorNotPermitted](#) may, but is not guaranteed to, be returned as a diagnostic in such case.
- ▶ Stream priorities are supported only on GPUs with compute capability 3.5 or higher.

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

See also:

[cudaStreamCreate](#), [cudaStreamCreateWithFlags](#), [cudaDeviceGetStreamPriorityRange](#), [cudaStreamGetPriority](#), [cudaStreamQuery](#), [cudaStreamWaitEvent](#), [cudaStreamAddCallback](#), [cudaStreamSynchronize](#), [cudaStreamDestroy](#), [cuStreamCreateWithPriority](#)

## `__host__ __device__ cudaError_t cudaStreamDestroy(cudaStream_t stream)`

Destroys and cleans up an asynchronous stream.

### Parameters

**stream**

- Stream identifier

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidResourceHandle](#)

### Description

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

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



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

See also:

[cudaStreamCreate](#), [cudaStreamCreateWithFlags](#), [cudaStreamQuery](#),  
[cudaStreamWaitEvent](#), [cudaStreamSynchronize](#), [cudaStreamAddCallback](#),  
[cuStreamDestroy](#)

## **\_\_host\_\_ cudaError\_t cudaStreamEndCapture (cudaStream\_t stream, cudaGraph\_t \*pGraph)**

Ends capture on a stream, returning the captured graph.

### **Parameters**

#### **stream**

- Stream to query

#### **pGraph**

- The captured graph

### **Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorStreamCaptureWrongThread](#)

### **Description**

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

If the `mode` argument to [cudaStreamBeginCapture](#) was not `cudaStreamCaptureModeRelaxed`, this call must be from the same thread as [cudaStreamBeginCapture](#).



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

### **See also:**

[cudaStreamCreate](#), [cudaStreamBeginCapture](#), [cudaStreamIsCapturing](#)

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

Query capture status of a stream.

### Parameters

#### **stream**

- Stream to query

#### **pCaptureStatus**

- Returns the stream's capture status

#### **pId**

- Returns the unique id of the capture sequence

### Returns

[cudaSuccess](#), [cudaErrorStreamCaptureImplicit](#)

### Description

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

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

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

- ▶ the call returns [cudaSuccess](#)
- ▶ captureStatus is set to [cudaStreamCaptureStatusActive](#)



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

### See also:

[cudaStreamBeginCapture](#), [cudaStreamIsCapturing](#)

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

Query the flags of a stream.

### Parameters

#### **hStream**

- Handle to the stream to be queried

#### **flags**

- Pointer to an unsigned integer in which the stream's flags are returned

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`

### Description

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



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

### See also:

`cudaStreamCreateWithPriority`, `cudaStreamCreateWithFlags`, `cudaStreamGetPriority`, `cuStreamGetFlags`

## `__host__ cudaError_t cudaStreamGetPriority(cudaStream_t hStream, int *priority)`

Query the priority of a stream.

### Parameters

#### **hStream**

- Handle to the stream to be queried

#### **priority**

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

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`

### Description

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



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

### See also:

`cudaStreamCreateWithPriority`, `cudaDeviceGetStreamPriorityRange`, `cudaStreamGetFlags`, `cuStreamGetPriority`

## `__host__ cudaError_t cudaStreamIsCapturing(cudaStream_t stream, cudaStreamCaptureStatus *pCaptureStatus)`

Returns a stream's capture status.

### Parameters

#### **stream**

- Stream to query

#### **pCaptureStatus**

- Returns the stream's capture status

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorStreamCaptureImplicit`

### Description

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

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

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

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



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

### See also:

`cudaStreamCreate`, `cudaStreamBeginCapture`, `cudaStreamEndCapture`



## `__host__ cudaError_t cudaStreamQuery (cudaStream_t stream)`

Queries an asynchronous stream for completion status.

### Parameters

**stream**

- Stream identifier

### Returns

`cudaSuccess`, `cudaErrorNotReady`, `cudaErrorInvalidResourceHandle`

### Description

Returns `cudaSuccess` if all operations in `stream` have completed, or `cudaErrorNotReady` if not.

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



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

### See also:

`cudaStreamCreate`, `cudaStreamCreateWithFlags`, `cudaStreamWaitEvent`, `cudaStreamSynchronize`, `cudaStreamAddCallback`, `cudaStreamDestroy`, `cuStreamQuery`

## `__host__ cudaError_t cudaStreamSynchronize(cudaStream_t stream)`

Waits for stream tasks to complete.

### Parameters

**stream**

- Stream identifier

### Returns

`cudaSuccess`, `cudaErrorInvalidResourceHandle`

### Description

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



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

### See also:

`cudaStreamCreate`, `cudaStreamCreateWithFlags`, `cudaStreamQuery`, `cudaStreamWaitEvent`, `cudaStreamAddCallback`, `cudaStreamDestroy`, `cuStreamSynchronize`

## `__host____device__cudaError_t cudaStreamWaitEvent(cudaStream_t stream, cudaEvent_t event, unsigned int flags)`

Make a compute stream wait on an event.

### Parameters

#### **stream**

- Stream to wait

#### **event**

- Event to wait on

#### **flags**

- Parameters for the operation (must be 0)

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`

### Description

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



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

### See also:

`cudaStreamCreate`, `cudaStreamCreateWithFlags`, `cudaStreamQuery`, `cudaStreamSynchronize`, `cudaStreamAddCallback`, `cudaStreamDestroy`, `cuStreamWaitEvent`

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

Swaps the stream capture interaction mode for a thread.

### Parameters

#### `mode`

- Pointer to mode value to swap with the current mode

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

### Description

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

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

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

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

A thread's mode is one of the following:

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

prohibited from potentially unsafe API calls. Concurrent capture sequences in other threads are ignored.

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



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

**See also:**

`cudaStreamBeginCapture`

## 5.5. Event Management

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

### `__host__ cudaError_t cudaEventCreate (cudaEvent_t *event)`

Creates an event object.

#### Parameters

**event**

- Newly created event

#### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorLaunchFailure`,  
`cudaErrorMemoryAllocation`

#### Description

Creates an event object for the current device using `cudaEventDefault`.



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

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

#### See also:

`cudaEventCreate` ( C++ API), `cudaEventCreateWithFlags`, `cudaEventRecord`, `cudaEventQuery`, `cudaEventSynchronize`, `cudaEventDestroy`, `cudaEventElapsedTime`, `cudaStreamWaitEvent`, `cuEventCreate`

## `__host__ __device__ cudaError_t cudaEventCreateWithFlags (cudaEvent_t *event, unsigned int flags)`

Creates an event object with the specified flags.

#### Parameters

##### `event`

- Newly created event

##### `flags`

- Flags for new event

#### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorLaunchFailure`, `cudaErrorMemoryAllocation`

#### Description

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

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

- ▶ `cudaEventInterprocess`: Specifies that the created event may be used as an interprocess event by `cudaIpcGetEventHandle()`. `cudaEventInterprocess` must be specified along with `cudaEventDisableTiming`.



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

#### See also:

`cudaEventCreate` ( C API), `cudaEventSynchronize`, `cudaEventDestroy`, `cudaEventElapsedTime`, `cudaStreamWaitEvent`, `cuEventCreate`

## `__host__ __device__ cudaError_t cudaEventDestroy(cudaEvent_t event)`

Destroys an event object.

### Parameters

#### `event`

- Event to destroy

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorLaunchFailure`

### Description

Destroys the event specified by `event`.

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



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

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

#### See also:

`cudaEventCreate` ( C API), `cudaEventCreateWithFlags`, `cudaEventQuery`, `cudaEventSynchronize`, `cudaEventRecord`, `cudaEventElapsedTime`, `cuEventDestroy`

## `__host__ cudaError_t cudaEventElapsedTime (float *ms, cudaEvent_t start, cudaEvent_t end)`

Computes the elapsed time between events.

#### Parameters

**ms**

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

**start**

- Starting event

**end**

- Ending event

#### Returns

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

#### Description

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

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

If `cudaEventRecord()` has not been called on either event, then `cudaErrorInvalidResourceHandle` is returned. If `cudaEventRecord()` has been called on both events but one or both of them has not yet been completed (that is, `cudaEventQuery()` would return `cudaErrorNotReady` on at least



one of the events), `cudaErrorNotReady` is returned. If either event was created with the `cudaEventDisableTiming` flag, then this function will return `cudaErrorInvalidResourceHandle`.



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

#### See also:

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

## `__host__ cudaError_t cudaEventQuery (cudaEvent_t event)`

Queries an event's status.

### Parameters

#### **event**

- Event to query

### Returns

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

### Description

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

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

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



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

### See also:

`cudaEventCreate` ( C API), `cudaEventCreateWithFlags`, `cudaEventRecord`, `cudaEventSynchronize`, `cudaEventDestroy`, `cudaEventElapsedTime`, `cuEventQuery`

## `__host__ __device__ cudaError_t cudaEventRecord (cudaEvent_t event, cudaStream_t stream)`

Records an event.

### Parameters

#### **event**

- Event to record

#### **stream**

- Stream in which to record event

### Returns

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

### Description

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

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



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

### See also:

[cudaEventCreate](#) ( C API), [cudaEventCreateWithFlags](#), [cudaEventQuery](#), [cudaEventSynchronize](#), [cudaEventDestroy](#), [cudaEventElapsedTime](#), [cudaStreamWaitEvent](#), [cuEventRecord](#)

## `__host__ cudaError_t cudaEventSynchronize(cudaEvent_t event)`

Waits for an event to complete.

### Parameters

#### `event`

- Event to wait for

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidResourceHandle](#), [cudaErrorLaunchFailure](#)

### Description

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

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



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

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

#### See also:

`cudaEventCreate` ( C API), `cudaEventCreateWithFlags`, `cudaEventRecord`, `cudaEventQuery`, `cudaEventDestroy`, `cudaEventElapsedTime`, `cuEventSynchronize`

## 5.6. External Resource Interoperability

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

### `__host__ cudaError_t cudaDestroyExternalMemory(cudaExternalMemory_t extMem)`

Destroys an external memory object.

#### Parameters

##### `extMem`

- External memory object to be destroyed

#### Returns

`cudaSuccess`, `cudaErrorInvalidResourceHandle`

#### Description

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



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

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

#### See also:

[cudaImportExternalMemory](#) [cudaExternalMemoryGetMappedBuffer](#),  
[cudaExternalMemoryGetMappedMipmappedArray](#)

## `__host__ cudaError_t cudaDestroyExternalSemaphore(cudaExternalSemaphore_t extSem)`

Destroys an external semaphore.

#### Parameters

**extSem**

- External semaphore to be destroyed

#### Returns

[cudaSuccess](#), [cudaErrorInvalidResourceHandle](#)

#### Description

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



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

#### See also:

[cudaImportExternalSemaphore](#), [cudaSignalExternalSemaphoresAsync](#),  
[cudaWaitExternalSemaphoresAsync](#)

## `__host__ cudaError_t`

`cudaExternalMemoryGetMappedBuffer (void  
 **devPtr, cudaExternalMemory_t extMem, const  
 cudaExternalMemoryBufferDesc *bufferDesc)`

Maps a buffer onto an imported memory object.

### Parameters

#### **devPtr**

- Returned device pointer to buffer

#### **extMem**

- Handle to external memory object

#### **bufferDesc**

- Buffer descriptor

### Returns

`cudaSuccess`, `cudaErrorInvalidResourceHandle`

### Description

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

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

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

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

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

The returned pointer `devPtr` must be freed using `cudaFree`.



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

#### See also:

`cudaImportExternalMemory` `cudaDestroyExternalMemory`,  
`cudaExternalMemoryGetMappedMipmappedArray`

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

Maps a CUDA mipmapped array onto an external memory object.

#### Parameters

##### **mipmap**

- Returned CUDA mipmapped array

##### **extMem**

- Handle to external memory object

##### **mipmapDesc**

- CUDA array descriptor

#### Returns

`cudaSuccess`, `cudaErrorInvalidResourceHandle`

#### Description

Maps a CUDA mipmapped array onto an external object and returns a handle to it in `mipmap`.

The properties of the CUDA mipmapped array being mapped must be described in `mipmapDesc`. The structure `cudaExternalMemoryMipmappedArrayDesc` is defined as follows:

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

where `cudaExternalMemoryMipmappedArrayDesc::offset` is the offset in the memory object where the base level of the mipmap chain is.

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

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



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

See also:

`cudaImportExternalMemory` `cudaDestroyExternalMemory`,  
`cudaExternalMemoryGetMappedBuffer`



If `cudaExternalMemoryHandleDesc::type` is `cudaExternalMemoryHandleTypeNvSciBuf`, then `cudaExternalMemoryMipmappedArrayDesc::numLevels` must not be greater than 1.



## `__host__ cudaError_t cudaImportExternalMemory` (`cudaExternalMemory_t *extMem_out`, `const` `cudaExternalMemoryHandleDesc *memHandleDesc`)

Imports an external memory object.

### Parameters

#### `extMem_out`

- Returned handle to an external memory object

#### `memHandleDesc`

- Memory import handle descriptor

### Returns

`cudaSuccess`, `cudaErrorInvalidResourceHandle`

### Description

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

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

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

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

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

If `cudaExternalMemoryHandleDesc::type` is `cudaExternalMemoryHandleTypeOpaqueFd`, then

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

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

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

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

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

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

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

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

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

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



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

- ▶ If the Vulkan memory imported into CUDA is mapped on the CPU then the application must use `vkInvalidateMappedMemoryRanges/`  
`vkFlushMappedMemoryRanges` as well as appropriate Vulkan pipeline barriers to maintain coherence between CPU and GPU. For more information on these APIs, please refer to "Synchronization and Cache Control" chapter from Vulkan specification.

See also:

`cudaDestroyExternalMemory`, `cudaExternalMemoryGetMappedBuffer`,  
`cudaExternalMemoryGetMappedMipmappedArray`

**`__host__ cudaError_t cudaImportExternalSemaphore`**  
**`(cudaExternalSemaphore_t *extSem_out, const`**  
**`cudaExternalSemaphoreHandleDesc *semHandleDesc)`**

Imports an external semaphore.

#### Parameters

##### **`extSem_out`**

- Returned handle to an external semaphore

##### **`semHandleDesc`**

- Semaphore import handle descriptor

#### Returns

`cudaSuccess`, `cudaErrorInvalidResourceHandle`

#### Description

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

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

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

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

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

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

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

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

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

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

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

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

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



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

#### See also:

`cudaDestroyExternalSemaphore`, `cudaSignalExternalSemaphoresAsync`,  
`cudaWaitExternalSemaphoresAsync`

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

Signals a set of external semaphore objects.

### Parameters

#### **extSemArray**

- Set of external semaphores to be signaled

#### **paramsArray**

- Array of semaphore parameters

#### **numExtSems**

- Number of semaphores to signal

#### **stream**

- Stream to enqueue the signal operations in

### Returns

[cudaSuccess](#), [cudaErrorInvalidResourceHandle](#)

### Description

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

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

If the semaphore object is any one of the following types: [cudaExternalSemaphoreHandleTypeOpaqueFd](#), [cudaExternalSemaphoreHandleTypeOpaqueWin32](#), [cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt](#) then signaling the semaphore will set it to the signaled state.

If the semaphore object is any one of the following types: [cudaExternalSemaphoreHandleTypeD3D12Fence](#), [cudaExternalSemaphoreHandleTypeD3D11Fence](#) then the semaphore will be set to the value specified in `cudaExternalSemaphoreSignalParams::params::fence::value`.

If the semaphore object is of the type [cudaExternalSemaphoreHandleTypeNvSciSync](#) this API sets `cudaExternalSemaphoreSignalParams::params::nvSciSync::fence` to a value that can be used by subsequent waiters of the same `NvSciSync` object to order operations with those currently submitted in `stream`. Such an update will overwrite



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

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



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

See also:

`cudaImportExternalSemaphore`, `cudaDestroyExternalSemaphore`,  
`cudaWaitExternalSemaphoresAsync`

```
__host__ cudaError_t cudaWaitExternalSemaphoresAsync
(const cudaExternalSemaphore_t *extSemArray, const
```



## `cudaExternalSemaphoreWaitParams *paramsArray, unsigned int numExtSems, cudaStream_t stream)`

Waits on a set of external semaphore objects.

### Parameters

#### **extSemArray**

- External semaphores to be waited on

#### **paramsArray**

- Array of semaphore parameters

#### **numExtSems**

- Number of semaphores to wait on

#### **stream**

- Stream to enqueue the wait operations in

### Returns

`cudaSuccess`, `cudaErrorInvalidResourceHandle` `cudaErrorTimeout`

### Description

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

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

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

If the semaphore object is any one of the following types: `cudaExternalSemaphoreHandleTypeD3D12Fence`, `cudaExternalSemaphoreHandleTypeD3D11Fence` then waiting on the semaphore will wait until the value of the semaphore is greater than or equal to `cudaExternalSemaphoreWaitParams::params::fence::value`.

If the semaphore object is of the type `cudaExternalSemaphoreHandleTypeNvSciSync` then, waiting on the semaphore will wait until the `cudaExternalSemaphoreSignalParams::params::nvSciSync::fence` is signaled by the signaler of the `NvSciSyncObj` that was associated with this semaphore object. By default, waiting on such an external semaphore object causes appropriate memory synchronization operations to be performed over all external memory objects that

are imported as `cudaExternalMemoryHandleTypeNvSciBuf`. This ensures that any subsequent accesses made by other importers of the same set of NvSciBuf memory object(s) are coherent. These operations can be skipped by specifying the flag `cudaExternalSemaphoreWaitSkipNvSciBufMemSync`, which can be used as a performance optimization when data coherency is not required. But specifying this flag in scenarios where data coherency is required results in undefined behavior. Also, for semaphore object of the type `cudaExternalSemaphoreHandleTypeNvSciSync`, if the NvSciSyncAttrList used to create the NvSciSyncObj had not set the flags in `cudaDeviceGetNvSciSyncAttributes` to `cudaNvSciSyncAttrWait`, this API will return `cudaErrorNotSupported`.

If the semaphore object is any one of the following types:

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



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

See also:

`cudaImportExternalSemaphore`, `cudaDestroyExternalSemaphore`,  
`cudaSignalExternalSemaphoresAsync`

## 5.7. Execution Control

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

Some functions have overloaded C++ API template versions documented separately in the [C++ API Routines](#) module.

## `__host__ __device__ cudaError_t cudaFuncGetAttributes(cudaFuncAttributes *attr, const void *func)`

Find out attributes for a given function.

### Parameters

#### **attr**

- Return pointer to function's attributes

#### **func**

- Device function symbol

### Returns

`cudaSuccess`, `cudaErrorInvalidDeviceFunction`

### Description

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

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



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

### See also:

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

## `__host__ cudaError_t cudaFuncSetAttribute (const void *func, cudaFuncAttribute attr, int value)`

Set attributes for a given function.

### Parameters

#### **func**

- Function to get attributes of

#### **attr**

- Attribute to set

#### **value**

- Value to set

### Returns

`cudaSuccess`, `cudaErrorInvalidDeviceFunction`, `cudaErrorInvalidValue`

### Description

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

Valid values for `attr` are:

- ▶ `cudaFuncAttributeMaxDynamicSharedMemorySize` - The requested maximum size in bytes of dynamically-allocated shared memory. The sum of this value and the function attribute `sharedSizeBytes` cannot exceed the device attribute `cudaDevAttrMaxSharedMemoryPerBlockOptin`. The maximal size of requestable dynamic shared memory may differ by GPU architecture.
- ▶ `cudaFuncAttributePreferredSharedMemoryCarveout` - On devices where the L1 cache and shared memory use the same hardware resources, this sets the shared memory carveout preference, in percent of the total shared memory. See `cudaDevAttrMaxSharedMemoryPerMultiprocessor`. This is only a hint, and the driver can choose a different ratio if required to execute the function.



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

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

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

## `__host__ cudaError_t cudaFuncSetCacheConfig (const void *func, cudaFuncCache cacheConfig)`

Sets the preferred cache configuration for a device function.

### Parameters

#### `func`

- Device function symbol

#### `cacheConfig`

- Requested cache configuration

### Returns

`cudaSuccess`, `cudaErrorInvalidDeviceFunction`

### Description

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

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

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

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

The supported cache configurations are:

- ▶ `cudaFuncCachePreferNone`: no preference for shared memory or L1 (default)

- ▶ `cudaFuncCachePreferShared`: prefer larger shared memory and smaller L1 cache
- ▶ `cudaFuncCachePreferL1`: prefer larger L1 cache and smaller shared memory
- ▶ `cudaFuncCachePreferEqual`: prefer equal size L1 cache and shared memory



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

#### See also:

`cudaFuncSetCacheConfig` ( C++ API), `cudaFuncGetAttributes` ( C API),  
`cudaLaunchKernel` ( C API), `cudaSetDoubleForDevice`, `cudaSetDoubleForHost`,  
`cudaThreadGetCacheConfig`, `cudaThreadSetCacheConfig`, `cuFuncSetCacheConfig`

## `__host__ cudaError_t cudaFuncSetSharedMemConfig` (const void \*func, cudaSharedMemConfig config)

Sets the shared memory configuration for a device function.

#### Parameters

##### **func**

- Device function symbol

##### **config**

- Requested shared memory configuration

#### Returns

`cudaSuccess`, `cudaErrorInvalidDeviceFunction`, `cudaErrorInvalidValue`,

#### Description

On devices with configurable shared memory banks, this function will force all subsequent launches of the specified device function to have the given shared memory bank size configuration. On any given launch of the function, the shared memory configuration of the device will be temporarily changed if needed to suit the function's

preferred configuration. Changes in shared memory configuration between subsequent launches of functions, may introduce a device side synchronization point.

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

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

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

For templated functions, pass the function symbol as follows:

`func_name<template_arg_0,...,template_arg_N>`

The supported bank configurations are:

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



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

**See also:**

`cudaDeviceSetSharedMemConfig`, `cudaDeviceGetSharedMemConfig`,  
`cudaDeviceSetCacheConfig`, `cudaDeviceGetCacheConfig`, `cudaFuncSetCacheConfig`,  
`cuFuncSetSharedMemConfig`

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

Obtains a parameter buffer.

### Parameters

#### **alignment**

- Specifies alignment requirement of the parameter buffer

#### **size**

- Specifies size requirement in bytes

### Returns

Returns pointer to the allocated parameterBuffer

### Description

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

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



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

### See also:

`cudaLaunchDevice`

## `__device__ void *cudaGetParameterBufferV2 (void *func, dim3 gridDimension, dim3 blockDim, unsigned int sharedMemSize)`

Launches a specified kernel.

### Parameters

#### **func**

- Pointer to the kernel to be launched

#### **gridDimension**

- Specifies grid dimensions

#### **blockDimension**

- Specifies block dimensions



**sharedMemSize**

- Specifies size of shared memory

**Returns**

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

**Description**

Launches a specified kernel with the specified parameter buffer. A parameter buffer can be obtained by calling `cudaGetParameterBuffer()`.

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



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

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

**See also:**

`cudaGetParameterBuffer`

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

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

**Parameters****func**

- Device function symbol

**gridDim**

- Grid dimensions

**blockDim**

- Block dimensions

**args**

- Arguments

**sharedMem**

- Shared memory

**stream**

- Stream identifier

**Returns**

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

**Description**

The function invokes kernel `func` on `gridDim` (`gridDim.x` × `gridDim.y` × `gridDim.z`) grid of blocks. Each block contains `blockDim` (`blockDim.x` × `blockDim.y` × `blockDim.z`) threads.

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

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

The kernel cannot make use of CUDA dynamic parallelism.

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

For templated functions, pass the function symbol as follows:

`func_name<template_arg_0,...,template_arg_N>`

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

`stream` specifies a stream the invocation is associated to.



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

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

See also:

[cudaLaunchCooperativeKernel](#) ( C++ API), [cudaLaunchCooperativeKernelMultiDevice](#), [cuLaunchCooperativeKernel](#)

## **\_\_host\_\_ cudaError\_t cudaLaunchCooperativeKernelMultiDevice ([cudaLaunchParams](#) \*launchParamsList, unsigned int numDevices, unsigned int flags)**

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

### **Parameters**

#### **launchParamsList**

- List of launch parameters, one per device

#### **numDevices**

- Size of the `launchParamsList` array

#### **flags**

- Flags to control launch behavior

### **Returns**

[cudaSuccess](#), [cudaErrorInvalidDeviceFunction](#), [cudaErrorInvalidConfiguration](#), [cudaErrorLaunchFailure](#), [cudaErrorLaunchTimeout](#), [cudaErrorLaunchOutOfResources](#), [cudaErrorCooperativeLaunchTooLarge](#), [cudaErrorSharedObjectInitFailed](#)

### **Description**

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

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

The same kernel must be launched on all devices. Note that any `__device__` or `__constant__` variables are independently instantiated on every device. It is the

application's responsibility to ensure these variables are initialized and used appropriately.

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

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

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

The kernel cannot make use of CUDA dynamic parallelism.

The `cudaLaunchParams` structure is defined as:

```

↑
struct cudaLaunchParams
{
    void *func;
    dim3 gridDim;
    dim3 blockDim;
    void **args;
    size_t sharedMem;
    cudaStream_t
    stream;
};

```

where:

- ▶ `cudaLaunchParams::func` specifies the kernel to be launched. This same functions must be launched on all devices. For templated functions, pass the function symbol as follows: `func_name<template_arg_0,...,template_arg_N>`
- ▶ `cudaLaunchParams::gridDim` specifies the width, height and depth of the grid in blocks. This must match across all kernels launched.
- ▶ `cudaLaunchParams::blockDim` is the width, height and depth of each thread block. This must match across all kernels launched.
- ▶ `cudaLaunchParams::args` specifies the arguments to the kernel. If the kernel has N parameters then `cudaLaunchParams::args` should point to array of N pointers. Each pointer, from `cudaLaunchParams::args[0]` to `cudaLaunchParams::args[N - 1]`, point to the region of memory from which the actual parameter will be copied.
- ▶ `cudaLaunchParams::sharedMem` is the dynamic shared-memory size per thread block in bytes. This must match across all kernels launched.

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

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

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



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

See also:

`cudaLaunchCooperativeKernel` ( C++ API), `cudaLaunchCooperativeKernel`,  
`cuLaunchCooperativeKernelMultiDevice`

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

Enqueues a host function call in a stream.

#### Parameters

**stream**

**fn**

- The function to call once preceding stream operations are complete

**userData**

- User-specified data to be passed to the function

**Returns**

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

**Description**

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

The host function must not make any CUDA API calls. Attempting to use a CUDA API may result in `cudaErrorNotPermitted`, but this is not required. The host function must not perform any synchronization that may depend on outstanding CUDA work not mandated to run earlier. Host functions without a mandated order (such as in independent streams) execute in undefined order and may be serialized.

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

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

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



- ▶ This function uses standard `default stream` semantics.
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

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

#### See also:

`cudaStreamCreate`, `cudaStreamQuery`, `cudaStreamSynchronize`, `cudaStreamWaitEvent`, `cudaStreamDestroy`, `cudaMallocManaged`, `cudaStreamAttachMemAsync`, `cudaStreamAddCallback`, `cuLaunchHostFunc`

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

Launches a device function.

#### Parameters

##### **func**

- Device function symbol

##### **gridDim**

- Grid dimensions

##### **blockDim**

- Block dimensions

##### **args**

- Arguments

##### **sharedMem**

- Shared memory

##### **stream**

- Stream identifier

#### Returns

`cudaSuccess`, `cudaErrorInvalidDeviceFunction`, `cudaErrorInvalidConfiguration`, `cudaErrorLaunchFailure`, `cudaErrorLaunchTimeout`, `cudaErrorLaunchOutOfResources`, `cudaErrorSharedObjectInitFailed`, `cudaErrorInvalidPtx`, `cudaErrorNoKernelImageForDevice`, `cudaErrorJitCompilerNotFound`

## Description

The function invokes kernel `func` on `gridDim` (`gridDim.x × gridDim.y × gridDim.z`) grid of blocks. Each block contains `blockDim` (`blockDim.x × blockDim.y × blockDim.z`) threads.

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

For templated functions, pass the function symbol as follows:

`func_name<template_arg_0,...,template_arg_N>`

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

`stream` specifies a stream the invocation is associated to.



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

See also:

[cudaLaunchKernel](#) ( C++ API), [cuLaunchKernel](#)

## `__host__ cudaError_t cudaSetDoubleForDevice (double *d)`

Converts a double argument to be executed on a device.

### Parameters

`d`

- Double to convert

### Returns

[cudaSuccess](#)



## Description

**Deprecated** This function is deprecated as of CUDA 7.5

Converts the double value of `d` to an internal float representation if the device does not support double arithmetic. If the device does natively support doubles, then this function does nothing.



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

## See also:

`cudaFuncSetCacheConfig` ( C API), `cudaFuncGetAttributes` ( C API),  
`cudaSetDoubleForHost`

## `__host__ cudaError_t cudaSetDoubleForHost (double *d)`

Converts a double argument after execution on a device.

## Parameters

`d`

- Double to convert

## Returns

`cudaSuccess`

## Description

**Deprecated** This function is deprecated as of CUDA 7.5

Converts the double value of `d` from a potentially internal float representation if the device does not support double arithmetic. If the device does natively support doubles, then this function does nothing.



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

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

See also:

`cudaFuncSetCacheConfig` ( C API), `cudaFuncGetAttributes` ( C API),  
`cudaSetDoubleForDevice`

## 5.8. Occupancy

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

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

See `cudaOccupancyMaxPotentialBlockSize` ( C++ API),  
`cudaOccupancyMaxPotentialBlockSize` ( C++ API),  
`cudaOccupancyMaxPotentialBlockSizeVariableSMem` ( C++ API),  
`cudaOccupancyMaxPotentialBlockSizeVariableSMem` ( C++ API)

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

Returns occupancy for a device function.

### Parameters

#### **numBlocks**

- Returned occupancy

#### **func**

- Kernel function for which occupancy is calculated

#### **blockSize**

- Block size the kernel is intended to be launched with

**dynamicSMemSize**

- Per-block dynamic shared memory usage intended, in bytes

**Returns**

[cudaSuccess](#), [cudaErrorInvalidDevice](#), [cudaErrorInvalidDeviceFunction](#),  
[cudaErrorInvalidValue](#), [cudaErrorUnknown](#),

**Description**

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



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

**See also:**

[cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags](#),  
[cudaOccupancyMaxPotentialBlockSize](#) ( C++ API),  
[cudaOccupancyMaxPotentialBlockSizeWithFlags](#) ( C++ API),  
[cudaOccupancyMaxPotentialBlockSizeVariableSMem](#) ( C++ API),  
[cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags](#) ( C++ API),  
[cuOccupancyMaxActiveBlocksPerMultiprocessor](#)

**\_\_host\_\_ cudaError\_t**

**[cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags](#)**  
(int \*numBlocks, const void \*func, int blockSize, size\_t  
dynamicSMemSize, unsigned int flags)

Returns occupancy for a device function with the specified flags.

**Parameters****numBlocks**

- Returned occupancy

**func**

- Kernel function for which occupancy is calculated

**blockSize**

- Block size the kernel is intended to be launched with

**dynamicSMemSize**

- Per-block dynamic shared memory usage intended, in bytes

**flags**

- Requested behavior for the occupancy calculator

**Returns**

[cudaSuccess](#), [cudaErrorInvalidDevice](#), [cudaErrorInvalidDeviceFunction](#),  
[cudaErrorInvalidValue](#), [cudaErrorUnknown](#),

**Description**

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

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

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



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

**See also:**

[cudaOccupancyMaxActiveBlocksPerMultiprocessor](#),  
[cudaOccupancyMaxPotentialBlockSize](#) ( C++ API),

[cudaOccupancyMaxPotentialBlockSizeWithFlags \( C++ API\)](#),  
[cudaOccupancyMaxPotentialBlockSizeVariableSMem \( C++ API\)](#),  
[cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags \( C++ API\)](#),  
[cuOccupancyMaxActiveBlocksPerMultiprocessorWithFlags](#)

## 5.9. Memory Management

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

Some functions have overloaded C++ API template versions documented separately in the [C++ API Routines](#) module.

**\_\_host\_\_ cudaError\_t cudaArrayGetInfo**  
**(cudaChannelFormatDesc \*desc, cudaExtent \*extent,**  
**unsigned int \*flags, cudaArray\_t array)**

Gets info about the specified cudaArray.

### Parameters

#### **desc**

- Returned array type

#### **extent**

- Returned array shape. 2D arrays will have depth of zero

#### **flags**

- Returned array flags

#### **array**

- The cudaArray to get info for

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

### Description

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

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



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

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

See also:

`cuArrayGetDescriptor`, `cuArray3DGetDescriptor`

## `__host__ __device__ cudaError_t cudaFree (void *devPtr)`

Frees memory on the device.

### Parameters

**devPtr**

- Device pointer to memory to free

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

### Description

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

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



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

See also:

[cudaMalloc](#), [cudaMallocPitch](#), [cudaMallocArray](#), [cudaFreeArray](#), [cudaMallocHost](#) ( C API), [cudaFreeHost](#), [cudaMalloc3D](#), [cudaMalloc3DArray](#), [cudaHostAlloc](#), [cuMemFree](#)

## `__host__ cudaError_t cudaFreeArray (cudaArray_t array)`

Frees an array on the device.

### Parameters

#### **array**

- Pointer to array to free

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

### Description

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



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

### See also:

[cudaMalloc](#), [cudaMallocPitch](#), [cudaFree](#), [cudaMallocArray](#), [cudaMallocHost](#) ( C API), [cudaFreeHost](#), [cudaHostAlloc](#), [cuArrayDestroy](#)

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

Frees page-locked memory.

### Parameters

#### **ptr**

- Pointer to memory to free

## Returns

`cudaSuccess`, `cudaErrorInvalidValue`

## Description

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



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

## See also:

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

## `__host__ cudaError_t cudaFreeMipmappedArray(cudaMipmappedArray_t mipmappedArray)`

Frees a mipmapped array on the device.

## Parameters

### `mipmappedArray`

- Pointer to mipmapped array to free

## Returns

`cudaSuccess`, `cudaErrorInvalidValue`

## Description

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





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

#### See also:

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

**`__host__ cudaError_t cudaGetMipmappedArrayLevel`  
(`cudaArray_t *levelArray`, `cudaMipmappedArray_const_t`  
`mipmappedArray`, unsigned int `level`)**

Gets a mipmap level of a CUDA mipmapped array.

#### Parameters

##### **`levelArray`**

- Returned mipmap level CUDA array

##### **`mipmappedArray`**

- CUDA mipmapped array

##### **`level`**

- Mipmap level

#### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

#### Description

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

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



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

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

#### See also:

`cudaMalloc3D`, `cudaMalloc`, `cudaMallocPitch`, `cudaFree`, `cudaFreeArray`, `cudaMallocHost` ( C API), `cudaFreeHost`, `cudaHostAlloc`, `make_cudaExtent`, `cuMipmappedArrayGetLevel`

## `__host__ cudaError_t cudaGetSymbolAddress (void **devPtr, const void *symbol)`

Finds the address associated with a CUDA symbol.

#### Parameters

##### `devPtr`

- Return device pointer associated with symbol

##### `symbol`

- Device symbol address

#### Returns

`cudaSuccess`, `cudaErrorInvalidSymbol`, `cudaErrorNoKernelImageForDevice`

#### Description

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



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

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

See also:

[cudaGetSymbolAddress](#) ( C++ API), [cudaGetSymbolSize](#) ( C API), [cuModuleGetGlobal](#)

## **\_\_host\_\_ cudaError\_t cudaGetSymbolSize (size\_t \*size, const void \*symbol)**

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

### Parameters

**size**

- Size of object associated with symbol

**symbol**

- Device symbol address

### Returns

[cudaSuccess](#), [cudaErrorInvalidSymbol](#), [cudaErrorNoKernelImageForDevice](#)

### Description

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



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

See also:

`cudaGetSymbolAddress` ( C API), `cudaGetSymbolSize` ( C++ API), `cuModuleGetGlobal`

## `__host__ cudaError_t cudaHostAlloc (void **pHost, size_t size, unsigned int flags)`

Allocates page-locked memory on the host.

### Parameters

#### **pHost**

- Device pointer to allocated memory

#### **size**

- Requested allocation size in bytes

#### **flags**

- Requested properties of allocated memory

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorMemoryAllocation`

### Description

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

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

- ▶ **`cudaHostAllocDefault`**: This flag's value is defined to be 0 and causes `cudaHostAlloc()` to emulate `cudaMallocHost()`.
- ▶ **`cudaHostAllocPortable`**: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
- ▶ **`cudaHostAllocMapped`**: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling `cudaHostGetDevicePointer()`.
- ▶ **`cudaHostAllocWriteCombined`**: Allocates the memory as write-combined (WC). WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs. WC memory is

a good option for buffers that will be written by the CPU and read by the device via mapped pinned memory or host->device transfers.

All of these flags are orthogonal to one another: a developer may allocate memory that is portable, mapped and/or write-combined with no restrictions.

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

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

Memory allocated by this function must be freed with `cudaFreeHost()`.



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

See also:

`cudaSetDeviceFlags`, `cudaMallocHost` ( C API), `cudaFreeHost`, `cudaGetDeviceFlags`, `cuMemHostAlloc`

## `__host__ cudaError_t cudaHostGetDevicePointer (void **pDevice, void *pHost, unsigned int flags)`

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

### Parameters

#### **pDevice**

- Returned device pointer for mapped memory

#### **pHost**

- Requested host pointer mapping

#### **flags**

- Flags for extensions (must be 0 for now)

## Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorMemoryAllocation`

## Description

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

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

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

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



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

## See also:

`cudaSetDeviceFlags`, `cudaHostAlloc`, `cuMemHostGetDevicePointer`

## `__host__ cudaError_t cudaHostGetFlags (unsigned int *pFlags, void *pHost)`

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

### Parameters

#### **pFlags**

- Returned flags word

#### **pHost**

- Host pointer

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

### Description

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



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

### See also:

`cudaHostAlloc`, `cuMemHostGetFlags`

## `__host__ cudaError_t cudaHostRegister (void *ptr, size_t size, unsigned int flags)`

Registers an existing host memory range for use by CUDA.

### Parameters

#### **ptr**

- Host pointer to memory to page-lock

**size**

- Size in bytes of the address range to page-lock in bytes

**flags**

- Flags for allocation request

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorMemoryAllocation`,  
`cudaErrorHostMemoryAlreadyRegistered`, `cudaErrorNotSupported`

**Description**

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

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

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

- ▶ **`cudaHostRegisterDefault`**: On a system with unified virtual addressing, the memory will be both mapped and portable. On a system with no unified virtual addressing, the memory will be neither mapped nor portable.
- ▶ **`cudaHostRegisterPortable`**: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
- ▶ **`cudaHostRegisterMapped`**: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling `cudaHostGetDevicePointer()`.
- ▶ **`cudaHostRegisterIoMemory`**: The passed memory pointer is treated as pointing to some memory-mapped I/O space, e.g. belonging to a third-party PCIe device, and it will be marked as non cache-coherent and contiguous.

All of these flags are orthogonal to one another: a developer may page-lock memory that is portable or mapped with no restrictions.

The CUDA context must have been created with the `cudaMapHost` flag in order for the `cudaHostRegisterMapped` flag to have any effect.



The `cudaHostRegisterMapped` flag may be specified on CUDA contexts for devices that do not support mapped pinned memory. The failure is deferred to `cudaHostGetDevicePointer()` because the memory may be mapped into other CUDA contexts via the `cudaHostRegisterPortable` flag.

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

The memory page-locked by this function must be unregistered with `cudaHostUnregister()`.



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

See also:

`cudaHostUnregister`, `cudaHostGetFlags`, `cudaHostGetDevicePointer`,  
`cuMemHostRegister`

## `__host__ cudaError_t cudaHostUnregister (void *ptr)`

Unregisters a memory range that was registered with `cudaHostRegister`.

### Parameters

**ptr**

- Host pointer to memory to unregister

## Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorHostMemoryNotRegistered](#)

## Description

Unmaps the memory range whose base address is specified by `ptr`, and makes it pageable again.

The base address must be the same one specified to [cudaHostRegister\(\)](#).



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

## See also:

[cudaHostUnregister](#), [cuMemHostUnregister](#)

**`__host__ __device__ cudaError_t cudaMalloc (void **devPtr, size_t size)`**

Allocate memory on the device.

## Parameters

### **`devPtr`**

- Pointer to allocated device memory

### **`size`**

- Requested allocation size in bytes

## Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorMemoryAllocation](#)

## Description

Allocates `size` bytes of linear memory on the device and returns in `*devPtr` a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. [cudaMalloc\(\)](#) returns [cudaErrorMemoryAllocation](#) in case of failure.

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



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

#### See also:

`cudaMallocPitch`, `cudaFree`, `cudaMallocArray`, `cudaFreeArray`, `cudaMalloc3D`, `cudaMalloc3DArray`, `cudaMallocHost` ( C API), `cudaFreeHost`, `cudaHostAlloc`, `cuMemAlloc`

## `__host__ cudaError_t cudaMalloc3D (cudaPitchedPtr *pitchedDevPtr, cudaExtent extent)`

Allocates logical 1D, 2D, or 3D memory objects on the device.

### Parameters

#### `pitchedDevPtr`

- Pointer to allocated pitched device memory

#### `extent`

- Requested allocation size (`width` field in bytes)

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorMemoryAllocation`

### Description

Allocates at least `width * height * depth` bytes of linear memory on the device and returns a `cudaPitchedPtr` in which `ptr` is a pointer to the allocated memory. The function may pad the allocation to ensure hardware alignment requirements are met. The pitch returned in the `pitch` field of `pitchedDevPtr` is the width in bytes of the allocation.

The returned `cudaPitchedPtr` contains additional fields `xsize` and `ysize`, the logical width and height of the allocation, which are equivalent to the `width` and `height` `extent` parameters provided by the programmer during allocation.

For allocations of 2D and 3D objects, it is highly recommended that programmers perform allocations using `cudaMalloc3D()` or `cudaMallocPitch()`. Due to alignment restrictions in the hardware, this is especially true if the application will be performing memory copies involving 2D or 3D objects (whether linear memory or CUDA arrays).



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

See also:

`cudaMallocPitch`, `cudaFree`, `cudaMemcpy3D`, `cudaMemset3D`, `cudaMalloc3DArray`, `cudaMallocArray`, `cudaFreeArray`, `cudaMallocHost` ( C API), `cudaFreeHost`, `cudaHostAlloc`, `make_cudaPitchedPtr`, `make_cudaExtent`, `cuMemAllocPitch`

**`__host__ cudaError_t cudaMalloc3DArray (cudaArray_t *array, const cudaChannelFormatDesc *desc, cudaExtent extent, unsigned int flags)`**

Allocate an array on the device.

### Parameters

#### **array**

- Pointer to allocated array in device memory

#### **desc**

- Requested channel format

#### **extent**

- Requested allocation size (`width` field in elements)

#### **flags**

- Flags for extensions

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorMemoryAllocation`

## Description

Allocates a CUDA array according to the `cudaChannelFormatDesc` structure `desc` and returns a handle to the new CUDA array in `*array`.

The `cudaChannelFormatDesc` is defined as:

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

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

`cudaMalloc3DArray()` can allocate the following:

- ▶ A 1D array is allocated if the height and depth extents are both zero.
- ▶ A 2D array is allocated if only the depth extent is zero.
- ▶ A 3D array is allocated if all three extents are non-zero.
- ▶ A 1D layered CUDA array is allocated if only the height extent is zero and the `cudaArrayLayered` flag is set. Each layer is a 1D array. The number of layers is determined by the depth extent.
- ▶ A 2D layered CUDA array is allocated if all three extents are non-zero and the `cudaArrayLayered` flag is set. Each layer is a 2D array. The number of layers is determined by the depth extent.
- ▶ A cubemap CUDA array is allocated if all three extents are non-zero and the `cudaArrayCubemap` flag is set. Width must be equal to height, and depth must be six. A cubemap is a special type of 2D layered CUDA array, where the six layers represent the six faces of a cube. The order of the six layers in memory is the same as that listed in `cudaGraphicsCubeFace`.
- ▶ A cubemap layered CUDA array is allocated if all three extents are non-zero, and both, `cudaArrayCubemap` and `cudaArrayLayered` flags are set. Width must be equal to height, and depth must be a multiple of six. A cubemap layered CUDA array is a special type of 2D layered CUDA array that consists of a collection of cubemaps. The first six layers represent the first cubemap, the next six layers form the second cubemap, and so on.

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

- ▶ `cudaArrayDefault`: This flag's value is defined to be 0 and provides default array allocation
- ▶ `cudaArrayLayered`: Allocates a layered CUDA array, with the depth extent indicating the number of layers
- ▶ `cudaArrayCubemap`: Allocates a cubemap CUDA array. Width must be equal to height, and depth must be six. If the `cudaArrayLayered` flag is also set, depth must be a multiple of six.

- **cudaArraySurfaceLoadStore**: Allocates a CUDA array that could be read from or written to using a surface reference.
- **cudaArrayTextureGather**: This flag indicates that texture gather operations will be performed on the CUDA array. Texture gather can only be performed on 2D CUDA arrays.

The width, height and depth extents must meet certain size requirements as listed in the following table. All values are specified in elements.

Note that 2D CUDA arrays have different size requirements if the **cudaArrayTextureGather** flag is set. In that case, the valid range for (width, height, depth) is ((1,maxTexture2DGather[0]), (1,maxTexture2DGather[1]), 0).

CUDA array type	Valid extents that must always be met {(width range in elements), (height range), (depth range)}	Valid extents with <b>cudaArraySurfaceLoadStore</b> set {(width range in elements), (height range), (depth range)}
1D	{ (1,maxTexture1D), 0, 0 }	{ (1,maxSurface1D), 0, 0 }
2D	{ (1,maxTexture2D[0]), (1,maxTexture2D[1]), 0 }	{ (1,maxSurface2D[0]), (1,maxSurface2D[1]), 0 }
3D	{ (1,maxTexture3D[0]), (1,maxTexture3D[1]), (1,maxTexture3D[2]) } OR { (1,maxTexture3DAlt[0]), (1,maxTexture3DAlt[1]), (1,maxTexture3DAlt[2]) }	{ (1,maxSurface3D[0]), (1,maxSurface3D[1]), (1,maxSurface3D[2]) }
1D Layered	{ (1,maxTexture1DLayered[0]), 0, (1,maxTexture1DLayered[1]) }	{ (1,maxSurface1DLayered[0]), 0, (1,maxSurface1DLayered[1]) }
2D Layered	{ (1,maxTexture2DLayered[0]), (1,maxTexture2DLayered[1]), (1,maxTexture2DLayered[2]) }	{ (1,maxSurface2DLayered[0]), (1,maxSurface2DLayered[1]), (1,maxSurface2DLayered[2]) }
Cubemap	{ (1,maxTextureCubemap), (1,maxTextureCubemap), 6 }	{ (1,maxSurfaceCubemap), (1,maxSurfaceCubemap), 6 }
Cubemap Layered	{ (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[1]) }	{ (1,maxSurfaceCubemapLayered[0]), (1,maxSurfaceCubemapLayered[0]), (1,maxSurfaceCubemapLayered[1]) }



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

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

#### See also:

`cudaMalloc3D`, `cudaMalloc`, `cudaMallocPitch`, `cudaFree`, `cudaFreeArray`, `cudaMallocHost` ( C API), `cudaFreeHost`, `cudaHostAlloc`, `make_cudaExtent`, `cuArray3DCreate`

**`__host__ cudaError_t cudaMallocArray (cudaArray_t *array, const cudaChannelFormatDesc *desc, size_t width, size_t height, unsigned int flags)`**

Allocate an array on the device.

#### Parameters

##### **array**

- Pointer to allocated array in device memory

##### **desc**

- Requested channel format

##### **width**

- Requested array allocation width

##### **height**

- Requested array allocation height

##### **flags**

- Requested properties of allocated array

#### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorMemoryAllocation`

#### Description

Allocates a CUDA array according to the `cudaChannelFormatDesc` structure `desc` and returns a handle to the new CUDA array in `*array`.

The `cudaChannelFormatDesc` is defined as:

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

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

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

- ▶ `cudaArrayDefault`: This flag's value is defined to be 0 and provides default array allocation
- ▶ `cudaArraySurfaceLoadStore`: Allocates an array that can be read from or written to using a surface reference
- ▶ `cudaArrayTextureGather`: This flag indicates that texture gather operations will be performed on the array.

`width` and `height` must meet certain size requirements. See `cudaMalloc3DArray()` for more details.



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

See also:

`cudaMalloc`, `cudaMallocPitch`, `cudaFree`, `cudaFreeArray`, `cudaMallocHost` ( C API), `cudaFreeHost`, `cudaMalloc3D`, `cudaMalloc3DArray`, `cudaHostAlloc`, `cuArrayCreate`

## `__host__ cudaError_t cudaMallocHost (void **ptr, size_t size)`

Allocates page-locked memory on the host.

### Parameters

**ptr**

- Pointer to allocated host memory

**size**

- Requested allocation size in bytes

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorMemoryAllocation`



## Description

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



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

## See also:

`cudaMalloc`, `cudaMallocPitch`, `cudaMallocArray`, `cudaMalloc3D`, `cudaMalloc3DArray`, `cudaHostAlloc`, `cudaFree`, `cudaFreeArray`, `cudaMallocHost` ( C++ API), `cudaFreeHost`, `cudaHostAlloc`, `cuMemAllocHost`

## `__host__ cudaError_t cudaMallocManaged (void **devPtr, size_t size, unsigned int flags)`

Allocates memory that will be automatically managed by the Unified Memory system.

## Parameters

### `devPtr`

- Pointer to allocated device memory

### `size`

- Requested allocation size in bytes

### `flags`

- Must be either `cudaMemAttachGlobal` or `cudaMemAttachHost` (defaults to `cudaMemAttachGlobal`)

## Returns

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

## Description

Allocates `size` bytes of managed memory on the device and returns in `*devPtr` a pointer to the allocated memory. If the device doesn't support allocating managed memory, `cudaErrorNotSupported` is returned. Support for managed memory can be queried using the device attribute `cudaDevAttrManagedMemory`. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. If `size` is 0, `cudaMallocManaged` returns `cudaErrorInvalidValue`. The pointer is valid on the CPU and on all GPUs in the system that support managed memory. All accesses to this pointer must obey the Unified Memory programming model.

`flags` specifies the default stream association for this allocation. `flags` must be one of `cudaMemAttachGlobal` or `cudaMemAttachHost`. The default value for `flags` is `cudaMemAttachGlobal`. If `cudaMemAttachGlobal` is specified, then this memory is accessible from any stream on any device. If `cudaMemAttachHost` is specified, then the allocation should not be accessed from devices that have a zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`; an explicit call to `cudaStreamAttachMemAsync` will be required to enable access on such devices.

If the association is later changed via `cudaStreamAttachMemAsync` to a single stream, the default association, as specified during `cudaMallocManaged`, is restored when that stream is destroyed. For `__managed__` variables, the default association is always `cudaMemAttachGlobal`. Note that destroying a stream is an asynchronous operation, and as a result, the change to default association won't happen until all work in the stream has completed.

Memory allocated with `cudaMallocManaged` should be released with `cudaFree`.

Device memory oversubscription is possible for GPUs that have a non-zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`. Managed memory on such GPUs may be evicted from device memory to host memory at any time by the Unified Memory driver in order to make room for other allocations.

In a multi-GPU system where all GPUs have a non-zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`, managed memory may not be populated when this API returns and instead may be populated on access. In such systems, managed memory can migrate to any processor's memory at any time. The Unified Memory driver will employ heuristics to maintain data locality and prevent excessive page faults to the extent possible. The application can also guide the driver about memory usage patterns via `cudaMemAdvise`. The application can also explicitly migrate memory to a desired processor's memory via `cudaMemPrefetchAsync`.

In a multi-GPU system where all of the GPUs have a zero value for the device attribute `cudaDevAttrConcurrentManagedAccess` and all the GPUs have peer-to-peer support with each other, the physical storage for managed memory is created on the GPU which is active at the time `cudaMallocManaged` is called. All other GPUs will reference the data at reduced bandwidth via peer mappings over the PCIe bus. The Unified Memory driver does not migrate memory among such GPUs.

In a multi-GPU system where not all GPUs have peer-to-peer support with each other and where the value of the device attribute `cudaDevAttrConcurrentManagedAccess` is zero for at least one of those GPUs, the location chosen for physical storage of managed memory is system-dependent.

- ▶ On Linux, the location chosen will be device memory as long as the current set of active contexts are on devices that either have peer-to-peer support with each other or have a non-zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`. If there is an active context on a GPU that does not have a non-zero value for that device attribute and it does not have peer-to-peer support with the other devices that have active contexts on them, then the location for physical storage will be 'zero-copy' or host memory. Note that this means that managed memory that is located in device memory is migrated to host memory if a new context is created on a GPU that doesn't have a non-zero value for the device attribute and does not support peer-to-peer with at least one of the other devices that has an active context. This in turn implies that context creation may fail if there is insufficient host memory to migrate all managed allocations.
- ▶ On Windows, the physical storage is always created in 'zero-copy' or host memory. All GPUs will reference the data at reduced bandwidth over the PCIe bus. In these circumstances, use of the environment variable `CUDA_VISIBLE_DEVICES` is recommended to restrict CUDA to only use those GPUs that have peer-to-peer support. Alternatively, users can also set `CUDA_MANAGED_FORCE_DEVICE_ALLOC` to a non-zero value to force the driver to always use device memory for physical storage. When this environment variable is set to a non-zero value, all devices used in that process that support managed memory have to be peer-to-peer compatible with each other. The error `cudaErrorInvalidDevice` will be returned if a device that supports managed memory is used and it is not peer-to-peer compatible with any of the other managed memory supporting devices that were previously used in that process, even if `cudaDeviceReset` has been called on those devices. These environment variables are described in the CUDA programming guide under the "CUDA environment variables" section.



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

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

#### See also:

`cudaMallocPitch`, `cudaFree`, `cudaMallocArray`, `cudaFreeArray`, `cudaMalloc3D`, `cudaMalloc3DArray`, `cudaMallocHost` ( C API), `cudaFreeHost`, `cudaHostAlloc`, `cudaDeviceGetAttribute`, `cudaStreamAttachMemAsync`, `cuMemAllocManaged`

**`__host__ cudaError_t cudaMallocMipmappedArray`**  
**(`cudaMipmappedArray_t *mipmappedArray`, `const cudaChannelFormatDesc *desc`, `cudaExtent extent`, `unsigned int numLevels`, `unsigned int flags`)**

Allocate a mipmapped array on the device.

#### Parameters

##### **`mipmappedArray`**

- Pointer to allocated mipmapped array in device memory

##### **`desc`**

- Requested channel format

##### **`extent`**

- Requested allocation size (`width` field in elements)

##### **`numLevels`**

- Number of mipmap levels to allocate

##### **`flags`**

- Flags for extensions

#### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorMemoryAllocation`

#### Description

Allocates a CUDA mipmapped array according to the `cudaChannelFormatDesc` structure `desc` and returns a handle to the new CUDA mipmapped array in `*mipmappedArray`. `numLevels` specifies the number of mipmap levels to be allocated. This value is clamped to the range `[1, 1 + floor(log2(max(width, height, depth)))]`.

The `cudaChannelFormatDesc` is defined as:

```

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

```

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

`cudaMallocMipmappedArray()` can allocate the following:

- ▶ A 1D mipmapped array is allocated if the height and depth extents are both zero.
- ▶ A 2D mipmapped array is allocated if only the depth extent is zero.
- ▶ A 3D mipmapped array is allocated if all three extents are non-zero.
- ▶ A 1D layered CUDA mipmapped array is allocated if only the height extent is zero and the `cudaArrayLayered` flag is set. Each layer is a 1D mipmapped array. The number of layers is determined by the depth extent.
- ▶ A 2D layered CUDA mipmapped array is allocated if all three extents are non-zero and the `cudaArrayLayered` flag is set. Each layer is a 2D mipmapped array. The number of layers is determined by the depth extent.
- ▶ A cubemap CUDA mipmapped array is allocated if all three extents are non-zero and the `cudaArrayCubemap` flag is set. Width must be equal to height, and depth must be six. The order of the six layers in memory is the same as that listed in `cudaGraphicsCubeFace`.
- ▶ A cubemap layered CUDA mipmapped array is allocated if all three extents are non-zero, and both, `cudaArrayCubemap` and `cudaArrayLayered` flags are set. Width must be equal to height, and depth must be a multiple of six. A cubemap layered CUDA mipmapped array is a special type of 2D layered CUDA mipmapped array that consists of a collection of cubemap mipmapped arrays. The first six layers represent the first cubemap mipmapped array, the next six layers form the second cubemap mipmapped array, and so on.

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

- ▶ **`cudaArrayDefault`**: This flag's value is defined to be 0 and provides default mipmapped array allocation
- ▶ **`cudaArrayLayered`**: Allocates a layered CUDA mipmapped array, with the depth extent indicating the number of layers
- ▶ **`cudaArrayCubemap`**: Allocates a cubemap CUDA mipmapped array. Width must be equal to height, and depth must be six. If the `cudaArrayLayered` flag is also set, depth must be a multiple of six.
- ▶ **`cudaArraySurfaceLoadStore`**: This flag indicates that individual mipmap levels of the CUDA mipmapped array will be read from or written to using a surface reference.

- **cudaArrayTextureGather**: This flag indicates that texture gather operations will be performed on the CUDA array. Texture gather can only be performed on 2D CUDA mipmapped arrays, and the gather operations are performed only on the most detailed mipmap level.

The width, height and depth extents must meet certain size requirements as listed in the following table. All values are specified in elements.

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



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

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

See also:

[cudaMalloc3D](#), [cudaMalloc](#), [cudaMallocPitch](#), [cudaFree](#), [cudaFreeArray](#),  
[cudaMallocHost](#) ( C API), [cudaFreeHost](#), [cudaHostAlloc](#), [make\\_cudaExtent](#),  
[cuMipmappedArrayCreate](#)

## **\_\_host\_\_ cudaError\_t cudaMallocPitch (void \*\*devPtr, size\_t \*pitch, size\_t width, size\_t height)**

Allocates pitched memory on the device.

### Parameters

#### **devPtr**

- Pointer to allocated pitched device memory

#### **pitch**

- Pitch for allocation

#### **width**

- Requested pitched allocation width (in bytes)

#### **height**

- Requested pitched allocation height

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorMemoryAllocation](#)

### Description

Allocates at least `width` (in bytes) \* `height` bytes of linear memory on the device and returns in `*devPtr` a pointer to the allocated memory. The function may pad the allocation to ensure that corresponding pointers in any given row will continue to meet the alignment requirements for coalescing as the address is updated from row to row. The pitch returned in `*pitch` by [cudaMallocPitch\(\)](#) is the width in bytes of the allocation. The intended usage of `pitch` is as a separate parameter of the allocation, used to compute addresses within the 2D array. Given the row and column of an array element of type `T`, the address is computed as:

```
↑ T* pElement = (T*)((char*)BaseAddress + Row * pitch) + Column;
```

For allocations of 2D arrays, it is recommended that programmers consider performing pitch allocations using [cudaMallocPitch\(\)](#). Due to pitch alignment restrictions in the hardware, this is especially true if the application will be performing 2D memory copies between different regions of device memory (whether linear memory or CUDA arrays).



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

#### See also:

`cudaMalloc`, `cudaFree`, `cudaMallocArray`, `cudaFreeArray`, `cudaMallocHost` ( C API), `cudaFreeHost`, `cudaMalloc3D`, `cudaMalloc3DArray`, `cudaHostAlloc`, `cuMemAllocPitch`

## `__host__ cudaError_t cudaMemAdvise (const void *devPtr, size_t count, cudaMemoryAdvise advice, int device)`

Advise about the usage of a given memory range.

#### Parameters

##### `devPtr`

- Pointer to memory to set the advice for

##### `count`

- Size in bytes of the memory range

##### `advice`

- Advice to be applied for the specified memory range

##### `device`

- Device to apply the advice for

#### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`

#### Description

Advise the Unified Memory subsystem about the usage pattern for the memory range starting at `devPtr` with a size of `count` bytes. The start address and end address of the memory range will be rounded down and rounded up respectively to be aligned to CPU page size before the advice is applied. The memory range must refer to managed memory allocated via `cudaMallocManaged` or declared via `__managed__` variables. The memory range could also refer to system-allocated pageable memory provided it represents a valid, host-accessible region of memory and all additional constraints



imposed by `advice` as outlined below are also satisfied. Specifying an invalid system-allocated pageable memory range results in an error being returned.

The `advice` parameter can take the following values:

- ▶ **`cudaMemAdviseSetReadMostly`**: This implies that the data is mostly going to be read from and only occasionally written to. Any read accesses from any processor to this region will create a read-only copy of at least the accessed pages in that processor's memory. Additionally, if `cudaMemPrefetchAsync` is called on this region, it will create a read-only copy of the data on the destination processor. If any processor writes to this region, all copies of the corresponding page will be invalidated except for the one where the write occurred. The `device` argument is ignored for this advice. Note that for a page to be read-duplicated, the accessing processor must either be the CPU or a GPU that has a non-zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`. Also, if a context is created on a device that does not have the device attribute `cudaDevAttrConcurrentManagedAccess` set, then read-duplication will not occur until all such contexts are destroyed. If the memory region refers to valid system-allocated pageable memory, then the accessing device must have a non-zero value for the device attribute `cudaDevAttrPageableMemoryAccess` for a read-only copy to be created on that device. Note however that if the accessing device also has a non-zero value for the device attribute `cudaDevAttrPageableMemoryAccessUsesHostPageTables`, then setting this advice will not create a read-only copy when that device accesses this memory region.
- ▶ **`cudaMemAdviseUnsetReadMostly`**: Undoes the effect of `cudaMemAdviseReadMostly` and also prevents the Unified Memory driver from attempting heuristic read-duplication on the memory range. Any read-duplicated copies of the data will be collapsed into a single copy. The location for the collapsed copy will be the preferred location if the page has a preferred location and one of the read-duplicated copies was resident at that location. Otherwise, the location chosen is arbitrary.
- ▶ **`cudaMemAdviseSetPreferredLocation`**: This advice sets the preferred location for the data to be the memory belonging to `device`. Passing in `cudaCpuDeviceId` for `device` sets the preferred location as host memory. If `device` is a GPU, then it must have a non-zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`. Setting the preferred location does not cause data to migrate to that location immediately. Instead, it guides the migration policy when a fault occurs on that memory region. If the data is already in its preferred location and the faulting processor can establish a mapping without requiring the data to be migrated, then data migration will be avoided. On the other hand, if the data is not in its preferred location or if a direct mapping cannot be established, then it will be migrated to the processor accessing it. It is important to note that setting the preferred location does not prevent data prefetching done using `cudaMemPrefetchAsync`. Having a preferred location can override the page

thrash detection and resolution logic in the Unified Memory driver. Normally, if a page is detected to be constantly thrashing between for example host and device memory, the page may eventually be pinned to host memory by the Unified Memory driver. But if the preferred location is set as device memory, then the page will continue to thrash indefinitely. If `cudaMemAdviseSetReadMostly` is also set on this memory region or any subset of it, then the policies associated with that advice will override the policies of this advice, unless read accesses from device will not result in a read-only copy being created on that device as outlined in description for the advice `cudaMemAdviseSetReadMostly`. If the memory region refers to valid system-allocated pageable memory, then device must have a non-zero value for the device attribute `cudaDevAttrPageableMemoryAccess`. Additionally, if device has a non-zero value for the device attribute `cudaDevAttrPageableMemoryAccessUsesHostPageTables`, then this call has no effect. Note however that this behavior may change in the future.

- ▶ `cudaMemAdviseUnsetPreferredLocation`: Undoes the effect of `cudaMemAdviseSetPreferredLocation` and changes the preferred location to none.
- ▶ `cudaMemAdviseSetAccessedBy`: This advice implies that the data will be accessed by device. Passing in `cudaCpuDeviceId` for device will set the advice for the CPU. If device is a GPU, then the device attribute `cudaDevAttrConcurrentManagedAccess` must be non-zero. This advice does not cause data migration and has no impact on the location of the data per se. Instead, it causes the data to always be mapped in the specified processor's page tables, as long as the location of the data permits a mapping to be established. If the data gets migrated for any reason, the mappings are updated accordingly. This advice is recommended in scenarios where data locality is not important, but avoiding faults is. Consider for example a system containing multiple GPUs with peer-to-peer access enabled, where the data located on one GPU is occasionally accessed by peer GPUs. In such scenarios, migrating data over to the other GPUs is not as important because the accesses are infrequent and the overhead of migration may be too high. But preventing faults can still help improve performance, and so having a mapping set up in advance is useful. Note that on CPU access of this data, the data may be migrated to host memory because the CPU typically cannot access device memory directly. Any GPU that had the `cudaMemAdviseSetAccessedBy` flag set for this data will now have its mapping updated to point to the page in host memory. If `cudaMemAdviseSetReadMostly` is also set on this memory region or any subset of it, then the policies associated with that advice will override the policies of this advice. Additionally, if the preferred location of this memory region or any subset of it is also device, then the policies associated with `cudaMemAdviseSetPreferredLocation` will override the policies of this advice. If the memory region refers to valid system-allocated pageable memory, then device must have a non-zero value for the device attribute `cudaDevAttrPageableMemoryAccess`. Additionally, if device has a non-zero value

for the device attribute `cudaDevAttrPageableMemoryAccessUsesHostPageTables`, then this call has no effect.

- ▶ `cudaMemAdviseUnsetAccessedBy`: Undoes the effect of `cudaMemAdviseSetAccessedBy`. Any mappings to the data from `device` may be removed at any time causing accesses to result in non-fatal page faults. If the memory region refers to valid system-allocated pageable memory, then `device` must have a non-zero value for the device attribute `cudaDevAttrPageableMemoryAccess`. Additionally, if `device` has a non-zero value for the device attribute `cudaDevAttrPageableMemoryAccessUsesHostPageTables`, then this call has no effect.



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

#### See also:

`cudaMemcpy`, `cudaMemcpyPeer`, `cudaMemcpyAsync`, `cudaMemcpy3DPeerAsync`, `cudaMemPrefetchAsync`, `cuMemAdvise`

## `__host__ cudaError_t cudaMemcpy (void *dst, const void *src, size_t count, cudaMemcpyKind kind)`

Copies data between host and device.

#### Parameters

##### **dst**

- Destination memory address

##### **src**

- Source memory address

##### **count**

- Size in bytes to copy

##### **kind**

- Type of transfer

## Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidMemcpyDirection`

## Description

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing. Calling `cudaMemcpy()` with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior.



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

## See also:

`cudaMemcpy2D`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`, `cuMemcpyDtoH`, `cuMemcpyHtoD`, `cuMemcpyDtoD`, `cuMemcpy`

**`__host__ cudaError_t cudaMemcpy2D (void *dst, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, cudaMemcpyKind kind)`**

Copies data between host and device.

## Parameters

**`dst`**

- Destination memory address

**dpitch**

- Pitch of destination memory

**src**

- Source memory address

**spitch**

- Pitch of source memory

**width**

- Width of matrix transfer (columns in bytes)

**height**

- Height of matrix transfer (rows)

**kind**

- Type of transfer

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidPitchValue`,  
`cudaErrorInvalidMemcpyDirection`

**Description**

Copies a matrix (`height` rows of `width` bytes each) from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing. `dpitch` and `spitch` are the widths in memory in bytes of the 2D arrays pointed to by `dst` and `src`, including any padding added to the end of each row. The memory areas may not overlap. `width` must not exceed either `dpitch` or `spitch`. Calling `cudaMemcpy2D()` with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior. `cudaMemcpy2D()` returns an error if `dpitch` or `spitch` exceeds the maximum allowed.



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

See also:

[cudaMemcpy](#), [cudaMemcpy2DToArray](#), [cudaMemcpy2DFromArray](#),  
[cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#),  
[cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpy2DToArrayAsync](#),  
[cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#),  
[cudaMemcpyFromSymbolAsync](#), [cuMemcpy2D](#), [cuMemcpy2DUnaligned](#)

**\_\_host\_\_ cudaError\_t cudaMemcpy2DArrayToArray**  
**(cudaArray\_t dst, size\_t wOffsetDst, size\_t**  
**hOffsetDst, cudaArray\_const\_t src, size\_t wOffsetSrc,**  
**size\_t hOffsetSrc, size\_t width, size\_t height,**  
**cudaMemcpyKind kind)**

Copies data between host and device.

### Parameters

**dst**

- Destination memory address

**wOffsetDst**

- Destination starting X offset

**hOffsetDst**

- Destination starting Y offset

**src**

- Source memory address

**wOffsetSrc**

- Source starting X offset

**hOffsetSrc**

- Source starting Y offset

**width**

- Width of matrix transfer (columns in bytes)

**height**

- Height of matrix transfer (rows)

**kind**

- Type of transfer

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidMemcpyDirection](#)

### Description

Copies a matrix (height rows of width bytes each) from the CUDA array `srcArray` starting at the upper left corner (`wOffsetSrc`, `hOffsetSrc`) to the CUDA array `dst` starting at the upper left corner (`wOffsetDst`, `hOffsetDst`), where `kind`

specifies the direction of the copy, and must be one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing. `wOffsetDst + width` must not exceed the width of the CUDA array `dst`. `wOffsetSrc + width` must not exceed the width of the CUDA array `src`.



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

#### See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`,  
`cudaMemcpy2DFromArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`,  
`cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpy2DToArrayAsync`,  
`cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`,  
`cudaMemcpyFromSymbolAsync`, `cuMemcpy2D`, `cuMemcpy2DUnaligned`

```
__host__ __device__ cudaError_t cudaMemcpy2DAsync(
    void *dst, size_t dpitch, const void *src, size_t spitch,
    size_t width, size_t height, cudaMemcpyKind kind,
    cudaStream_t stream)
```

Copies data between host and device.

#### Parameters

##### **dst**

- Destination memory address

##### **dpitch**

- Pitch of destination memory

##### **src**

- Source memory address



**spitch**

- Pitch of source memory

**width**

- Width of matrix transfer (columns in bytes)

**height**

- Height of matrix transfer (rows)

**kind**

- Type of transfer

**stream**

- Stream identifier

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidPitchValue](#),  
[cudaErrorInvalidMemcpyDirection](#)

**Description**

Copies a matrix (`height` rows of `width` bytes each) from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of [cudaMemcpyHostToHost](#), [cudaMemcpyHostToDevice](#), [cudaMemcpyDeviceToHost](#), [cudaMemcpyDeviceToDevice](#), or [cudaMemcpyDefault](#). Passing [cudaMemcpyDefault](#) is recommended, in which case the type of transfer is inferred from the pointer values. However, [cudaMemcpyDefault](#) is only allowed on systems that support unified virtual addressing. `dpitch` and `spitch` are the widths in memory in bytes of the 2D arrays pointed to by `dst` and `src`, including any padding added to the end of each row. The memory areas may not overlap. `width` must not exceed either `dpitch` or `spitch`.

Calling [cudaMemcpy2DAsync\(\)](#) with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior. [cudaMemcpy2DAsync\(\)](#) returns an error if `dpitch` or `spitch` is greater than the maximum allowed.

[cudaMemcpy2DAsync\(\)](#) is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is [cudaMemcpyHostToDevice](#) or [cudaMemcpyDeviceToHost](#) and `stream` is non-zero, the copy may overlap with operations in other streams.

The device version of this function only handles device to device copies and cannot be given local or shared pointers.



- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits [asynchronous](#) behavior for most use cases.



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

#### See also:

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpy2DToArray](#), [cudaMemcpy2DFromArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#), [cuMemcpy2DAsync](#)

**\_\_host\_\_ cudaError\_t cudaMemcpy2DFromArray (void \*dst, size\_t dpitch, cudaArray\_const\_t src, size\_t wOffset, size\_t hOffset, size\_t width, size\_t height, cudaMemcpyKind kind)**

Copies data between host and device.

#### Parameters

##### **dst**

- Destination memory address

##### **dpitch**

- Pitch of destination memory

##### **src**

- Source memory address

##### **wOffset**

- Source starting X offset

##### **hOffset**

- Source starting Y offset

##### **width**

- Width of matrix transfer (columns in bytes)

##### **height**

- Height of matrix transfer (rows)

##### **kind**

- Type of transfer

## Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidPitchValue`,  
`cudaErrorInvalidMemcpyDirection`

## Description

Copies a matrix (height rows of width bytes each) from the CUDA array `srcArray` starting at the upper left corner (`wOffset`, `hOffset`) to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing. `dpitch` is the width in memory in bytes of the 2D array pointed to by `dst`, including any padding added to the end of each row. `wOffset + width` must not exceed the width of the CUDA array `src`. `width` must not exceed `dpitch`. `cudaMemcpy2DFromArray()` returns an error if `dpitch` exceeds the maximum allowed.



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

## See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`,  
`cudaMemcpy2DFromArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`,  
`cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpy2DToArrayAsync`,  
`cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`,  
`cudaMemcpyFromSymbolAsync`, `cuMemcpy2D`, `cuMemcpy2DUnaligned`

```
__host__ cudaError_t cudaMemcpy2DFromArrayAsync  
(void *dst, size_t dpitch, cudaArray_const_t src, size_t
```

**wOffset, size\_t hOffset, size\_t width, size\_t height,  
cudaMemcpyKind kind, cudaStream\_t stream)**

Copies data between host and device.

### Parameters

**dst**

- Destination memory address

**dpitch**

- Pitch of destination memory

**src**

- Source memory address

**wOffset**

- Source starting X offset

**hOffset**

- Source starting Y offset

**width**

- Width of matrix transfer (columns in bytes)

**height**

- Height of matrix transfer (rows)

**kind**

- Type of transfer

**stream**

- Stream identifier

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidPitchValue](#),  
[cudaErrorInvalidMemcpyDirection](#)

### Description

Copies a matrix (*height* rows of *width* bytes each) from the CUDA array *srcArray* starting at the upper left corner (*wOffset*, *hOffset*) to the memory area pointed to by *dst*, where *kind* specifies the direction of the copy, and must be one of [cudaMemcpyHostToHost](#), [cudaMemcpyHostToDevice](#), [cudaMemcpyDeviceToHost](#), [cudaMemcpyDeviceToDevice](#), or [cudaMemcpyDefault](#). Passing [cudaMemcpyDefault](#) is recommended, in which case the type of transfer is inferred from the pointer values. However, [cudaMemcpyDefault](#) is only allowed on systems that support unified virtual addressing. *dpitch* is the width in memory in bytes of the 2D array pointed to by *dst*, including any padding added to the end of each row. *wOffset* + *width* must not exceed the width of the CUDA array *src*. *width* must not exceed *dpitch*. [cudaMemcpy2DFromArrayAsync\(\)](#) returns an error if *dpitch* exceeds the maximum allowed.

`cudaMemcpy2DFromArrayAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost` and `stream` is non-zero, the copy may overlap with operations in other streams.



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

#### See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`, `cuMemcpy2DAsync`

**`__host__ cudaError_t cudaMemcpy2DToArray`**  
**`(cudaArray_t dst, size_t wOffset, size_t hOffset, const void *src, size_t spitch, size_t width, size_t height, cudaMemcpyKind kind)`**

Copies data between host and device.

#### Parameters

##### **`dst`**

- Destination memory address

##### **`wOffset`**

- Destination starting X offset

##### **`hOffset`**

- Destination starting Y offset

##### **`src`**

- Source memory address

**spitch**

- Pitch of source memory

**width**

- Width of matrix transfer (columns in bytes)

**height**

- Height of matrix transfer (rows)

**kind**

- Type of transfer

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidPitchValue](#),  
[cudaErrorInvalidMemcpyDirection](#)

**Description**

Copies a matrix (`height` rows of `width` bytes each) from the memory area pointed to by `src` to the CUDA array `dst` starting at the upper left corner (`wOffset`, `hOffset`) where `kind` specifies the direction of the copy, and must be one of [cudaMemcpyHostToHost](#), [cudaMemcpyHostToDevice](#), [cudaMemcpyDeviceToHost](#), [cudaMemcpyDeviceToDevice](#), or [cudaMemcpyDefault](#). Passing [cudaMemcpyDefault](#) is recommended, in which case the type of transfer is inferred from the pointer values. However, [cudaMemcpyDefault](#) is only allowed on systems that support unified virtual addressing. `spitch` is the width in memory in bytes of the 2D array pointed to by `src`, including any padding added to the end of each row. `wOffset + width` must not exceed the width of the CUDA array `dst`. `width` must not exceed `spitch`. [cudaMemcpy2DToArray\(\)](#) returns an error if `spitch` exceeds the maximum allowed.



- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits [synchronous](#) behavior for most use cases.
- Note that this function may also return [cudaErrorInitializationError](#), [cudaErrorInsufficientDriver](#) or [cudaErrorNoDevice](#) if this call tries to initialize internal CUDA RT state.
- Note that as specified by [cudaStreamAddCallback](#) no CUDA function may be called from callback. [cudaErrorNotPermitted](#) may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpy2DFromArray](#),  
[cudaMemcpy2DFromArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#),  
[cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpy2DToArrayAsync](#),

`cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`,  
`cudaMemcpyFromSymbolAsync`, `cuMemcpy2D`, `cuMemcpy2DUnaligned`

```
__host__ cudaError_t cudaMemcpy2DToArrayAsync  

(cudaArray_t dst, size_t wOffset, size_t hOffset, const  

void *src, size_t spitch, size_t width, size_t height,  

cudaMemcpyKind kind, cudaStream_t stream)
```

Copies data between host and device.

### Parameters

#### **dst**

- Destination memory address

#### **wOffset**

- Destination starting X offset

#### **hOffset**

- Destination starting Y offset

#### **src**

- Source memory address

#### **spitch**

- Pitch of source memory

#### **width**

- Width of matrix transfer (columns in bytes)

#### **height**

- Height of matrix transfer (rows)

#### **kind**

- Type of transfer

#### **stream**

- Stream identifier

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidPitchValue`,  
`cudaErrorInvalidMemcpyDirection`

### Description

Copies a matrix (`height` rows of `width` bytes each) from the memory area pointed to by `src` to the CUDA array `dst` starting at the upper left corner (`wOffset`, `hOffset`) where `kind` specifies the direction of the copy, and must be one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified

virtual addressing. `spitch` is the width in memory in bytes of the 2D array pointed to by `src`, including any padding added to the end of each row. `wOffset + width` must not exceed the width of the CUDA array `dst`. `width` must not exceed `spitch`. `cudaMemcpy2DToArrayAsync()` returns an error if `spitch` exceeds the maximum allowed.

`cudaMemcpy2DToArrayAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost` and `stream` is non-zero, the copy may overlap with operations in other streams.



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

#### See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`,

`cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`, `cuMemcpy2DAsync`

## `__host__ cudaError_t cudaMemcpy3D (const cudaMemcpy3DParms *p)`

Copies data between 3D objects.

### Parameters

**p**

- 3D memory copy parameters

## Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidPitchValue`,  
`cudaErrorInvalidMemcpyDirection`

## Description

```

struct cudaExtent {
    size_t width;
    size_t height;
    size_t depth;
};
struct cudaExtent
    make_cudaExtent(size_t w, size_t h, size_t d);

struct cudaPos {
    size_t x;
    size_t y;
    size_t z;
};
struct cudaPos
    make_cudaPos(size_t x, size_t y, size_t z);

struct cudaMemcpy3DParms {
    cudaArray_t
        srcArray;
    struct cudaPos
        srcPos;
    struct cudaPitchedPtr
        srcPtr;
    cudaArray_t
        dstArray;
    struct cudaPos
        dstPos;
    struct cudaPitchedPtr
        dstPtr;
    struct cudaExtent
        extent;
    enum cudaMemcpyKind
        kind;
};

```

`cudaMemcpy3D()` copies data between two 3D objects. The source and destination objects may be in either host memory, device memory, or a CUDA array. The source, destination, extent, and kind of copy performed is specified by the `cudaMemcpy3DParms` struct which should be initialized to zero before use:

```

cudaMemcpy3DParms myParms = {0};

```

The struct passed to `cudaMemcpy3D()` must specify one of `srcArray` or `srcPtr` and one of `dstArray` or `dstPtr`. Passing more than one non-zero source or destination will cause `cudaMemcpy3D()` to return an error.

The `srcPos` and `dstPos` fields are optional offsets into the source and destination objects and are defined in units of each object's elements. The element for a host or device pointer is assumed to be **unsigned char**.

The `extent` field defines the dimensions of the transferred area in elements. If a CUDA array is participating in the copy, the extent is defined in terms of that array's elements.



If no CUDA array is participating in the copy then the extents are defined in elements of **unsigned char**.

The `kind` field defines the direction of the copy. It must be one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing. For `cudaMemcpyHostToHost` or `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost` passed as `kind` and `cudaArray` type passed as source or destination, if the `kind` implies `cudaArray` type to be present on the host, `cudaMemcpy3D()` will disregard that implication and silently correct the `kind` based on the fact that `cudaArray` type can only be present on the device.

If the source and destination are both arrays, `cudaMemcpy3D()` will return an error if they do not have the same element size.

The source and destination object may not overlap. If overlapping source and destination objects are specified, undefined behavior will result.

The source object must lie entirely within the region defined by `srcPos` and `extent`. The destination object must lie entirely within the region defined by `dstPos` and `extent`.

`cudaMemcpy3D()` returns an error if the pitch of `srcPtr` or `dstPtr` exceeds the maximum allowed. The pitch of a `cudaPitchedPtr` allocated with `cudaMalloc3D()` will always be valid.



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

#### See also:

`cudaMalloc3D`, `cudaMalloc3DArray`, `cudaMemset3D`, `cudaMemcpy3DAsync`, `cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpy2DToArrayAsync`,

`cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`,  
`cudaMemcpyFromSymbolAsync`, `make_cudaExtent`, `make_cudaPos`, `cuMemcpy3D`

**\_\_host\_\_ \_\_device\_\_ cudaError\_t cudaMemcpy3DAsync  
 (const cudaMemcpy3DParms \*p, cudaStream\_t stream)**

Copies data between 3D objects.

### Parameters

**p**

- 3D memory copy parameters

**stream**

- Stream identifier

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidPitchValue`,  
`cudaErrorInvalidMemcpyDirection`

### Description

```

struct cudaExtent {
    size_t width;
    size_t height;
    size_t depth;
};
struct cudaExtent
    make_cudaExtent(size_t w, size_t h, size_t d);

struct cudaPos {
    size_t x;
    size_t y;
    size_t z;
};
struct cudaPos
    make_cudaPos(size_t x, size_t y, size_t z);

struct cudaMemcpy3DParms {
    cudaArray_t
        srcArray;
    struct cudaPos
        srcPos;
    struct cudaPitchedPtr
        srcPtr;
    cudaArray_t
        dstArray;
    struct cudaPos
        dstPos;
    struct cudaPitchedPtr
        dstPtr;
    struct cudaExtent
        extent;
    enum cudaMemcpyKind
        kind;
};
  
```

`cudaMemcpy3DAsync()` copies data between two 3D objects. The source and destination objects may be in either host memory, device memory, or a CUDA array. The source, destination, extent, and kind of copy performed is specified by the `cudaMemcpy3DParms` struct which should be initialized to zero before use:

```
cudaMemcpy3DParms myParms = {0};
```

The struct passed to `cudaMemcpy3DAsync()` must specify one of `srcArray` or `srcPtr` and one of `dstArray` or `dstPtr`. Passing more than one non-zero source or destination will cause `cudaMemcpy3DAsync()` to return an error.

The `srcPos` and `dstPos` fields are optional offsets into the source and destination objects and are defined in units of each object's elements. The element for a host or device pointer is assumed to be **unsigned char**. For CUDA arrays, positions must be in the range `[0, 2048)` for any dimension.

The `extent` field defines the dimensions of the transferred area in elements. If a CUDA array is participating in the copy, the extent is defined in terms of that array's elements. If no CUDA array is participating in the copy then the extents are defined in elements of **unsigned char**.

The `kind` field defines the direction of the copy. It must be one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing. For `cudaMemcpyHostToHost` or `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost` passed as `kind` and `cudaArray` type passed as source or destination, if the `kind` implies `cudaArray` type to be present on the host, `cudaMemcpy3DAsync()` will disregard that implication and silently correct the `kind` based on the fact that `cudaArray` type can only be present on the device.

If the source and destination are both arrays, `cudaMemcpy3DAsync()` will return an error if they do not have the same element size.

The source and destination object may not overlap. If overlapping source and destination objects are specified, undefined behavior will result.

The source object must lie entirely within the region defined by `srcPos` and `extent`. The destination object must lie entirely within the region defined by `dstPos` and `extent`.

`cudaMemcpy3DAsync()` returns an error if the pitch of `srcPtr` or `dstPtr` exceeds the maximum allowed. The pitch of a `cudaPitchedPtr` allocated with `cudaMalloc3D()` will always be valid.

`cudaMemcpy3DAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is `cudaMemcpyHostToDevice` or

`cudaMemcpyDeviceToHost` and `stream` is non-zero, the copy may overlap with operations in other streams.

The device version of this function only handles device to device copies and cannot be given local or shared pointers.



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

#### See also:

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

## `__host__ cudaError_t cudaMemcpy3DPeer (const cudaMemcpy3DPeerParms *p)`

Copies memory between devices.

### Parameters

**p**

- Parameters for the memory copy

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`

### Description

Perform a 3D memory copy according to the parameters specified in `p`. See the definition of the `cudaMemcpy3DPeerParms` structure for documentation of its parameters.

Note that this function is synchronous with respect to the host only if the source or destination of the transfer is host memory. Note also that this copy is serialized with respect to all pending and future asynchronous work in to the current device, the copy's source device, and the copy's destination device (use `cudaMemcpy3DPeerAsync` to avoid this synchronization).



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

#### See also:

`cudaMemcpy`, `cudaMemcpyPeer`, `cudaMemcpyAsync`, `cudaMemcpyPeerAsync`, `cudaMemcpy3DPeerAsync`, `cuMemcpy3DPeer`

## `__host__ cudaError_t cudaMemcpy3DPeerAsync (const cudaMemcpy3DPeerParms *p, cudaStream_t stream)`

Copies memory between devices asynchronously.

### Parameters

**p**

- Parameters for the memory copy

**stream**

- Stream identifier

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`

### Description

Perform a 3D memory copy according to the parameters specified in `p`. See the definition of the `cudaMemcpy3DPeerParms` structure for documentation of its parameters.



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

### See also:

[cudaMemcpy](#), [cudaMemcpyPeer](#), [cudaMemcpyAsync](#), [cudaMemcpyPeerAsync](#), [cudaMemcpy3DPeerAsync](#), [cuMemcpy3DPeerAsync](#)

**`__host____device__cudaError_t cudaMemcpyAsync (void *dst, const void *src, size_t count, cudaMemcpyKind kind, cudaStream_t stream)`**

Copies data between host and device.

### Parameters

#### **dst**

- Destination memory address

#### **src**

- Source memory address

#### **count**

- Size in bytes to copy

#### **kind**

- Type of transfer

#### **stream**

- Stream identifier

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidMemcpyDirection](#)

### Description

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of

`cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing.

The memory areas may not overlap. Calling `cudaMemcpyAsync()` with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior.

`cudaMemcpyAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost` and the `stream` is non-zero, the copy may overlap with operations in other streams.

The device version of this function only handles device to device copies and cannot be given local or shared pointers.



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

#### See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`,  
`cudaMemcpy2DFromArray`, `cudaMemcpy2DFromArrayToArray`, `cudaMemcpyToSymbol`,  
`cudaMemcpyFromSymbol`, `cudaMemcpy2DAsync`, `cudaMemcpy2DToArrayAsync`,  
`cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`,  
`cudaMemcpyFromSymbolAsync` `cuMemcpyAsync`, `cuMemcpyDtoHAsync`,  
`cuMemcpyHtoDAsync`, `cuMemcpyDtoDAsync`

**\_\_host\_\_ cudaError\_t cudaMemcpyFromSymbol (void \*dst, const void \*symbol, size\_t count, size\_t offset, cudaMemcpyKind kind)**

Copies data from the given symbol on the device.

### Parameters

#### dst

- Destination memory address

#### symbol

- Device symbol address

#### count

- Size in bytes to copy

#### offset

- Offset from start of symbol in bytes

#### kind

- Type of transfer

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidSymbol](#), [cudaErrorInvalidMemcpyDirection](#), [cudaErrorNoKernelImageForDevice](#)

### Description

Copies `count` bytes from the memory area pointed to by `offset` bytes from the start of symbol `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either [cudaMemcpyDeviceToHost](#), [cudaMemcpyDeviceToDevice](#), or [cudaMemcpyDefault](#). Passing [cudaMemcpyDefault](#) is recommended, in which case the type of transfer is inferred from the pointer values. However, [cudaMemcpyDefault](#) is only allowed on systems that support unified virtual addressing.



- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits [synchronous](#) behavior for most use cases.
- Use of a string naming a variable as the `symbol` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.
- Note that this function may also return [cudaErrorInitializationError](#), [cudaErrorInsufficientDriver](#) or [cudaErrorNoDevice](#) if this call tries to initialize internal CUDA RT state.



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

#### See also:

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpy2DToArray](#),  
[cudaMemcpy2DFromArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#),  
[cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpy2DToArrayAsync](#),  
[cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#),  
[cudaMemcpyFromSymbolAsync](#), [cuMemcpy](#), [cuMemcpyDtoH](#), [cuMemcpyDtoD](#)

**`__host__ cudaError_t cudaMemcpyFromSymbolAsync`**  
**`(void *dst, const void *symbol, size_t count, size_t`**  
**`offset, cudaMemcpyKind kind, cudaStream_t stream)`**

Copies data from the given symbol on the device.

#### Parameters

##### **dst**

- Destination memory address

##### **symbol**

- Device symbol address

##### **count**

- Size in bytes to copy

##### **offset**

- Offset from start of symbol in bytes

##### **kind**

- Type of transfer

##### **stream**

- Stream identifier

#### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidSymbol](#),  
[cudaErrorInvalidMemcpyDirection](#), [cudaErrorNoKernelImageForDevice](#)

#### Description

Copies `count` bytes from the memory area pointed to by `offset` bytes from the start of symbol `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either [cudaMemcpyDeviceToHost](#), [cudaMemcpyDeviceToDevice](#), or [cudaMemcpyDefault](#). Passing [cudaMemcpyDefault](#) is recommended, in which case the

type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing.

`cudaMemcpyFromSymbolAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is `cudaMemcpyDeviceToHost` and `stream` is non-zero, the copy may overlap with operations in other streams.



- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ This function exhibits `asynchronous` behavior for most use cases.
- ▶ This function uses standard `default stream` semantics.
- ▶ Use of a string naming a variable as the `symbol` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.
- ▶ Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- ▶ Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cuMemcpyAsync`, `cuMemcpyDtoHAsync`, `cuMemcpyDtoDAsync`

## `__host__ cudaError_t cudaMemcpyPeer (void *dst, int dstDevice, const void *src, int srcDevice, size_t count)`

Copies memory between two devices.

### Parameters

**dst**

- Destination device pointer

**dstDevice**

- Destination device

**src**

- Source device pointer

**srcDevice**

- Source device

**count**

- Size of memory copy in bytes

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidDevice](#)

**Description**

Copies memory from one device to memory on another device. `dst` is the base device pointer of the destination memory and `dstDevice` is the destination device. `src` is the base device pointer of the source memory and `srcDevice` is the source device. `count` specifies the number of bytes to copy.

Note that this function is asynchronous with respect to the host, but serialized with respect all pending and future asynchronous work in to the current device, `srcDevice`, and `dstDevice` (use [cudaMemcpyPeerAsync](#) to avoid this synchronization).



- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ This function exhibits [synchronous](#) behavior for most use cases.
- ▶ Note that this function may also return [cudaErrorInitializationError](#), [cudaErrorInsufficientDriver](#) or [cudaErrorNoDevice](#) if this call tries to initialize internal CUDA RT state.
- ▶ Note that as specified by [cudaStreamAddCallback](#) no CUDA function may be called from callback. [cudaErrorNotPermitted](#) may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**

[cudaMemcpy](#), [cudaMemcpyAsync](#), [cudaMemcpyPeerAsync](#),  
[cudaMemcpy3DPeerAsync](#), [cuMemcpyPeer](#)

```
__host__ cudaError_t cudaMemcpyPeerAsync (void *dst,
int dstDevice, const void *src, int srcDevice, size_t
count, cudaStream_t stream)
```

Copies memory between two devices asynchronously.

### Parameters

**dst**

- Destination device pointer

**dstDevice**

- Destination device

**src**

- Source device pointer

**srcDevice**

- Source device

**count**

- Size of memory copy in bytes

**stream**

- Stream identifier

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidDevice](#)

### Description

Copies memory from one device to memory on another device. `dst` is the base device pointer of the destination memory and `dstDevice` is the destination device. `src` is the base device pointer of the source memory and `srcDevice` is the source device. `count` specifies the number of bytes to copy.

Note that this function is asynchronous with respect to the host and all work on other devices.



- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ This function exhibits [asynchronous](#) behavior for most use cases.
- ▶ This function uses standard [default stream](#) semantics.
- ▶ Note that this function may also return [cudaErrorInitializationError](#), [cudaErrorInsufficientDriver](#) or [cudaErrorNoDevice](#) if this call tries to initialize internal CUDA RT state.

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

See also:

`cudaMemcpy`, `cudaMemcpyPeer`, `cudaMemcpyAsync`, `cudaMemcpy3DPeerAsync`, `cuMemcpyPeerAsync`

**`__host__ cudaError_t cudaMemcpyToSymbol (const void *symbol, const void *src, size_t count, size_t offset, cudaMemcpyKind kind)`**

Copies data to the given symbol on the device.

### Parameters

#### **symbol**

- Device symbol address

#### **src**

- Source memory address

#### **count**

- Size in bytes to copy

#### **offset**

- Offset from start of symbol in bytes

#### **kind**

- Type of transfer

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidSymbol`, `cudaErrorInvalidMemcpyDirection`, `cudaErrorNoKernelImageForDevice`

### Description

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing.



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

#### See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`, `cuMemcpy`, `cuMemcpyHtoD`, `cuMemcpyDtoD`

**`__host__ cudaError_t cudaMemcpyToSymbolAsync (const void *symbol, const void *src, size_t count, size_t offset, cudaMemcpyKind kind, cudaStream_t stream)`**

Copies data to the given symbol on the device.

#### Parameters

##### **symbol**

- Device symbol address

##### **src**

- Source memory address

##### **count**

- Size in bytes to copy

##### **offset**

- Offset from start of symbol in bytes

##### **kind**

- Type of transfer

##### **stream**

- Stream identifier

## Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidSymbol`,  
`cudaErrorInvalidMemcpyDirection`, `cudaErrorNoKernelImageForDevice`

## Description

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing.

`cudaMemcpyToSymbolAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is `cudaMemcpyHostToDevice` and `stream` is non-zero, the copy may overlap with operations in other streams.



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

## See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`,  
`cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`,  
`cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpy2DToArrayAsync`,  
`cudaMemcpy2DFromArrayAsync`, `cudaMemcpyFromSymbolAsync`, `cuMemcpyAsync`,  
`cuMemcpyHtoDAsync`, `cuMemcpyDtoDAsync`

## `__host__ cudaError_t cudaMemGetInfo (size_t *free, size_t *total)`

Gets free and total device memory.

### Parameters

#### `free`

- Returned free memory in bytes

#### `total`

- Returned total memory in bytes

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorLaunchFailure`

### Description

Returns in `*free` and `*total` respectively, the free and total amount of memory available for allocation by the device in bytes.



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

### See also:

`cuMemGetInfo`

## `__host__ cudaError_t cudaMemPrefetchAsync (const void *devPtr, size_t count, int dstDevice, cudaStream_t stream)`

Prefetches memory to the specified destination device.

### Parameters

#### `devPtr`

- Pointer to be prefetched



**count**

- Size in bytes

**dstDevice**

- Destination device to prefetch to

**stream**

- Stream to enqueue prefetch operation

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidDevice](#)

**Description**

Prefetches memory to the specified destination device. `devPtr` is the base device pointer of the memory to be prefetched and `dstDevice` is the destination device. `count` specifies the number of bytes to copy. `stream` is the stream in which the operation is enqueued. The memory range must refer to managed memory allocated via [cudaMallocManaged](#) or declared via `__managed__` variables.

Passing in `cudaCpuDeviceId` for `dstDevice` will prefetch the data to host memory. If `dstDevice` is a GPU, then the device attribute [cudaDevAttrConcurrentManagedAccess](#) must be non-zero. Additionally, `stream` must be associated with a device that has a non-zero value for the device attribute [cudaDevAttrConcurrentManagedAccess](#).

The start address and end address of the memory range will be rounded down and rounded up respectively to be aligned to CPU page size before the prefetch operation is enqueued in the stream.

If no physical memory has been allocated for this region, then this memory region will be populated and mapped on the destination device. If there's insufficient memory to prefetch the desired region, the Unified Memory driver may evict pages from other [cudaMallocManaged](#) allocations to host memory in order to make room. Device memory allocated using [cudaMalloc](#) or [cudaMallocArray](#) will not be evicted.

By default, any mappings to the previous location of the migrated pages are removed and mappings for the new location are only setup on `dstDevice`. The exact behavior however also depends on the settings applied to this memory range via [cudaMemAdvise](#) as described below:

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

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

If [cudaMemAdviseSetAccessedBy](#) was called on any subset of this memory range, then mappings to those pages from all the appropriate processors are updated to refer to the

new location if establishing such a mapping is possible. Otherwise, those mappings are cleared.

Note that this API is not required for functionality and only serves to improve performance by allowing the application to migrate data to a suitable location before it is accessed. Memory accesses to this range are always coherent and are allowed even when the data is actively being migrated.

Note that this function is asynchronous with respect to the host and all work on other devices.



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

See also:

[cudaMemcpy](#), [cudaMemcpyPeer](#), [cudaMemcpyAsync](#), [cudaMemcpy3DPeerAsync](#), [cudaMemAdvise](#), [cuMemPrefetchAsync](#)

**\_\_host\_\_ cudaError\_t cudaMemcpyRangeGetAttribute**  
**(void \*data, size\_t dataSize, cudaMemcpyRangeAttribute**  
**attribute, const void \*devPtr, size\_t count)**

Query an attribute of a given memory range.

### Parameters

#### **data**

- A pointers to a memory location where the result of each attribute query will be written to.

#### **dataSize**

- Array containing the size of data

#### **attribute**

- The attribute to query

**devPtr**

- Start of the range to query

**count**

- Size of the range to query

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#)

**Description**

Query an attribute about the memory range starting at `devPtr` with a size of `count` bytes. The memory range must refer to managed memory allocated via [cudaMallocManaged](#) or declared via `__managed__` variables.

The `attribute` parameter can take the following values:

- ▶ [cudaMemRangeAttributeReadMostly](#): If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. The result returned will be 1 if all pages in the given memory range have read-duplication enabled, or 0 otherwise.
- ▶ [cudaMemRangeAttributePreferredLocation](#): If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. The result returned will be a GPU device id if all pages in the memory range have that GPU as their preferred location, or it will be `cudaCpuDeviceId` if all pages in the memory range have the CPU as their preferred location, or it will be `cudaInvalidDeviceId` if either all the pages don't have the same preferred location or some of the pages don't have a preferred location at all. Note that the actual location of the pages in the memory range at the time of the query may be different from the preferred location.
- ▶ [cudaMemRangeAttributeAccessedBy](#): If this attribute is specified, `data` will be interpreted as an array of 32-bit integers, and `dataSize` must be a non-zero multiple of 4. The result returned will be a list of device ids that had `cudaMemAdviceSetAccessedBy` set for that entire memory range. If any device does not have that advice set for the entire memory range, that device will not be included. If `data` is larger than the number of devices that have that advice set for that memory range, `cudaInvalidDeviceId` will be returned in all the extra space provided. For ex., if `dataSize` is 12 (i.e. `data` has 3 elements) and only device 0 has the advice set, then the result returned will be { 0, `cudaInvalidDeviceId`, `cudaInvalidDeviceId` }. If `data` is smaller than the number of devices that have that advice set, then only as many devices will be returned as can fit in the array. There is no guarantee on which specific devices will be returned, however.
- ▶ [cudaMemRangeAttributeLastPrefetchLocation](#): If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. The result returned will be the last location to which all pages in the memory range were prefetched explicitly via [cudaMemPrefetchAsync](#). This will either be a GPU id

or `cudaCpuDeviceId` depending on whether the last location for prefetch was a GPU or the CPU respectively. If any page in the memory range was never explicitly prefetched or if all pages were not prefetched to the same location, `cudaInvalidDeviceId` will be returned. Note that this simply returns the last location that the application requested to prefetch the memory range to. It gives no indication as to whether the prefetch operation to that location has completed or even begun.



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

#### See also:

[cudaMemRangeGetAttributes](#), [cudaMemPrefetchAsync](#), [cudaMemAdvise](#), [cuMemRangeGetAttribute](#)

```
__host__ cudaError_t cudaMemRangeGetAttributes
(void **data, size_t *dataSizes, cudaMemRangeAttribute
*attributes, size_t numAttributes, const void *devPtr,
size_t count)
```

Query attributes of a given memory range.

#### Parameters

##### **data**

- A two-dimensional array containing pointers to memory locations where the result of each attribute query will be written to.

##### **dataSizes**

- Array containing the sizes of each result

##### **attributes**

- An array of attributes to query (`numAttributes` and the number of attributes in this array should match)

##### **numAttributes**

- Number of attributes to query

**devPtr**

- Start of the range to query

**count**

- Size of the range to query

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#)

**Description**

Query attributes of the memory range starting at `devPtr` with a size of `count` bytes. The memory range must refer to managed memory allocated via [cudaMallocManaged](#) or declared via `__managed__` variables. The `attributes` array will be interpreted to have `numAttributes` entries. The `dataSizes` array will also be interpreted to have `numAttributes` entries. The results of the query will be stored in `data`.

The list of supported attributes are given below. Please refer to [cudaMemRangeGetAttribute](#) for attribute descriptions and restrictions.

- ▶ [cudaMemRangeAttributeReadMostly](#)
- ▶ [cudaMemRangeAttributePreferredLocation](#)
- ▶ [cudaMemRangeAttributeAccessedBy](#)
- ▶ [cudaMemRangeAttributeLastPrefetchLocation](#)



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

**See also:**

[cudaMemRangeGetAttribute](#), [cudaMemAdvise](#) [cudaMemPrefetchAsync](#),  
[cuMemRangeGetAttributes](#)

## `__host__ cudaError_t cudaMemcpy (void *devPtr, int value, size_t count)`

Initializes or sets device memory to a value.

### Parameters

#### **devPtr**

- Pointer to device memory

#### **value**

- Value to set for each byte of specified memory

#### **count**

- Size in bytes to set

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`,

### Description

Fills the first `count` bytes of the memory area pointed to by `devPtr` with the constant byte value `value`.

Note that this function is asynchronous with respect to the host unless `devPtr` refers to pinned host memory.



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

### See also:

[cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

## `__host__ cudaError_t cudaMemset2D (void *devPtr, size_t pitch, int value, size_t width, size_t height)`

Initializes or sets device memory to a value.

### Parameters

#### **devPtr**

- Pointer to 2D device memory

#### **pitch**

- Pitch in bytes of 2D device memory

#### **value**

- Value to set for each byte of specified memory

#### **width**

- Width of matrix set (columns in bytes)

#### **height**

- Height of matrix set (rows)

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#),

### Description

Sets to the specified value `value` a matrix (`height` rows of `width` bytes each) pointed to by `dstPtr`. `pitch` is the width in bytes of the 2D array pointed to by `dstPtr`, including any padding added to the end of each row. This function performs fastest when the pitch is one that has been passed back by [cudaMallocPitch\(\)](#).

Note that this function is asynchronous with respect to the host unless `devPtr` refers to pinned host memory.



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

See also:

`cudaMemset`, `cudaMemset3D`, `cudaMemsetAsync`, `cudaMemset2DAsync`,  
`cudaMemset3DAsync`, `cuMemsetD2D8`, `cuMemsetD2D16`, `cuMemsetD2D32`

**\_\_host\_\_ \_\_device\_\_ cudaError\_t cudaMemset2DAsync**  
**(void \*devPtr, size\_t pitch, int value, size\_t width,**  
**size\_t height, cudaStream\_t stream)**

Initializes or sets device memory to a value.

### Parameters

#### **devPtr**

- Pointer to 2D device memory

#### **pitch**

- Pitch in bytes of 2D device memory

#### **value**

- Value to set for each byte of specified memory

#### **width**

- Width of matrix set (columns in bytes)

#### **height**

- Height of matrix set (rows)

#### **stream**

- Stream identifier

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`,

### Description

Sets to the specified value `value` a matrix (`height` rows of `width` bytes each) pointed to by `dstPtr`. `pitch` is the width in bytes of the 2D array pointed to by `dstPtr`, including any padding added to the end of each row. This function performs fastest when the pitch is one that has been passed back by `cudaMallocPitch()`.

`cudaMemset2DAsync()` is asynchronous with respect to the host, so the call may return before the memset is complete. The operation can optionally be associated to a stream by passing a non-zero `stream` argument. If `stream` is non-zero, the operation may overlap with operations in other streams.

The device version of this function only handles device to device copies and cannot be given local or shared pointers.



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



- ▶ See also [memset synchronization details](#).
- ▶ This function uses standard [default stream](#) semantics.
- ▶ Note that this function may also return [cudaErrorInitializationError](#), [cudaErrorInsufficientDriver](#) or [cudaErrorNoDevice](#) if this call tries to initialize internal CUDA RT state.
- ▶ Note that as specified by [cudaStreamAddCallback](#) no CUDA function may be called from callback. [cudaErrorNotPermitted](#) may, but is not guaranteed to, be returned as a diagnostic in such case.

#### See also:

[cudaMemset](#), [cudaMemset2D](#), [cudaMemset3D](#), [cudaMemsetAsync](#),  
[cudaMemset3DAsync](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16Async](#),  
[cuMemsetD2D32Async](#)

## **\_\_host\_\_ cudaError\_t cudaMemset3D (cudaPitchedPtr pitchedDevPtr, int value, cudaExtent extent)**

Initializes or sets device memory to a value.

### Parameters

#### **pitchedDevPtr**

- Pointer to pitched device memory

#### **value**

- Value to set for each byte of specified memory

#### **extent**

- Size parameters for where to set device memory (*width* field in bytes)

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#),

### Description

Initializes each element of a 3D array to the specified value *value*. The object to initialize is defined by *pitchedDevPtr*. The *pitch* field of *pitchedDevPtr* is the width in memory in bytes of the 3D array pointed to by *pitchedDevPtr*, including any padding added to the end of each row. The *xsize* field specifies the logical width of each row in bytes, while the *ysize* field specifies the height of each 2D slice in rows.

The extents of the initialized region are specified as a *width* in bytes, a *height* in rows, and a *depth* in slices.

Extents with *width* greater than or equal to the *xsize* of *pitchedDevPtr* may perform significantly faster than extents narrower than the *xsize*. Secondly, extents

with `height` equal to the `ysize` of `pitchedDevPtr` will perform faster than when the `height` is shorter than the `ysize`.

This function performs fastest when the `pitchedDevPtr` has been allocated by `cudaMalloc3D()`.

Note that this function is asynchronous with respect to the host unless `pitchedDevPtr` refers to pinned host memory.



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

See also:

[cudaMemset](#), [cudaMemset2D](#), [cudaMemsetAsync](#), [cudaMemset2DAsync](#), [cudaMemset3DAsync](#), [cudaMalloc3D](#), [make\\_cudaPitchedPtr](#), [make\\_cudaExtent](#)

**\_\_host\_\_ \_\_device\_\_ cudaError\_t cudaMemset3DAsync**  
([cudaPitchedPtr](#) pitchedDevPtr, int value, [cudaExtent](#) extent, [cudaStream\\_t](#) stream)

Initializes or sets device memory to a value.

### Parameters

#### **pitchedDevPtr**

- Pointer to pitched device memory

#### **value**

- Value to set for each byte of specified memory

#### **extent**

- Size parameters for where to set device memory (`width` field in bytes)

#### **stream**

- Stream identifier

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#),

## Description

Initializes each element of a 3D array to the specified value `value`. The object to initialize is defined by `pitchedDevPtr`. The `pitch` field of `pitchedDevPtr` is the width in memory in bytes of the 3D array pointed to by `pitchedDevPtr`, including any padding added to the end of each row. The `xsize` field specifies the logical width of each row in bytes, while the `ysize` field specifies the height of each 2D slice in rows.

The extents of the initialized region are specified as a width in bytes, a height in rows, and a depth in slices.

Extents with `width` greater than or equal to the `xsize` of `pitchedDevPtr` may perform significantly faster than extents narrower than the `xsize`. Secondly, extents with `height` equal to the `ysize` of `pitchedDevPtr` will perform faster than when the height is shorter than the `ysize`.

This function performs fastest when the `pitchedDevPtr` has been allocated by `cudaMalloc3D()`.

`cudaMemset3DAsync()` is asynchronous with respect to the host, so the call may return before the memset is complete. The operation can optionally be associated to a stream by passing a non-zero `stream` argument. If `stream` is non-zero, the operation may overlap with operations in other streams.

The device version of this function only handles device to device copies and cannot be given local or shared pointers.



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

## See also:

[cudaMemset](#), [cudaMemset2D](#), [cudaMemset3D](#), [cudaMemsetAsync](#), [cudaMemset2DAsync](#), [cudaMalloc3D](#), [make\\_cudaPitchedPtr](#), [make\\_cudaExtent](#)

**\_\_host\_\_ \_\_device\_\_ cudaError\_t cudaMemsetAsync (void \*devPtr, int value, size\_t count, cudaStream\_t stream)**

Initializes or sets device memory to a value.

### Parameters

#### **devPtr**

- Pointer to device memory

#### **value**

- Value to set for each byte of specified memory

#### **count**

- Size in bytes to set

#### **stream**

- Stream identifier

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#),

### Description

Fills the first `count` bytes of the memory area pointed to by `devPtr` with the constant byte value `value`.

[cudaMemsetAsync\(\)](#) is asynchronous with respect to the host, so the call may return before the memset is complete. The operation can optionally be associated to a stream by passing a non-zero `stream` argument. If `stream` is non-zero, the operation may overlap with operations in other streams.

The device version of this function only handles device to device copies and cannot be given local or shared pointers.



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

**See also:**

[cudaMemset](#), [cudaMemset2D](#), [cudaMemset3D](#), [cudaMemset2DAsync](#),  
[cudaMemset3DAsync](#), [cuMemsetD8Async](#), [cuMemsetD16Async](#), [cuMemsetD32Async](#)

**\_\_host\_\_ make\_cudaExtent (size\_t w, size\_t h, size\_t d)**

Returns a [cudaExtent](#) based on input parameters.

**Parameters****w**

- Width in elements when referring to array memory, in bytes when referring to linear memory

**h**

- Height in elements

**d**

- Depth in elements

**Returns**

[cudaExtent](#) specified by w, h, and d

**Description**

Returns a [cudaExtent](#) based on the specified input parameters w, h, and d.

**See also:**

[make\\_cudaPitchedPtr](#), [make\\_cudaPos](#)

**\_\_host\_\_ make\_cudaPitchedPtr (void \*d, size\_t p, size\_t xsz, size\_t ysz)**

Returns a [cudaPitchedPtr](#) based on input parameters.

**Parameters****d**

- Pointer to allocated memory

**p**

- Pitch of allocated memory in bytes

**xsz**

- Logical width of allocation in elements

**ysz**

- Logical height of allocation in elements

**Returns**

`cudaPitchedPtr` specified by `d`, `p`, `xsz`, and `ysz`

**Description**

Returns a `cudaPitchedPtr` based on the specified input parameters `d`, `p`, `xsz`, and `ysz`.

**See also:**

`make_cudaExtent`, `make_cudaPos`

**\_\_host\_\_ make\_cudaPos (size\_t x, size\_t y, size\_t z)**

Returns a `cudaPos` based on input parameters.

**Parameters**

- x**  
- X position
- y**  
- Y position
- z**  
- Z position

**Returns**

`cudaPos` specified by `x`, `y`, and `z`

**Description**

Returns a `cudaPos` based on the specified input parameters `x`, `y`, and `z`.

**See also:**

`make_cudaExtent`, `make_cudaPitchedPtr`

## 5.10. Memory Management [DEPRECATED]

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

Some functions have overloaded C++ API template versions documented separately in the [C++ API Routines](#) module.

```
__host__ cudaError_t cudaMemcpyArrayToArray
(cudaArray_t dst, size_t wOffsetDst, size_t hOffsetDst,
cudaArray_const_t src, size_t wOffsetSrc, size_t
hOffsetSrc, size_t count, cudaMemcpyKind kind)
```

Copies data between host and device.

### Parameters

#### **dst**

- Destination memory address

#### **wOffsetDst**

- Destination starting X offset

#### **hOffsetDst**

- Destination starting Y offset

#### **src**

- Source memory address

#### **wOffsetSrc**

- Source starting X offset

#### **hOffsetSrc**

- Source starting Y offset

#### **count**

- Size in bytes to copy

#### **kind**

- Type of transfer

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidMemcpyDirection](#)

### Description

#### Deprecated

Copies `count` bytes from the CUDA array `src` starting at the upper left corner (`wOffsetSrc`, `hOffsetSrc`) to the CUDA array `dst` starting at the upper left corner (`wOffsetDst`, `hOffsetDst`) where `kind` specifies the direction of the copy, and must be one of [cudaMemcpyHostToHost](#), [cudaMemcpyHostToDevice](#), [cudaMemcpyDeviceToHost](#), [cudaMemcpyDeviceToDevice](#), or [cudaMemcpyDefault](#). Passing [cudaMemcpyDefault](#) is recommended, in which case the type of transfer is inferred from the pointer values. However, [cudaMemcpyDefault](#) is only allowed on systems that support unified virtual addressing.



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

### See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`, `cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`, `cuMemcpyAtoA`

**`__host__ cudaError_t cudaMemcpyFromArray (void *dst, cudaArray_const_t src, size_t wOffset, size_t hOffset, size_t count, cudaMemcpyKind kind)`**

Copies data between host and device.

### Parameters

#### **dst**

- Destination memory address

#### **src**

- Source memory address

#### **wOffset**

- Source starting X offset

#### **hOffset**

- Source starting Y offset

#### **count**

- Size in bytes to copy

#### **kind**

- Type of transfer

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidMemcpyDirection`



## Description

### Deprecated

Copies `count` bytes from the CUDA array `src` starting at the upper left corner (`wOffset`, `hOffset`) to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing.



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

### See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyArrayToArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`, `cuMemcpyAtoH`, `cuMemcpyAtoD`

```
__host__ cudaError_t cudaMemcpyFromArrayAsync  
(void *dst, cudaArray_const_t src, size_t wOffset,  
size_t hOffset, size_t count, cudaMemcpyKind kind,  
cudaStream_t stream)
```

Copies data between host and device.

### Parameters

#### `dst`

- Destination memory address

**src**

- Source memory address

**wOffset**

- Source starting X offset

**hOffset**

- Source starting Y offset

**count**

- Size in bytes to copy

**kind**

- Type of transfer

**stream**

- Stream identifier

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidMemcpyDirection](#)

**Description****Deprecated**

Copies `count` bytes from the CUDA array `src` starting at the upper left corner (`wOffset`, `hOffset`) to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of [cudaMemcpyHostToHost](#), [cudaMemcpyHostToDevice](#), [cudaMemcpyDeviceToHost](#), [cudaMemcpyDeviceToDevice](#), or [cudaMemcpyDefault](#). Passing [cudaMemcpyDefault](#) is recommended, in which case the type of transfer is inferred from the pointer values. However, [cudaMemcpyDefault](#) is only allowed on systems that support unified virtual addressing.

[cudaMemcpyFromArrayAsync\(\)](#) is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is [cudaMemcpyHostToDevice](#) or [cudaMemcpyDeviceToHost](#) and `stream` is non-zero, the copy may overlap with operations in other streams.



- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ This function exhibits [asynchronous](#) behavior for most use cases.
- ▶ This function uses standard [default stream](#) semantics.
- ▶ Note that this function may also return [cudaErrorInitializationError](#), [cudaErrorInsufficientDriver](#) or [cudaErrorNoDevice](#) if this call tries to initialize internal CUDA RT state.

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

#### See also:

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpyToArray](#), [cudaMemcpy2DToArray](#), [cudaMemcpyFromArray](#), [cudaMemcpy2DFromArray](#), [cudaMemcpyArrayToArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#), [cuMemcpyAtoHAsync](#), [cuMemcpy2DAsync](#)

**\_\_host\_\_ cudaError\_t cudaMemcpyToArray (cudaArray\_t dst, size\_t wOffset, size\_t hOffset, const void \*src, size\_t count, cudaMemcpyKind kind)**

Copies data between host and device.

#### Parameters

##### **dst**

- Destination memory address

##### **wOffset**

- Destination starting X offset

##### **hOffset**

- Destination starting Y offset

##### **src**

- Source memory address

##### **count**

- Size in bytes to copy

##### **kind**

- Type of transfer

#### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidMemcpyDirection](#)

#### Description

##### Deprecated

Copies `count` bytes from the memory area pointed to by `src` to the CUDA array `dst` starting at the upper left corner (`wOffset`, `hOffset`), where `kind` specifies the direction of the copy, and must be one of [cudaMemcpyHostToHost](#),

`cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing.



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

#### See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`,  
`cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyArrayToArray`,  
`cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`,  
`cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`,  
`cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`,  
`cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`,  
`cudaMemcpyFromSymbolAsync`, `cuMemcpyHtoA`, `cuMemcpyDtoA`

**`__host__ cudaError_t cudaMemcpyToArrayAsync`**  
**`(cudaArray_t dst, size_t wOffset, size_t hOffset,`**  
**`const void *src, size_t count, cudaMemcpyKind kind,`**  
**`cudaStream_t stream)`**

Copies data between host and device.

#### Parameters

##### **`dst`**

- Destination memory address

##### **`wOffset`**

- Destination starting X offset

##### **`hOffset`**

- Destination starting Y offset

##### **`src`**

- Source memory address

**count**

- Size in bytes to copy

**kind**

- Type of transfer

**stream**

- Stream identifier

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidMemcpyDirection`

**Description**

## Deprecated

Copies `count` bytes from the memory area pointed to by `src` to the CUDA array `dst` starting at the upper left corner (`wOffset`, `hOffset`), where `kind` specifies the direction of the copy, and must be one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`. Passing `cudaMemcpyDefault` is recommended, in which case the type of transfer is inferred from the pointer values. However, `cudaMemcpyDefault` is only allowed on systems that support unified virtual addressing.

`cudaMemcpyToArrayAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost` and `stream` is non-zero, the copy may overlap with operations in other streams.



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

**See also:**

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`, `cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyArrayToArray`,

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

## 5.11. Unified Addressing

This section describes the unified addressing functions of the CUDA runtime application programming interface.

### Overview

CUDA devices can share a unified address space with the host. For these devices there is no distinction between a device pointer and a host pointer -- the same pointer value may be used to access memory from the host program and from a kernel running on the device (with exceptions enumerated below).

### Supported Platforms

Whether or not a device supports unified addressing may be queried by calling `cudaGetDeviceProperties()` with the device property `cudaDeviceProp::unifiedAddressing`.

Unified addressing is automatically enabled in 64-bit processes .

Unified addressing is not yet supported on Windows Vista or Windows 7 for devices that do not use the TCC driver model.

### Looking Up Information from Pointer Values

It is possible to look up information about the memory which backs a pointer value. For instance, one may want to know if a pointer points to host or device memory. As another example, in the case of device memory, one may want to know on which CUDA device the memory resides. These properties may be queried using the function `cudaPointerGetAttributes()`

Since pointers are unique, it is not necessary to specify information about the pointers specified to `cudaMemcpy()` and other copy functions. The copy direction `cudaMemcpyDefault` may be used to specify that the CUDA runtime should infer the location of the pointer from its value.

### Automatic Mapping of Host Allocated Host Memory

All host memory allocated through all devices using `cudaMallocHost()` and `cudaHostAlloc()` is always directly accessible from all devices that support unified addressing. This is the case regardless of whether or not the flags `cudaHostAllocPortable` and `cudaHostAllocMapped` are specified.

The pointer value through which allocated host memory may be accessed in kernels on all devices that support unified addressing is the same as the pointer value through which that memory is accessed on the host. It is not necessary to call `cudaHostGetDevicePointer()` to get the device pointer for these allocations.

Note that this is not the case for memory allocated using the flag `cudaHostAllocWriteCombined`, as discussed below.

### Direct Access of Peer Memory

Upon enabling direct access from a device that supports unified addressing to another peer device that supports unified addressing using `cudaDeviceEnablePeerAccess()` all memory allocated in the peer device using `cudaMalloc()` and `cudaMallocPitch()` will immediately be accessible by the current device. The device pointer value through which any peer's memory may be accessed in the current device is the same pointer value through which that memory may be accessed from the peer device.

### Exceptions, Disjoint Addressing

Not all memory may be accessed on devices through the same pointer value through which they are accessed on the host. These exceptions are host memory registered using `cudaHostRegister()` and host memory allocated using the flag `cudaHostAllocWriteCombined`. For these exceptions, there exists a distinct host and device address for the memory. The device address is guaranteed to not overlap any valid host pointer range and is guaranteed to have the same value across all devices that support unified addressing.

This device address may be queried using `cudaHostGetDevicePointer()` when a device using unified addressing is current. Either the host or the unified device pointer value may be used to refer to this memory in `cudaMemcpy()` and similar functions using the `cudaMemcpyDefault` memory direction.

## `__host__ cudaError_t cudaPointerGetAttributes (cudaPointerAttributes *attributes, const void *ptr)`

Returns attributes about a specified pointer.

### Parameters

#### **attributes**

- Attributes for the specified pointer

#### **ptr**

- Pointer to get attributes for

### Returns

`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorInvalidValue`

## Description

Returns in `*attributes` the attributes of the pointer `ptr`. If pointer was not allocated in, mapped by or registered with context supporting unified addressing `cudaErrorInvalidValue` is returned.



In CUDA 11.0 forward passing host pointer will return `cudaMemoryTypeUnregistered` in `cudaPointerAttributes::type` and call will return `cudaSuccess`.

The `cudaPointerAttributes` structure is defined as:

```

struct cudaPointerAttributes {
    enum cudaMemoryType
        memoryType;
    enum cudaMemoryType
        type;
    int device;
    void *devicePointer;
    void *hostPointer;
    int isManaged;
}

```

In this structure, the individual fields mean

- ▶ `cudaPointerAttributes::memoryType` identifies the location of the memory associated with pointer `ptr`. It can be `cudaMemoryTypeHost` for host memory or `cudaMemoryTypeDevice` for device and managed memory. It has been deprecated in favour of `cudaPointerAttributes::type`.
- ▶ `cudaPointerAttributes::type` identifies type of memory. It can be `cudaMemoryTypeUnregistered` for unregistered host memory, `cudaMemoryTypeHost` for registered host memory, `cudaMemoryTypeDevice` for device memory or `cudaMemoryTypeManaged` for managed memory.
- ▶ `device` is the device against which `ptr` was allocated. If `ptr` has memory type `cudaMemoryTypeDevice` then this identifies the device on which the memory referred to by `ptr` physically resides. If `ptr` has memory type `cudaMemoryTypeHost` then this identifies the device which was current when the allocation was made (and if that device is deinitialized then this allocation will vanish with that device's state).
- ▶ `devicePointer` is the device pointer alias through which the memory referred to by `ptr` may be accessed on the current device. If the memory referred to by `ptr` cannot be accessed directly by the current device then this is NULL.
- ▶ `hostPointer` is the host pointer alias through which the memory referred to by `ptr` may be accessed on the host. If the memory referred to by `ptr` cannot be accessed directly by the host then this is NULL.
- ▶ `isManaged` indicates if the pointer `ptr` points to managed memory or not. It has been deprecated in favour of `cudaPointerAttributes::type`.





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

See also:

`cudaGetDeviceCount`, `cudaGetDevice`, `cudaSetDevice`, `cudaChooseDevice`,  
`cuPointerGetAttributes`

## 5.12. Peer Device Memory Access

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

**`__host__ cudaError_t cudaDeviceCanAccessPeer (int *canAccessPeer, int device, int peerDevice)`**

Queries if a device may directly access a peer device's memory.

### Parameters

#### **`canAccessPeer`**

- Returned access capability

#### **`device`**

- Device from which allocations on `peerDevice` are to be directly accessed.

#### **`peerDevice`**

- Device on which the allocations to be directly accessed by `device` reside.

### Returns

`cudaSuccess`, `cudaErrorInvalidDevice`

### Description

Returns in `*canAccessPeer` a value of 1 if device `device` is capable of directly accessing memory from `peerDevice` and 0 otherwise. If direct access of `peerDevice` from `device` is possible, then access may be enabled by calling `cudaDeviceEnablePeerAccess()`.



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

#### See also:

`cudaDeviceEnablePeerAccess`, `cudaDeviceDisablePeerAccess`, `cuDeviceCanAccessPeer`

## `__host__ cudaError_t cudaDeviceDisablePeerAccess (int peerDevice)`

Disables direct access to memory allocations on a peer device.

#### Parameters

##### `peerDevice`

- Peer device to disable direct access to

#### Returns

`cudaSuccess`, `cudaErrorPeerAccessNotEnabled`, `cudaErrorInvalidDevice`

#### Description

Returns `cudaErrorPeerAccessNotEnabled` if direct access to memory on `peerDevice` has not yet been enabled from the current device.



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

#### See also:

[cudaDeviceCanAccessPeer](#), [cudaDeviceEnablePeerAccess](#), [cuCtxDisablePeerAccess](#)

## `__host__ cudaError_t cudaDeviceEnablePeerAccess (int peerDevice, unsigned int flags)`

Enables direct access to memory allocations on a peer device.

### Parameters

#### **peerDevice**

- Peer device to enable direct access to from the current device

#### **flags**

- Reserved for future use and must be set to 0

### Returns

[cudaSuccess](#), [cudaErrorInvalidDevice](#), [cudaErrorPeerAccessAlreadyEnabled](#), [cudaErrorInvalidValue](#)

### Description

On success, all allocations from `peerDevice` will immediately be accessible by the current device. They will remain accessible until access is explicitly disabled using [cudaDeviceDisablePeerAccess\(\)](#) or either device is reset using [cudaDeviceReset\(\)](#).

Note that access granted by this call is unidirectional and that in order to access memory on the current device from `peerDevice`, a separate symmetric call to [cudaDeviceEnablePeerAccess\(\)](#) is required.

Note that there are both device-wide and system-wide limitations per system configuration, as noted in the CUDA Programming Guide under the section "Peer-to-Peer Memory Access".

Returns [cudaErrorInvalidDevice](#) if [cudaDeviceCanAccessPeer\(\)](#) indicates that the current device cannot directly access memory from `peerDevice`.

Returns [cudaErrorPeerAccessAlreadyEnabled](#) if direct access of `peerDevice` from the current device has already been enabled.

Returns [cudaErrorInvalidValue](#) if `flags` is not 0.



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

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

See also:

[cudaDeviceCanAccessPeer](#), [cudaDeviceDisablePeerAccess](#), [cuCtxEnablePeerAccess](#)

## 5.13. OpenGL Interoperability

This section describes the OpenGL interoperability functions of the CUDA runtime application programming interface. Note that mapping of OpenGL resources is performed with the graphics API agnostic, resource mapping interface described in [Graphics Interopability](#).

### enum cudaGLDeviceList

CUDA devices corresponding to the current OpenGL context

#### Values

**cudaGLDeviceListAll = 1**

The CUDA devices for all GPUs used by the current OpenGL context

**cudaGLDeviceListCurrentFrame = 2**

The CUDA devices for the GPUs used by the current OpenGL context in its currently rendering frame

**cudaGLDeviceListNextFrame = 3**

The CUDA devices for the GPUs to be used by the current OpenGL context in the next frame

**\_\_host\_\_ cudaError\_t cudaGLGetDevices (unsigned int \*pCudaDeviceCount, int \*pCudaDevices, unsigned int cudaDeviceCount, cudaGLDeviceList deviceList)**

Gets the CUDA devices associated with the current OpenGL context.

#### Parameters

**pCudaDeviceCount**

- Returned number of CUDA devices corresponding to the current OpenGL context

**pCudaDevices**

- Returned CUDA devices corresponding to the current OpenGL context

**cudaDeviceCount**

- The size of the output device array `pCudaDevices`

**deviceList**

- The set of devices to return. This set may be `cudaGLDeviceListAll` for all devices, `cudaGLDeviceListCurrentFrame` for the devices used to render the current frame (in SLI), or `cudaGLDeviceListNextFrame` for the devices used to render the next frame (in SLI).

**Returns**

`cudaSuccess`, `cudaErrorNoDevice`, `cudaErrorInvalidGraphicsContext`, `cudaErrorUnknown`

**Description**

Returns in `*pCudaDeviceCount` the number of CUDA-compatible devices corresponding to the current OpenGL context. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of the CUDA-compatible devices corresponding to the current OpenGL context. If any of the GPUs being used by the current OpenGL context are not CUDA capable then the call will return `cudaErrorNoDevice`.



- This function is not supported on Mac OS X.
- Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

`cudaGraphicsUnregisterResource`, `cudaGraphicsMapResources`, `cudaGraphicsSubResourceGetMappedArray`, `cudaGraphicsResourceGetMappedPointer`, `cuGLGetDevices`

## `__host__ cudaError_t cudaGraphicsGLRegisterBuffer(cudaGraphicsResource **resource, GLuint buffer, unsigned int flags)`

Registers an OpenGL buffer object.

**Parameters****resource**

- Pointer to the returned object handle

**buffer**

- name of buffer object to be registered

**flags**

- Register flags

## Returns

`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorInvalidValue`,  
`cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

## Description

Registers the buffer object specified by `buffer` for access by CUDA. A handle to the registered object is returned as `resource`. The register flags `flags` specify the intended usage, as follows:

- ▶ `cudaGraphicsRegisterFlagsNone`: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- ▶ `cudaGraphicsRegisterFlagsReadOnly`: Specifies that CUDA will not write to this resource.
- ▶ `cudaGraphicsRegisterFlagsWriteDiscard`: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.



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

## See also:

`cudaGraphicsUnregisterResource`, `cudaGraphicsMapResources`,  
`cudaGraphicsResourceGetMappedPointer`, `cuGraphicsGLRegisterBuffer`

**`__host__ cudaError_t cudaGraphicsGLRegisterImage`**  
**`(cudaGraphicsResource **resource, GLuint image,`**  
**`GLenum target, unsigned int flags)`**

Register an OpenGL texture or renderbuffer object.

## Parameters

### **resource**

- Pointer to the returned object handle

### **image**

- name of texture or renderbuffer object to be registered

### **target**

- Identifies the type of object specified by `image`

### **flags**

- Register flags

## Returns

`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorInvalidValue`,  
`cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

## Description

Registers the texture or renderbuffer object specified by `image` for access by CUDA. A handle to the registered object is returned as `resource`.

`target` must match the type of the object, and must be one of `GL_TEXTURE_2D`, `GL_TEXTURE_RECTANGLE`, `GL_TEXTURE_CUBE_MAP`, `GL_TEXTURE_3D`, `GL_TEXTURE_2D_ARRAY`, or `GL_RENDERBUFFER`.

The register flags `flags` specify the intended usage, as follows:

- ▶ `cudaGraphicsRegisterFlagsNone`: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- ▶ `cudaGraphicsRegisterFlagsReadOnly`: Specifies that CUDA will not write to this resource.
- ▶ `cudaGraphicsRegisterFlagsWriteDiscard`: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.
- ▶ `cudaGraphicsRegisterFlagsSurfaceLoadStore`: Specifies that CUDA will bind this resource to a surface reference.
- ▶ `cudaGraphicsRegisterFlagsTextureGather`: Specifies that CUDA will perform texture gather operations on this resource.

The following image formats are supported. For brevity's sake, the list is abbreviated. For ex., {`GL_R`, `GL_RG`} X {8, 16} would expand to the following 4 formats {`GL_R8`, `GL_R16`, `GL_RG8`, `GL_RG16`} :

- ▶ `GL_RED`, `GL_RG`, `GL_RGBA`, `GL_LUMINANCE`, `GL_ALPHA`,  
`GL_LUMINANCE_ALPHA`, `GL_INTENSITY`
- ▶ {`GL_R`, `GL_RG`, `GL_RGBA`} X {8, 16, 16F, 32F, 8UI, 16UI, 32UI, 8I, 16I, 32I}
- ▶ {`GL_LUMINANCE`, `GL_ALPHA`, `GL_LUMINANCE_ALPHA`, `GL_INTENSITY`}  
X {8, 16, 16F\_ARB, 32F\_ARB, 8UI\_EXT, 16UI\_EXT, 32UI\_EXT, 8I\_EXT, 16I\_EXT,  
32I\_EXT}

The following image classes are currently disallowed:

- ▶ Textures with borders
- ▶ Multisampled renderbuffers



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

#### See also:

[cudaGraphicsUnregisterResource](#), [cudaGraphicsMapResources](#),  
[cudaGraphicsSubResourceGetMappedArray](#), [cuGraphicsGLRegisterImage](#)

## `__host__ cudaError_t cudaWGLGetDevice (int *device, HGPUNV hGpu)`

Gets the CUDA device associated with hGpu.

#### Parameters

##### **device**

- Returns the device associated with hGpu, or -1 if hGpu is not a compute device.

##### **hGpu**

- Handle to a GPU, as queried via `WGL_NV_gpu_affinity`

#### Returns

[cudaSuccess](#)

#### Description

Returns the CUDA device associated with a hGpu, if applicable.



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

#### See also:

[WGL\\_NV\\_gpu\\_affinity](#), [cuWGLGetDevice](#)

## 5.14. OpenGL Interoperability [DEPRECATED]

This section describes deprecated OpenGL interoperability functionality.

### `enum cudaGLMapFlags`

CUDA GL Map Flags



## Values

**cudaGLMapFlagsNone = 0**

Default; Assume resource can be read/written

**cudaGLMapFlagsReadOnly = 1**

CUDA kernels will not write to this resource

**cudaGLMapFlagsWriteDiscard = 2**

CUDA kernels will only write to and will not read from this resource

**\_\_host\_\_ cudaError\_t cudaGLMapBufferObject (void  
\*\*devPtr, GLuint bufObj)**

Maps a buffer object for access by CUDA.

## Parameters

**devPtr**

- Returned device pointer to CUDA object

**bufObj**

- Buffer object ID to map

## Returns

[cudaSuccess](#), [cudaErrorMapBufferObjectFailed](#)

## Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Maps the buffer object of ID `bufObj` into the address space of CUDA and returns in `*devPtr` the base pointer of the resulting mapping. The buffer must have previously been registered by calling [cudaGLRegisterBufferObject\(\)](#). While a buffer is mapped by CUDA, any OpenGL operation which references the buffer will result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

All streams in the current thread are synchronized with the current GL context.



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

## See also:

[cudaGraphicsMapResources](#)

```
__host__ cudaError_t cudaGLMapBufferObjectAsync  
(void **devPtr, GLuint bufObj, cudaStream_t stream)
```

Maps a buffer object for access by CUDA.

### Parameters

#### **devPtr**

- Returned device pointer to CUDA object

#### **bufObj**

- Buffer object ID to map

#### **stream**

- Stream to synchronize

### Returns

[cudaSuccess](#), [cudaErrorMapBufferObjectFailed](#)

### Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Maps the buffer object of ID `bufObj` into the address space of CUDA and returns in `*devPtr` the base pointer of the resulting mapping. The buffer must have previously been registered by calling [cudaGLRegisterBufferObject\(\)](#). While a buffer is mapped by CUDA, any OpenGL operation which references the buffer will result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

Stream /p stream is synchronized with the current GL context.



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

### See also:

[cudaGraphicsMapResources](#)

## `__host__ cudaError_t cudaGLRegisterBufferObject (GLuint bufObj)`

Registers a buffer object for access by CUDA.

### Parameters

#### **bufObj**

- Buffer object ID to register

### Returns

`cudaSuccess`, `cudaErrorInitializationError`

### Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Registers the buffer object of ID `bufObj` for access by CUDA. This function must be called before CUDA can map the buffer object. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.



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

### See also:

`cudaGraphicsGLRegisterBuffer`

## `__host__ cudaError_t cudaGLSetBufferObjectMapFlags (GLuint bufObj, unsigned int flags)`

Set usage flags for mapping an OpenGL buffer.

### Parameters

#### **bufObj**

- Registered buffer object to set flags for

#### **flags**

- Parameters for buffer mapping

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

## Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Set flags for mapping the OpenGL buffer `bufObj`

Changes to flags will take effect the next time `bufObj` is mapped. The `flags` argument may be any of the following:

- ▶ **`cudaGLMapFlagsNone`**: Specifies no hints about how this buffer will be used. It is therefore assumed that this buffer will be read from and written to by CUDA kernels. This is the default value.
- ▶ **`cudaGLMapFlagsReadOnly`**: Specifies that CUDA kernels which access this buffer will not write to the buffer.
- ▶ **`cudaGLMapFlagsWriteDiscard`**: Specifies that CUDA kernels which access this buffer will not read from the buffer and will write over the entire contents of the buffer, so none of the data previously stored in the buffer will be preserved.

If `bufObj` has not been registered for use with CUDA, then **`cudaErrorInvalidResourceHandle`** is returned. If `bufObj` is presently mapped for access by CUDA, then **`cudaErrorUnknown`** is returned.



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

See also:

**`cudaGraphicsResourceSetMapFlags`**

## **`__host__ cudaError_t cudaGLSetGLDevice (int device)`**

Sets a CUDA device to use OpenGL interoperability.

### Parameters

**device**

- Device to use for OpenGL interoperability

### Returns

**`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorSetOnActiveProcess`**

## Description

**Deprecated** This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with an OpenGL context in order to achieve maximum interoperability performance.



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

**See also:**

[cudaGraphicsGLRegisterBuffer](#), [cudaGraphicsGLRegisterImage](#)

## **\_\_host\_\_ cudaError\_t cudaGLUnmapBufferObject (GLuint bufObj)**

Unmaps a buffer object for access by CUDA.

### **Parameters**

**bufObj**

- Buffer object to unmap

### **Returns**

[cudaSuccess](#), [cudaErrorUnmapBufferObjectFailed](#)

### **Description**

**Deprecated** This function is deprecated as of CUDA 3.0.

Unmaps the buffer object of ID `bufObj` for access by CUDA. When a buffer is unmapped, the base address returned by [cudaGLMapBufferObject\(\)](#) is invalid and subsequent references to the address result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

All streams in the current thread are synchronized with the current GL context.



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

**See also:**

[cudaGraphicsUnmapResources](#)

## `__host__ cudaError_t cudaGLUnmapBufferObjectAsync` (GLuint bufObj, cudaStream\_t stream)

Unmaps a buffer object for access by CUDA.

### Parameters

#### **bufObj**

- Buffer object to unmap

#### **stream**

- Stream to synchronize

### Returns

`cudaSuccess`, `cudaErrorUnmapBufferObjectFailed`

### Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Unmaps the buffer object of ID `bufObj` for access by CUDA. When a buffer is unmapped, the base address returned by `cudaGLMapBufferObject()` is invalid and subsequent references to the address result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

Stream /p stream is synchronized with the current GL context.



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

### See also:

`cudaGraphicsUnmapResources`

## `__host__ cudaError_t cudaGLUnregisterBufferObject` (GLuint bufObj)

Unregisters a buffer object for access by CUDA.

### Parameters

#### **bufObj**

- Buffer object to unregister

**Returns**`cudaSuccess`**Description**

**Deprecated** This function is deprecated as of CUDA 3.0.

Unregisters the buffer object of ID `bufObj` for access by CUDA and releases any CUDA resources associated with the buffer. Once a buffer is unregistered, it may no longer be mapped by CUDA. The GL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.



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

**See also:**`cudaGraphicsUnregisterResource`

## 5.15. Direct3D 9 Interoperability

This section describes the Direct3D 9 interoperability functions of the CUDA runtime application programming interface. Note that mapping of Direct3D 9 resources is performed with the graphics API agnostic, resource mapping interface described in [Graphics Interopability](#).

### enum `cudaD3D9DeviceList`

CUDA devices corresponding to a D3D9 device

**Values**

**`cudaD3D9DeviceListAll = 1`**

The CUDA devices for all GPUs used by a D3D9 device

**`cudaD3D9DeviceListCurrentFrame = 2`**

The CUDA devices for the GPUs used by a D3D9 device in its currently rendering frame

**`cudaD3D9DeviceListNextFrame = 3`**

The CUDA devices for the GPUs to be used by a D3D9 device in the next frame

## **\_\_host\_\_ cudaError\_t cudaD3D9GetDevice (int \*device, const char \*pszAdapterName)**

Gets the device number for an adapter.

### **Parameters**

#### **device**

- Returns the device corresponding to pszAdapterName

#### **pszAdapterName**

- D3D9 adapter to get device for

### **Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorUnknown](#)

### **Description**

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



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

### **See also:**

[cudaD3D9SetDirect3DDevice](#), [cudaGraphicsD3D9RegisterResource](#), [cuD3D9GetDevice](#)

## **\_\_host\_\_ cudaError\_t cudaD3D9GetDevices (unsigned int \*pCudaDeviceCount, int \*pCudaDevices, unsigned int cudaDeviceCount, IDirect3DDevice9 \*pD3D9Device, cudaD3D9DeviceList deviceList)**

Gets the CUDA devices corresponding to a Direct3D 9 device.

### **Parameters**

#### **pCudaDeviceCount**

- Returned number of CUDA devices corresponding to pD3D9Device

#### **pCudaDevices**

- Returned CUDA devices corresponding to pD3D9Device



**cudaDeviceCount**

- The size of the output device array `pCudaDevices`

**pD3D9Device**

- Direct3D 9 device to query for CUDA devices

**deviceList**

- The set of devices to return. This set may be `cudaD3D9DeviceListAll` for all devices, `cudaD3D9DeviceListCurrentFrame` for the devices used to render the current frame (in SLI), or `cudaD3D9DeviceListNextFrame` for the devices used to render the next frame (in SLI).

**Returns**

`cudaSuccess`, `cudaErrorNoDevice`, `cudaErrorUnknown`

**Description**

Returns in `*pCudaDeviceCount` the number of CUDA-compatible devices corresponding to the Direct3D 9 device `pD3D9Device`. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of the the CUDA-compatible devices corresponding to the Direct3D 9 device `pD3D9Device`.

If any of the GPUs being used to render `pDevice` are not CUDA capable then the call will return `cudaErrorNoDevice`.



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

**See also:**

`cudaGraphicsUnregisterResource`, `cudaGraphicsMapResources`,  
`cudaGraphicsSubResourceGetMappedArray`, `cudaGraphicsResourceGetMappedPointer`,  
`cuD3D9GetDevices`

## **\_\_host\_\_ cudaError\_t cudaD3D9GetDirect3DDevice (IDirect3DDevice9 \*\*ppD3D9Device)**

Gets the Direct3D device against which the current CUDA context was created.

**Parameters****ppD3D9Device**

- Returns the Direct3D device for this thread

**Returns**

`cudaSuccess`, `cudaErrorInvalidGraphicsContext`, `cudaErrorUnknown`

## Description

Returns in `*ppD3D9Device` the Direct3D device against which this CUDA context was created in `cudaD3D9SetDirect3DDevice()`.



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

## See also:

`cudaD3D9SetDirect3DDevice`, `cuD3D9GetDirect3DDevice`

## `__host__ cudaError_t cudaD3D9SetDirect3DDevice` (IDirect3DDevice9 \*pD3D9Device, int device)

Sets the Direct3D 9 device to use for interoperability with a CUDA device.

## Parameters

### `pD3D9Device`

- Direct3D device to use for this thread

### `device`

- The CUDA device to use. This device must be among the devices returned when querying `cudaD3D9DeviceListAll` from `cudaD3D9GetDevices`, may be set to -1 to automatically select an appropriate CUDA device.

## Returns

`cudaSuccess`, `cudaErrorInitializationError`, `cudaErrorInvalidValue`,  
`cudaErrorSetOnActiveProcess`

## Description

Records `pD3D9Device` as the Direct3D 9 device to use for Direct3D 9 interoperability with the CUDA device `device` and sets `device` as the current device for the calling host thread.

If `device` has already been initialized then this call will fail with the error `cudaErrorSetOnActiveProcess`. In this case it is necessary to reset `device` using `cudaDeviceReset()` before Direct3D 9 interoperability on `device` may be enabled.

Successfully initializing CUDA interoperability with `pD3D9Device` will increase the internal reference count on `pD3D9Device`. This reference count will be decremented when `device` is reset using `cudaDeviceReset()`.

Note that this function is never required for correct functionality. Use of this function will result in accelerated interoperability only when the operating system is Windows

Vista or Windows 7, and the device `pD3DDevice` is not an `IDirect3DDevice9Ex`. In all other circumstances, this function is not necessary.



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

See also:

[cudaD3D9GetDevice](#), [cudaGraphicsD3D9RegisterResource](#), [cudaDeviceReset](#)

**\_\_host\_\_ cudaError\_t  
cudaGraphicsD3D9RegisterResource  
(cudaGraphicsResource \*\*resource, IDirect3DResource9  
\*pD3DResource, unsigned int flags)**

Register a Direct3D 9 resource for access by CUDA.

### Parameters

#### **resource**

- Pointer to returned resource handle

#### **pD3DResource**

- Direct3D resource to register

#### **flags**

- Parameters for resource registration

### Returns

[cudaSuccess](#), [cudaErrorInvalidDevice](#), [cudaErrorInvalidValue](#),  
[cudaErrorInvalidResourceHandle](#), [cudaErrorUnknown](#)

### Description

Registers the Direct3D 9 resource `pD3DResource` for access by CUDA.

If this call is successful then the application will be able to map and unmap this resource until it is unregistered through [cudaGraphicsUnregisterResource\(\)](#). Also on success, this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through [cudaGraphicsUnregisterResource\(\)](#).

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of `pD3DResource` must be one of the following.

- ▶ `IDirect3DVertexBuffer9`: may be accessed through a device pointer
- ▶ `IDirect3DIndexBuffer9`: may be accessed through a device pointer
- ▶ `IDirect3DSurface9`: may be accessed through an array. Only stand-alone objects of type `IDirect3DSurface9` may be explicitly shared. In particular, individual mipmap levels and faces of cube maps may not be registered directly. To access individual surfaces associated with a texture, one must register the base texture object.
- ▶ `IDirect3DBaseTexture9`: individual surfaces on this texture may be accessed through an array.

The `flags` argument may be used to specify additional parameters at register time. The valid values for this parameter are

- ▶ `cudaGraphicsRegisterFlagsNone`: Specifies no hints about how this resource will be used.
- ▶ `cudaGraphicsRegisterFlagsSurfaceLoadStore`: Specifies that CUDA will bind this resource to a surface reference.
- ▶ `cudaGraphicsRegisterFlagsTextureGather`: Specifies that CUDA will perform texture gather operations on this resource.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- ▶ The primary rendertarget may not be registered with CUDA.
- ▶ Resources allocated as shared may not be registered with CUDA.
- ▶ Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- ▶ Surfaces of depth or stencil formats cannot be shared.

A complete list of supported formats is as follows:

- ▶ `D3DFMT_L8`
- ▶ `D3DFMT_L16`
- ▶ `D3DFMT_A8R8G8B8`
- ▶ `D3DFMT_X8R8G8B8`
- ▶ `D3DFMT_G16R16`
- ▶ `D3DFMT_A8B8G8R8`
- ▶ `D3DFMT_A8`
- ▶ `D3DFMT_A8L8`
- ▶ `D3DFMT_Q8W8V8U8`
- ▶ `D3DFMT_V16U16`
- ▶ `D3DFMT_A16B16G16R16F`
- ▶ `D3DFMT_A16B16G16R16`
- ▶ `D3DFMT_R32F`
- ▶ `D3DFMT_G16R16F`

- ▶ D3DFMT\_A32B32G32R32F
- ▶ D3DFMT\_G32R32F
- ▶ D3DFMT\_R16F

If `pD3DResource` is of incorrect type or is already registered, then `cudaErrorInvalidResourceHandle` is returned. If `pD3DResource` cannot be registered, then `cudaErrorUnknown` is returned.



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

See also:

`cudaD3D9SetDirect3DDevice`, `cudaGraphicsUnregisterResource`,  
`cudaGraphicsMapResources`, `cudaGraphicsSubResourceGetMappedArray`,  
`cudaGraphicsResourceGetMappedPointer`, `cuGraphicsD3D9RegisterResource`

## 5.16. Direct3D 9 Interoperability [DEPRECATED]

This section describes deprecated Direct3D 9 interoperability functions.

### enum `cudaD3D9MapFlags`

CUDA D3D9 Map Flags

#### Values

**`cudaD3D9MapFlagsNone = 0`**

Default; Assume resource can be read/written

**`cudaD3D9MapFlagsReadOnly = 1`**

CUDA kernels will not write to this resource

**`cudaD3D9MapFlagsWriteDiscard = 2`**

CUDA kernels will only write to and will not read from this resource

### enum `cudaD3D9RegisterFlags`

CUDA D3D9 Register Flags

#### Values

**`cudaD3D9RegisterFlagsNone = 0`**

Default; Resource can be accessed through a void\*

**`cudaD3D9RegisterFlagsArray = 1`**

Resource can be accessed through a CUarray\*

**\_\_host\_\_ cudaError\_t cudaD3D9MapResources (int count, IDirect3DResource9 \*\*ppResources)**

Map Direct3D resources for access by CUDA.

### Parameters

#### count

- Number of resources to map for CUDA

#### ppResources

- Resources to map for CUDA

### Returns

[cudaSuccess](#), [cudaErrorInvalidResourceHandle](#), [cudaErrorUnknown](#)

### Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Maps the `count` Direct3D resources in `ppResources` for access by CUDA.

The resources in `ppResources` may be accessed in CUDA kernels until they are unmapped. Direct3D should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any Direct3D calls issued before [cudaD3D9MapResources\(\)](#) will complete before any CUDA kernels issued after [cudaD3D9MapResources\(\)](#) begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries then [cudaErrorInvalidResourceHandle](#) is returned. If any of `ppResources` are presently mapped for access by CUDA then [cudaErrorUnknown](#) is returned.



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

### See also:

[cudaGraphicsMapResources](#)

## `__host__ cudaError_t cudaD3D9RegisterResource` (`IDirect3DResource9 *pResource`, unsigned int flags)

Registers a Direct3D resource for access by CUDA.

### Parameters

#### **pResource**

- Resource to register

#### **flags**

- Parameters for resource registration

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`,  
`cudaErrorUnknown`

### Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Registers the Direct3D resource `pResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through `cudaD3D9UnregisterResource()`. Also on success, this call will increase the internal reference count on `pResource`. This reference count will be decremented when this resource is unregistered through `cudaD3D9UnregisterResource()`.

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of `pResource` must be one of the following.

- ▶ `IDirect3DVertexBuffer9`: No notes.
- ▶ `IDirect3DIndexBuffer9`: No notes.
- ▶ `IDirect3DSurface9`: Only stand-alone objects of type `IDirect3DSurface9` may be explicitly shared. In particular, individual mipmap levels and faces of cube maps may not be registered directly. To access individual surfaces associated with a texture, one must register the base texture object.
- ▶ `IDirect3DBaseTexture9`: When a texture is registered, all surfaces associated with all mipmap levels of all faces of the texture will be accessible to CUDA.

The `flags` argument specifies the mechanism through which CUDA will access the Direct3D resource. The following value is allowed:

- ▶ `cudaD3D9RegisterFlagsNone`: Specifies that CUDA will access this resource through a `void*`. The pointer, size, and pitch for each subresource of this

resource may be queried through `cudaD3D9ResourceGetMappedPointer()`, `cudaD3D9ResourceGetMappedSize()`, and `cudaD3D9ResourceGetMappedPitch()` respectively. This option is valid for all resource types.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations:

- ▶ The primary rendertarget may not be registered with CUDA.
- ▶ Resources allocated as shared may not be registered with CUDA.
- ▶ Any resources allocated in `D3DPOOL_SYSTEMMEM` or `D3DPOOL_MANAGED` may not be registered with CUDA.
- ▶ Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- ▶ Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized on this context, then `cudaErrorInvalidDevice` is returned. If `pResource` is of incorrect type (e.g. is a non-stand-alone `IDirect3DSurface9`) or is already registered, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` cannot be registered then `cudaErrorUnknown` is returned.



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

See also:

`cudaGraphicsD3D9RegisterResource`

## `__host__ cudaError_t cudaD3D9ResourceGetMappedArray(cudaArray **ppArray, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)`

Get an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

### Parameters

#### **ppArray**

- Returned array corresponding to subresource

#### **pResource**

- Mapped resource to access

#### **face**

- Face of resource to access



**level**

- Level of resource to access

**Returns**

[cudaSuccess](#), [cudaErrorInvalidResourceHandle](#), [cudaErrorUnknown](#)

**Description**

**Deprecated** This function is deprecated as of CUDA 3.0.

Returns in `*pArray` an array through which the subresource of the mapped Direct3D resource `pResource`, which corresponds to `face` and `level` may be accessed. The value set in `pArray` may change every time that `pResource` is mapped.

If `pResource` is not registered then [cudaErrorInvalidResourceHandle](#) is returned. If `pResource` was not registered with usage flags [cudaD3D9RegisterFlagsArray](#), then [cudaErrorInvalidResourceHandle](#) is returned. If `pResource` is not mapped, then [cudaErrorUnknown](#) is returned.

For usage requirements of `face` and `level` parameters, see [cudaD3D9ResourceGetMappedPointer\(\)](#).



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

**See also:**

[cudaGraphicsSubResourceGetMappedArray](#)

**`__host__ cudaError_t cudaD3D9ResourceGetMappedPitch(`  
**`size_t *pPitch, size_t *pPitchSlice, IDirect3DResource9`  
**`*pResource, unsigned int face, unsigned int level)`******

Get the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.

**Parameters****pPitch**

- Returned pitch of subresource

**pPitchSlice**

- Returned Z-slice pitch of subresource

**pResource**

- Mapped resource to access

**face**

- Face of resource to access

**level**

- Level of resource to access

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidResourceHandle](#),  
[cudaErrorUnknown](#)

**Description**

**Deprecated** This function is deprecated as of CUDA 3.0.

Returns in `*pPitch` and `*pPitchSlice` the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource `pResource`, which corresponds to `face` and `level`. The values set in `pPitch` and `pPitchSlice` may change every time that `pResource` is mapped.

The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position `x`, `y` from the base pointer of the surface is:

**`y * pitch + (bytes per pixel) * x`**

For a 3D surface, the byte offset of the sample at position `x`, `y`, `z` from the base pointer of the surface is:

**`z * slicePitch + y * pitch + (bytes per pixel) * x`**

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

If `pResource` is not of type `IDirect3DBaseTexture9` or one of its sub-types or if `pResource` has not been registered for use with CUDA, then [cudaErrorInvalidResourceHandle](#) is returned. If `pResource` was not registered with usage flags [cudaD3D9RegisterFlagsNone](#), then [cudaErrorInvalidResourceHandle](#) is returned. If `pResource` is not mapped for access by CUDA then [cudaErrorUnknown](#) is returned.

For usage requirements of `face` and `level` parameters, see [cudaD3D9ResourceGetMappedPointer\(\)](#).



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

**See also:**

`cudaGraphicsResourceGetMappedPointer`

`__host__ cudaError_t`

`cudaD3D9ResourceGetMappedPointer (void **pPointer,  
IDirect3DResource9 *pResource, unsigned int face,  
unsigned int level)`

Get a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

### Parameters

#### **pPointer**

- Returned pointer corresponding to subresource

#### **pResource**

- Mapped resource to access

#### **face**

- Face of resource to access

#### **level**

- Level of resource to access

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`,  
`cudaErrorUnknown`

### Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Returns in `*pPointer` the base pointer of the subresource of the mapped Direct3D resource `pResource`, which corresponds to `face` and `level`. The value set in `pPointer` may change every time that `pResource` is mapped.

If `pResource` is not registered, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` was not registered with usage flags `cudaD3D9RegisterFlagsNone`, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` is not mapped, then `cudaErrorUnknown` is returned.

If `pResource` is of type `IDirect3DCubeTexture9`, then `face` must one of the values enumerated by type `D3DCUBEMAP_FACES`. For all other types, `face` must be 0. If `face` is invalid, then `cudaErrorInvalidValue` is returned.

If `pResource` is of type `IDirect3DBaseTexture9`, then `level` must correspond to a valid mipmap level. Only mipmap level 0 is supported for now. For all other types `level` must be 0. If `level` is invalid, then `cudaErrorInvalidValue` is returned.



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

See also:

[cudaGraphicsResourceGetMappedPointer](#)

## **\_\_host\_\_ cudaError\_t cudaD3D9ResourceGetMappedSize (size\_t \*pSize, IDirect3DResource9 \*pResource, unsigned int face, unsigned int level)**

Get the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.

### **Parameters**

#### **pSize**

- Returned size of subresource

#### **pResource**

- Mapped resource to access

#### **face**

- Face of resource to access

#### **level**

- Level of resource to access

### **Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidResourceHandle](#),  
[cudaErrorUnknown](#)

### **Description**

**Deprecated** This function is deprecated as of CUDA 3.0.

Returns in `*pSize` the size of the subresource of the mapped Direct3D resource `pResource`, which corresponds to `face` and `level`. The value set in `pSize` may change every time that `pResource` is mapped.

If `pResource` has not been registered for use with CUDA then [cudaErrorInvalidResourceHandle](#) is returned. If `pResource` was not registered with usage flags [cudaD3D9RegisterFlagsNone](#), then [cudaErrorInvalidResourceHandle](#) is returned. If `pResource` is not mapped for access by CUDA then [cudaErrorUnknown](#) is returned.

For usage requirements of `face` and `level` parameters, see [cudaD3D9ResourceGetMappedPointer\(\)](#).



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

See also:

[cudaGraphicsResourceGetMappedPointer](#)

```
__host__ cudaError_t
cudaD3D9ResourceGetSurfaceDimensions (size_t
*pWidth, size_t *pHeight, size_t *pDepth,
IDirect3DResource9 *pResource, unsigned int face,
unsigned int level)
```

Get the dimensions of a registered Direct3D surface.

### Parameters

#### **pWidth**

- Returned width of surface

#### **pHeight**

- Returned height of surface

#### **pDepth**

- Returned depth of surface

#### **pResource**

- Registered resource to access

#### **face**

- Face of resource to access

#### **level**

- Level of resource to access

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidResourceHandle](#),

### Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Returns in *\*pWidth*, *\*pHeight*, and *\*pDepth* the dimensions of the subresource of the mapped Direct3D resource *pResource* which corresponds to *face* and *level*.

Since anti-aliased surfaces may have multiple samples per pixel, it is possible that the dimensions of a resource will be an integer factor larger than the dimensions reported by the Direct3D runtime.

The parameters `pWidth`, `pHeight`, and `pDepth` are optional. For 2D surfaces, the value returned in `*pDepth` will be 0.

If `pResource` is not of type `IDirect3DBaseTexture9` or `IDirect3DSurface9` or if `pResource` has not been registered for use with CUDA, then `cudaErrorInvalidResourceHandle` is returned.

For usage requirements of `face` and `level` parameters, see `cudaD3D9ResourceGetMappedPointer`.



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

See also:

`cudaGraphicsSubResourceGetMappedArray`

## `__host__ cudaError_t cudaD3D9ResourceSetMapFlags` (`IDirect3DResource9 *pResource`, unsigned int flags)

Set usage flags for mapping a Direct3D resource.

### Parameters

#### `pResource`

- Registered resource to set flags for

#### flags

- Parameters for resource mapping

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

### Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Set flags for mapping the Direct3D resource `pResource`.

Changes to flags will take effect the next time `pResource` is mapped. The `flags` argument may be any of the following:

- `cudaD3D9MapFlagsNone`: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.

- ▶ `cudaD3D9MapFlagsReadOnly`: Specifies that CUDA kernels which access this resource will not write to this resource.
- ▶ `cudaD3D9MapFlagsWriteDiscard`: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If `pResource` has not been registered for use with CUDA, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` is presently mapped for access by CUDA, then `cudaErrorUnknown` is returned.



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

See also:

`cudaInteropResourceSetMapFlags`

## `__host__ cudaError_t cudaD3D9UnmapResources (int count, IDirect3DResource9 **ppResources)`

Unmap Direct3D resources for access by CUDA.

### Parameters

#### **count**

- Number of resources to unmap for CUDA

#### **ppResources**

- Resources to unmap for CUDA

### Returns

`cudaSuccess`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

### Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Unmaps the `count` Direct3D resources in `ppResources`.

This function provides the synchronization guarantee that any CUDA kernels issued before `cudaD3D9UnmapResources()` will complete before any Direct3D calls issued after `cudaD3D9UnmapResources()` begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries, then `cudaErrorInvalidResourceHandle` is

returned. If any of `ppResources` are not presently mapped for access by CUDA then `cudaErrorUnknown` is returned.



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

See also:

[cudaGraphicsUnmapResources](#)

## `__host__ cudaError_t cudaD3D9UnregisterResource(IDirect3DResource9 *pResource)`

Unregisters a Direct3D resource for access by CUDA.

### Parameters

**pResource**

- Resource to unregister

### Returns

[cudaSuccess](#), [cudaErrorInvalidResourceHandle](#), [cudaErrorUnknown](#)

### Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Unregisters the Direct3D resource `pResource` so it is not accessible by CUDA unless registered again.

If `pResource` is not registered, then [cudaErrorInvalidResourceHandle](#) is returned.



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

See also:

[cudaGraphicsUnregisterResource](#)



## 5.17. Direct3D 10 Interoperability

This section describes the Direct3D 10 interoperability functions of the CUDA runtime application programming interface. Note that mapping of Direct3D 10 resources is performed with the graphics API agnostic, resource mapping interface described in [Graphics Interoperability](#).

### enum cudaD3D10DeviceList

CUDA devices corresponding to a D3D10 device

#### Values

**cudaD3D10DeviceListAll = 1**

The CUDA devices for all GPUs used by a D3D10 device

**cudaD3D10DeviceListCurrentFrame = 2**

The CUDA devices for the GPUs used by a D3D10 device in its currently rendering frame

**cudaD3D10DeviceListNextFrame = 3**

The CUDA devices for the GPUs to be used by a D3D10 device in the next frame

**\_\_host\_\_ cudaError\_t cudaD3D10GetDevice (int \*device, IDXGIAdapter \*pAdapter)**

Gets the device number for an adapter.

#### Parameters

**device**

- Returns the device corresponding to pAdapter

**pAdapter**

- D3D10 adapter to get device for

#### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorUnknown](#)

#### Description

Returns in \*device the CUDA-compatible device corresponding to the adapter pAdapter obtained from IDXGIFactory::EnumAdapters. This call will succeed only if a device on adapter pAdapter is CUDA-compatible.



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

See also:

[cudaGraphicsD3D10RegisterResource](#), [cuD3D10GetDevice](#)

**\_\_host\_\_ cudaError\_t cudaD3D10GetDevices (unsigned int \*pCudaDeviceCount, int \*pCudaDevices, unsigned int cudaDeviceCount, ID3D10Device \*pD3D10Device, cudaD3D10DeviceList deviceList)**

Gets the CUDA devices corresponding to a Direct3D 10 device.

### Parameters

#### **pCudaDeviceCount**

- Returned number of CUDA devices corresponding to pD3D10Device

#### **pCudaDevices**

- Returned CUDA devices corresponding to pD3D10Device

#### **cudaDeviceCount**

- The size of the output device array pCudaDevices

#### **pD3D10Device**

- Direct3D 10 device to query for CUDA devices

#### **deviceList**

- The set of devices to return. This set may be [cudaD3D10DeviceListAll](#) for all devices, [cudaD3D10DeviceListCurrentFrame](#) for the devices used to render the current frame (in SLI), or [cudaD3D10DeviceListNextFrame](#) for the devices used to render the next frame (in SLI).

### Returns

[cudaSuccess](#), [cudaErrorNoDevice](#), [cudaErrorUnknown](#)

### Description

Returns in \*pCudaDeviceCount the number of CUDA-compatible devices corresponding to the Direct3D 10 device pD3D10Device. Also returns in \*pCudaDevices at most cudaDeviceCount of the the CUDA-compatible devices corresponding to the Direct3D 10 device pD3D10Device.

If any of the GPUs being used to render pDevice are not CUDA capable then the call will return [cudaErrorNoDevice](#).



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

#### See also:

[cudaGraphicsUnregisterResource](#), [cudaGraphicsMapResources](#),  
[cudaGraphicsSubResourceGetMappedArray](#), [cudaGraphicsResourceGetMappedPointer](#),  
[cuD3D10GetDevices](#)

**\_\_host\_\_ cudaError\_t  
 cudaGraphicsD3D10RegisterResource  
 (cudaGraphicsResource \*\*resource, ID3D10Resource  
 \*pD3DResource, unsigned int flags)**

Registers a Direct3D 10 resource for access by CUDA.

#### Parameters

##### **resource**

- Pointer to returned resource handle

##### **pD3DResource**

- Direct3D resource to register

##### **flags**

- Parameters for resource registration

#### Returns

[cudaSuccess](#), [cudaErrorInvalidDevice](#), [cudaErrorInvalidValue](#),  
[cudaErrorInvalidResourceHandle](#), [cudaErrorUnknown](#)

#### Description

Registers the Direct3D 10 resource `pD3DResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through [cudaGraphicsUnregisterResource\(\)](#). Also on success, this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through [cudaGraphicsUnregisterResource\(\)](#).

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of `pD3DResource` must be one of the following.

- `ID3D10Buffer`: may be accessed via a device pointer

- ▶ ID3D10Texture1D: individual subresources of the texture may be accessed via arrays
- ▶ ID3D10Texture2D: individual subresources of the texture may be accessed via arrays
- ▶ ID3D10Texture3D: individual subresources of the texture may be accessed via arrays

The `flags` argument may be used to specify additional parameters at register time. The valid values for this parameter are

- ▶ `cudaGraphicsRegisterFlagsNone`: Specifies no hints about how this resource will be used.
- ▶ `cudaGraphicsRegisterFlagsSurfaceLoadStore`: Specifies that CUDA will bind this resource to a surface reference.
- ▶ `cudaGraphicsRegisterFlagsTextureGather`: Specifies that CUDA will perform texture gather operations on this resource.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- ▶ The primary rendertarget may not be registered with CUDA.
- ▶ Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- ▶ Surfaces of depth or stencil formats cannot be shared.

A complete list of supported DXGI formats is as follows. For compactness the notation `A_{B,C,D}` represents `A_B`, `A_C`, and `A_D`.

- ▶ `DXGI_FORMAT_A8_UNORM`
- ▶ `DXGI_FORMAT_B8G8R8A8_UNORM`
- ▶ `DXGI_FORMAT_B8G8R8X8_UNORM`
- ▶ `DXGI_FORMAT_R16_FLOAT`
- ▶ `DXGI_FORMAT_R16G16B16A16_{FLOAT,SINT,SNORM,UINT,UNORM}`
- ▶ `DXGI_FORMAT_R16G16_{FLOAT,SINT,SNORM,UINT,UNORM}`
- ▶ `DXGI_FORMAT_R16_{SINT,SNORM,UINT,UNORM}`
- ▶ `DXGI_FORMAT_R32_FLOAT`
- ▶ `DXGI_FORMAT_R32G32B32A32_{FLOAT,SINT,UINT}`
- ▶ `DXGI_FORMAT_R32G32_{FLOAT,SINT,UINT}`
- ▶ `DXGI_FORMAT_R32_{SINT,UINT}`
- ▶ `DXGI_FORMAT_R8G8B8A8_{SINT,SNORM,UINT,UNORM,UNORM_SRGB}`
- ▶ `DXGI_FORMAT_R8G8_{SINT,SNORM,UINT,UNORM}`
- ▶ `DXGI_FORMAT_R8_{SINT,SNORM,UINT,UNORM}`

If `pD3DResource` is of incorrect type or is already registered, then `cudaErrorInvalidResourceHandle` is returned. If `pD3DResource` cannot be registered, then `cudaErrorUnknown` is returned.



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

See also:

`cudaGraphicsUnregisterResource`, `cudaGraphicsMapResources`,  
`cudaGraphicsSubResourceGetMappedArray`, `cudaGraphicsResourceGetMappedPointer`,  
`cuGraphicsD3D10RegisterResource`

## 5.18. Direct3D 10 Interoperability [DEPRECATED]

This section describes deprecated Direct3D 10 interoperability functions.

### enum `cudaD3D10MapFlags`

CUDA D3D10 Map Flags

#### Values

**`cudaD3D10MapFlagsNone = 0`**

Default; Assume resource can be read/written

**`cudaD3D10MapFlagsReadOnly = 1`**

CUDA kernels will not write to this resource

**`cudaD3D10MapFlagsWriteDiscard = 2`**

CUDA kernels will only write to and will not read from this resource

### enum `cudaD3D10RegisterFlags`

CUDA D3D10 Register Flags

#### Values

**`cudaD3D10RegisterFlagsNone = 0`**

Default; Resource can be accessed through a `void*`

**`cudaD3D10RegisterFlagsArray = 1`**

Resource can be accessed through a `CUarray*`

## **\_\_host\_\_ cudaError\_t cudaD3D10GetDirect3DDevice (ID3D10Device \*\*ppD3D10Device)**

Gets the Direct3D device against which the current CUDA context was created.

### Parameters

#### **ppD3D10Device**

- Returns the Direct3D device for this thread

### Returns

[cudaSuccess](#), [cudaErrorUnknown](#)

### Description

**Deprecated** This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with a D3D10 device in order to achieve maximum interoperability performance.



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

### See also:

[cudaD3D10SetDirect3DDevice](#)

## **\_\_host\_\_ cudaError\_t cudaD3D10MapResources (int count, ID3D10Resource \*\*ppResources)**

Maps Direct3D Resources for access by CUDA.

### Parameters

#### **count**

- Number of resources to map for CUDA

#### **ppResources**

- Resources to map for CUDA

### Returns

[cudaSuccess](#), [cudaErrorInvalidResourceHandle](#), [cudaErrorUnknown](#)

## Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Maps the `count` Direct3D resources in `ppResources` for access by CUDA.

The resources in `ppResources` may be accessed in CUDA kernels until they are unmapped. Direct3D should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any Direct3D calls issued before `cudaD3D10MapResources()` will complete before any CUDA kernels issued after `cudaD3D10MapResources()` begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries then `cudaErrorInvalidResourceHandle` is returned. If any of `ppResources` are presently mapped for access by CUDA then `cudaErrorUnknown` is returned.



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

See also:

`cudaGraphicsMapResources`

## `__host__ cudaError_t cudaD3D10RegisterResource(ID3D10Resource *pResource, unsigned int flags)`

Registers a Direct3D 10 resource for access by CUDA.

### Parameters

#### `pResource`

- Resource to register

#### `flags`

- Parameters for resource registration

### Returns

`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

## Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Registers the Direct3D resource `pResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through `cudaD3D10UnregisterResource()`. Also on success, this call will increase the internal reference count on `pResource`. This reference count will be decremented when this resource is unregistered through `cudaD3D10UnregisterResource()`.

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of `pResource` must be one of the following:

- ▶ `ID3D10Buffer`: Cannot be used with `flags` set to `cudaD3D10RegisterFlagsArray`.
- ▶ `ID3D10Texture1D`: No restrictions.
- ▶ `ID3D10Texture2D`: No restrictions.
- ▶ `ID3D10Texture3D`: No restrictions.

The `flags` argument specifies the mechanism through which CUDA will access the Direct3D resource. The following values are allowed.

- ▶ `cudaD3D10RegisterFlagsNone`: Specifies that CUDA will access this resource through a `void*`. The pointer, size, and pitch for each subresource of this resource may be queried through `cudaD3D10ResourceGetMappedPointer()`, `cudaD3D10ResourceGetMappedSize()`, and `cudaD3D10ResourceGetMappedPitch()` respectively. This option is valid for all resource types.
- ▶ `cudaD3D10RegisterFlagsArray`: Specifies that CUDA will access this resource through a `CUarray` queried on a sub-resource basis through `cudaD3D10ResourceGetMappedArray()`. This option is only valid for resources of type `ID3D10Texture1D`, `ID3D10Texture2D`, and `ID3D10Texture3D`.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- ▶ The primary rendertarget may not be registered with CUDA.
- ▶ Resources allocated as shared may not be registered with CUDA.
- ▶ Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- ▶ Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized on this context then `cudaErrorInvalidDevice` is returned. If `pResource` is of incorrect type or is already registered then `cudaErrorInvalidResourceHandle` is returned. If `pResource` cannot be registered then `cudaErrorUnknown` is returned.





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

See also:

[cudaGraphicsD3D10RegisterResource](#)

**\_\_host\_\_ cudaError\_t  
cudaD3D10ResourceGetMappedArray (cudaArray  
\*\*ppArray, ID3D10Resource \*pResource, unsigned int  
subResource)**

Gets an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

#### Parameters

**ppArray**

- Returned array corresponding to subresource

**pResource**

- Mapped resource to access

**subResource**

- Subresource of pResource to access

#### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidResourceHandle](#),  
[cudaErrorUnknown](#)

#### Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Returns in \*ppArray an array through which the subresource of the mapped Direct3D resource pResource which corresponds to subResource may be accessed. The value set in ppArray may change every time that pResource is mapped.

If pResource is not registered, then [cudaErrorInvalidResourceHandle](#) is returned.

If pResource was not registered with usage flags [cudaD3D10RegisterFlagsArray](#), then [cudaErrorInvalidResourceHandle](#) is returned. If pResource is not mapped then [cudaErrorUnknown](#) is returned.

For usage requirements of the subResource parameter, see [cudaD3D10ResourceGetMappedPointer\(\)](#).



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

See also:

[cudaGraphicsSubResourceGetMappedArray](#)

**\_\_host\_\_ cudaError\_t  
cudaD3D10ResourceGetMappedPitch (size\_t \*pPitch,  
size\_t \*pPitchSlice, ID3D10Resource \*pResource,  
unsigned int subResource)**

Gets the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.

#### Parameters

##### **pPitch**

- Returned pitch of subresource

##### **pPitchSlice**

- Returned Z-slice pitch of subresource

##### **pResource**

- Mapped resource to access

##### **subResource**

- Subresource of pResource to access

#### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidResourceHandle](#),  
[cudaErrorUnknown](#)

#### Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Returns in *pPitch* and *pPitchSlice* the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource *pResource*, which corresponds to *subResource*. The values set in *pPitch* and *pPitchSlice* may change every time that *pResource* is mapped.

The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position *x*, *y* from the base pointer of the surface is:

$y * \text{pitch} + (\text{bytes per pixel}) * x$

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 `ID3D10Texture1D`, `ID3D10Texture2D`, or `ID3D10Texture3D`, or if `pResource` has not been registered for use with CUDA, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` was not registered with usage flags `cudaD3D10RegisterFlagsNone`, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` is not mapped for access by CUDA then `cudaErrorUnknown` is returned.

For usage requirements of the `subResource` parameter see `cudaD3D10ResourceGetMappedPointer()`.



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

See also:

`cudaGraphicsSubResourceGetMappedArray`

`__host__ cudaError_t`

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

Gets a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

### Parameters

#### `pPointer`

- Returned pointer corresponding to subresource

#### `pResource`

- Mapped resource to access

#### `subResource`

- Subresource of `pResource` to access

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

## Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Returns in `*pPointer` the base pointer of the subresource of the mapped Direct3D resource `pResource` which corresponds to `subResource`. The value set in `pPointer` may change every time that `pResource` is mapped.

If `pResource` is not registered, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` was not registered with usage flags `cudaD3D9RegisterFlagsNone`, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` is not mapped then `cudaErrorUnknown` is returned.

If `pResource` is of type `ID3D10Buffer` then `subResource` must be 0. If `pResource` is of any other type, then the value of `subResource` must come from the subresource calculation in `D3D10CalcSubResource()`.



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

## See also:

`cudaGraphicsResourceGetMappedPointer`

## `__host__ cudaError_t cudaD3D10ResourceGetMappedSize(size_t *pSize, ID3D10Resource *pResource, unsigned int subResource)`

Gets the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.

## Parameters

### `pSize`

- Returned size of subresource

### `pResource`

- Mapped resource to access

### `subResource`

- Subresource of `pResource` to access

## Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

## Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Returns in `*pSize` the size of the subresource of the mapped Direct3D resource `pResource` which corresponds to `subResource`. The value set in `pSize` may change every time that `pResource` is mapped.

If `pResource` has not been registered for use with CUDA then `cudaErrorInvalidHandle` is returned. If `pResource` was not registered with usage flags `cudaD3D10RegisterFlagsNone`, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` is not mapped for access by CUDA then `cudaErrorUnknown` is returned.

For usage requirements of the `subResource` parameter see `cudaD3D10ResourceGetMappedPointer()`.



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

See also:

`cudaGraphicsResourceGetMappedPointer`

**\_\_host\_\_ cudaError\_t  
 cudaD3D10ResourceGetSurfaceDimensions  
 (size\_t \*pWidth, size\_t \*pHeight, size\_t \*pDepth,  
 ID3D10Resource \*pResource, unsigned int subResource)**

Gets the dimensions of a registered Direct3D surface.

## Parameters

### **pWidth**

- Returned width of surface

### **pHeight**

- Returned height of surface

### **pDepth**

- Returned depth of surface

### **pResource**

- Registered resource to access

### **subResource**

- Subresource of `pResource` to access

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidResourceHandle](#),

**Description**

**Deprecated** This function is deprecated as of CUDA 3.0.

Returns in `*pWidth`, `*pHeight`, and `*pDepth` the dimensions of the subresource of the mapped Direct3D resource `pResource` which corresponds to `subResource`.

Since anti-aliased surfaces may have multiple samples per pixel, it is possible that the dimensions of a resource will be an integer factor larger than the dimensions reported by the Direct3D runtime.

The parameters `pWidth`, `pHeight`, and `pDepth` are optional. For 2D surfaces, the value returned in `*pDepth` will be 0.

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

For usage requirements of `subResource` parameters see [cudaD3D10ResourceGetMappedPointer\(\)](#).



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

**See also:**

[cudaGraphicsSubResourceGetMappedArray](#)

## **`__host__ cudaError_t cudaD3D10ResourceSetMapFlags` (`ID3D10Resource *pResource`, unsigned int flags)**

Set usage flags for mapping a Direct3D resource.

**Parameters****`pResource`**

- Registered resource to set flags for

**`flags`**

- Parameters for resource mapping

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidResourceHandle](#),  
[cudaErrorUnknown](#),

## Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Set usage flags for mapping the Direct3D resource `pResource`.

Changes to flags will take effect the next time `pResource` is mapped. The `flags` argument may be any of the following:

- ▶ **`cudaD3D10MapFlagsNone`**: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- ▶ **`cudaD3D10MapFlagsReadOnly`**: Specifies that CUDA kernels which access this resource will not write to this resource.
- ▶ **`cudaD3D10MapFlagsWriteDiscard`**: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If `pResource` has not been registered for use with CUDA then `cudaErrorInvalidHandle` is returned. If `pResource` is presently mapped for access by CUDA then `cudaErrorUnknown` is returned.



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

See also:

`cudaGraphicsResourceSetMapFlags`

## `__host__ cudaError_t cudaD3D10SetDirect3DDevice (ID3D10Device *pD3D10Device, int device)`

Sets the Direct3D 10 device to use for interoperability with a CUDA device.

### Parameters

#### **`pD3D10Device`**

- Direct3D device to use for interoperability

#### **`device`**

- The CUDA device to use. This device must be among the devices returned when querying `cudaD3D10DeviceListAll` from `cudaD3D10GetDevices`, may be set to -1 to automatically select an appropriate CUDA device.

## Returns

[cudaSuccess](#), [cudaErrorInitializationError](#), [cudaErrorInvalidValue](#),  
[cudaErrorSetOnActiveProcess](#)

## Description

**Deprecated** This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with a D3D10 device in order to achieve maximum interoperability performance.



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

## See also:

[cudaD3D10GetDevice](#), [cudaGraphicsD3D10RegisterResource](#), [cudaDeviceReset](#)

# **\_\_host\_\_ cudaError\_t cudaD3D10UnmapResources (int count, ID3D10Resource \*\*ppResources)**

Unmaps Direct3D resources.

## Parameters

### count

- Number of resources to unmap for CUDA

### ppResources

- Resources to unmap for CUDA

## Returns

[cudaSuccess](#), [cudaErrorInvalidResourceHandle](#), [cudaErrorUnknown](#)

## Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Unmaps the `count` Direct3D resource in `ppResources`.

This function provides the synchronization guarantee that any CUDA kernels issued before [cudaD3D10UnmapResources\(\)](#) will complete before any Direct3D calls issued after [cudaD3D10UnmapResources\(\)](#) begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries, then [cudaErrorInvalidResourceHandle](#) is



returned. If any of `ppResources` are not presently mapped for access by CUDA then `cudaErrorUnknown` is returned.



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

See also:

[cudaGraphicsUnmapResources](#)

## `__host__ cudaError_t cudaD3D10UnregisterResource(ID3D10Resource *pResource)`

Unregisters a Direct3D resource.

### Parameters

**pResource**

- Resource to unregister

### Returns

[cudaSuccess](#), [cudaErrorInvalidResourceHandle](#), [cudaErrorUnknown](#)

### Description

**Deprecated** This function is deprecated as of CUDA 3.0.

Unregisters the Direct3D resource `resource` so it is not accessible by CUDA unless registered again.

If `pResource` is not registered, then [cudaErrorInvalidResourceHandle](#) is returned.



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

See also:

[cudaGraphicsUnregisterResource](#)

## 5.19. Direct3D 11 Interoperability

This section describes the Direct3D 11 interoperability functions of the CUDA runtime application programming interface. Note that mapping of Direct3D 11 resources is performed with the graphics API agnostic, resource mapping interface described in [Graphics Interoperability](#).

### enum cudaD3D11DeviceList

CUDA devices corresponding to a D3D11 device

#### Values

**cudaD3D11DeviceListAll = 1**

The CUDA devices for all GPUs used by a D3D11 device

**cudaD3D11DeviceListCurrentFrame = 2**

The CUDA devices for the GPUs used by a D3D11 device in its currently rendering frame

**cudaD3D11DeviceListNextFrame = 3**

The CUDA devices for the GPUs to be used by a D3D11 device in the next frame

**\_\_host\_\_ cudaError\_t cudaD3D11GetDevice (int \*device, IDXGIAdapter \*pAdapter)**

Gets the device number for an adapter.

#### Parameters

**device**

- Returns the device corresponding to pAdapter

**pAdapter**

- D3D11 adapter to get device for

#### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorUnknown](#)

#### Description

Returns in \*device the CUDA-compatible device corresponding to the adapter pAdapter obtained from IDXGIFactory::EnumAdapters. This call will succeed only if a device on adapter pAdapter is CUDA-compatible.



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

#### See also:

[cudaGraphicsUnregisterResource](#), [cudaGraphicsMapResources](#),  
[cudaGraphicsSubResourceGetMappedArray](#), [cudaGraphicsResourceGetMappedPointer](#),  
[cuD3D11GetDevice](#)

**\_\_host\_\_ cudaError\_t cudaD3D11GetDevices (unsigned int \*pCudaDeviceCount, int \*pCudaDevices, unsigned int cudaDeviceCount, ID3D11Device \*pD3D11Device, cudaD3D11DeviceList deviceList)**

Gets the CUDA devices corresponding to a Direct3D 11 device.

#### Parameters

##### **pCudaDeviceCount**

- Returned number of CUDA devices corresponding to pD3D11Device

##### **pCudaDevices**

- Returned CUDA devices corresponding to pD3D11Device

##### **cudaDeviceCount**

- The size of the output device array pCudaDevices

##### **pD3D11Device**

- Direct3D 11 device to query for CUDA devices

##### **deviceList**

- The set of devices to return. This set may be [cudaD3D11DeviceListAll](#) for all devices, [cudaD3D11DeviceListCurrentFrame](#) for the devices used to render the current frame (in SLI), or [cudaD3D11DeviceListNextFrame](#) for the devices used to render the next frame (in SLI).

#### Returns

[cudaSuccess](#), [cudaErrorNoDevice](#), [cudaErrorUnknown](#)

#### Description

Returns in \*pCudaDeviceCount the number of CUDA-compatible devices corresponding to the Direct3D 11 device pD3D11Device. Also returns in \*pCudaDevices at most cudaDeviceCount of the the CUDA-compatible devices corresponding to the Direct3D 11 device pD3D11Device.

If any of the GPUs being used to render `pDevice` are not CUDA capable then the call will return `cudaErrorNoDevice`.



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

See also:

`cudaGraphicsUnregisterResource`, `cudaGraphicsMapResources`,  
`cudaGraphicsSubResourceGetMappedArray`, `cudaGraphicsResourceGetMappedPointer`,  
`cuD3D11GetDevices`

**\_\_host\_\_ cudaError\_t  
 cudaGraphicsD3D11RegisterResource  
 (cudaGraphicsResource \*\*resource, ID3D11Resource  
 \*pD3DResource, unsigned int flags)**

Register a Direct3D 11 resource for access by CUDA.

### Parameters

#### **resource**

- Pointer to returned resource handle

#### **pD3DResource**

- Direct3D resource to register

#### **flags**

- Parameters for resource registration

### Returns

`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorInvalidValue`,  
`cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

### Description

Registers the Direct3D 11 resource `pD3DResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through `cudaGraphicsUnregisterResource()`. Also on success, this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through `cudaGraphicsUnregisterResource()`.

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of `pD3DResource` must be one of the following.

- ▶ `ID3D11Buffer`: may be accessed via a device pointer
- ▶ `ID3D11Texture1D`: individual subresources of the texture may be accessed via arrays
- ▶ `ID3D11Texture2D`: individual subresources of the texture may be accessed via arrays
- ▶ `ID3D11Texture3D`: individual subresources of the texture may be accessed via arrays

The `flags` argument may be used to specify additional parameters at register time. The valid values for this parameter are

- ▶ `cudaGraphicsRegisterFlagsNone`: Specifies no hints about how this resource will be used.
- ▶ `cudaGraphicsRegisterFlagsSurfaceLoadStore`: Specifies that CUDA will bind this resource to a surface reference.
- ▶ `cudaGraphicsRegisterFlagsTextureGather`: Specifies that CUDA will perform texture gather operations on this resource.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- ▶ The primary rendertarget may not be registered with CUDA.
- ▶ Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- ▶ Surfaces of depth or stencil formats cannot be shared.

A complete list of supported DXGI formats is as follows. For compactness the notation `A_{B,C,D}` represents `A_B`, `A_C`, and `A_D`.

- ▶ `DXGI_FORMAT_A8_UNORM`
- ▶ `DXGI_FORMAT_B8G8R8A8_UNORM`
- ▶ `DXGI_FORMAT_B8G8R8X8_UNORM`
- ▶ `DXGI_FORMAT_R16_FLOAT`
- ▶ `DXGI_FORMAT_R16G16B16A16_{FLOAT,SINT,SNORM,UINT,UNORM}`
- ▶ `DXGI_FORMAT_R16G16_{FLOAT,SINT,SNORM,UINT,UNORM}`
- ▶ `DXGI_FORMAT_R16_{SINT,SNORM,UINT,UNORM}`
- ▶ `DXGI_FORMAT_R32_FLOAT`
- ▶ `DXGI_FORMAT_R32G32B32A32_{FLOAT,SINT,UINT}`
- ▶ `DXGI_FORMAT_R32G32_{FLOAT,SINT,UINT}`
- ▶ `DXGI_FORMAT_R32_{SINT,UINT}`
- ▶ `DXGI_FORMAT_R8G8B8A8_{SINT,SNORM,UINT,UNORM,UNORM_SRGB}`
- ▶ `DXGI_FORMAT_R8G8_{SINT,SNORM,UINT,UNORM}`
- ▶ `DXGI_FORMAT_R8_{SINT,SNORM,UINT,UNORM}`

If `pD3DResource` is of incorrect type or is already registered, then `cudaErrorInvalidResourceHandle` is returned. If `pD3DResource` cannot be registered, then `cudaErrorUnknown` is returned.



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

**See also:**

[cudaGraphicsUnregisterResource](#), [cudaGraphicsMapResources](#),  
[cudaGraphicsSubResourceGetMappedArray](#), [cudaGraphicsResourceGetMappedPointer](#),  
[cuGraphicsD3D11RegisterResource](#)

## 5.20. Direct3D 11 Interoperability [DEPRECATED]

This section describes deprecated Direct3D 11 interoperability functions.

### `__host__ cudaError_t cudaD3D11GetDirect3DDevice (ID3D11Device **ppD3D11Device)`

Gets the Direct3D device against which the current CUDA context was created.

**Parameters**

**ppD3D11Device**

- Returns the Direct3D device for this thread

**Returns**

[cudaSuccess](#), [cudaErrorUnknown](#)

**Description**

**Deprecated** This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with a D3D11 device in order to achieve maximum interoperability performance.



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

**See also:**

[cudaD3D11SetDirect3DDevice](#)

## `__host__ cudaError_t cudaD3D11SetDirect3DDevice (ID3D11Device *pD3D11Device, int device)`

Sets the Direct3D 11 device to use for interoperability with a CUDA device.

### Parameters

#### `pD3D11Device`

- Direct3D device to use for interoperability

#### `device`

- The CUDA device to use. This device must be among the devices returned when querying `cudaD3D11DeviceListAll` from `cudaD3D11GetDevices`, may be set to -1 to automatically select an appropriate CUDA device.

### Returns

`cudaSuccess`, `cudaErrorInitializationError`, `cudaErrorInvalidValue`,  
`cudaErrorSetOnActiveProcess`

### Description

**Deprecated** This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with a D3D11 device in order to achieve maximum interoperability performance.



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

### See also:

`cudaD3D11GetDevice`, `cudaGraphicsD3D11RegisterResource`, `cudaDeviceReset`

## 5.21. VDPAU Interoperability

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

### `__host__ cudaError_t cudaGraphicsVDPAURegisterOutputSurface`

## ([cudaGraphicsResource](#) \*\*resource, [VdpOutputSurface](#) vdpSurface, unsigned int flags)

Register a [VdpOutputSurface](#) object.

### Parameters

#### **resource**

- Pointer to the returned object handle

#### **vdpSurface**

- VDPAU object to be registered

#### **flags**

- Map flags

### Returns

[cudaSuccess](#), [cudaErrorInvalidDevice](#), [cudaErrorInvalidValue](#),  
[cudaErrorInvalidResourceHandle](#), [cudaErrorUnknown](#)

### Description

Registers the [VdpOutputSurface](#) specified by `vdpSurface` for access by CUDA. A handle to the registered object is returned as `resource`. The surface's intended usage is specified using `flags`, as follows:

- ▶ [cudaGraphicsMapFlagsNone](#): Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- ▶ [cudaGraphicsMapFlagsReadOnly](#): Specifies that CUDA will not write to this resource.
- ▶ [cudaGraphicsMapFlagsWriteDiscard](#): Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.



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

### See also:

[cudaVDPAUSetVDPAUDevice](#), [cudaGraphicsUnregisterResource](#),  
[cudaGraphicsSubResourceGetMappedArray](#), [cuGraphicsVDPAURegisterOutputSurface](#)

## [\\_\\_host\\_\\_ cudaError\\_t](#) [cudaGraphicsVDPAURegisterVideoSurface](#)



## ([cudaGraphicsResource](#) \*\*resource, [VdpVideoSurface](#) vdpSurface, unsigned int flags)

Register a [VdpVideoSurface](#) object.

### Parameters

#### **resource**

- Pointer to the returned object handle

#### **vdpSurface**

- VDPAU object to be registered

#### **flags**

- Map flags

### Returns

[cudaSuccess](#), [cudaErrorInvalidDevice](#), [cudaErrorInvalidValue](#),  
[cudaErrorInvalidResourceHandle](#), [cudaErrorUnknown](#)

### Description

Registers the [VdpVideoSurface](#) specified by `vdpSurface` for access by CUDA. A handle to the registered object is returned as `resource`. The surface's intended usage is specified using `flags`, as follows:

- ▶ [cudaGraphicsMapFlagsNone](#): Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- ▶ [cudaGraphicsMapFlagsReadOnly](#): Specifies that CUDA will not write to this resource.
- ▶ [cudaGraphicsMapFlagsWriteDiscard](#): Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.



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

### See also:

[cudaVDPAUSetVDPAUDevice](#), [cudaGraphicsUnregisterResource](#),  
[cudaGraphicsSubResourceGetMappedArray](#), [cuGraphicsVDPAURegisterVideoSurface](#)

```
__host__ cudaError_t cudaVDPAUGetDevice (int  
*device, VdpDevice vdpDevice, VdpGetProcAddress  
*vdpGetProcAddress)
```

Gets the CUDA device associated with a VdpDevice.

### Parameters

#### **device**

- Returns the device associated with vdpDevice, or -1 if the device associated with vdpDevice is not a compute device.

#### **vdpDevice**

- A VdpDevice handle

#### **vdpGetProcAddress**

- VDPAU's VdpGetProcAddress function pointer

### Returns

[cudaSuccess](#)

### Description

Returns the CUDA device associated with a VdpDevice, if applicable.



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

### See also:

[cudaVDPAUSetVDPAUDevice](#), [cuVDPAUGetDevice](#)

```
__host__ cudaError_t cudaVDPAUSetVDPAUDevice (int  
device, VdpDevice vdpDevice, VdpGetProcAddress  
*vdpGetProcAddress)
```

Sets a CUDA device to use VDPAU interoperability.

### Parameters

#### **device**

- Device to use for VDPAU interoperability

#### **vdpDevice**

- The VdpDevice to interoperate with

**vdpGetProcAddress**

- VDPAU's VdpGetProcAddress function pointer

**Returns**

[cudaSuccess](#), [cudaErrorInvalidDevice](#), [cudaErrorSetOnActiveProcess](#)

**Description**

Records `vdpDevice` as the `VdpDevice` for VDPAU interoperability with the CUDA device `device` and sets `device` as the current device for the calling host thread.

If `device` has already been initialized then this call will fail with the error [cudaErrorSetOnActiveProcess](#). In this case it is necessary to reset `device` using [cudaDeviceReset\(\)](#) before VDPAU interoperability on `device` may be enabled.



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

**See also:**

[cudaGraphicsVDPAURegisterVideoSurface](#),  
[cudaGraphicsVDPAURegisterOutputSurface](#), [cudaDeviceReset](#)

## 5.22. EGL Interoperability

This section describes the EGL interoperability functions of the CUDA runtime application programming interface. Note that mapping of EGL resources is performed with the graphics API agnostic, resource mapping interface described in [Graphics Interopability](#).

**\_\_host\_\_ cudaError\_t**

**cudaEGLStreamConsumerAcquireFrame**

([cudaEglStreamConnection](#) \*conn,  
[cudaGraphicsResource\\_t](#) \*pCudaResource, [cudaStream\\_t](#) \*pStream, unsigned int timeout)

Acquire an image frame from the EGLStream with CUDA as a consumer.

**Parameters**

**conn**

- Connection on which to acquire

**pCudaResource**

- CUDA resource on which the EGLStream frame will be mapped for use.

**pStream**

- CUDA stream for synchronization and any data migrations implied by [cudaEglResourceLocationFlags](#).

**timeout**

- Desired timeout in usec.

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorUnknown](#), [cudaErrorLaunchTimeout](#)

**Description**

Acquire an image frame from EGLStreamKHR.

[cudaGraphicsResourceGetMappedEglFrame](#) can be called on `pCudaResource` to get `cudaEglFrame`.

**See also:**

[cudaEGLStreamConsumerConnect](#), [cudaEGLStreamConsumerDisconnect](#),  
[cudaEGLStreamConsumerReleaseFrame](#), [cuEGLStreamConsumerAcquireFrame](#)

**`__host__ cudaError_t cudaEGLStreamConsumerConnect  
(cudaEglStreamConnection *conn, EGLStreamKHR  
eglStream)`**

Connect CUDA to EGLStream as a consumer.

**Parameters****conn**

- Pointer to the returned connection handle

**eglStream**

- EGLStreamKHR handle

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorUnknown](#)

**Description**

Connect CUDA as a consumer to EGLStreamKHR specified by `eglStream`.

The EGLStreamKHR is an EGL object that transfers a sequence of image frames from one API to another.

**See also:**

[cudaEGLStreamConsumerDisconnect](#), [cudaEGLStreamConsumerAcquireFrame](#),  
[cudaEGLStreamConsumerReleaseFrame](#), [cuEGLStreamConsumerConnect](#)

## **\_\_host\_\_ cudaError\_t cudaEGLStreamConsumerConnectWithFlags (cudaEglStreamConnection \*conn, EGLStreamKHR eglStream, unsigned int flags)**

Connect CUDA to EGLStream as a consumer with given flags.

**Parameters****conn**

- Pointer to the returned connection handle

**eglStream**

- EGLStreamKHR handle

**flags**

- Flags denote intended location - system or video.

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorUnknown](#)

**Description**

Connect CUDA as a consumer to EGLStreamKHR specified by `stream` with specified `flags` defined by [cudaEglResourceLocationFlags](#).

The flags specify whether the consumer wants to access frames from system memory or video memory. Default is [cudaEglResourceLocationVidmem](#).

**See also:**

[cudaEGLStreamConsumerDisconnect](#), [cudaEGLStreamConsumerAcquireFrame](#),  
[cudaEGLStreamConsumerReleaseFrame](#), [cuEGLStreamConsumerConnectWithFlags](#)

## \_\_host\_\_ cudaError\_t cudaEGLStreamConsumerDisconnect (cudaEglStreamConnection \*conn)

Disconnect CUDA as a consumer to EGLStream .

### Parameters

**conn**

- Conection to disconnect.

### Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown

### Description

Disconnect CUDA as a consumer to EGLStreamKHR.

### See also:

cudaEGLStreamConsumerConnect, cudaEGLStreamConsumerAcquireFrame,  
cudaEGLStreamConsumerReleaseFrame, cuEGLStreamConsumerDisconnect

## \_\_host\_\_ cudaError\_t cudaEGLStreamConsumerReleaseFrame (cudaEglStreamConnection \*conn, cudaGraphicsResource\_t pCudaResource, cudaStream\_t \*pStream)

Releases the last frame acquired from the EGLStream.

### Parameters

**conn**

- Connection on which to release

**pCudaResource**

- CUDA resource whose corresponding frame is to be released

**pStream**

- CUDA stream on which release will be done.

### Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown

**Description**

Release the acquired image frame specified by `pCudaResource` to `EGLStreamKHR`.

**See also:**

`cudaEGLStreamConsumerConnect`, `cudaEGLStreamConsumerDisconnect`,  
`cudaEGLStreamConsumerAcquireFrame`, `cuEGLStreamConsumerReleaseFrame`

**`__host__ cudaError_t cudaEGLStreamProducerConnect`  
`(cudaEglStreamConnection *conn, EGLStreamKHR`  
`eglStream, EGLint width, EGLint height)`**

Connect CUDA to EGLStream as a producer.

**Parameters****`conn`**

- Pointer to the returned connection handle

**`eglStream`**

- EGLStreamKHR handle

**`width`**

- width of the image to be submitted to the stream

**`height`**

- height of the image to be submitted to the stream

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorUnknown`

**Description**

Connect CUDA as a producer to `EGLStreamKHR` specified by `stream`.

The `EGLStreamKHR` is an EGL object that transfers a sequence of image frames from one API to another.

**See also:**

`cudaEGLStreamProducerDisconnect`, `cudaEGLStreamProducerPresentFrame`,  
`cudaEGLStreamProducerReturnFrame`, `cuEGLStreamProducerConnect`

## `__host__ cudaError_t cudaEGLStreamProducerDisconnect (cudaEglStreamConnection *conn)`

Disconnect CUDA as a producer to EGLStream .

### Parameters

**conn**

- Connection to disconnect.

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorUnknown`

### Description

Disconnect CUDA as a producer to EGLStreamKHR.

### See also:

`cudaEGLStreamProducerConnect`, `cudaEGLStreamProducerPresentFrame`,  
`cudaEGLStreamProducerReturnFrame`, `cuEGLStreamProducerDisconnect`

## `__host__ cudaError_t cudaEGLStreamProducerPresentFrame (cudaEglStreamConnection *conn, cudaEglFrame eglframe, cudaStream_t *pStream)`

Present a CUDA eglFrame to the EGLStream with CUDA as a producer.

### Parameters

**conn**

- Connection on which to present the CUDA array

**eglframe**

- CUDA Eglstream Proucer Frame handle to be sent to the consumer over EglStream.

**pStream**

- CUDA stream on which to present the frame.

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorUnknown`



## Description

The `cudaEglFrame` is defined as:

```
typedef struct cudaEglFrame_st {
    union {
        cudaArray_t          pArray[CUDA_EGL_MAX_PLANES];
        struct cudaPitchedPtr pPitch[CUDA_EGL_MAX_PLANES];
    } frame;
    cudaEglPlaneDesc planeDesc[CUDA_EGL_MAX_PLANES];
    unsigned int planeCount;
    cudaEglFrameType frameType;
    cudaEglColorFormat eglColorFormat;
} cudaEglFrame;
```

For `cudaEglFrame` of type `cudaEglFrameTypePitch`, the application may present sub-region of a memory allocation. In that case, `cudaPitchedPtr::ptr` will specify the start address of the sub-region in the allocation and `cudaEglPlaneDesc` will specify the dimensions of the sub-region.

See also:

`cudaEGLStreamProducerConnect`, `cudaEGLStreamProducerDisconnect`,  
`cudaEGLStreamProducerReturnFrame`, `cuEGLStreamProducerPresentFrame`

## \_\_host\_\_ cudaError\_t

### cudaEGLStreamProducerReturnFrame

(`cudaEglStreamConnection *conn`, `cudaEglFrame *eglframe`, `cudaStream_t *pStream`)

Return the CUDA `eglFrame` to the EGLStream last released by the consumer.

## Parameters

### conn

- Connection on which to present the CUDA array

### eglframe

- CUDA Eglstream Proucer Frame handle returned from the consumer over EglStream.

### pStream

- CUDA stream on which to return the frame.

## Returns

`cudaSuccess`, `cudaErrorLaunchTimeout`, `cudaErrorInvalidValue`, `cudaErrorUnknown`

## Description

This API can potentially return `cudaErrorLaunchTimeout` if the consumer has not returned a frame to EGL stream. If timeout is returned the application can retry.

**See also:**

[cudaEGLStreamProducerConnect](#), [cudaEGLStreamProducerDisconnect](#),  
[cudaEGLStreamProducerPresentFrame](#), [cuEGLStreamProducerReturnFrame](#)

**\_\_host\_\_ cudaError\_t cudaEventCreateFromEGLSync  
 (cudaEvent\_t \*phEvent, EGLSyncKHR eglSync, unsigned  
 int flags)**

Creates an event from EGLSync object.

**Parameters****phEvent**

- Returns newly created event

**eglSync**

- Opaque handle to EGLSync object

**flags**

- Event creation flags

**Returns**

[cudaSuccess](#), [cudaErrorInitializationError](#), [cudaErrorInvalidValue](#),  
[cudaErrorLaunchFailure](#), [cudaErrorMemoryAllocation](#)

**Description**

Creates an event \*phEvent from an EGLSyncKHR eglSync with the flages specified via flags. Valid flags include:

- ▶ [cudaEventDefault](#): Default event creation flag.
- ▶ [cudaEventBlockingSync](#): Specifies that the created event should use blocking synchronization. A CPU thread that uses [cudaEventSynchronize\(\)](#) to wait on an event created with this flag will block until the event has actually been completed.

[cudaEventRecord](#) and TimingData are not supported for events created from EGLSync.

The EGLSyncKHR is an opaque handle to an EGL sync object. typedef void\* EGLSyncKHR

**See also:**

[cudaEventQuery](#), [cudaEventSynchronize](#), [cudaEventDestroy](#)

## `__host__ cudaError_t cudaGraphicsEGLRegisterImage(cudaGraphicsResource **pCudaResource, EGLImageKHR image, unsigned int flags)`

Registers an EGL image.

### Parameters

#### **pCudaResource**

- Pointer to the returned object handle

#### **image**

- An EGLImageKHR image which can be used to create target resource.

#### **flags**

- Map flags

### Returns

`cudaSuccess`, `cudaErrorInvalidResourceHandle`, `cudaErrorInvalidValue`, `cudaErrorUnknown`

### Description

Registers the EGLImageKHR specified by `image` for access by CUDA. A handle to the registered object is returned as `pCudaResource`. Additional Mapping/Unmapping is not required for the registered resource and `cudaGraphicsResourceGetMappedEglFrame` can be directly called on the `pCudaResource`.

The application will be responsible for synchronizing access to shared objects. The application must ensure that any pending operation which access the objects have completed before passing control to CUDA. This may be accomplished by issuing and waiting for `glFinish` command on all GLcontexts (for OpenGL and likewise for other APIs). The application will be also responsible for ensuring that any pending operation on the registered CUDA resource has completed prior to executing subsequent commands in other APIs accessing the same memory objects. This can be accomplished by calling `cuCtxSynchronize` or `cuEventSynchronize` (preferably).

The surface's intended usage is specified using `flags`, as follows:

- ▶ `cudaGraphicsRegisterFlagsNone`: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- ▶ `cudaGraphicsRegisterFlagsReadOnly`: Specifies that CUDA will not write to this resource.

- ▶ `cudaGraphicsRegisterFlagsWriteDiscard`: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

The `EGLImageKHR` is an object which can be used to create `EGLImage` target resource. It is defined as a void pointer. `typedef void* EGLImageKHR`

**See also:**

`cudaGraphicsUnregisterResource`, `cudaGraphicsResourceGetMappedEglFrame`, `cuGraphicsEGLRegisterImage`

**`__host__ cudaError_t  
cudaGraphicsResourceGetMappedEglFrame  
(cudaEglFrame *eglFrame, cudaGraphicsResource_t  
resource, unsigned int index, unsigned int mipLevel)`**

Get an `eglFrame` through which to access a registered EGL graphics resource.

#### Parameters

##### **`eglFrame`**

- Returned `eglFrame`.

##### **`resource`**

- Registered resource to access.

##### **`index`**

- Index for cubemap surfaces.

##### **`mipLevel`**

- Mipmap level for the subresource to access.

#### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorUnknown`

#### Description

Returns in `*eglFrame` an `eglFrame` pointer through which the registered graphics resource `resource` may be accessed. This API can only be called for EGL graphics resources.

The `cudaEglFrame` is defined as

```
typedef struct cudaEglFrame_st {
    union {
        cudaArray_t          pArray[CUDA_EGL_MAX_PLANES];
        struct cudaPitchedPtr pPitch[CUDA_EGL_MAX_PLANES];
    } frame;
    cudaEglPlaneDesc planeDesc[CUDA_EGL_MAX_PLANES];
    unsigned int planeCount;
    cudaEglFrameType frameType;
    cudaEglColorFormat eglColorFormat;
} cudaEglFrame;
```

See also:

`cudaGraphicsSubResourceGetMappedArray`, `cudaGraphicsResourceGetMappedPointer`,  
`cuGraphicsResourceGetMappedEglFrame`

## 5.23. Graphics Interoperability

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

**`__host__ cudaError_t cudaGraphicsMapResources`  
`(int count, cudaGraphicsResource_t *resources,`  
`cudaStream_t stream)`**

Map graphics resources for access by CUDA.

### Parameters

#### **count**

- Number of resources to map

#### **resources**

- Resources to map for CUDA

#### **stream**

- Stream for synchronization

### Returns

`cudaSuccess`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

### Description

Maps the `count` graphics resources in `resources` for access by CUDA.

The resources in `resources` may be accessed by CUDA until they are unmapped. The graphics API from which `resources` were registered should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any graphics calls issued before `cudaGraphicsMapResources()` will complete before any subsequent CUDA work issued in `stream` begins.

If `resources` contains any duplicate entries then `cudaErrorInvalidResourceHandle` is returned. If any of `resources` are presently mapped for access by CUDA then `cudaErrorUnknown` is returned.



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

See also:

`cudaGraphicsResourceGetMappedPointer`, `cudaGraphicsSubResourceGetMappedArray`, `cudaGraphicsUnmapResources`, `cuGraphicsMapResources`

## `__host__ cudaError_t` `cudaGraphicsResourceGetMappedMipmappedArray` `(cudaMipmappedArray_t *mipmappedArray,` `cudaGraphicsResource_t resource)`

Get a mipmapped array through which to access a mapped graphics resource.

### Parameters

#### **mipmappedArray**

- Returned mipmapped array through which `resource` may be accessed

#### **resource**

- Mapped resource to access

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

## Description

Returns in `*mipmappedArray` a mipmapped array through which the mapped graphics resource `resource` may be accessed. The value set in `mipmappedArray` may change every time that `resource` is mapped.

If `resource` is not a texture then it cannot be accessed via an array and `cudaErrorUnknown` is returned. If `resource` is not mapped then `cudaErrorUnknown` is returned.



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

## See also:

`cudaGraphicsResourceGetMappedPointer`,  
`cuGraphicsResourceGetMappedMipmappedArray`

## \_\_host\_\_ cudaError\_t

**`cudaGraphicsResourceGetMappedPointer (void **devPtr, size_t *size, cudaGraphicsResource_t resource)`**

Get an device pointer through which to access a mapped graphics resource.

## Parameters

### **`devPtr`**

- Returned pointer through which `resource` may be accessed

### **`size`**

- Returned size of the buffer accessible starting at `*devPtr`

### **`resource`**

- Mapped resource to access

## Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`,  
`cudaErrorUnknown`

## Description

Returns in `*devPtr` a pointer through which the mapped graphics resource `resource` may be accessed. Returns in `*size` the size of the memory in bytes which may be accessed from that pointer. The value set in `devPtr` may change every time that `resource` is mapped.

If `resource` is not a buffer then it cannot be accessed via a pointer and `cudaErrorUnknown` is returned. If `resource` is not mapped then `cudaErrorUnknown` is returned. \*



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

## See also:

`cudaGraphicsMapResources`, `cudaGraphicsSubResourceGetMappedArray`,  
`cuGraphicsResourceGetMappedPointer`

## `__host__ cudaError_t cudaGraphicsResourceSetMapFlags` (`cudaGraphicsResource_t resource`, unsigned int flags)

Set usage flags for mapping a graphics resource.

## Parameters

### **resource**

- Registered resource to set flags for

### **flags**

- Parameters for resource mapping

## Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`,  
`cudaErrorUnknown`,

## Description

Set flags for mapping the graphics resource `resource`.



Changes to `flags` will take effect the next time `resource` is mapped. The `flags` argument may be any of the following:

- ▶ `cudaGraphicsMapFlagsNone`: Specifies no hints about how `resource` will be used. It is therefore assumed that CUDA may read from or write to `resource`.
- ▶ `cudaGraphicsMapFlagsReadOnly`: Specifies that CUDA will not write to `resource`.
- ▶ `cudaGraphicsMapFlagsWriteDiscard`: Specifies CUDA will not read from `resource` and will write over the entire contents of `resource`, so none of the data previously stored in `resource` will be preserved.

If `resource` is presently mapped for access by CUDA then `cudaErrorUnknown` is returned. If `flags` is not one of the above values then `cudaErrorInvalidValue` is returned.



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

See also:

`cudaGraphicsMapResources`, `cuGraphicsResourceSetMapFlags`

`__host__ cudaError_t`

`cudaGraphicsSubResourceGetMappedArray (cudaArray_t *array, cudaGraphicsResource_t resource, unsigned int arrayIndex, unsigned int mipLevel)`

Get an array through which to access a subresource of a mapped graphics resource.

#### Parameters

##### **array**

- Returned array through which a subresource of `resource` may be accessed

##### **resource**

- Mapped resource to access

##### **arrayIndex**

- Array index for array textures or cubemap face index as defined by `cudaGraphicsCubeFace` for cubemap textures for the subresource to access

**mipLevel**

- Mipmap level for the subresource to access

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidResourceHandle](#),  
[cudaErrorUnknown](#)

**Description**

Returns in `*array` an array through which the subresource of the mapped graphics resource `resource` which corresponds to array index `arrayIndex` and mipmap level `mipLevel` may be accessed. The value set in `array` may change every time that `resource` is mapped.

If `resource` is not a texture then it cannot be accessed via an array and [cudaErrorUnknown](#) is returned. If `arrayIndex` is not a valid array index for `resource` then [cudaErrorInvalidValue](#) is returned. If `mipLevel` is not a valid mipmap level for `resource` then [cudaErrorInvalidValue](#) is returned. If `resource` is not mapped then [cudaErrorUnknown](#) is returned.



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

**See also:**

[cudaGraphicsResourceGetMappedPointer](#), [cuGraphicsSubResourceGetMappedArray](#)

**`__host__ cudaError_t cudaGraphicsUnmapResources`**  
**`(int count, cudaGraphicsResource_t *resources,`**  
**`cudaStream_t stream)`**

Unmap graphics resources.

**Parameters****count**

- Number of resources to unmap

**resources**

- Resources to unmap

**stream**

- Stream for synchronization

**Returns**

[cudaSuccess](#), [cudaErrorInvalidResourceHandle](#), [cudaErrorUnknown](#)

**Description**

Unmaps the `count` graphics resources in `resources`.

Once unmapped, the resources in `resources` may not be accessed by CUDA until they are mapped again.

This function provides the synchronization guarantee that any CUDA work issued in `stream` before [cudaGraphicsUnmapResources\(\)](#) will complete before any subsequently issued graphics work begins.

If `resources` contains any duplicate entries then [cudaErrorInvalidResourceHandle](#) is returned. If any of `resources` are not presently mapped for access by CUDA then [cudaErrorUnknown](#) is returned.



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

**See also:**

[cudaGraphicsMapResources](#), [cuGraphicsUnmapResources](#)

## `__host__ cudaError_t cudaGraphicsUnregisterResource(cudaGraphicsResource_t resource)`

Unregisters a graphics resource for access by CUDA.

### Parameters

#### **resource**

- Resource to unregister

### Returns

`cudaSuccess`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

### Description

Unregisters the graphics resource `resource` so it is not accessible by CUDA unless registered again.

If `resource` is invalid then `cudaErrorInvalidResourceHandle` is returned.



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

### See also:

`cudaGraphicsD3D9RegisterResource`, `cudaGraphicsD3D10RegisterResource`,  
`cudaGraphicsD3D11RegisterResource`, `cudaGraphicsGLRegisterBuffer`,  
`cudaGraphicsGLRegisterImage`, `cuGraphicsUnregisterResource`

## 5.24. Texture Reference Management [DEPRECATED]

This section describes the low level texture reference management functions of the CUDA runtime application programming interface.

Some functions have overloaded C++ API template versions documented separately in the [C++ API Routines](#) module.

```
__host__ cudaError_t cudaBindTexture (size_t *offset,  
const textureReference *texref, const void *devPtr,  
const cudaChannelFormatDesc *desc, size_t size)
```

Binds a memory area to a texture.

### Parameters

#### **offset**

- Offset in bytes

#### **texref**

- Texture to bind

#### **devPtr**

- Memory area on device

#### **desc**

- Channel format

#### **size**

- Size of the memory area pointed to by devPtr

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidTexture](#)

### Description

#### Deprecated

Binds `size` bytes of the memory area pointed to by `devPtr` to the texture reference `texref`. `desc` describes how the memory is interpreted when fetching values from the texture. Any memory previously bound to `texref` is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, [cudaBindTexture\(\)](#) returns in `*offset` a byte offset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the `tex1Dfetch()` function. If the device memory pointer was returned from [cudaMalloc\(\)](#), the offset is guaranteed to be 0 and NULL may be passed as the `offset` parameter.

The total number of elements (or texels) in the linear address range cannot exceed [cudaDeviceProp::maxTexture1DLinear\[0\]](#). The number of elements is computed as  $(size / elementSize)$ , where `elementSize` is determined from `desc`.



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

### See also:

`cudaCreateChannelDesc` ( C API), `cudaGetChannelDesc`, `cudaGetTextureReference`, `cudaBindTexture` ( C++ API), `cudaBindTexture2D` ( C API), `cudaBindTextureToArray` ( C API), `cudaUnbindTexture` ( C API), `cudaGetTextureAlignmentOffset` ( C API), `cuTexRefSetAddress`, `cuTexRefSetAddressMode`, `cuTexRefSetFormat`, `cuTexRefSetFlags`, `cuTexRefSetBorderColor`

**\_\_host\_\_ cudaError\_t cudaBindTexture2D (size\_t \*offset, const textureReference \*texref, const void \*devPtr, const cudaChannelFormatDesc \*desc, size\_t width, size\_t height, size\_t pitch)**

Binds a 2D memory area to a texture.

### Parameters

#### **offset**

- Offset in bytes

#### **texref**

- Texture reference to bind

#### **devPtr**

- 2D memory area on device

#### **desc**

- Channel format

#### **width**

- Width in texel units

#### **height**

- Height in texel units

#### **pitch**

- Pitch in bytes

## Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidTexture`

## Description

### Deprecated

Binds the 2D memory area pointed to by `devPtr` to the texture reference `texref`. The size of the area is constrained by `width` in texel units, `height` in texel units, and `pitch` in byte units. `desc` describes how the memory is interpreted when fetching values from the texture. Any memory previously bound to `texref` is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, `cudaBindTexture2D()` returns in `*offset` a byte offset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the `tex2D()` function. If the device memory pointer was returned from `cudaMalloc()`, the offset is guaranteed to be 0 and NULL may be passed as the `offset` parameter.

`width` and `height`, which are specified in elements (or texels), cannot exceed `cudaDeviceProp::maxTexture2DLinear[0]` and `cudaDeviceProp::maxTexture2DLinear[1]` respectively. `pitch`, which is specified in bytes, cannot exceed `cudaDeviceProp::maxTexture2DLinear[2]`.

The driver returns `cudaErrorInvalidValue` if `pitch` is not a multiple of `cudaDeviceProp::texturePitchAlignment`.



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

## See also:

`cudaCreateChannelDesc` ( C API), `cudaGetChannelDesc`, `cudaGetTextureReference`, `cudaBindTexture` ( C API), `cudaBindTexture2D` ( C++ API), `cudaBindTexture2D` ( C++ API, inherited channel descriptor), `cudaBindTextureToArray` ( C API), `cudaBindTextureToArray` ( C API), `cudaGetTextureAlignmentOffset` ( C API), `cuTexRefSetAddress2D`, `cuTexRefSetFormat`, `cuTexRefSetFlags`, `cuTexRefSetAddressMode`, `cuTexRefSetBorderColor`

**\_\_host\_\_ cudaError\_t cudaBindTextureToArray (const textureReference \*texref, cudaArray\_const\_t array, const cudaChannelFormatDesc \*desc)**

Binds an array to a texture.

### Parameters

#### **texref**

- Texture to bind

#### **array**

- Memory array on device

#### **desc**

- Channel format

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidTexture](#)

### Description

#### Deprecated

Binds the CUDA array `array` to the texture reference `texref`. `desc` describes how the memory is interpreted when fetching values from the texture. Any CUDA array previously bound to `texref` is unbound.



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

### See also:

[cudaCreateChannelDesc](#) ( C API), [cudaGetChannelDesc](#), [cudaGetTextureReference](#), [cudaBindTexture](#) ( C API), [cudaBindTexture2D](#) ( C API), [cudaBindTextureToArray](#) ( C++ API), [cudaUnbindTexture](#) ( C API), [cudaGetTextureAlignmentOffset](#) ( C API), [cuTexRefSetArray](#), [cuTexRefSetFormat](#), [cuTexRefSetFlags](#), [cuTexRefSetAddressMode](#), [cuTexRefSetFilterMode](#), [cuTexRefSetBorderColor](#), [cuTexRefSetMaxAnisotropy](#)



`__host__ cudaError_t`

`cudaBindTextureToMipmappedArray (const textureReference *texref, cudaMipmappedArray_const_t mipmappedArray, const cudaChannelFormatDesc *desc)`

Binds a mipmapped array to a texture.

### Parameters

**texref**

- Texture to bind

**mipmappedArray**

- Memory mipmapped array on device

**desc**

- Channel format

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidTexture`

### Description

Deprecated

Binds the CUDA mipmapped array `mipmappedArray` to the texture reference `texref`. `desc` describes how the memory is interpreted when fetching values from the texture. Any CUDA mipmapped array previously bound to `texref` is unbound.



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

### See also:

`cudaCreateChannelDesc` ( C API), `cudaGetChannelDesc`, `cudaGetTextureReference`, `cudaBindTexture` ( C API), `cudaBindTexture2D` ( C API), `cudaBindTextureToArray` ( C++ API), `cudaUnbindTexture` ( C API), `cudaGetTextureAlignmentOffset` ( C API), `cuTexRefSetMipmappedArray`, `cuTexRefSetMipmapFilterMode`

`cuTexRefSetMipmapLevelClamp`, `cuTexRefSetMipmapLevelBias`, `cuTexRefSetFormat`,  
`cuTexRefSetFlags`, `cuTexRefSetAddressMode`, `cuTexRefSetBorderColor`,  
`cuTexRefSetMaxAnisotropy`

## `__host__ cudaError_t cudaGetTextureAlignmentOffset` `(size_t *offset, const textureReference *texref)`

Get the alignment offset of a texture.

### Parameters

#### `offset`

- Offset of texture reference in bytes

#### `texref`

- Texture to get offset of

### Returns

`cudaSuccess`, `cudaErrorInvalidTexture`, `cudaErrorInvalidTextureBinding`

### Description

#### Deprecated

Returns in `*offset` the offset that was returned when texture reference `texref` was bound.



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

### See also:

`cudaCreateChannelDesc` ( C API), `cudaGetChannelDesc`, `cudaGetTextureReference`,  
`cudaBindTexture` ( C API), `cudaBindTexture2D` ( C API), `cudaBindTextureToArray` ( C  
API), `cudaUnbindTexture` ( C API), `cudaGetTextureAlignmentOffset` ( C++ API)

## `__host__ cudaError_t cudaGetTextureReference (const textureReference **texref, const void *symbol)`

Get the texture reference associated with a symbol.

### Parameters

#### **texref**

- Texture reference associated with symbol

#### **symbol**

- Texture to get reference for

### Returns

`cudaSuccess`, `cudaErrorInvalidTexture`

### Description

#### Deprecated

Returns in `*texref` the structure associated to the texture reference defined by symbol `symbol`.



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

### See also:

`cudaCreateChannelDesc` ( C API), `cudaGetChannelDesc`,  
`cudaGetTextureAlignmentOffset` ( C API), `cudaBindTexture` ( C API),  
`cudaBindTexture2D` ( C API), `cudaBindTextureToArray` ( C API), `cudaUnbindTexture`  
 ( C API), `cuModuleGetTexRef`

## `__host__ cudaError_t cudaUnbindTexture (const textureReference *texref)`

Unbinds a texture.

### Parameters

#### `texref`

- Texture to unbind

### Returns

`cudaSuccess`, `cudaErrorInvalidTexture`

### Description

#### Deprecated

Unbinds the texture bound to `texref`. If `texref` is not currently bound, no operation is performed.



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

### See also:

`cudaCreateChannelDesc` ( C API), `cudaGetChannelDesc`, `cudaGetTextureReference`, `cudaBindTexture` ( C API), `cudaBindTexture2D` ( C API), `cudaBindTextureToArray` ( C API), `cudaUnbindTexture` ( C++ API), `cudaGetTextureAlignmentOffset` ( C API)

## 5.25. Surface Reference Management [DEPRECATED]

This section describes the low level surface reference management functions of the CUDA runtime application programming interface.

Some functions have overloaded C++ API template versions documented separately in the [C++ API Routines](#) module.

```
__host__ cudaError_t cudaBindSurfaceToArray (const
surfaceReference *surfref, cudaArray_const_t array,
const cudaChannelFormatDesc *desc)
```

Binds an array to a surface.

### Parameters

#### **surfref**

- Surface to bind

#### **array**

- Memory array on device

#### **desc**

- Channel format

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidSurface](#)

### Description

#### Deprecated

Binds the CUDA array `array` to the surface reference `surfref`. `desc` describes how the memory is interpreted when fetching values from the surface. Any CUDA array previously bound to `surfref` is unbound.



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

### See also:

[cudaBindSurfaceToArray \( C++ API\)](#), [cudaBindSurfaceToArray \( C++ API, inherited channel descriptor\)](#), [cudaGetSurfaceReference](#), [cuSurfRefSetArray](#)

## `__host__ cudaError_t cudaGetSurfaceReference (const surfaceReference **surfref, const void *symbol)`

Get the surface reference associated with a symbol.

### Parameters

#### **surfref**

- Surface reference associated with symbol

#### **symbol**

- Surface to get reference for

### Returns

`cudaSuccess`, `cudaErrorInvalidSurface`

### Description

#### Deprecated

Returns in `*surfref` the structure associated to the surface reference defined by symbol `symbol`.



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

### See also:

`cudaBindSurfaceToArray` ( C API), `cuModuleGetSurfRef`

## 5.26. Texture Object Management

This section describes the low level texture object management functions of the CUDA runtime application programming interface. The texture object API is only supported on devices of compute capability 3.0 or higher.

## `__host__ cudaCreateChannelDesc (int x, int y, int z, int w, cudaChannelFormatKind f)`

Returns a channel descriptor using the specified format.

### Parameters

- x**  
- X component
- y**  
- Y component
- z**  
- Z component
- w**  
- W component
- f**  
- Channel format

### Returns

Channel descriptor with format `f`

### Description

Returns a channel descriptor with format `f` and number of bits of each component `x`, `y`, `z`, and `w`. The `cudaChannelFormatDesc` is defined as:

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

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

### See also:

`cudaCreateChannelDesc` ( C++ API), `cudaGetChannelDesc`, `cudaCreateTextureObject`, `cudaCreateSurfaceObject`

## `__host__ cudaError_t cudaCreateTextureObject (cudaTextureObject_t *pTexObject, const cudaResourceDesc *pResDesc, const cudaTextureDesc`

**\*pTexDesc, const cudaResourceViewDesc  
\*pResViewDesc)**

Creates a texture object.

### Parameters

**pTexObject**

- Texture object to create

**pResDesc**

- Resource descriptor

**pTexDesc**

- Texture descriptor

**pResViewDesc**

- Resource view descriptor

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

### Description

Creates a texture object and returns it in `pTexObject`. `pResDesc` describes the data to texture from. `pTexDesc` describes how the data should be sampled. `pResViewDesc` is an optional argument that specifies an alternate format for the data described by `pResDesc`, and also describes the subresource region to restrict access to when texturing. `pResViewDesc` can only be specified if the type of resource is a CUDA array or a CUDA mipmapped array.

Texture objects are only supported on devices of compute capability 3.0 or higher. Additionally, a texture object is an opaque value, and, as such, should only be accessed through CUDA API calls.



The `cudaResourceDesc` structure is defined as:

```

struct cudaResourceDesc {
    enum cudaResourceType
    resType;

    union {
        struct {
            cudaArray_t
            array;
        } array;
        struct {
            cudaMipmappedArray_t
            mipmap;
        } mipmap;
        struct {
            void *devPtr;
            struct cudaChannelFormatDesc
            desc;
            size_t sizeInBytes;
        } linear;
        struct {
            void *devPtr;
            struct cudaChannelFormatDesc
            desc;
            size_t width;
            size_t height;
            size_t pitchInBytes;
        } pitch2D;
    } res;
};

```

where:

- `cudaResourceDesc::resType` specifies the type of resource to texture from. `CUresourceType` is defined as:

```

enum cudaResourceType {
    cudaResourceTypeArray          = 0x00,
    cudaResourceTypeMipmappedArray = 0x01,
    cudaResourceTypeLinear         = 0x02,
    cudaResourceTypePitch2D       = 0x03
};

```

If `cudaResourceDesc::resType` is set to `cudaResourceTypeArray`, `cudaResourceDesc::res::array::array` must be set to a valid CUDA array handle.

If `cudaResourceDesc::resType` is set to `cudaResourceTypeMipmappedArray`, `cudaResourceDesc::res::mipmap::mipmap` must be set to a valid CUDA mipmapped array handle and `cudaTextureDesc::normalizedCoords` must be set to true.

If `cudaResourceDesc::resType` is set to `cudaResourceTypeLinear`, `cudaResourceDesc::res::linear::devPtr` must be set to a valid device pointer, that is aligned to `cudaDeviceProp::textureAlignment`. `cudaResourceDesc::res::linear::desc` describes the format and the number of components per array element. `cudaResourceDesc::res::linear::sizeInBytes` specifies the size of the array in bytes. The total number of elements in the linear address range cannot exceed `cudaDeviceProp::maxTexture1DLinear`. The number of elements is computed as  $(\text{sizeInBytes} / \text{sizeof}(\text{desc}))$ .

If `cudaResourceDesc::resType` is set to `cudaResourceTypePitch2D`, `cudaResourceDesc::res::pitch2D::devPtr` must be set to a valid device pointer, that is aligned to `cudaDeviceProp::textureAlignment`. `cudaResourceDesc::res::pitch2D::desc` describes the format and the number of components per array element. `cudaResourceDesc::res::pitch2D::width` and `cudaResourceDesc::res::pitch2D::height` specify the width and height of the array in elements, and cannot exceed `cudaDeviceProp::maxTexture2DLinear[0]` and `cudaDeviceProp::maxTexture2DLinear[1]` respectively. `cudaResourceDesc::res::pitch2D::pitchInBytes` specifies the pitch between two rows in bytes and has to be aligned to `cudaDeviceProp::texturePitchAlignment`. Pitch cannot exceed `cudaDeviceProp::maxTexture2DLinear[2]`.

The `cudaTextureDesc` struct is defined as

```

struct cudaTextureDesc {
    enum cudaTextureAddressMode
    addressMode[3];
    enum cudaTextureFilterMode
    filterMode;
    enum cudaTextureReadMode
    readMode;
    int sRGB;
    float borderColor[4];
    int normalizedCoords;
    unsigned int maxAnisotropy;
    enum cudaTextureFilterMode
    mipmapFilterMode;
    float mipmapLevelBias;
    float minMipmapLevelClamp;
    float maxMipmapLevelClamp;
};

```

where

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

```

enum cudaTextureAddressMode {
    cudaAddressModeWrap = 0,
    cudaAddressModeClamp = 1,
    cudaAddressModeMirror = 2,
    cudaAddressModeBorder = 3
};

```

This is ignored if `cudaResourceDesc::resType` is `cudaResourceTypeLinear`. Also, if `cudaTextureDesc::normalizedCoords` is set to zero, `cudaAddressModeWrap` and `cudaAddressModeMirror` won't be supported and will be switched to `cudaAddressModeClamp`.

- `cudaTextureDesc::filterMode` specifies the filtering mode to be used when fetching from the texture. `cudaTextureFilterMode` is defined as:

```

enum cudaTextureFilterMode {
    cudaFilterModePoint = 0,
    cudaFilterModeLinear = 1
};

```

This is ignored if `cudaResourceDesc::resType` is `cudaResourceTypeLinear`.

- ▶ `cudaTextureDesc::readMode` specifies whether integer data should be converted to floating point or not. `cudaTextureReadMode` is defined as:

```

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

```

Note that this applies only to 8-bit and 16-bit integer formats. 32-bit integer format would not be promoted, regardless of whether or not this `cudaTextureDesc::readMode` is set `cudaReadModeNormalizedFloat` is specified.

- ▶ `cudaTextureDesc::sRGB` specifies whether sRGB to linear conversion should be performed during texture fetch.
- ▶ `cudaTextureDesc::borderColor` specifies the float values of color. where: `cudaTextureDesc::borderColor[0]` contains value of 'R', `cudaTextureDesc::borderColor[1]` contains value of 'G', `cudaTextureDesc::borderColor[2]` contains value of 'B', `cudaTextureDesc::borderColor[3]` contains value of 'A' Note that application using integer border color values will need to `<reinterpret_cast>` these values to float. The values are set only when the addressing mode specified by `cudaTextureDesc::addressMode` is `cudaAddressModeBorder`.
- ▶ `cudaTextureDesc::normalizedCoords` specifies whether the texture coordinates will be normalized or not.
- ▶ `cudaTextureDesc::maxAnisotropy` specifies the maximum anisotropy ratio to be used when doing anisotropic filtering. This value will be clamped to the range [1,16].
- ▶ `cudaTextureDesc::mipmapFilterMode` specifies the filter mode when the calculated mipmap level lies between two defined mipmap levels.
- ▶ `cudaTextureDesc::mipmapLevelBias` specifies the offset to be applied to the calculated mipmap level.
- ▶ `cudaTextureDesc::minMipmapLevelClamp` specifies the lower end of the mipmap level range to clamp access to.
- ▶ `cudaTextureDesc::maxMipmapLevelClamp` specifies the upper end of the mipmap level range to clamp access to.

The `cudaResourceViewDesc` struct is defined as

```

struct cudaResourceViewDesc {
    enum cudaResourceViewFormat
    format;
    size_t          width;
    size_t          height;
    size_t          depth;
    unsigned int    firstMipmapLevel;
    unsigned int    lastMipmapLevel;
    unsigned int    firstLayer;
    unsigned int    lastLayer;
};

```

where:

- ▶ `cudaResourceViewDesc::format` specifies how the data contained in the CUDA array or CUDA mipmapped array should be interpreted. Note that this can incur a change in size of the texture data. If the resource view format is a block compressed format, then the underlying CUDA array or CUDA mipmapped array has to have a 32-bit unsigned integer format with 2 or 4 channels, depending on the block compressed format. For ex., BC1 and BC4 require the underlying CUDA array to have a 32-bit unsigned int with 2 channels. The other BC formats require the underlying resource to have the same 32-bit unsigned int format but with 4 channels.
- ▶ `cudaResourceViewDesc::width` specifies the new width of the texture data. If the resource view format is a block compressed format, this value has to be 4 times the original width of the resource. For non block compressed formats, this value has to be equal to that of the original resource.
- ▶ `cudaResourceViewDesc::height` specifies the new height of the texture data. If the resource view format is a block compressed format, this value has to be 4 times the original height of the resource. For non block compressed formats, this value has to be equal to that of the original resource.
- ▶ `cudaResourceViewDesc::depth` specifies the new depth of the texture data. This value has to be equal to that of the original resource.
- ▶ `cudaResourceViewDesc::firstMipmapLevel` specifies the most detailed mipmap level. This will be the new mipmap level zero. For non-mipmapped resources, this value has to be zero. `cudaTextureDesc::minMipmapLevelClamp` and `cudaTextureDesc::maxMipmapLevelClamp` will be relative to this value. For ex., if the `firstMipmapLevel` is set to 2, and a `minMipmapLevelClamp` of 1.2 is specified, then the actual minimum mipmap level clamp will be 3.2.
- ▶ `cudaResourceViewDesc::lastMipmapLevel` specifies the least detailed mipmap level. For non-mipmapped resources, this value has to be zero.
- ▶ `cudaResourceViewDesc::firstLayer` specifies the first layer index for layered textures. This will be the new layer zero. For non-layered resources, this value has to be zero.
- ▶ `cudaResourceViewDesc::lastLayer` specifies the last layer index for layered textures. For non-layered resources, this value has to be zero.



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

See also:

[cudaDestroyTextureObject](#), [cuTexObjectCreate](#)

## **\_\_host\_\_ cudaError\_t cudaDestroyTextureObject (cudaTextureObject\_t texObject)**

Destroys a texture object.

### Parameters

**texObject**

- Texture object to destroy

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

### Description

Destroys the texture object specified by `texObject`.



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

See also:

[cudaCreateTextureObject](#), [cuTexObjectDestroy](#)

## **\_\_host\_\_ cudaError\_t cudaGetChannelDesc (cudaChannelFormatDesc \*desc, cudaArray\_const\_t array)**

Get the channel descriptor of an array.

### Parameters

**desc**

- Channel format

**array**

- Memory array on device

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`

**Description**

Returns in `*desc` the channel descriptor of the CUDA array `array`.



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

**See also:**

`cudaCreateChannelDesc` ( C API), `cudaCreateTextureObject`, `cudaCreateSurfaceObject`

**\_\_host\_\_ cudaError\_t**

**`cudaGetTextureObjectResourceDesc` (`cudaResourceDesc *pResDesc`, `cudaTextureObject_t texObject`)**

Returns a texture object's resource descriptor.

**Parameters****`pResDesc`**

- Resource descriptor

**`texObject`**

- Texture object

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`

**Description**

Returns the resource descriptor for the texture object specified by `texObject`.



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

See also:

`cudaCreateTextureObject`, `cuTexObjectGetResourceDesc`

`__host__ cudaError_t`  
`cudaGetTextureObjectResourceViewDesc`  
`(cudaResourceViewDesc *pResViewDesc,`  
`cudaTextureObject_t texObject)`

Returns a texture object's resource view descriptor.

#### Parameters

**pResViewDesc**

- Resource view descriptor

**texObject**

- Texture object

#### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

#### Description

Returns the resource view descriptor for the texture object specified by `texObject`. If no resource view was specified, `cudaErrorInvalidValue` is returned.



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

See also:

[cudaCreateTextureObject](#), [cuTexObjectGetResourceViewDesc](#)

**\_\_host\_\_ cudaError\_t cudaGetTextureObjectTextureDesc**  
**(cudaTextureDesc \*pTexDesc, cudaTextureObject\_t**  
**texObject)**

Returns a texture object's texture descriptor.

#### Parameters

**pTexDesc**

- Texture descriptor

**texObject**

- Texture object

#### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

#### Description

Returns the texture descriptor for the texture object specified by `texObject`.



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

#### See also:

[cudaCreateTextureObject](#), [cuTexObjectGetTextureDesc](#)

## 5.27. Surface Object Management

This section describes the low level texture object management functions of the CUDA runtime application programming interface. The surface object API is only supported on devices of compute capability 3.0 or higher.



```
__host__ cudaError_t cudaCreateSurfaceObject
(cudaSurfaceObject_t *pSurfObject, const
cudaResourceDesc *pResDesc)
```

Creates a surface object.

### Parameters

#### **pSurfObject**

- Surface object to create

#### **pResDesc**

- Resource descriptor

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidChannelDescriptor](#),  
[cudaErrorInvalidResourceHandle](#)

### Description

Creates a surface object and returns it in `pSurfObject`. `pResDesc` describes the data to perform surface load/stores on. [cudaResourceDesc::resType](#) must be [cudaResourceTypeArray](#) and `cudaResourceDesc::res::array::array` must be set to a valid CUDA array handle.

Surface objects are only supported on devices of compute capability 3.0 or higher. Additionally, a surface object is an opaque value, and, as such, should only be accessed through CUDA API calls.



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

### See also:

[cudaDestroySurfaceObject](#), [cuSurfObjectCreate](#)

## `__host__ cudaError_t cudaDestroySurfaceObject(cudaSurfaceObject_t surfObject)`

Destroys a surface object.

### Parameters

#### **surfObject**

- Surface object to destroy

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

### Description

Destroys the surface object specified by `surfObject`.



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

### See also:

`cudaCreateSurfaceObject`, `cuSurfObjectDestroy`

## `__host__ cudaError_t cudaGetSurfaceObjectResourceDesc(cudaResourceDesc *pResDesc, cudaSurfaceObject_t surfObject)`

Returns a surface object's resource descriptor Returns the resource descriptor for the surface object specified by `surfObject`.

### Parameters

#### **pResDesc**

- Resource descriptor

#### **surfObject**

- Surface object

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`

**Description**

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

**See also:**

`cudaCreateSurfaceObject`, `cuSurfObjectGetResourceDesc`

## 5.28. Version Management

`__host__ cudaError_t cudaDriverGetVersion (int *driverVersion)`

Returns the latest version of CUDA supported by the driver.

**Parameters****driverVersion**

- Returns the CUDA driver version.

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`

**Description**

Returns in `*driverVersion` the latest version of CUDA supported by the driver. The version is returned as (1000 major + 10 minor). For example, CUDA 9.2 would be represented by 9020. If no driver is installed, then 0 is returned as the driver version.

This function automatically returns `cudaErrorInvalidValue` if `driverVersion` is NULL.



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

### See also:

`cudaRuntimeGetVersion`, `cuDriverGetVersion`

## `__host__ __device__ cudaError_t cudaRuntimeGetVersion(int *runtimeVersion)`

Returns the CUDA Runtime version.

### Parameters

#### `runtimeVersion`

- Returns the CUDA Runtime version.

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

### Description

Returns in `*runtimeVersion` the version number of the current CUDA Runtime instance. The version is returned as (1000 major + 10 minor). For example, CUDA 9.2 would be represented by 9020.

This function automatically returns `cudaErrorInvalidValue` if the `runtimeVersion` argument is NULL.



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

See also:

[cudaDriverGetVersion](#), [cuDriverGetVersion](#)

## 5.29. Graph Management

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

```
__host__ cudaError_t cudaGraphAddChildGraphNode(
cudaGraphNode_t *pGraphNode, cudaGraph_t graph,
const cudaGraphNode_t *pDependencies, size_t
numDependencies, cudaGraph_t childGraph)
```

Creates a child graph node and adds it to a graph.

### Parameters

#### **pGraphNode**

- Returns newly created node

#### **graph**

- Graph to which to add the node

#### **pDependencies**

- Dependencies of the node

#### **numDependencies**

- Number of dependencies

#### **childGraph**

- The graph to clone into this node

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

### Description

Creates a new node which executes an embedded graph, and adds it to `graph` with `numDependencies` dependencies specified via `pDependencies`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

The node executes an embedded child graph. The child graph is cloned in this call.



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

#### See also:

`cudaGraphChildGraphNodeGetGraph`, `cudaGraphCreate`, `cudaGraphDestroyNode`, `cudaGraphAddEmptyNode`, `cudaGraphAddKernelNode`, `cudaGraphAddHostNode`, `cudaGraphAddMemcpyNode`, `cudaGraphAddMemsetNode`, `cudaGraphClone`

## `__host__ cudaError_t cudaGraphAddDependencies` (`cudaGraph_t graph`, `const cudaGraphNode_t *from`, `const cudaGraphNode_t *to`, `size_t numDependencies`)

Adds dependency edges to a graph.

### Parameters

#### **graph**

- Graph to which dependencies are added

#### **from**

- Array of nodes that provide the dependencies

#### **to**

- Array of dependent nodes

#### **numDependencies**

- Number of dependencies to be added

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

### Description

The number of dependencies to be added is defined by `numDependencies`. Elements in `pFrom` and `pTo` at corresponding indices define a dependency. Each node in `pFrom` and `pTo` must belong to `graph`.

If `numDependencies` is 0, elements in `pFrom` and `pTo` will be ignored. Specifying an existing dependency will return an error.



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

See also:

`cudaGraphRemoveDependencies`, `cudaGraphGetEdges`,  
`cudaGraphNodeGetDependencies`, `cudaGraphNodeGetDependentNodes`

**`__host__ cudaError_t cudaGraphAddEmptyNode`**  
**`(cudaGraphNode_t *pGraphNode, cudaGraph_t graph,`**  
**`const cudaGraphNode_t *pDependencies, size_t`**  
**`numDependencies)`**

Creates an empty node and adds it to a graph.

#### Parameters

##### **`pGraphNode`**

- Returns newly created node

##### **`graph`**

- Graph to which to add the node

##### **`pDependencies`**

- Dependencies of the node

##### **`numDependencies`**

- Number of dependencies

#### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

## Description

Creates a new node which performs no operation, and adds it to graph with `numDependencies` dependencies specified via `pDependencies`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

An empty node performs no operation during execution, but can be used for transitive ordering. For example, a phased execution graph with 2 groups of  $n$  nodes with a barrier between them can be represented using an empty node and  $2*n$  dependency edges, rather than no empty node and  $n^2$  dependency edges.



- ▶ Graph objects are not threadsafe. [More here.](#)
- ▶ Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.
- ▶ Note that as specified by `cudaStreamAddCallback` no CUDA function may be called from callback. `cudaErrorNotPermitted` may, but is not guaranteed to, be returned as a diagnostic in such case.

## See also:

`cudaGraphCreate`, `cudaGraphDestroyNode`, `cudaGraphAddChildGraphNode`, `cudaGraphAddKernelNode`, `cudaGraphAddHostNode`, `cudaGraphAddMemcpyNode`, `cudaGraphAddMemsetNode`

```
__host__ cudaError_t cudaGraphAddHostNode(
    cudaGraphNode_t *pGraphNode, cudaGraph_t
    graph, const cudaGraphNode_t *pDependencies,
    size_t numDependencies, const cudaHostNodeParams
    *pNodeParams)
```

Creates a host execution node and adds it to a graph.

## Parameters

### **pGraphNode**

- Returns newly created node

### **graph**

- Graph to which to add the node

### **pDependencies**

- Dependencies of the node



**numDependencies**

- Number of dependencies

**pNodeParams**

- Parameters for the host node

**Returns**

[cudaSuccess](#), [cudaErrorNotSupported](#), [cudaErrorInvalidValue](#)

**Description**

Creates a new CPU execution node and adds it to graph with `numDependencies` dependencies specified via `pDependencies` and arguments specified in `pNodeParams`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

When the graph is launched, the node will invoke the specified CPU function. Host nodes are not supported under MPS with pre-Volta GPUs.



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

**See also:**

[cudaLaunchHostFunc](#), [cudaGraphHostNodeGetParams](#),  
[cudaGraphHostNodeSetParams](#), [cudaGraphCreate](#), [cudaGraphDestroyNode](#),  
[cudaGraphAddChildGraphNode](#), [cudaGraphAddEmptyNode](#),  
[cudaGraphAddKernelNode](#), [cudaGraphAddMemcpyNode](#),  
[cudaGraphAddMemsetNode](#)

```
__host__ cudaError_t cudaGraphAddKernelNode
(cudaGraphNode_t *pGraphNode, cudaGraph_t graph,
const cudaGraphNode_t *pDependencies, size_t
```

## numDependencies, const cudaKernelNodeParams \*pNodeParams)

Creates a kernel execution node and adds it to a graph.

### Parameters

#### pGraphNode

- Returns newly created node

#### graph

- Graph to which to add the node

#### pDependencies

- Dependencies of the node

#### numDependencies

- Number of dependencies

#### pNodeParams

- Parameters for the GPU execution node

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidDeviceFunction](#)

### Description

Creates a new kernel execution node and adds it to graph with numDependencies dependencies specified via pDependencies and arguments specified in pNodeParams. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. pDependencies may not have any duplicate entries. A handle to the new node will be returned in pGraphNode.

The [cudaKernelNodeParams](#) structure is defined as:

```
struct cudaKernelNodeParams
{
    void* func;
    dim3 gridDim;
    dim3 blockDim;
    unsigned int sharedMemBytes;
    void **kernelParams;
    void **extra;
};
```

When the graph is launched, the node will invoke kernel func on a (gridDim.x x gridDim.y x gridDim.z) grid of blocks. Each block contains (blockDim.x x blockDim.y x blockDim.z) threads.

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

Kernel parameters to func can be specified in one of two ways:

1) Kernel parameters can be specified via `kernelParams`. If the kernel has `N` parameters, then `kernelParams` needs to be an array of `N` pointers. Each pointer, from `kernelParams[0]` to `kernelParams[N-1]`, points to the region of memory from which the actual parameter will be copied. The number of kernel parameters and their offsets and sizes do not need to be specified as that information is retrieved directly from the kernel's image.

2) Kernel parameters can also be packaged by the application into a single buffer that is passed in via `extra`. This places the burden on the application of knowing each kernel parameter's size and alignment/padding within the buffer. The `extra` parameter exists to allow this function to take additional less commonly used arguments. `extra` specifies a list of names of extra settings and their corresponding values. Each extra setting name is immediately followed by the corresponding value. The list must be terminated with either `NULL` or `CU_LAUNCH_PARAM_END`.

- ▶ `CU_LAUNCH_PARAM_END`, which indicates the end of the `extra` array;
- ▶ `CU_LAUNCH_PARAM_BUFFER_POINTER`, which specifies that the next value in `extra` will be a pointer to a buffer containing all the kernel parameters for launching kernel `func`;
- ▶ `CU_LAUNCH_PARAM_BUFFER_SIZE`, which specifies that the next value in `extra` will be a pointer to a `size_t` containing the size of the buffer specified with `CU_LAUNCH_PARAM_BUFFER_POINTER`;

The error `cudaErrorInvalidValue` will be returned if kernel parameters are specified with both `kernelParams` and `extra` (i.e. both `kernelParams` and `extra` are non-NULL).

The `kernelParams` or `extra` array, as well as the argument values it points to, are copied during this call.



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.



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

**See also:**

[cudaLaunchKernel](#), [cudaGraphKernelNodeGetParams](#),  
[cudaGraphKernelNodeSetParams](#), [cudaGraphCreate](#), [cudaGraphDestroyNode](#),  
[cudaGraphAddChildGraphNode](#), [cudaGraphAddEmptyNode](#),  
[cudaGraphAddHostNode](#), [cudaGraphAddMemcpyNode](#), [cudaGraphAddMemsetNode](#)

```
__host__ cudaError_t cudaGraphAddMemcpyNode  

(cudaGraphNode_t *pGraphNode, cudaGraph_t  

graph, const cudaGraphNode_t *pDependencies,  

size_t numDependencies, const cudaMemcpy3DParms  

*pCopyParams)
```

Creates a memcpy node and adds it to a graph.

**Parameters****pGraphNode**

- Returns newly created node

**graph**

- Graph to which to add the node

**pDependencies**

- Dependencies of the node

**numDependencies**

- Number of dependencies

**pCopyParams**

- Parameters for the memory copy

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#)

**Description**

Creates a new memcpy node and adds it to graph with numDependencies dependencies specified via pDependencies. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. pDependencies may not have any duplicate entries. A handle to the new node will be returned in pGraphNode.

When the graph is launched, the node will perform the memcpy described by pCopyParams. See [cudaMemcpy3D\(\)](#) for a description of the structure and its restrictions.

Memcpy nodes have some additional restrictions with regards to managed memory, if the system contains at least one device which has a zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`.



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

See also:

`cudaMemcpy3D`, `cudaGraphMemcpyNodeGetParams`,  
`cudaGraphMemcpyNodeSetParams`, `cudaGraphCreate`, `cudaGraphDestroyNode`,  
`cudaGraphAddChildGraphNode`, `cudaGraphAddEmptyNode`,  
`cudaGraphAddKernelNode`, `cudaGraphAddHostNode`, `cudaGraphAddMemsetNode`

```
__host__ cudaError_t cudaGraphAddMemsetNode(
    cudaGraphNode_t *pGraphNode, cudaGraph_t
    graph, const cudaGraphNode_t *pDependencies,
    size_t numDependencies, const cudaMemcpyParams
    *pMemcpyParams)
```

Creates a memset node and adds it to a graph.

### Parameters

#### **pGraphNode**

- Returns newly created node

#### **graph**

- Graph to which to add the node

#### **pDependencies**

- Dependencies of the node

#### **numDependencies**

- Number of dependencies

#### **pMemcpyParams**

- Parameters for the memory set

## Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidDevice](#)

## Description

Creates a new memset node and adds it to graph with `numDependencies` dependencies specified via `pDependencies`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

The element size must be 1, 2, or 4 bytes. When the graph is launched, the node will perform the memset described by `pMemsetParams`.



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

## See also:

[cudaMemset2D](#), [cudaGraphMemsetNodeGetParams](#),  
[cudaGraphMemsetNodeSetParams](#), [cudaGraphCreate](#), [cudaGraphDestroyNode](#),  
[cudaGraphAddChildGraphNode](#), [cudaGraphAddEmptyNode](#),  
[cudaGraphAddKernelNode](#), [cudaGraphAddHostNode](#), [cudaGraphAddMemcpyNode](#)

## \_\_host\_\_ cudaError\_t

### cudaGraphChildGraphNodeGetGraph (cudaGraphNode\_t node, cudaGraph\_t \*pGraph)

Gets a handle to the embedded graph of a child graph node.

## Parameters

### node

- Node to get the embedded graph for

### pGraph

- Location to store a handle to the graph

## Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

## Description

Gets a handle to the embedded graph in a child graph node. This call does not clone the graph. Changes to the graph will be reflected in the node, and the node retains ownership of the graph.



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

## See also:

[cudaGraphAddChildGraphNode](#), [cudaGraphNodeFindInClone](#)

## `__host__ cudaError_t cudaGraphClone (cudaGraph_t *pGraphClone, cudaGraph_t originalGraph)`

Clones a graph.

## Parameters

### **pGraphClone**

- Returns newly created cloned graph

### **originalGraph**

- Graph to clone

## Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorMemoryAllocation](#)

## Description

This function creates a copy of `originalGraph` and returns it in `pGraphClone`. All parameters are copied into the cloned graph. The original graph may be modified after this call without affecting the clone.

Child graph nodes in the original graph are recursively copied into the clone.



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

See also:

`cudaGraphCreate`, `cudaGraphNodeFindInClone`

## `__host__ cudaError_t cudaGraphCreate (cudaGraph_t *pGraph, unsigned int flags)`

Creates a graph.

### Parameters

#### `pGraph`

- Returns newly created graph

#### `flags`

- Graph creation flags, must be 0

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorMemoryAllocation`

### Description

Creates an empty graph, which is returned via `pGraph`.



- ▶ Graph objects are not threadsafe. [More here.](#)
- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ Note that this function may also return `cudaErrorInitializationError`, `cudaErrorInsufficientDriver` or `cudaErrorNoDevice` if this call tries to initialize internal CUDA RT state.



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

#### See also:

[cudaGraphAddChildGraphNode](#), [cudaGraphAddEmptyNode](#),  
[cudaGraphAddKernelNode](#), [cudaGraphAddHostNode](#), [cudaGraphAddMemcpyNode](#),  
[cudaGraphAddMemsetNode](#), [cudaGraphInstantiate](#), [cudaGraphDestroy](#),  
[cudaGraphGetNodes](#), [cudaGraphGetRootNodes](#), [cudaGraphGetEdges](#),  
[cudaGraphClone](#)

## `__host__ cudaError_t cudaGraphDestroy (cudaGraph_t graph)`

Destroys a graph.

#### Parameters

##### **graph**

- Graph to destroy

#### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

#### Description

Destroys the graph specified by `graph`, as well as all of its nodes.



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

#### See also:

[cudaGraphCreate](#)

## `__host__ cudaError_t cudaGraphDestroyNode(cudaGraphNode_t node)`

Remove a node from the graph.

### Parameters

**node**

- Node to remove

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

### Description

Removes `node` from its graph. This operation also severs any dependencies of other nodes on `node` and vice versa.



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

### See also:

`cudaGraphAddChildGraphNode`, `cudaGraphAddEmptyNode`,  
`cudaGraphAddKernelNode`, `cudaGraphAddHostNode`, `cudaGraphAddMemcpyNode`,  
`cudaGraphAddMemsetNode`

## `__host__ cudaError_t cudaGraphExecDestroy(cudaGraphExec_t graphExec)`

Destroys an executable graph.

### Parameters

**graphExec**

- Executable graph to destroy

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`

**Description**

Destroys the executable graph specified by `graphExec`.



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

**See also:**

`cudaGraphInstantiate`, `cudaGraphLaunch`

**`__host__ cudaError_t cudaGraphExecHostNodeSetParams`  
**`(cudaGraphExec_t hGraphExec, cudaGraphNode_t node,`  
**`const cudaHostNodeParams *pNodeParams)`******

Sets the parameters for a host node in the given `graphExec`.

**Parameters****`hGraphExec`**

- The executable graph in which to set the specified node

**`node`**

- Host node from the graph which was used to instantiate `graphExec`

**`pNodeParams`**

- Updated Parameters to set

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`,

## Description

Updates the work represented by `node` in `hGraphExec` as though `node` had contained `pNodeParams` at instantiation. `node` must remain in the graph which was used to instantiate `hGraphExec`. Changed edges to and from `node` are ignored.

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



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

## See also:

[cudaGraphAddHostNode](#), [cudaGraphHostNodeSetParams](#) [cudaGraphInstantiate](#)  
[cudaGraphExecKernelNodeSetParams](#) [cudaGraphExecMemcpyNodeSetParams](#)  
[cudaGraphExecMemsetNodeSetParams](#)

## \_\_host\_\_ cudaError\_t

[cudaGraphExecKernelNodeSetParams](#) ([cudaGraphExec\\_t](#)  
[hGraphExec](#), [cudaGraphNode\\_t](#) node, const  
[cudaKernelNodeParams](#) \*pNodeParams)

Sets the parameters for a kernel node in the given graphExec.

## Parameters

### **hGraphExec**

- The executable graph in which to set the specified node

### **node**

- kernel node from the graph from which graphExec was instantiated

### **pNodeParams**

- Updated Parameters to set

## Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#),

## Description

Sets the parameters of a kernel node in an executable graph `hGraphExec`. The node is identified by the corresponding node `node` in the non-executable graph, from which the executable graph was instantiated.

`node` must not have been removed from the original graph. The `func` field of `nodeParams` cannot be modified and must match the original value. All other values can be modified.

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



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

See also:

[cudaGraphAddKernelNode](#), [cudaGraphKernelNodeSetParams](#), [cudaGraphInstantiate](#)

`__host__ cudaError_t`

`cudaGraphExecMemcpyNodeSetParams`

(`cudaGraphExec_t hGraphExec`, `cudaGraphNode_t node`,  
`const cudaMemcpy3DParms *pNodeParams`)

Sets the parameters for a memcpy node in the given `graphExec`.

## Parameters

**`hGraphExec`**

- The executable graph in which to set the specified node

**node****pNodeParams**

- Updated Parameters to set

**Returns**[cudaSuccess](#), [cudaErrorInvalidValue](#),**Description**

Updates the work represented by `node` in `hGraphExec` as though `node` had contained `pNodeParams` at instantiation. `node` must remain in the graph which was used to instantiate `hGraphExec`. Changed edges to and from `node` are ignored.

The source and destination memory in `pNodeParams` must be allocated from the same contexts as the original source and destination memory. Both the instantiation-time memory operands and the memory operands in `pNodeParams` must be 1-dimensional. Zero-length operations are not supported.

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

Returns [cudaErrorInvalidValue](#) if the memory operands' mappings changed or either the original or new memory operands are multidimensional.



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

**See also:**[cudaGraphAddMemcpyNode](#), [cudaGraphMemcpyNodeSetParams](#)[cudaGraphInstantiate](#) [cudaGraphExecKernelNodeSetParams](#)[cudaGraphExecMemsetNodeSetParams](#) [cudaGraphExecHostNodeSetParams](#)**`__host__ cudaError_t`****`cudaGraphExecMemsetNodeSetParams`**

## (`cudaGraphExec_t hGraphExec`, `cudaGraphNode_t node`, `const cudaMemcpyParams *pNodeParams`)

Sets the parameters for a memset node in the given graphExec.

### Parameters

#### **hGraphExec**

- The executable graph in which to set the specified node

#### **node**

- Memset node from the graph which was used to instantiate graphExec

#### **pNodeParams**

- Updated Parameters to set

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`,

### Description

Updates the work represented by `node` in `hGraphExec` as though `node` had contained `pNodeParams` at instantiation. `node` must remain in the graph which was used to instantiate `hGraphExec`. Changed edges to and from `node` are ignored.

The destination memory in `pNodeParams` must be allocated from the same context as the original destination memory. Both the instantiation-time memory operand and the memory operand in `pNodeParams` must be 1-dimensional. Zero-length operations are not supported.

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

Returns `cudaErrorInvalidValue` if the memory operand's mappings changed or either the original or new memory operand are multidimensional.



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

**See also:**

[cudaGraphAddMemsetNode](#), [cudaGraphMemsetNodeSetParams](#) [cudaGraphInstantiate](#)  
[cudaGraphExecKernelNodeSetParams](#) [cudaGraphExecMemcpyNodeSetParams](#)  
[cudaGraphExecHostNodeSetParams](#)

```
__host__ cudaError_t cudaGraphExecUpdate  

(cudaGraphExec_t hGraphExec, cudaGraph_t  

hGraph, cudaGraphNode_t *hErrorNode_out,  

cudaGraphExecUpdateResult *updateResult_out)
```

Check whether an executable graph can be updated with a graph and perform the update if possible.

**Parameters****hGraphExec**

The instantiated graph to be updated

**hGraph**

The graph containing the updated parameters

**hErrorNode\_out**

The node which caused the permissibility check to forbid the update, if any

**updateResult\_out**

Whether the graph update was permitted. If was forbidden, the reason why

**Returns**

[cudaSuccess](#), [cudaErrorGraphExecUpdateFailure](#),

**Description**

Updates the node parameters in the instantiated graph specified by `hGraphExec` with the node parameters in a topologically identical graph specified by `hGraph`.

Limitations:

- ▶ Kernel nodes:
  - ▶ The function must not change (same restriction as [cudaGraphExecKernelNodeSetParams\(\)](#))
- ▶ Memset and memcpy nodes:
  - ▶ The CUDA device(s) to which the operand(s) was allocated/mapped cannot change.
  - ▶ The source/destination memory must be allocated from the same contexts as the original source/destination memory.
  - ▶ Only 1D memsets can be changed.



- ▶ Additional memcpy node restrictions:
  - ▶ Changing either the source or destination memory type(i.e. CU\_MEMORYTYPE\_DEVICE, CU\_MEMORYTYPE\_ARRAY, etc.) is not supported.

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

Some node types are not currently supported:

- ▶ Empty graph nodes(cudaGraphNodeEmpty)
- ▶ Child graphs(cudaGraphNodeGraph).

cudaGraphExecUpdate sets `updateResult_out` to `cudaGraphExecUpdateErrorTopologyChanged` under the following conditions:

- ▶ The count of nodes directly in `hGraphExec` and `hGraph` differ, in which case `hErrorNode_out` is NULL.
- ▶ A node is deleted in `hGraph` but not its pair from `hGraphExec`, in which case `hErrorNode_out` is NULL.
- ▶ A node is deleted in `hGraphExec` but not its pair from `hGraph`, in which case `hErrorNode_out` is the pairless node from `hGraph`.
- ▶ The dependent nodes of a pair differ, in which case `hErrorNode_out` is the node from `hGraph`.

cudaGraphExecUpdate sets `updateResult_out` to:

- ▶ `cudaGraphExecUpdateError` if passed an invalid value.
- ▶ `cudaGraphExecUpdateErrorTopologyChanged` if the graph topology changed
- ▶ `cudaGraphExecUpdateErrorNodeTypeChanged` if the type of a node changed, in which case `hErrorNode_out` is set to the node from `hGraph`.
- ▶ `cudaGraphExecUpdateErrorFunctionChanged` if the func field of a kernel changed, in which case `hErrorNode_out` is set to the node from `hGraph`
- ▶ `cudaGraphExecUpdateErrorParametersChanged` if any parameters to a node changed in a way that is not supported, in which case `hErrorNode_out` is set to the node from `hGraph`
- ▶ `cudaGraphExecUpdateErrorNotSupported` if something about a node is unsupported, like the node's type or configuration, in which case `hErrorNode_out` is set to the node from `hGraph`

If `updateResult_out` isn't set in one of the situations described above, the update check passes and `cudaGraphExecUpdate` updates `hGraphExec` to match the contents of `hGraph`. If an error happens during the update, `updateResult_out` will be set to `cudaGraphExecUpdateError`; otherwise, `updateResult_out` is set to `cudaGraphExecUpdateSuccess`.

`cudaGraphExecUpdate` returns `cudaSuccess` when the update was performed successfully. It returns `cudaErrorGraphExecUpdateFailure` if the graph update was not performed because it included changes which violated constraints specific to instantiated graph update.



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

See also:

`cudaGraphInstantiate`,

**`__host__ cudaError_t cudaGraphGetEdges (cudaGraph_t graph, cudaGraphNode_t *from, cudaGraphNode_t *to, size_t *numEdges)`**

Returns a graph's dependency edges.

### Parameters

#### **graph**

- Graph to get the edges from

#### **from**

- Location to return edge endpoints

#### **to**

- Location to return edge endpoints

#### **numEdges**

- See description

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

### Description

Returns a list of `graph`'s dependency edges. Edges are returned via corresponding indices in `from` and `to`; that is, the node in `to[i]` has a dependency on the node in

`from[i]`. `from` and `to` may both be `NULL`, in which case this function only returns the number of edges in `numEdges`. Otherwise, `numEdges` entries will be filled in. If `numEdges` is higher than the actual number of edges, the remaining entries in `from` and `to` will be set to `NULL`, and the number of edges actually returned will be written to `numEdges`.



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

#### See also:

`cudaGraphGetNodes`, `cudaGraphGetRootNodes`, `cudaGraphAddDependencies`, `cudaGraphRemoveDependencies`, `cudaGraphNodeGetDependencies`, `cudaGraphNodeGetDependentNodes`

## `__host__ cudaError_t cudaGraphGetNodes (cudaGraph_t graph, cudaGraphNode_t *nodes, size_t *numNodes)`

Returns a graph's nodes.

### Parameters

#### **graph**

- Graph to query

#### **nodes**

- Pointer to return the nodes

#### **numNodes**

- See description

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

### Description

Returns a list of `graph`'s nodes. `nodes` may be `NULL`, in which case this function will return the number of nodes in `numNodes`. Otherwise, `numNodes` entries will be filled

in. If `numNodes` is higher than the actual number of nodes, the remaining entries in `nodes` will be set to `NULL`, and the number of nodes actually obtained will be returned in `numNodes`.



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

#### See also:

[cudaGraphCreate](#), [cudaGraphGetRootNodes](#), [cudaGraphGetEdges](#),  
[cudaGraphNodeGetType](#), [cudaGraphNodeGetDependencies](#),  
[cudaGraphNodeGetDependentNodes](#)

## `__host__ cudaError_t cudaGraphGetRootNodes` `(cudaGraph_t graph, cudaGraphNode_t *pRootNodes,` `size_t *pNumRootNodes)`

Returns a graph's root nodes.

#### Parameters

##### **graph**

- Graph to query

##### **pRootNodes**

- Pointer to return the root nodes

##### **pNumRootNodes**

- See description

#### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

#### Description

Returns a list of `graph`'s root nodes. `pRootNodes` may be `NULL`, in which case this function will return the number of root nodes in `pNumRootNodes`. Otherwise, `pNumRootNodes` entries will be filled in. If `pNumRootNodes` is higher than the actual

number of root nodes, the remaining entries in `pRootNodes` will be set to `NULL`, and the number of nodes actually obtained will be returned in `pNumRootNodes`.



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

See also:

`cudaGraphCreate`, `cudaGraphGetNodes`, `cudaGraphGetEdges`,  
`cudaGraphNodeGetType`, `cudaGraphNodeGetDependencies`,  
`cudaGraphNodeGetDependentNodes`

## `__host__ cudaError_t cudaGraphHostNodeGetParams` `(cudaGraphNode_t node, cudaHostNodeParams` `*pNodeParams)`

Returns a host node's parameters.

### Parameters

**node**

- Node to get the parameters for

**pNodeParams**

- Pointer to return the parameters

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

### Description

Returns the parameters of host node `node` in `pNodeParams`.



- ▶ Graph objects are not threadsafe. [More here.](#)

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

See also:

`cudaLaunchHostFunc`, `cudaGraphAddHostNode`, `cudaGraphHostNodeSetParams`

**`__host__ cudaError_t cudaGraphHostNodeSetParams`  
(`cudaGraphNode_t node`, `const cudaHostNodeParams *pNodeParams`)**

Sets a host node's parameters.

### Parameters

**node**

- Node to set the parameters for

**pNodeParams**

- Parameters to copy

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

### Description

Sets the parameters of host node `node` to `nodeParams`.



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

See also:

[cudaLaunchHostFunc](#), [cudaGraphAddHostNode](#), [cudaGraphHostNodeGetParams](#)

```
__host__ cudaError_t cudaGraphInstantiate(
    (cudaGraphExec_t *pGraphExec, cudaGraph_t graph,
    cudaGraphNode_t *pErrorNode, char *pLogBuffer, size_t
    bufferSize)
```

Creates an executable graph from a graph.

### Parameters

#### **pGraphExec**

- Returns instantiated graph

#### **graph**

- Graph to instantiate

#### **pErrorNode**

- In case of an instantiation error, this may be modified to indicate a node contributing to the error

#### **pLogBuffer**

- A character buffer to store diagnostic messages

#### **bufferSize**

- Size of the log buffer in bytes

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

### Description

Instantiates `graph` as an executable graph. The graph is validated for any structural constraints or intra-node constraints which were not previously validated. If instantiation is successful, a handle to the instantiated graph is returned in `pGraphExec`.

If there are any errors, diagnostic information may be returned in `pErrorNode` and `pLogBuffer`. This is the primary way to inspect instantiation errors. The output will be null terminated unless the diagnostics overflow the buffer. In this case, they will be truncated, and the last byte can be inspected to determine if truncation occurred.



- ▶ Graph objects are not threadsafe. [More here](#).
- ▶ Note that this function may also return error codes from previous, asynchronous launches.

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

See also:

`cudaGraphCreate`, `cudaGraphLaunch`, `cudaGraphExecDestroy`

**`__host__ cudaError_t cudaGraphKernelNodeGetParams`  
(`cudaGraphNode_t node`, `cudaKernelNodeParams`  
`*pNodeParams`)**

Returns a kernel node's parameters.

### Parameters

#### **node**

- Node to get the parameters for

#### **pNodeParams**

- Pointer to return the parameters

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDeviceFunction`

### Description

Returns the parameters of kernel node `node` in `pNodeParams`. The `kernelParams` or `extra` array returned in `pNodeParams`, as well as the argument values it points to, are owned by the node. This memory remains valid until the node is destroyed or its parameters are modified, and should not be modified directly. Use `cudaGraphKernelNodeSetParams` to update the parameters of this node.

The params will contain either `kernelParams` or `extra`, according to which of these was most recently set on the node.



- ▶ Graph objects are not threadsafe. [More here.](#)
- ▶ Note that this function may also return error codes from previous, asynchronous launches.



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

**See also:**

`cudaLaunchKernel`, `cudaGraphAddKernelNode`, `cudaGraphKernelNodeSetParams`

**`__host__ cudaError_t cudaGraphKernelNodeSetParams`  
`(cudaGraphNode_t node, const cudaKernelNodeParams`  
`*pNodeParams)`**

Sets a kernel node's parameters.

**Parameters****node**

- Node to set the parameters for

**pNodeParams**

- Parameters to copy

**Returns**

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`,  
`cudaErrorMemoryAllocation`

**Description**

Sets the parameters of kernel node `node` to `pNodeParams`.



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

See also:

[cudaLaunchKernel](#), [cudaGraphAddKernelNode](#), [cudaGraphKernelNodeGetParams](#)

## `__host__ cudaError_t cudaGraphLaunch` (`cudaGraphExec_t graphExec`, `cudaStream_t stream`)

Launches an executable graph in a stream.

### Parameters

#### **graphExec**

- Executable graph to launch

#### **stream**

- Stream in which to launch the graph

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

### Description

Executes `graphExec` in `stream`. Only one instance of `graphExec` may be executing at a time. Each launch is ordered behind both any previous work in `stream` and any previous launches of `graphExec`. To execute a graph concurrently, it must be instantiated multiple times into multiple executable graphs.



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

See also:

[cudaGraphInstantiate](#), [cudaGraphExecDestroy](#)

```
__host__ cudaError_t cudaGraphMemcpyNodeGetParams
(cudaGraphNode_t node, cudaMemcpy3DParms
*pNodeParams)
```

Returns a memcpy node's parameters.

### Parameters

#### node

- Node to get the parameters for

#### pNodeParams

- Pointer to return the parameters

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

### Description

Returns the parameters of memcpy node `node` in `pNodeParams`.



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

### See also:

[cudaMemcpy3D](#), [cudaGraphAddMemcpyNode](#), [cudaGraphMemcpyNodeSetParams](#)

```
__host__ cudaError_t cudaGraphMemcpyNodeSetParams
(cudaGraphNode_t node, const cudaMemcpy3DParms
*pNodeParams)
```

Sets a memcpy node's parameters.

### Parameters

#### **node**

- Node to set the parameters for

#### **pNodeParams**

- Parameters to copy

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#),

### Description

Sets the parameters of memcpy node `node` to `pNodeParams`.



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

### See also:

[cudaMemcpy3D](#), [cudaGraphAddMemcpyNode](#), [cudaGraphMemcpyNodeGetParams](#)

```
__host__ cudaError_t cudaGraphMemsetNodeGetParams
(cudaGraphNode_t node, cudaMemcpyParams
*pNodeParams)
```

Returns a memset node's parameters.

### Parameters

#### node

- Node to get the parameters for

#### pNodeParams

- Pointer to return the parameters

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

### Description

Returns the parameters of memset node `node` in `pNodeParams`.



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

### See also:

[cudaMemset2D](#), [cudaGraphAddMemsetNode](#), [cudaGraphMemsetNodeSetParams](#)

```
__host__ cudaError_t cudaGraphMemsetNodeSetParams
(cudaGraphNode_t node, const cudaMemcpyParams
*pNodeParams)
```

Sets a memset node's parameters.

### Parameters

#### **node**

- Node to set the parameters for

#### **pNodeParams**

- Parameters to copy

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

### Description

Sets the parameters of memset node `node` to `pNodeParams`.



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

### See also:

[cudaMemset2D](#), [cudaGraphAddMemsetNode](#), [cudaGraphMemsetNodeGetParams](#)

## `__host__ cudaError_t cudaGraphNodeFindInClone` (`cudaGraphNode_t *pNode`, `cudaGraphNode_t` `originalNode`, `cudaGraph_t clonedGraph`)

Finds a cloned version of a node.

### Parameters

#### **pNode**

- Returns handle to the cloned node

#### **originalNode**

- Handle to the original node

#### **clonedGraph**

- Cloned graph to query

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

### Description

This function returns the node in `clonedGraph` corresponding to `originalNode` in the original graph.

`clonedGraph` must have been cloned from `originalGraph` via `cudaGraphClone`. `originalNode` must have been in `originalGraph` at the time of the call to `cudaGraphClone`, and the corresponding cloned node in `clonedGraph` must not have been removed. The cloned node is then returned via `pClonedNode`.



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

### See also:

`cudaGraphClone`

```
__host__ cudaError_t cudaGraphNodeGetDependencies
(cudaGraphNode_t node, cudaGraphNode_t
*pDependencies, size_t *pNumDependencies)
```

Returns a node's dependencies.

### Parameters

#### **node**

- Node to query

#### **pDependencies**

- Pointer to return the dependencies

#### **pNumDependencies**

- See description

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

### Description

Returns a list of `node`'s dependencies. `pDependencies` may be NULL, in which case this function will return the number of dependencies in `pNumDependencies`. Otherwise, `pNumDependencies` entries will be filled in. If `pNumDependencies` is higher than the actual number of dependencies, the remaining entries in `pDependencies` will be set to NULL, and the number of nodes actually obtained will be returned in `pNumDependencies`.



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

### See also:

[cudaGraphNodeGetDependentNodes](#), [cudaGraphGetNodes](#), [cudaGraphGetRootNodes](#), [cudaGraphGetEdges](#), [cudaGraphAddDependencies](#), [cudaGraphRemoveDependencies](#)



`__host__ cudaError_t  
 cudaGraphNodeGetDependentNodes (cudaGraphNode_t  
 node, cudaGraphNode_t *pDependentNodes, size_t  
 *pNumDependentNodes)`

Returns a node's dependent nodes.

### Parameters

#### **node**

- Node to query

#### **pDependentNodes**

- Pointer to return the dependent nodes

#### **pNumDependentNodes**

- See description

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

### Description

Returns a list of `node`'s dependent nodes. `pDependentNodes` may be `NULL`, in which case this function will return the number of dependent nodes in `pNumDependentNodes`. Otherwise, `pNumDependentNodes` entries will be filled in. If `pNumDependentNodes` is higher than the actual number of dependent nodes, the remaining entries in `pDependentNodes` will be set to `NULL`, and the number of nodes actually obtained will be returned in `pNumDependentNodes`.



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

### See also:

`cudaGraphNodeGetDependencies`, `cudaGraphGetNodes`, `cudaGraphGetRootNodes`, `cudaGraphGetEdges`, `cudaGraphAddDependencies`, `cudaGraphRemoveDependencies`

## `__host__ cudaError_t cudaGraphNodeGetType` `(cudaGraphNode_t node, cudaGraphNodeType *pType)`

Returns a node's type.

### Parameters

#### **node**

- Node to query

#### **pType**

- Pointer to return the node type

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`

### Description

Returns the node type of `node` in `pType`.



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

### See also:

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

```
__host__ cudaError_t cudaGraphRemoveDependencies(
cudaGraph_t graph, const cudaGraphNode_t *from,
const cudaGraphNode_t *to, size_t numDependencies)
```

Removes dependency edges from a graph.

### Parameters

#### **graph**

- Graph from which to remove dependencies

#### **from**

- Array of nodes that provide the dependencies

#### **to**

- Array of dependent nodes

#### **numDependencies**

- Number of dependencies to be removed

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)

### Description

The number of `pDependencies` to be removed is defined by `numDependencies`. Elements in `pFrom` and `pTo` at corresponding indices define a dependency. Each node in `pFrom` and `pTo` must belong to `graph`.

If `numDependencies` is 0, elements in `pFrom` and `pTo` will be ignored. Specifying a non-existing dependency will return an error.



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

### See also:

[cudaGraphAddDependencies](#), [cudaGraphGetEdges](#), [cudaGraphNodeGetDependencies](#), [cudaGraphNodeGetDependentNodes](#)

## 5.30. C++ API Routines

C++-style interface built on top of CUDA runtime API.

This section describes the C++ high level API functions of the CUDA runtime application programming interface. To use these functions, your application needs to be compiled with the `nvcc` compiler.

### `__cudaOccupancyB2DHelper`

`cppClassifierVisibility: visibility=public`

```
template < class T, int dim > __host__ cudaError_t
cudaBindSurfaceToArray (const surfaceTdim surf,
cudaArray_const_t array)
```

[C++ API] Binds an array to a surface

#### Parameters

**surf**

- Surface to bind

**array**

- Memory array on device

#### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidSurface`

#### Description

Binds the CUDA array `array` to the surface reference `surf`. The channel descriptor is inherited from the CUDA array. Any CUDA array previously bound to `surf` is unbound.



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

**See also:**

[cudaBindSurfaceToArray \( C API\)](#), [cudaBindSurfaceToArray \( C++ API\)](#)

```
template < class T, int dim > __host__ cudaError_t
cudaBindSurfaceToArray (const surfaceTdim surf,
cudaArray_const_t array, const cudaChannelFormatDesc
desc)
```

[C++ API] Binds an array to a surface

**Parameters****surf**

- Surface to bind

**array**

- Memory array on device

**desc**

- Channel format

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidSurface](#)

**Description**

Binds the CUDA array `array` to the surface reference `surf`. `desc` describes how the memory is interpreted when dealing with the surface. Any CUDA array previously bound to `surf` is unbound.



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

**See also:**

[cudaBindSurfaceToArray \( C API\)](#), [cudaBindSurfaceToArray \( C++ API, inherited channel descriptor\)](#)

```
template < class T, int dim, enum cudaTextureReadMode
readMode > __host__ cudaError_t cudaBindTexture
(size_t *offset, const textureTdimreadMode tex, const
void *devPtr, size_t size)
```

[C++ API] Binds a memory area to a texture

### Parameters

#### offset

- Offset in bytes

#### tex

- Texture to bind

#### devPtr

- Memory area on device

#### size

- Size of the memory area pointed to by devPtr

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidTexture](#)

### Description

Binds `size` bytes of the memory area pointed to by `devPtr` to texture reference `tex`. The channel descriptor is inherited from the texture reference type. The `offset` parameter is an optional byte offset as with the low-level `cudaBindTexture( size_t*, const struct textureReference*, const void*, const struct cudaChannelFormatDesc*, size_t)` function. Any memory previously bound to `tex` is unbound.



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

### See also:

[cudaCreateChannelDesc](#) ( C++ API), [cudaGetChannelDesc](#), [cudaGetTextureReference](#), [cudaBindTexture](#) ( C API), [cudaBindTexture](#) ( C++ API), [cudaBindTexture2D](#) ( C++ API),

`cudaBindTexture2D` ( C++ API, inherited channel descriptor), `cudaBindTextureToArray` ( C++ API), `cudaBindTextureToArray` ( C++ API, inherited channel descriptor), `cudaUnbindTexture` ( C++ API), `cudaGetTextureAlignmentOffset` ( C++ API)

```
template < class T, int dim, enum cudaTextureReadMode
readMode > __host__ cudaError_t cudaBindTexture
(size_t *offset, const textureTdimreadMode tex, const
void *devPtr, const cudaChannelFormatDesc desc, size_t
size)
```

[C++ API] Binds a memory area to a texture

### Parameters

#### **offset**

- Offset in bytes

#### **tex**

- Texture to bind

#### **devPtr**

- Memory area on device

#### **desc**

- Channel format

#### **size**

- Size of the memory area pointed to by devPtr

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidTexture`

### Description

Binds `size` bytes of the memory area pointed to by `devPtr` to texture reference `tex`. `desc` describes how the memory is interpreted when fetching values from the texture. The `offset` parameter is an optional byte offset as with the low-level `cudaBindTexture()` function. Any memory previously bound to `tex` is unbound.



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

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

#### See also:

`cudaCreateChannelDesc` ( C++ API), `cudaGetChannelDesc`, `cudaGetTextureReference`, `cudaBindTexture` ( C API), `cudaBindTexture` ( C++ API, inherited channel descriptor), `cudaBindTexture2D` ( C++ API), `cudaBindTexture2D` ( C++ API, inherited channel descriptor), `cudaBindTextureToArray` ( C++ API), `cudaBindTextureToArray` ( C++ API, inherited channel descriptor), `cudaUnbindTexture` ( C++ API), `cudaGetTextureAlignmentOffset` ( C++ API)

```
template < class T, int dim, enum cudaTextureReadMode
readMode > __host__ cudaError_t cudaBindTexture2D
(size_t *offset, const textureTdimreadMode tex, const
void *devPtr, size_t width, size_t height, size_t pitch)
```

[C++ API] Binds a 2D memory area to a texture

#### Parameters

##### **offset**

- Offset in bytes

##### **tex**

- Texture reference to bind

##### **devPtr**

- 2D memory area on device

##### **width**

- Width in texel units

##### **height**

- Height in texel units

##### **pitch**

- Pitch in bytes

#### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidTexture`

#### Description

Binds the 2D memory area pointed to by `devPtr` to the texture reference `tex`. The size of the area is constrained by `width` in texel units, `height` in texel units, and `pitch` in byte units. The channel descriptor is inherited from the texture reference type. Any memory previously bound to `tex` is unbound.



Since the hardware enforces an alignment requirement on texture base addresses, `cudaBindTexture2D()` returns in `*offset` a byte offset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the `tex2D()` function. If the device memory pointer was returned from `cudaMalloc()`, the offset is guaranteed to be 0 and NULL may be passed as the `offset` parameter.



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

See also:

`cudaCreateChannelDesc` ( C++ API), `cudaGetChannelDesc`, `cudaGetTextureReference`, `cudaBindTexture` ( C++ API), `cudaBindTexture` ( C++ API, inherited channel descriptor), `cudaBindTexture2D` ( C API), `cudaBindTexture2D` ( C++ API), `cudaBindTextureToArray` ( C++ API), `cudaBindTextureToArray` ( C++ API, inherited channel descriptor), `cudaUnbindTexture` ( C++ API), `cudaGetTextureAlignmentOffset` ( C++ API)

```
template < class T, int dim, enum cudaTextureReadMode
readMode > __host__ cudaError_t cudaBindTexture2D
(size_t *offset, const textureTdimreadMode tex, const
void *devPtr, const cudaChannelFormatDesc desc, size_t
width, size_t height, size_t pitch)
```

[C++ API] Binds a 2D memory area to a texture

### Parameters

#### **offset**

- Offset in bytes

#### **tex**

- Texture reference to bind

#### **devPtr**

- 2D memory area on device

#### **desc**

- Channel format

**width**

- Width in texel units

**height**

- Height in texel units

**pitch**

- Pitch in bytes

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidTexture](#)

**Description**

Binds the 2D memory area pointed to by `devPtr` to the texture reference `tex`. The size of the area is constrained by `width` in texel units, `height` in texel units, and `pitch` in byte units. `desc` describes how the memory is interpreted when fetching values from the texture. Any memory previously bound to `tex` is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, [cudaBindTexture2D\(\)](#) returns in `*offset` a byte offset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the `tex2D()` function. If the device memory pointer was returned from [cudaMalloc\(\)](#), the offset is guaranteed to be 0 and NULL may be passed as the `offset` parameter.



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

**See also:**

[cudaCreateChannelDesc](#) ( C++ API), [cudaGetChannelDesc](#), [cudaGetTextureReference](#), [cudaBindTexture](#) ( C++ API), [cudaBindTexture](#) ( C++ API, inherited channel descriptor), [cudaBindTexture2D](#) ( C API), [cudaBindTexture2D](#) ( C++ API, inherited channel descriptor), [cudaBindTextureToArray](#) ( C++ API), [cudaBindTextureToArray](#) ( C++ API, inherited channel descriptor), [cudaUnbindTexture](#) ( C++ API), [cudaGetTextureAlignmentOffset](#) ( C++ API)

```
template < class T, int dim, enum
cudaTextureReadMode readMode > __host__ cudaError_t
cudaBindTextureToArray (const textureTdimreadMode
tex, cudaArray_const_t array)
```

[C++ API] Binds an array to a texture

### Parameters

**tex**

- Texture to bind

**array**

- Memory array on device

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidTexture](#)

### Description

Binds the CUDA array `array` to the texture reference `tex`. The channel descriptor is inherited from the CUDA array. Any CUDA array previously bound to `tex` is unbound.



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

### See also:

[cudaCreateChannelDesc](#) ( C++ API), [cudaGetChannelDesc](#), [cudaGetTextureReference](#), [cudaBindTexture](#) ( C++ API), [cudaBindTexture](#) ( C++ API, inherited channel descriptor), [cudaBindTexture2D](#) ( C++ API), [cudaBindTexture2D](#) ( C++ API, inherited channel descriptor), [cudaBindTextureToArray](#) ( C API), [cudaBindTextureToArray](#) ( C++ API), [cudaUnbindTexture](#) ( C++ API), [cudaGetTextureAlignmentOffset](#) ( C++ API)

```
template < class T, int dim, enum
cudaTextureReadMode readMode > __host__ cudaError_t
```

## cudaBindTextureToArray (const textureTdimreadMode tex, cudaArray\_const\_t array, const cudaChannelFormatDesc desc)

[C++ API] Binds an array to a texture

### Parameters

**tex**

- Texture to bind

**array**

- Memory array on device

**desc**

- Channel format

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidTexture](#)

### Description

Binds the CUDA array `array` to the texture reference `tex`. `desc` describes how the memory is interpreted when fetching values from the texture. Any CUDA array previously bound to `tex` is unbound.



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

### See also:

[cudaCreateChannelDesc](#) ( C++ API), [cudaGetChannelDesc](#), [cudaGetTextureReference](#), [cudaBindTexture](#) ( C++ API), [cudaBindTexture](#) ( C++ API, inherited channel descriptor), [cudaBindTexture2D](#) ( C++ API), [cudaBindTexture2D](#) ( C++ API, inherited channel descriptor), [cudaBindTextureToArray](#) ( C API), [cudaBindTextureToArray](#) ( C++ API, inherited channel descriptor), [cudaUnbindTexture](#) ( C++ API), [cudaGetTextureAlignmentOffset](#) ( C++ API)

```
template < class T, int dim, enum
cudaTextureReadMode readMode > __host__ cudaError_t
cudaBindTextureToMipmappedArray
(const textureTdimreadMode tex,
cudaMipmappedArray_const_t mipmappedArray)
```

[C++ API] Binds a mipmapped array to a texture

### Parameters

**tex**

- Texture to bind

**mipmappedArray**

- Memory mipmapped array on device

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidTexture](#)

### Description

Binds the CUDA mipmapped array `mipmappedArray` to the texture reference `tex`. The channel descriptor is inherited from the CUDA array. Any CUDA mipmapped array previously bound to `tex` is unbound.



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

### See also:

[cudaCreateChannelDesc](#) ( C++ API), [cudaGetChannelDesc](#), [cudaGetTextureReference](#), [cudaBindTexture](#) ( C++ API), [cudaBindTexture](#) ( C++ API, inherited channel descriptor), [cudaBindTexture2D](#) ( C++ API), [cudaBindTexture2D](#) ( C++ API, inherited channel descriptor), [cudaBindTextureToArray](#) ( C API), [cudaBindTextureToArray](#) ( C++ API), [cudaUnbindTexture](#) ( C++ API), [cudaGetTextureAlignmentOffset](#) ( C++ API)

```
template < class T, int dim, enum
cudaTextureReadMode readMode > __host__ cudaError_t
cudaBindTextureToMipmappedArray
(const textureTdimreadMode tex,
cudaMipmappedArray_const_t mipmappedArray, const
cudaChannelFormatDesc desc)
```

[C++ API] Binds a mipmapped array to a texture

### Parameters

**tex**

- Texture to bind

**mipmappedArray**

- Memory mipmapped array on device

**desc**

- Channel format

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidTexture](#)

### Description

Binds the CUDA mipmapped array `mipmappedArray` to the texture reference `tex`. `desc` describes how the memory is interpreted when fetching values from the texture. Any CUDA mipmapped array previously bound to `tex` is unbound.



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

### See also:

[cudaCreateChannelDesc](#) ( C++ API), [cudaGetChannelDesc](#), [cudaGetTextureReference](#), [cudaBindTexture](#) ( C++ API), [cudaBindTexture](#) ( C++ API, inherited channel descriptor), [cudaBindTexture2D](#) ( C++ API), [cudaBindTexture2D](#) ( C++ API, inherited

channel descriptor), `cudaBindTextureToArray` ( C API), `cudaBindTextureToArray` ( C++ API, inherited channel descriptor), `cudaUnbindTexture` ( C++ API), `cudaGetTextureAlignmentOffset` ( C++ API)

## `template < class T > __host__ cudaCreateChannelDesc (void)`

[C++ API] Returns a channel descriptor using the specified format

### Returns

Channel descriptor with format `f`

### Description

Returns a channel descriptor with format `f` and number of bits of each component `x`, `y`, `z`, and `w`. The `cudaChannelFormatDesc` is defined as:

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

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

### See also:

`cudaCreateChannelDesc` ( Low level), `cudaGetChannelDesc`, `cudaGetTextureReference`, `cudaBindTexture` ( High level), `cudaBindTexture` ( High level, inherited channel descriptor), `cudaBindTexture2D` ( High level), `cudaBindTextureToArray` ( High level), `cudaBindTextureToArray` ( High level, inherited channel descriptor), `cudaUnbindTexture` ( High level), `cudaGetTextureAlignmentOffset` ( High level)

## `__host__ cudaError_t cudaEventCreate (cudaEvent_t *event, unsigned int flags)`

[C++ API] Creates an event object with the specified flags

### Parameters

#### **event**

- Newly created event

#### **flags**

- Flags for new event

## Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorLaunchFailure`,  
`cudaErrorMemoryAllocation`

## Description

Creates an event object with the specified flags. Valid flags include:

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



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

## See also:

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

```
template < class T > __host__ __cudaError_t
cudaFuncGetAttributes (cudaFuncAttributes *attr, T
*entry)
```

[C++ API] Find out attributes for a given function

## Parameters

**attr**

- Return pointer to function's attributes



**entry**

- Function to get attributes of

**Returns**

`cudaSuccess`, `cudaErrorInvalidDeviceFunction`

**Description**

This function obtains the attributes of a function specified via `entry`. The parameter `entry` must be a pointer to a function that executes on the device. The parameter specified by `entry` must be declared as a `__global__` function. The fetched attributes are placed in `attr`. If the specified function does not exist, then `cudaErrorInvalidDeviceFunction` is returned.

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



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

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

```
template < class T > __host__ cudaError_t
cudaFuncSetAttribute (T *entry, cudaFuncAttribute attr,
int value)
```

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

**Parameters****entry**

- Function to get attributes of

**attr**

- Attribute to set

**value**

- Value to set

## Returns

[cudaSuccess](#), [cudaErrorInvalidDeviceFunction](#), [cudaErrorInvalidValue](#)

## Description

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

Valid values for `attr` are:

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



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

[cudaLaunchKernel](#) ( C++ API), [cudaFuncSetCacheConfig](#) ( C++ API),  
[cudaFuncGetAttributes](#) ( C API), [cudaSetDoubleForDevice](#), [cudaSetDoubleForHost](#)

```
template < class T > __host__ cudaError_t
cudaFuncSetCacheConfig (T *func, cudaFuncCache
cacheConfig)
```

[C++ API] Sets the preferred cache configuration for a device function

### Parameters

#### **func**

- device function pointer

#### **cacheConfig**

- Requested cache configuration

### Returns

[cudaSuccess](#), [cudaErrorInvalidDeviceFunction](#)

### Description

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

`func` must be a pointer to a function that executes on the device. The parameter specified by `func` must be declared as a `__global__` function. If the specified function does not exist, then [cudaErrorInvalidDeviceFunction](#) is returned.

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

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

The supported cache configurations are:

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



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

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

[cudaLaunchKernel](#) ( C++ API), [cudaFuncSetCacheConfig](#) ( C API),  
[cudaFuncGetAttributes](#) ( C++ API), [cudaSetDoubleForDevice](#), [cudaSetDoubleForHost](#),  
[cudaThreadGetCacheConfig](#), [cudaThreadSetCacheConfig](#)

**template < class T > \_\_host\_\_ cudaError\_t  
[cudaGetSymbolAddress](#) (void \*\*devPtr, const T symbol)**

[C++ API] Finds the address associated with a CUDA symbol

### Parameters

#### **devPtr**

- Return device pointer associated with symbol

#### **symbol**

- Device symbol reference

### Returns

[cudaSuccess](#), [cudaErrorInvalidSymbol](#), [cudaErrorNoKernelImageForDevice](#)

### Description

Returns in \*devPtr the address of symbol `symbol` on the device. `symbol` can either be a variable that resides in global or constant memory space. If `symbol` cannot be found, or if `symbol` is not declared in the global or constant memory space, \*devPtr is unchanged and the error [cudaErrorInvalidSymbol](#) is returned.



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

### See also:

[cudaGetSymbolAddress](#) ( C API), [cudaGetSymbolSize](#) ( C++ API)

```
template < class T > __host__ cudaError_t
cudaGetSymbolSize (size_t *size, const T symbol)
```

[C++ API] Finds the size of the object associated with a CUDA symbol

### Parameters

#### **size**

- Size of object associated with symbol

#### **symbol**

- Device symbol reference

### Returns

[cudaSuccess](#), [cudaErrorInvalidSymbol](#), [cudaErrorNoKernelImageForDevice](#)

### Description

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



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

### See also:

[cudaGetSymbolAddress](#) ( C++ API), [cudaGetSymbolSize](#) ( C API)

```
template < class T, int dim, enum
cudaTextureReadMode readMode > __host__ cudaError_t
```

## cudaGetTextureAlignmentOffset (size\_t \*offset, const textureTdimreadMode tex)

[C++ API] Get the alignment offset of a texture

### Parameters

#### offset

- Offset of texture reference in bytes

#### tex

- Texture to get offset of

### Returns

cudaSuccess, cudaErrorInvalidTexture, cudaErrorInvalidTextureBinding

### Description

Returns in \*offset the offset that was returned when texture reference tex was bound.



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

### See also:

[cudaCreateChannelDesc](#) ( C++ API), [cudaGetChannelDesc](#), [cudaGetTextureReference](#), [cudaBindTexture](#) ( C++ API), [cudaBindTexture](#) ( C++ API, inherited channel descriptor), [cudaBindTexture2D](#) ( C++ API), [cudaBindTexture2D](#) ( C++ API, inherited channel descriptor), [cudaBindTextureToArray](#) ( C++ API), [cudaBindTextureToArray](#) ( C++ API, inherited channel descriptor), [cudaUnbindTexture](#) ( C++ API), [cudaGetTextureAlignmentOffset](#) ( C API)

template < class T > \_\_host\_\_ cudaError\_t  
cudaLaunchCooperativeKernel (const T \*func, dim3

## `gridDim, dim3 blockDim, void **args, size_t sharedMem, cudaStream_t stream)`

Launches a device function.

### Parameters

#### **func**

- Device function symbol

#### **gridDim**

- Grid dimensions

#### **blockDim**

- Block dimensions

#### **args**

- Arguments

#### **sharedMem**

- Shared memory (defaults to 0)

#### **stream**

- Stream identifier (defaults to NULL)

### Returns

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

### Description

The function invokes kernel `func` on `gridDim` (`gridDim.x × gridDim.y × gridDim.z`) grid of blocks. Each block contains `blockDim` (`blockDim.x × blockDim.y × blockDim.z`) threads.

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

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

The kernel cannot make use of CUDA dynamic parallelism.

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

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

`stream` specifies a stream the invocation is associated to.



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

[cudaLaunchCooperativeKernel](#) ( C API)

```
template < class T > __host__ cudaError_t
cudaLaunchKernel (const T *func, dim3 gridDim, dim3
blockDim, void **args, size_t sharedMem, cudaStream_t
stream)
```

Launches a device function.

### Parameters

#### **func**

- Device function symbol

#### **gridDim**

- Grid dimensions

#### **blockDim**

- Block dimensions

#### **args**

- Arguments

#### **sharedMem**

- Shared memory (defaults to 0)

#### **stream**

- Stream identifier (defaults to NULL)

### Returns

[cudaSuccess](#), [cudaErrorInvalidDeviceFunction](#), [cudaErrorInvalidConfiguration](#),  
[cudaErrorLaunchFailure](#), [cudaErrorLaunchTimeout](#), [cudaErrorLaunchOutOfResources](#),  
[cudaErrorSharedObjectInitFailed](#), [cudaErrorInvalidPtx](#),  
[cudaErrorNoKernelImageForDevice](#), [cudaErrorJitCompilerNotFound](#)



## Description

The function invokes kernel `func` on `gridDim` (`gridDim.x × gridDim.y × gridDim.z`) grid of blocks. Each block contains `blockDim` (`blockDim.x × blockDim.y × blockDim.z`) threads.

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

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

`stream` specifies a stream the invocation is associated to.



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

[cudaLaunchKernel](#) ( C API)

## `__host__ cudaError_t cudaMallocHost (void **ptr, size_t size, unsigned int flags)`

[C++ API] Allocates page-locked memory on the host

### Parameters

**ptr**

- Device pointer to allocated memory

**size**

- Requested allocation size in bytes

**flags**

- Requested properties of allocated memory

### Returns

[cudaSuccess](#), [cudaErrorMemoryAllocation](#)

## Description

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

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

- ▶ `cudaHostAllocDefault`: This flag's value is defined to be 0.
- ▶ `cudaHostAllocPortable`: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
- ▶ `cudaHostAllocMapped`: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling `cudaHostGetDevicePointer()`.
- ▶ `cudaHostAllocWriteCombined`: Allocates the memory as write-combined (WC). WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs. WC memory is a good option for buffers that will be written by the CPU and read by the device via mapped pinned memory or host->device transfers.

All of these flags are orthogonal to one another: a developer may allocate memory that is portable, mapped and/or write-combined with no restrictions.

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

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

Memory allocated by this function must be freed with `cudaFreeHost()`.



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

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

See also:

`cudaSetDeviceFlags`, `cudaMallocHost` ( C API), `cudaFreeHost`, `cudaHostAlloc`

```
template < class T > __host__ cudaError_t
cudaMallocManaged (T **devPtr, size_t size, unsigned int
flags)
```

Allocates memory that will be automatically managed by the Unified Memory system.

### Parameters

#### **devPtr**

- Pointer to allocated device memory

#### **size**

- Requested allocation size in bytes

#### **flags**

- Must be either `cudaMemAttachGlobal` or `cudaMemAttachHost` (defaults to `cudaMemAttachGlobal`)

### Returns

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

### Description

Allocates `size` bytes of managed memory on the device and returns in `*devPtr` a pointer to the allocated memory. If the device doesn't support allocating managed memory, `cudaErrorNotSupported` is returned. Support for managed memory can be queried using the device attribute `cudaDevAttrManagedMemory`. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. If `size` is 0, `cudaMallocManaged` returns `cudaErrorInvalidValue`. The pointer is valid on the CPU and on all GPUs in the system that support managed memory. All accesses to this pointer must obey the Unified Memory programming model.

`flags` specifies the default stream association for this allocation. `flags` must be one of `cudaMemAttachGlobal` or `cudaMemAttachHost`. The default value for `flags` is `cudaMemAttachGlobal`. If `cudaMemAttachGlobal` is specified, then this memory is accessible from any stream on any device. If `cudaMemAttachHost` is specified, then the allocation should not be accessed from devices that have a zero value for

the device attribute `cudaDevAttrConcurrentManagedAccess`; an explicit call to `cudaStreamAttachMemAsync` will be required to enable access on such devices.

If the association is later changed via `cudaStreamAttachMemAsync` to a single stream, the default association, as specified during `cudaMallocManaged`, is restored when that stream is destroyed. For `__managed__` variables, the default association is always `cudaMemAttachGlobal`. Note that destroying a stream is an asynchronous operation, and as a result, the change to default association won't happen until all work in the stream has completed.

Memory allocated with `cudaMallocManaged` should be released with `cudaFree`.

Device memory oversubscription is possible for GPUs that have a non-zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`. Managed memory on such GPUs may be evicted from device memory to host memory at any time by the Unified Memory driver in order to make room for other allocations.

In a multi-GPU system where all GPUs have a non-zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`, managed memory may not be populated when this API returns and instead may be populated on access. In such systems, managed memory can migrate to any processor's memory at any time. The Unified Memory driver will employ heuristics to maintain data locality and prevent excessive page faults to the extent possible. The application can also guide the driver about memory usage patterns via `cudaMemAdvise`. The application can also explicitly migrate memory to a desired processor's memory via `cudaMemPrefetchAsync`.

In a multi-GPU system where all of the GPUs have a zero value for the device attribute `cudaDevAttrConcurrentManagedAccess` and all the GPUs have peer-to-peer support with each other, the physical storage for managed memory is created on the GPU which is active at the time `cudaMallocManaged` is called. All other GPUs will reference the data at reduced bandwidth via peer mappings over the PCIe bus. The Unified Memory driver does not migrate memory among such GPUs.

In a multi-GPU system where not all GPUs have peer-to-peer support with each other and where the value of the device attribute `cudaDevAttrConcurrentManagedAccess` is zero for at least one of those GPUs, the location chosen for physical storage of managed memory is system-dependent.

- ▶ On Linux, the location chosen will be device memory as long as the current set of active contexts are on devices that either have peer-to-peer support with each other or have a non-zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`. If there is an active context on a GPU that does not have a non-zero value for that device attribute and it does not have peer-to-peer support with the other devices that have active contexts on them, then the location for physical storage will be 'zero-copy' or host memory. Note that this means that managed memory that is located in device memory is migrated to host memory if a new context is created on a GPU that doesn't have a non-zero value for

the device attribute and does not support peer-to-peer with at least one of the other devices that has an active context. This in turn implies that context creation may fail if there is insufficient host memory to migrate all managed allocations.

- ▶ On Windows, the physical storage is always created in 'zero-copy' or host memory. All GPUs will reference the data at reduced bandwidth over the PCIe bus. In these circumstances, use of the environment variable `CUDA_VISIBLE_DEVICES` is recommended to restrict CUDA to only use those GPUs that have peer-to-peer support. Alternatively, users can also set `CUDA_MANAGED_FORCE_DEVICE_ALLOC` to a non-zero value to force the driver to always use device memory for physical storage. When this environment variable is set to a non-zero value, all devices used in that process that support managed memory have to be peer-to-peer compatible with each other. The error `cudaErrorInvalidDevice` will be returned if a device that supports managed memory is used and it is not peer-to-peer compatible with any of the other managed memory supporting devices that were previously used in that process, even if `cudaDeviceReset` has been called on those devices. These environment variables are described in the CUDA programming guide under the "CUDA environment variables" section.
- ▶ On ARM, managed memory is not available on discrete gpu with Drive PX-2.



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

#### See also:

`cudaMallocPitch`, `cudaFree`, `cudaMallocArray`, `cudaFreeArray`, `cudaMalloc3D`, `cudaMalloc3DArray`, `cudaMallocHost` ( C API), `cudaFreeHost`, `cudaHostAlloc`, `cudaDeviceGetAttribute`, `cudaStreamAttachMemAsync`

```
template < class T > __host__ cudaError_t
cudaMemcpyFromSymbol (void *dst, const T symbol,
size_t count, size_t offset, cudaMemcpyKind kind)
```

[C++ API] Copies data from the given symbol on the device

### Parameters

#### **dst**

- Destination memory address

#### **symbol**

- Device symbol reference

#### **count**

- Size in bytes to copy

#### **offset**

- Offset from start of symbol in bytes

#### **kind**

- Type of transfer

### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidSymbol](#),  
[cudaErrorInvalidMemcpyDirection](#), [cudaErrorNoKernelImageForDevice](#)

### Description

Copies `count` bytes from the memory area `offset` bytes from the start of symbol `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either [cudaMemcpyDeviceToHost](#) or [cudaMemcpyDeviceToDevice](#).



- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits [synchronous](#) behavior for most use cases.
- Use of a string naming a variable as the `symbol` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.
- Note that this function may also return [cudaErrorInitializationError](#), [cudaErrorInsufficientDriver](#) or [cudaErrorNoDevice](#) if this call tries to initialize internal CUDA RT state.
- Note that as specified by [cudaStreamAddCallback](#) no CUDA function may be called from callback. [cudaErrorNotPermitted](#) may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpy2DToArray](#),  
[cudaMemcpy2DFromArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#),  
[cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpy2DToArrayAsync](#),  
[cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#),  
[cudaMemcpyFromSymbolAsync](#)

```
template < class T > __host__ cudaError_t
cudaMemcpyFromSymbolAsync (void *dst, const T
symbol, size_t count, size_t offset, cudaMemcpyKind
kind, cudaStream_t stream)
```

[C++ API] Copies data from the given symbol on the device

**Parameters****dst**

- Destination memory address

**symbol**

- Device symbol reference

**count**

- Size in bytes to copy

**offset**

- Offset from start of symbol in bytes

**kind**

- Type of transfer

**stream**

- Stream identifier

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidSymbol](#),  
[cudaErrorInvalidMemcpyDirection](#), [cudaErrorNoKernelImageForDevice](#)

**Description**

Copies `count` bytes from the memory area `offset` bytes from the start of `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either [cudaMemcpyDeviceToHost](#) or [cudaMemcpyDeviceToDevice](#).

[cudaMemcpyFromSymbolAsync\(\)](#) is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is [cudaMemcpyDeviceToHost](#) and `stream` is non-zero, the copy may overlap with operations in other streams.



- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ This function exhibits [asynchronous](#) behavior for most use cases.
- ▶ Use of a string naming a variable as the `symbol` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.
- ▶ Note that this function may also return [cudaErrorInitializationError](#), [cudaErrorInsufficientDriver](#) or [cudaErrorNoDevice](#) if this call tries to initialize internal CUDA RT state.
- ▶ Note that as specified by [cudaStreamAddCallback](#) no CUDA function may be called from callback. [cudaErrorNotPermitted](#) may, but is not guaranteed to, be returned as a diagnostic in such case.

#### See also:

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpy2DToArray](#), [cudaMemcpy2DFromArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#)

```
template < class T > __host__ cudaError_t
cudaMemcpyToSymbol (const T symbol, const void *src,
size_t count, size_t offset, cudaMemcpyKind kind)
```

[C++ API] Copies data to the given symbol on the device

#### Parameters

##### **symbol**

- Device symbol reference

##### **src**

- Source memory address

##### **count**

- Size in bytes to copy

##### **offset**

- Offset from start of symbol in bytes

##### **kind**

- Type of transfer

#### Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidSymbol](#),  
[cudaErrorInvalidMemcpyDirection](#), [cudaErrorNoKernelImageForDevice](#)



## Description

Copies `count` bytes from the memory area pointed to by `src` to the memory area `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToDevice`.



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

## See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`

```
template < class T > __host__ cudaError_t
cudaMemcpyToSymbolAsync (const T symbol, const void
*src, size_t count, size_t offset, cudaMemcpyKind kind,
cudaStream_t stream)
```

[C++ API] Copies data to the given symbol on the device

## Parameters

### `symbol`

- Device symbol reference

### `src`

- Source memory address

### `count`

- Size in bytes to copy

**offset**

- Offset from start of symbol in bytes

**kind**

- Type of transfer

**stream**

- Stream identifier

**Returns**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidSymbol](#),  
[cudaErrorInvalidMemcpyDirection](#), [cudaErrorNoKernelImageForDevice](#)

**Description**

Copies `count` bytes from the memory area pointed to by `src` to the memory area `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either [cudaMemcpyHostToDevice](#) or [cudaMemcpyDeviceToDevice](#).

[cudaMemcpyToSymbolAsync\(\)](#) is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is [cudaMemcpyHostToDevice](#) and `stream` is non-zero, the copy may overlap with operations in other streams.



- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ This function exhibits [asynchronous](#) behavior for most use cases.
- ▶ Use of a string naming a variable as the `symbol` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.
- ▶ Note that this function may also return [cudaErrorInitializationError](#), [cudaErrorInsufficientDriver](#) or [cudaErrorNoDevice](#) if this call tries to initialize internal CUDA RT state.
- ▶ Note that as specified by [cudaStreamAddCallback](#) no CUDA function may be called from callback. [cudaErrorNotPermitted](#) may, but is not guaranteed to, be returned as a diagnostic in such case.

**See also:**

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpy2DToArray](#), [cudaMemcpy2DFromArray](#),  
[cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#),  
[cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpy2DToArrayAsync](#),  
[cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyFromSymbolAsync](#)

```
template < class T > __host__ cudaError_t
cudaOccupancyMaxActiveBlocksPerMultiprocessor
(int *numBlocks, T func, int blockSize, size_t
dynamicSMemSize)
```

Returns occupancy for a device function.

### Parameters

#### **numBlocks**

- Returned occupancy

#### **func**

- Kernel function for which occupancy is calculated

#### **blockSize**

- Block size the kernel is intended to be launched with

#### **dynamicSMemSize**

- Per-block dynamic shared memory usage intended, in bytes

### Returns

[cudaSuccess](#), [cudaErrorInvalidDevice](#), [cudaErrorInvalidDeviceFunction](#),  
[cudaErrorInvalidValue](#), [cudaErrorUnknown](#),

### Description

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



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

### See also:

[cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags](#)

[cudaOccupancyMaxPotentialBlockSize](#)

[cudaOccupancyMaxPotentialBlockSizeWithFlags](#)

`cudaOccupancyMaxPotentialBlockSizeVariableSMem`

`cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags`

```
template < class T > __host__ cudaError_t
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
(int *numBlocks, T func, int blockSize, size_t
dynamicSMemSize, unsigned int flags)
```

Returns occupancy for a device function with the specified flags.

### Parameters

#### **numBlocks**

- Returned occupancy

#### **func**

- Kernel function for which occupancy is calculated

#### **blockSize**

- Block size the kernel is intended to be launched with

#### **dynamicSMemSize**

- Per-block dynamic shared memory usage intended, in bytes

#### **flags**

- Requested behavior for the occupancy calculator

### Returns

`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorInvalidDeviceFunction`,  
`cudaErrorInvalidValue`, `cudaErrorUnknown`,

### Description

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

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

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



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

### See also:

`cudaOccupancyMaxActiveBlocksPerMultiprocessor`

`cudaOccupancyMaxPotentialBlockSize`

`cudaOccupancyMaxPotentialBlockSizeWithFlags`

`cudaOccupancyMaxPotentialBlockSizeVariableSMem`

`cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags`

```
template < class T > __host__ cudaError_t
cudaOccupancyMaxPotentialBlockSize (int *minGridSize,
int *blockSize, T func, size_t dynamicSMemSize, int
blockSizeLimit)
```

Returns grid and block size that achieves maximum potential occupancy for a device function.

### Parameters

#### **minGridSize**

- Returned minimum grid size needed to achieve the best potential occupancy

#### **blockSize**

- Returned block size

#### **func**

- Device function symbol

#### **dynamicSMemSize**

- Per-block dynamic shared memory usage intended, in bytes

#### **blockSizeLimit**

- The maximum block size `func` is designed to work with. 0 means no limit.

**Returns**

[cudaSuccess](#), [cudaErrorInvalidDevice](#), [cudaErrorInvalidDeviceFunction](#),  
[cudaErrorInvalidValue](#), [cudaErrorUnknown](#),

**Description**

Returns in `*minGridSize` and `*blocksize` a suggested grid / block size pair that achieves the best potential occupancy (i.e. the maximum number of active warps with the smallest number of blocks).

Use

**See also:**

[cudaOccupancyMaxPotentialBlockSizeVariableSMem](#) if the amount of per-block dynamic shared memory changes with different block sizes.



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

**See also:**

[cudaOccupancyMaxPotentialBlockSizeWithFlags](#)

[cudaOccupancyMaxActiveBlocksPerMultiprocessor](#)

[cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags](#)

[cudaOccupancyMaxPotentialBlockSizeVariableSMem](#)

[cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags](#)

```
template < typename UnaryFunction, class T >
__host__ __cudaError_t
cudaOccupancyMaxPotentialBlockSizeVariableSMem (int
```

## **\*minGridSize, int \*blockSize, T func, UnaryFunction blockSizeToDynamicSMemSize, int blockSizeLimit)**

Returns grid and block size that achieves maximum potential occupancy for a device function.

### **Parameters**

#### **minGridSize**

- Returned minimum grid size needed to achieve the best potential occupancy

#### **blockSize**

- Returned block size

#### **func**

- Device function symbol

#### **blockSizeToDynamicSMemSize**

- A unary function / functor that takes block size, and returns the size, in bytes, of dynamic shared memory needed for a block

#### **blockSizeLimit**

- The maximum block size `func` is designed to work with. 0 means no limit.

### **Returns**

[cudaSuccess](#), [cudaErrorInvalidDevice](#), [cudaErrorInvalidDeviceFunction](#),  
[cudaErrorInvalidValue](#), [cudaErrorUnknown](#),

### **Description**

Returns in `*minGridSize` and `*blocksize` a suggested grid / block size pair that achieves the best potential occupancy (i.e. the maximum number of active warps with the smallest number of blocks).



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

### **See also:**

[cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags](#)

[cudaOccupancyMaxActiveBlocksPerMultiprocessor](#)

`cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags`

`cudaOccupancyMaxPotentialBlockSize`

`cudaOccupancyMaxPotentialBlockSizeWithFlags`

```
template < typename UnaryFunction, class T >
__host__ cudaError_t
cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags
(int *minGridSize, int *blockSize, T func, UnaryFunction
blockSizeToDynamicSMemSize, int blockSizeLimit,
unsigned int flags)
```

Returns grid and block size that achieves maximum potential occupancy for a device function.

### Parameters

#### **minGridSize**

- Returned minimum grid size needed to achieve the best potential occupancy

#### **blockSize**

- Returned block size

#### **func**

- Device function symbol

#### **blockSizeToDynamicSMemSize**

- A unary function / functor that takes block size, and returns the size, in bytes, of dynamic shared memory needed for a block

#### **blockSizeLimit**

- The maximum block size `func` is designed to work with. 0 means no limit.

#### **flags**

- Requested behavior for the occupancy calculator

### Returns

`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorInvalidDeviceFunction`,  
`cudaErrorInvalidValue`, `cudaErrorUnknown`,

### Description

Returns in `*minGridSize` and `*blocksize` a suggested grid / block size pair that achieves the best potential occupancy (i.e. the maximum number of active warps with the smallest number of blocks).

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



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



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

#### See also:

`cudaOccupancyMaxPotentialBlockSizeVariableSMem`

`cudaOccupancyMaxActiveBlocksPerMultiprocessor`

`cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags`

`cudaOccupancyMaxPotentialBlockSize`

`cudaOccupancyMaxPotentialBlockSizeWithFlags`

```
template < class T > __host__ cudaError_t
cudaOccupancyMaxPotentialBlockSizeWithFlags
(int *minGridSize, int *blockSize, T func, size_t
dynamicSMemSize, int blockSizeLimit, unsigned int
flags)
```

Returns grid and block size that achieved maximum potential occupancy for a device function with the specified flags.

#### Parameters

##### **minGridSize**

- Returned minimum grid size needed to achieve the best potential occupancy

**blockSize**

- Returned block size

**func**

- Device function symbol

**dynamicSMemSize**

- Per-block dynamic shared memory usage intended, in bytes

**blockSizeLimit**

- The maximum block size `func` is designed to work with. 0 means no limit.

**flags**

- Requested behavior for the occupancy calculator

**Returns**

[cudaSuccess](#), [cudaErrorInvalidDevice](#), [cudaErrorInvalidDeviceFunction](#),  
[cudaErrorInvalidValue](#), [cudaErrorUnknown](#),

**Description**

Returns in `*minGridSize` and `*blocksize` a suggested grid / block size pair that achieves the best potential occupancy (i.e. the maximum number of active warps with the smallest number of blocks).

The `flags` parameter controls how special cases are handle. Valid flags include:

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

Use

**See also:**

[cudaOccupancyMaxPotentialBlockSizeVariableSMem](#) if the amount of per-block dynamic shared memory changes with different block sizes.



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

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

#### See also:

`cudaOccupancyMaxPotentialBlockSize`

`cudaOccupancyMaxActiveBlocksPerMultiprocessor`

`cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags`

`cudaOccupancyMaxPotentialBlockSizeVariableSMem`

`cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags`

```
template < class T > __host__ cudaError_t
cudaStreamAttachMemAsync (cudaStream_t stream, T
*devPtr, size_t length, unsigned int flags)
```

Attach memory to a stream asynchronously.

#### Parameters

##### **stream**

- Stream in which to enqueue the attach operation

##### **devPtr**

- Pointer to memory (must be a pointer to managed memory or to a valid host-accessible region of system-allocated memory)

##### **length**

- Length of memory (defaults to zero)

##### **flags**

- Must be one of `cudaMemAttachGlobal`, `cudaMemAttachHost` or `cudaMemAttachSingle` (defaults to `cudaMemAttachSingle`)

#### Returns

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

#### Description

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

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

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

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

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

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

When memory is associated with a single stream, the Unified Memory system will allow CPU access to this memory region so long as all operations in `stream` have completed, regardless of whether other streams are active. In effect, this constrains exclusive ownership of the managed memory region by an active GPU to per-stream activity instead of whole-GPU activity.

Accessing memory on the device from streams that are not associated with it will produce undefined results. No error checking is performed by the Unified Memory system to ensure that kernels launched into other streams do not access this region.

It is a program's responsibility to order calls to `cudaStreamAttachMemAsync` via events, synchronization or other means to ensure legal access to memory at all times. Data visibility and coherency will be changed appropriately for all kernels which follow a stream-association change.

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



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

### See also:

`cudaStreamCreate`, `cudaStreamCreateWithFlags`, `cudaStreamWaitEvent`, `cudaStreamSynchronize`, `cudaStreamAddCallback`, `cudaStreamDestroy`, `cudaMallocManaged`

```
template < class T, int dim, enum cudaTextureReadMode
readMode > __host__ cudaError_t cudaUnbindTexture
(const textureTdimreadMode tex)
```

[C++ API] Unbinds a texture

### Parameters

**tex**

- Texture to unbind

### Returns

`cudaSuccess`, `cudaErrorInvalidTexture`

### Description

Unbinds the texture bound to `tex`. If `texref` is not currently bound, no operation is performed.



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

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

#### See also:

`cudaCreateChannelDesc` ( C++ API), `cudaGetChannelDesc`, `cudaGetTextureReference`, `cudaBindTexture` ( C++ API), `cudaBindTexture` ( C++ API, inherited channel descriptor), `cudaBindTexture2D` ( C++ API), `cudaBindTexture2D` ( C++ API, inherited channel descriptor), `cudaBindTextureToArray` ( C++ API), `cudaBindTextureToArray` ( C++ API, inherited channel descriptor), `cudaUnbindTexture` ( C API), `cudaGetTextureAlignmentOffset` ( C++ API)

## 5.31. Interactions with the CUDA Driver API

This section describes the interactions between the CUDA Driver API and the CUDA Runtime API

### Primary Contexts

There exists a one to one relationship between CUDA devices in the CUDA Runtime API and `CUcontext`s in the CUDA Driver API within a process. The specific context which the CUDA Runtime API uses for a device is called the device's primary context. From the perspective of the CUDA Runtime API, a device and its primary context are synonymous.

### Initialization and Tear-Down

CUDA Runtime API calls operate on the CUDA Driver API `CUcontext` which is current to to the calling host thread.

The function `cudaSetDevice()` makes the primary context for the specified device current to the calling thread by calling `cuCtxSetCurrent()`.

The CUDA Runtime API will automatically initialize the primary context for a device at the first CUDA Runtime API call which requires an active context. If no `CUcontext` is current to the calling thread when a CUDA Runtime API call which requires an active context is made, then the primary context for a device will be selected, made current to the calling thread, and initialized.

The context which the CUDA Runtime API initializes will be initialized using the parameters specified by the CUDA Runtime API functions `cudaSetDeviceFlags()`, `cudaD3D9SetDirect3DDevice()`, `cudaD3D10SetDirect3DDevice()`, `cudaD3D11SetDirect3DDevice()`, `cudaGLSetGLDevice()`, and `cudaVDPAUSetVDPAUDevice()`. Note that these functions will fail with `cudaErrorSetOnActiveProcess` if they are called when the primary context for the

specified device has already been initialized. (or if the current device has already been initialized, in the case of `cudaSetDeviceFlags()`).

Primary contexts will remain active until they are explicitly deinitialized using `cudaDeviceReset()`. The function `cudaDeviceReset()` will deinitialize the primary context for the calling thread's current device immediately. The context will remain current to all of the threads that it was current to. The next CUDA Runtime API call on any thread which requires an active context will trigger the reinitialization of that device's primary context.

Note that there is no reference counting of the primary context's lifetime. It is recommended that the primary context not be deinitialized except just before exit or to recover from an unspecified launch failure.

### Context Interoperability

Note that the use of multiple `CUcontext`s per device within a single process will substantially degrade performance and is strongly discouraged. Instead, it is highly recommended that the implicit one-to-one device-to-context mapping for the process provided by the CUDA Runtime API be used.

If a non-primary `CUcontext` created by the CUDA Driver API is current to a thread then the CUDA Runtime API calls to that thread will operate on that `CUcontext`, with some exceptions listed below. Interoperability between data types is discussed in the following sections.

The function `cudaPointerGetAttributes()` will return the error `cudaErrorIncompatibleDriverContext` if the pointer being queried was allocated by a non-primary context. The function `cudaDeviceEnablePeerAccess()` and the rest of the peer access API may not be called when a non-primary `CUcontext` is current. To use the pointer query and peer access APIs with a context created using the CUDA Driver API, it is necessary that the CUDA Driver API be used to access these features.

All CUDA Runtime API state (e.g, global variables' addresses and values) travels with its underlying `CUcontext`. In particular, if a `CUcontext` is moved from one thread to another then all CUDA Runtime API state will move to that thread as well.

Please note that attaching to legacy contexts (those with a version of 3010 as returned by `cuCtxGetApiVersion()`) is not possible. The CUDA Runtime will return `cudaErrorIncompatibleDriverContext` in such cases.

### Interactions between `CUstream` and `cudaStream_t`

The types `CUstream` and `cudaStream_t` are identical and may be used interchangeably.

### Interactions between `CUevent` and `cudaEvent_t`

The types `CUevent` and `cudaEvent_t` are identical and may be used interchangeably.

### Interactions between `CUarray` and `cudaArray_t`



The types `CUarray` and `struct cudaArray *` represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a `CUarray` in a CUDA Runtime API function which takes a `struct cudaArray *`, it is necessary to explicitly cast the `CUarray` to a `struct cudaArray *`.

In order to use a `struct cudaArray *` in a CUDA Driver API function which takes a `CUarray`, it is necessary to explicitly cast the `struct cudaArray *` to a `CUarray`.

### Interactions between `CUgraphicsResource` and `cudaGraphicsResource_t`

The types `CUgraphicsResource` and `cudaGraphicsResource_t` represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a `CUgraphicsResource` in a CUDA Runtime API function which takes a `cudaGraphicsResource_t`, it is necessary to explicitly cast the `CUgraphicsResource` to a `cudaGraphicsResource_t`.

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

## 5.32. Profiler Control

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

**`__host__ cudaError_t cudaProfilerInitialize (const char *configFile, const char *outputFile, cudaOutputMode_t outputMode)`**

Initialize the CUDA profiler.

### Parameters

#### **`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 `cudaKeyValuePair` OR `cudaCSV`.

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorProfilerDisabled`



## Description

Using this API user can initialize the CUDA profiler by specifying the configuration file, output file and output file format. This API is generally used to profile different set of counters by looping the kernel launch. The `configFile` parameter can be used to select profiling options including profiler counters. Refer to the "Compute Command Line Profiler User Guide" for supported profiler options and counters.

Limitation: The CUDA profiler cannot be initialized with this API if another profiling tool is already active, as indicated by the `cudaErrorProfilerDisabled` return code.

Typical usage of the profiling APIs is as follows:

```
for each set of counters/options { cudaProfilerInitialize(); //Initialize profiling,set
the counters/options in the config file ... cudaProfilerStart(); // code to be profiled
cudaProfilerStop(); ... cudaProfilerStart(); // code to be profiled cudaProfilerStop(); ... }
```



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

## See also:

`cudaProfilerStart`, `cudaProfilerStop`, `cuProfilerInitialize`

## `__host__ cudaError_t cudaProfilerStart (void)`

Enable profiling.

## Returns

`cudaSuccess`

## Description

Enables profile collection by the active profiling tool for the current context. If profiling is already enabled, then `cudaProfilerStart()` has no effect.

`cudaProfilerStart` and `cudaProfilerStop` APIs are used to programmatically control the profiling granularity by allowing profiling to be done only on selective pieces of code.



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

## See also:

`cudaProfilerInitialize`, `cudaProfilerStop`, `cuProfilerStart`

## `__host__ cudaError_t cudaProfilerStop (void)`

Disable profiling.

### Returns

`cudaSuccess`

### Description

Disables profile collection by the active profiling tool for the current context. If profiling is already disabled, then `cudaProfilerStop()` has no effect.

`cudaProfilerStart` and `cudaProfilerStop` APIs are used to programmatically control the profiling granularity by allowing profiling to be done only on selective pieces of code.



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

### See also:

`cudaProfilerInitialize`, `cudaProfilerStart`, `cuProfilerStop`

## 5.33. Data types used by CUDA Runtime

struct cudaChannelFormatDesc  
struct cudaDeviceProp  
struct cudaEglFrame  
struct cudaEglPlaneDesc  
struct cudaExtent  
struct cudaExternalMemoryBufferDesc  
struct cudaExternalMemoryHandleDesc  
struct cudaExternalMemoryMipmappedArrayDesc  
struct cudaExternalSemaphoreHandleDesc  
struct cudaExternalSemaphoreSignalParams  
struct cudaExternalSemaphoreWaitParams  
struct cudaFuncAttributes  
struct cudaHostNodeParams  
struct cudalpcEventHandle\_t  
struct cudalpcMemHandle\_t  
struct cudaKernelNodeParams  
struct cudaLaunchParams  
struct cudaMemcpy3DParms

`struct cudaMemcpy3DPeerParms`

`struct cudaMemcpyParams`

`struct cudaPitchedPtr`

`struct cudaPointerAttributes`

`struct cudaPos`

`struct cudaResourceDesc`

`struct cudaResourceViewDesc`

`struct cudaTextureDesc`

`struct CUuuid_st`

`struct surfaceReference`

`struct textureReference`

`enum cudaCGScope`

CUDA cooperative group scope

#### Values

**`cudaCGScopeInvalid = 0`**

Invalid cooperative group scope

**`cudaCGScopeGrid = 1`**

Scope represented by a `grid_group`

**`cudaCGScopeMultiGrid = 2`**

Scope represented by a `multi_grid_group`

`enum cudaChannelFormatKind`

Channel format kind

## Values

**cudaChannelFormatKindSigned = 0**

Signed channel format

**cudaChannelFormatKindUnsigned = 1**

Unsigned channel format

**cudaChannelFormatKindFloat = 2**

Float channel format

**cudaChannelFormatKindNone = 3**

No channel format

## enum cudaComputeMode

CUDA device compute modes

### Values

**cudaComputeModeDefault = 0**

Default compute mode (Multiple threads can use [cudaSetDevice\(\)](#) with this device)

**cudaComputeModeExclusive = 1**

Compute-exclusive-thread mode (Only one thread in one process will be able to use [cudaSetDevice\(\)](#) with this device)

**cudaComputeModeProhibited = 2**

Compute-prohibited mode (No threads can use [cudaSetDevice\(\)](#) with this device)

**cudaComputeModeExclusiveProcess = 3**

Compute-exclusive-process mode (Many threads in one process will be able to use [cudaSetDevice\(\)](#) with this device)

## enum cudaDeviceAttr

CUDA device attributes

### Values

**cudaDevAttrMaxThreadsPerBlock = 1**

Maximum number of threads per block

**cudaDevAttrMaxBlockDimX = 2**

Maximum block dimension X

**cudaDevAttrMaxBlockDimY = 3**

Maximum block dimension Y

**cudaDevAttrMaxBlockDimZ = 4**

Maximum block dimension Z

**cudaDevAttrMaxGridDimX = 5**

Maximum grid dimension X

**cudaDevAttrMaxGridDimY = 6**

Maximum grid dimension Y

**cudaDevAttrMaxGridDimZ = 7**

Maximum grid dimension Z

**cudaDevAttrMaxSharedMemoryPerBlock = 8**

Maximum shared memory available per block in bytes

**cudaDevAttrTotalConstantMemory = 9**

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

**cudaDevAttrWarpSize = 10**

Warp size in threads

**cudaDevAttrMaxPitch = 11**

Maximum pitch in bytes allowed by memory copies

**cudaDevAttrMaxRegistersPerBlock = 12**

Maximum number of 32-bit registers available per block

**cudaDevAttrClockRate = 13**

Peak clock frequency in kilohertz

**cudaDevAttrTextureAlignment = 14**

Alignment requirement for textures

**cudaDevAttrGpuOverlap = 15**

Device can possibly copy memory and execute a kernel concurrently

**cudaDevAttrMultiProcessorCount = 16**

Number of multiprocessors on device

**cudaDevAttrKernelExecTimeout = 17**

Specifies whether there is a run time limit on kernels

**cudaDevAttrIntegrated = 18**

Device is integrated with host memory

**cudaDevAttrCanMapHostMemory = 19**

Device can map host memory into CUDA address space

**cudaDevAttrComputeMode = 20**

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

**cudaDevAttrMaxTexture1DWidth = 21**

Maximum 1D texture width

**cudaDevAttrMaxTexture2DWidth = 22**

Maximum 2D texture width

**cudaDevAttrMaxTexture2DHeight = 23**

Maximum 2D texture height

**cudaDevAttrMaxTexture3DWidth = 24**

Maximum 3D texture width

**cudaDevAttrMaxTexture3DHeight = 25**

Maximum 3D texture height

**cudaDevAttrMaxTexture3DDepth = 26**

Maximum 3D texture depth

**cudaDevAttrMaxTexture2DLayeredWidth = 27**

Maximum 2D layered texture width

**cudaDevAttrMaxTexture2DLayeredHeight = 28**

Maximum 2D layered texture height  
**cudaDevAttrMaxTexture2DLayeredLayers = 29**  
 Maximum layers in a 2D layered texture  
**cudaDevAttrSurfaceAlignment = 30**  
 Alignment requirement for surfaces  
**cudaDevAttrConcurrentKernels = 31**  
 Device can possibly execute multiple kernels concurrently  
**cudaDevAttrEccEnabled = 32**  
 Device has ECC support enabled  
**cudaDevAttrPciBusId = 33**  
 PCI bus ID of the device  
**cudaDevAttrPciDeviceId = 34**  
 PCI device ID of the device  
**cudaDevAttrTccDriver = 35**  
 Device is using TCC driver model  
**cudaDevAttrMemoryClockRate = 36**  
 Peak memory clock frequency in kilohertz  
**cudaDevAttrGlobalMemoryBusWidth = 37**  
 Global memory bus width in bits  
**cudaDevAttrL2CacheSize = 38**  
 Size of L2 cache in bytes  
**cudaDevAttrMaxThreadsPerMultiProcessor = 39**  
 Maximum resident threads per multiprocessor  
**cudaDevAttrAsyncEngineCount = 40**  
 Number of asynchronous engines  
**cudaDevAttrUnifiedAddressing = 41**  
 Device shares a unified address space with the host  
**cudaDevAttrMaxTexture1DLayeredWidth = 42**  
 Maximum 1D layered texture width  
**cudaDevAttrMaxTexture1DLayeredLayers = 43**  
 Maximum layers in a 1D layered texture  
**cudaDevAttrMaxTexture2DGatherWidth = 45**  
 Maximum 2D texture width if cudaArrayTextureGather is set  
**cudaDevAttrMaxTexture2DGatherHeight = 46**  
 Maximum 2D texture height if cudaArrayTextureGather is set  
**cudaDevAttrMaxTexture3DWidthAlt = 47**  
 Alternate maximum 3D texture width  
**cudaDevAttrMaxTexture3DHeightAlt = 48**  
 Alternate maximum 3D texture height  
**cudaDevAttrMaxTexture3DDepthAlt = 49**  
 Alternate maximum 3D texture depth  
**cudaDevAttrPciDomainId = 50**  
 PCI domain ID of the device

**cudaDevAttrTexturePitchAlignment = 51**

Pitch alignment requirement for textures

**cudaDevAttrMaxTextureCubemapWidth = 52**

Maximum cubemap texture width/height

**cudaDevAttrMaxTextureCubemapLayeredWidth = 53**

Maximum cubemap layered texture width/height

**cudaDevAttrMaxTextureCubemapLayeredLayers = 54**

Maximum layers in a cubemap layered texture

**cudaDevAttrMaxSurface1DWidth = 55**

Maximum 1D surface width

**cudaDevAttrMaxSurface2DWidth = 56**

Maximum 2D surface width

**cudaDevAttrMaxSurface2DHeight = 57**

Maximum 2D surface height

**cudaDevAttrMaxSurface3DWidth = 58**

Maximum 3D surface width

**cudaDevAttrMaxSurface3DHeight = 59**

Maximum 3D surface height

**cudaDevAttrMaxSurface3DDepth = 60**

Maximum 3D surface depth

**cudaDevAttrMaxSurface1DLayeredWidth = 61**

Maximum 1D layered surface width

**cudaDevAttrMaxSurface1DLayeredLayers = 62**

Maximum layers in a 1D layered surface

**cudaDevAttrMaxSurface2DLayeredWidth = 63**

Maximum 2D layered surface width

**cudaDevAttrMaxSurface2DLayeredHeight = 64**

Maximum 2D layered surface height

**cudaDevAttrMaxSurface2DLayeredLayers = 65**

Maximum layers in a 2D layered surface

**cudaDevAttrMaxSurfaceCubemapWidth = 66**

Maximum cubemap surface width

**cudaDevAttrMaxSurfaceCubemapLayeredWidth = 67**

Maximum cubemap layered surface width

**cudaDevAttrMaxSurfaceCubemapLayeredLayers = 68**

Maximum layers in a cubemap layered surface

**cudaDevAttrMaxTexture1DLinearWidth = 69**

Maximum 1D linear texture width

**cudaDevAttrMaxTexture2DLinearWidth = 70**

Maximum 2D linear texture width

**cudaDevAttrMaxTexture2DLinearHeight = 71**

Maximum 2D linear texture height

**cudaDevAttrMaxTexture2DLinearPitch = 72**



Maximum 2D linear texture pitch in bytes  
**cudaDevAttrMaxTexture2DMipmappedWidth = 73**  
 Maximum mipmapped 2D texture width  
**cudaDevAttrMaxTexture2DMipmappedHeight = 74**  
 Maximum mipmapped 2D texture height  
**cudaDevAttrComputeCapabilityMajor = 75**  
 Major compute capability version number  
**cudaDevAttrComputeCapabilityMinor = 76**  
 Minor compute capability version number  
**cudaDevAttrMaxTexture1DMipmappedWidth = 77**  
 Maximum mipmapped 1D texture width  
**cudaDevAttrStreamPrioritiesSupported = 78**  
 Device supports stream priorities  
**cudaDevAttrGlobalL1CacheSupported = 79**  
 Device supports caching globals in L1  
**cudaDevAttrLocalL1CacheSupported = 80**  
 Device supports caching locals in L1  
**cudaDevAttrMaxSharedMemoryPerMultiprocessor = 81**  
 Maximum shared memory available per multiprocessor in bytes  
**cudaDevAttrMaxRegistersPerMultiprocessor = 82**  
 Maximum number of 32-bit registers available per multiprocessor  
**cudaDevAttrManagedMemory = 83**  
 Device can allocate managed memory on this system  
**cudaDevAttrIsMultiGpuBoard = 84**  
 Device is on a multi-GPU board  
**cudaDevAttrMultiGpuBoardGroupID = 85**  
 Unique identifier for a group of devices on the same multi-GPU board  
**cudaDevAttrHostNativeAtomicSupported = 86**  
 Link between the device and the host supports native atomic operations  
**cudaDevAttrSingleToDoublePrecisionPerfRatio = 87**  
 Ratio of single precision performance (in floating-point operations per second) to double precision performance  
**cudaDevAttrPageableMemoryAccess = 88**  
 Device supports coherently accessing pageable memory without calling `cudaHostRegister` on it  
**cudaDevAttrConcurrentManagedAccess = 89**  
 Device can coherently access managed memory concurrently with the CPU  
**cudaDevAttrComputePreemptionSupported = 90**  
 Device supports Compute Preemption  
**cudaDevAttrCanUseHostPointerForRegisteredMem = 91**  
 Device can access host registered memory at the same virtual address as the CPU  
**cudaDevAttrReserved92 = 92**  
**cudaDevAttrReserved93 = 93**

**cudaDevAttrReserved94 = 94**

**cudaDevAttrCooperativeLaunch = 95**

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

**cudaDevAttrCooperativeMultiDeviceLaunch = 96**

Device can participate in cooperative kernels launched via

[cudaLaunchCooperativeKernelMultiDevice](#)

**cudaDevAttrMaxSharedMemoryPerBlockOptin = 97**

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

[cudaFuncSetAttribute](#)

**cudaDevAttrCanFlushRemoteWrites = 98**

Device supports flushing of outstanding remote writes.

**cudaDevAttrHostRegisterSupported = 99**

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

**cudaDevAttrPageableMemoryAccessUsesHostPageTables = 100**

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

**cudaDevAttrDirectManagedMemAccessFromHost = 101**

Host can directly access managed memory on the device without migration.

## enum cudaDeviceP2PAttr

CUDA device P2P attributes

### Values

**cudaDevP2PAttrPerformanceRank = 1**

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

**cudaDevP2PAttrAccessSupported = 2**

Peer access is enabled

**cudaDevP2PAttrNativeAtomicSupported = 3**

Native atomic operation over the link supported

**cudaDevP2PAttrCudaArrayAccessSupported = 4**

Accessing CUDA arrays over the link supported

## enum cudaEglColorFormat

CUDA EGL Color Format - The different planar and multiplanar formats currently supported for CUDA\_EGL interops.

### Values

**cudaEglColorFormatYUV420Planar = 0**

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

**cudaEglColorFormatYUV420SemiPlanar = 1**

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

**cudaEglColorFormatYUV422Planar = 2**

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

**cudaEglColorFormatYUV422SemiPlanar = 3**

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

**cudaEglColorFormatRGB = 4**

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

**cudaEglColorFormatBGR = 5**

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

**cudaEglColorFormatARGB = 6**

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

**cudaEglColorFormatRGBA = 7**

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

**cudaEglColorFormatL = 8**

single luminance channel in one surface.

**cudaEglColorFormatR = 9**

single color channel in one surface.

**cudaEglColorFormatYUV444Planar = 10**

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

**cudaEglColorFormatYUV444SemiPlanar = 11**

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

**cudaEglColorFormatYUYV422 = 12**

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

**cudaEglColorFormatUYVY422 = 13**

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

**cudaEglColorFormatABGR = 14**

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

**cudaEglColorFormatBGRA = 15**

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

**cudaEglColorFormatA = 16**

Alpha color format - one channel in one surface.

**cudaEglColorFormatRG = 17**

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

**cudaEglColorFormatAYUV = 18**

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

**cudaEglColorFormatYVU444SemiPlanar = 19**

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

**cudaEglColorFormatYVU422SemiPlanar = 20**

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

**cudaEglColorFormatYVU420SemiPlanar = 21**

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

**cudaEglColorFormatY10V10U10\_444SemiPlanar = 22**

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

**cudaEglColorFormatY10V10U10\_420SemiPlanar = 23**

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

**cudaEglColorFormatY12V12U12\_444SemiPlanar = 24**

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

**cudaEglColorFormatY12V12U12\_420SemiPlanar = 25**

Y12, V12U12 in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

**cudaEglColorFormatVYUY\_ER = 26**

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

**cudaEglColorFormatUYVY\_ER = 27**

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

**cudaEglColorFormatYUYV\_ER = 28**

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

**cudaEglColorFormatYVYU\_ER = 29**

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

**cudaEglColorFormatYUV\_ER = 30**

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

**cudaEglColorFormatYUVA\_ER = 31**

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

**cudaEglColorFormatAYUV\_ER = 32**

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

**cudaEglColorFormatYUV444Planar\_ER = 33**

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

**cudaEglColorFormatYUV422Planar\_ER = 34**

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

**cudaEglColorFormatYUV420Planar\_ER = 35**

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

**cudaEglColorFormatYUV444SemiPlanar\_ER = 36**

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

**cudaEglColorFormatYUV422SemiPlanar\_ER = 37**

Extended Range Y, UV in two surfaces (UV as one surface) with VU byte ordering, U/V width = 1/2 Y width, U/V height = Y height.

**cudaEglColorFormatYUV420SemiPlanar\_ER = 38**

Extended Range Y, UV in two surfaces (UV as one surface) with VU byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

**cudaEglColorFormatYVU444Planar\_ER = 39**

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

**cudaEglColorFormatYVU422Planar\_ER = 40**

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

**cudaEglColorFormatYVU420Planar\_ER = 41**

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

**cudaEglColorFormatYVU444SemiPlanar\_ER = 42**

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

**cudaEglColorFormatYVU422SemiPlanar\_ER = 43**

Extended Range Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = Y height.

**cudaEglColorFormatYVU420SemiPlanar\_ER = 44**

Extended Range Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

**cudaEglColorFormatBayerRGGB = 45**

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

**cudaEglColorFormatBayerBGGR = 46**

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

**cudaEglColorFormatBayerGRBG = 47**

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

**cudaEglColorFormatBayerGBRG = 48**

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

**cudaEglColorFormatBayer10RGGB = 49**

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

**cudaEglColorFormatBayer10BGGR = 50**

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

**cudaEglColorFormatBayer10GRBG = 51**

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

**cudaEglColorFormatBayer10GBRG = 52**

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

**cudaEglColorFormatBayer12RGGB = 53**

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

**cudaEglColorFormatBayer12BGGR = 54**

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

**cudaEglColorFormatBayer12GRBG = 55**

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

**cudaEglColorFormatBayer12GBRG = 56**

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

**cudaEglColorFormatBayer14RGGB = 57**

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

**cudaEglColorFormatBayer14BGGR = 58**

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

**cudaEglColorFormatBayer14GRBG = 59**

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

**cudaEglColorFormatBayer14GBRG = 60**

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

**cudaEglColorFormatBayer20RGGB = 61**

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

**cudaEglColorFormatBayer20BGGR = 62**

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

**cudaEglColorFormatBayer20GRBG = 63**

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

**cudaEglColorFormatBayer20GBRG = 64**

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

**cudaEglColorFormatYVU444Planar = 65**

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

#### **cudaEglColorFormatYVU422Planar = 66**

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

#### **cudaEglColorFormatYVU420Planar = 67**

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

#### **cudaEglColorFormatBayerIspRGGB = 68**

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

#### **cudaEglColorFormatBayerIspBGGR = 69**

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

#### **cudaEglColorFormatBayerIspGRBG = 70**

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

#### **cudaEglColorFormatBayerIspGBRG = 71**

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

## enum cudaEglFrameType

CUDA EglFrame type - array or pointer

### Values

#### **cudaEglFrameTypeArray = 0**

Frame type CUDA array

#### **cudaEglFrameTypePitch = 1**

Frame type CUDA pointer

## enum cudaEglResourceLocationFlags

Resource location flags- system or vidmem

For CUDA context on iGPU, since video and system memory are equivalent - these flags will not have an effect on the execution.

For CUDA context on dGPU, applications can use the flag [cudaEglResourceLocationFlags](#) to give a hint about the desired location.

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

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



There may be an additional latency due to new allocation and data migration, if the frame is produced on a different memory.

## Values

**cudaEglResourceLocationSystemem = 0x00**

Resource location systemem

**cudaEglResourceLocationVidmem = 0x01**

Resource location vidmem

## enum cudaError

CUDA error types

## Values

**cudaSuccess = 0**

The API call returned with no errors. In the case of query calls, this also means that the operation being queried is complete (see [cudaEventQuery\(\)](#) and [cudaStreamQuery\(\)](#)).

**cudaErrorInvalidValue = 1**

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

**cudaErrorMemoryAllocation = 2**

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

**cudaErrorInitializationError = 3**

The API call failed because the CUDA driver and runtime could not be initialized.

**cudaErrorCudartUnloading = 4**

This indicates that a CUDA Runtime API call cannot be executed because it is being called during process shut down, at a point in time after CUDA driver has been unloaded.

**cudaErrorProfilerDisabled = 5**

This indicates profiler is not initialized for this run. This can happen when the application is running with external profiling tools like visual profiler.

**cudaErrorProfilerNotInitialized = 6**

**Deprecated** This error return is deprecated as of CUDA 5.0. It is no longer an error to attempt to enable/disable the profiling via [cudaProfilerStart](#) or [cudaProfilerStop](#) without initialization.

**cudaErrorProfilerAlreadyStarted = 7**

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

**cudaErrorProfilerAlreadyStopped = 8**

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



**cudaErrorInvalidConfiguration = 9**

This indicates that a kernel launch is requesting resources that can never be satisfied by the current device. Requesting more shared memory per block than the device supports will trigger this error, as will requesting too many threads or blocks. See [cudaDeviceProp](#) for more device limitations.

**cudaErrorInvalidPitchValue = 12**

This indicates that one or more of the pitch-related parameters passed to the API call is not within the acceptable range for pitch.

**cudaErrorInvalidSymbol = 13**

This indicates that the symbol name/identifier passed to the API call is not a valid name or identifier.

**cudaErrorInvalidHostPointer = 16**

This indicates that at least one host pointer passed to the API call is not a valid host pointer. [Deprecated](#) This error return is deprecated as of CUDA 10.1.

**cudaErrorInvalidDevicePointer = 17**

This indicates that at least one device pointer passed to the API call is not a valid device pointer. [Deprecated](#) This error return is deprecated as of CUDA 10.1.

**cudaErrorInvalidTexture = 18**

This indicates that the texture passed to the API call is not a valid texture.

**cudaErrorInvalidTextureBinding = 19**

This indicates that the texture binding is not valid. This occurs if you call [cudaGetTextureAlignmentOffset\(\)](#) with an unbound texture.

**cudaErrorInvalidChannelDescriptor = 20**

This indicates that the channel descriptor passed to the API call is not valid. This occurs if the format is not one of the formats specified by [cudaChannelFormatKind](#), or if one of the dimensions is invalid.

**cudaErrorInvalidMemcpyDirection = 21**

This indicates that the direction of the memcpy passed to the API call is not one of the types specified by [cudaMemcpyKind](#).

**cudaErrorAddressOfConstant = 22**

This indicated that the user has taken the address of a constant variable, which was forbidden up until the CUDA 3.1 release. [Deprecated](#) This error return is deprecated as of CUDA 3.1. Variables in constant memory may now have their address taken by the runtime via [cudaGetSymbolAddress\(\)](#).

**cudaErrorTextureFetchFailed = 23**

This indicated that a texture fetch was not able to be performed. This was previously used for device emulation of texture operations. [Deprecated](#) This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

**cudaErrorTextureNotBound = 24**

This indicated that a texture was not bound for access. This was previously used for device emulation of texture operations. [Deprecated](#) This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

**cudaErrorSynchronizationError = 25**

This indicated that a synchronization operation had failed. This was previously used for some device emulation functions. **Deprecated** This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

**cudaErrorInvalidFilterSetting = 26**

This indicates that a non-float texture was being accessed with linear filtering. This is not supported by CUDA.

**cudaErrorInvalidNormSetting = 27**

This indicates that an attempt was made to read a non-float texture as a normalized float. This is not supported by CUDA.

**cudaErrorMixedDeviceExecution = 28**

Mixing of device and device emulation code was not allowed. **Deprecated** This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

**cudaErrorNotYetImplemented = 31**

This indicates that the API call is not yet implemented. Production releases of CUDA will never return this error. **Deprecated** This error return is deprecated as of CUDA 4.1.

**cudaErrorMemoryValueTooLarge = 32**

This indicated that an emulated device pointer exceeded the 32-bit address range. **Deprecated** This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

**cudaErrorInsufficientDriver = 35**

This indicates that the installed NVIDIA CUDA driver is older than the CUDA runtime library. This is not a supported configuration. Users should install an updated NVIDIA display driver to allow the application to run.

**cudaErrorInvalidSurface = 37**

This indicates that the surface passed to the API call is not a valid surface.

**cudaErrorDuplicateVariableName = 43**

This indicates that multiple global or constant variables (across separate CUDA source files in the application) share the same string name.

**cudaErrorDuplicateTextureName = 44**

This indicates that multiple textures (across separate CUDA source files in the application) share the same string name.

**cudaErrorDuplicateSurfaceName = 45**

This indicates that multiple surfaces (across separate CUDA source files in the application) share the same string name.

**cudaErrorDevicesUnavailable = 46**

This indicates that all CUDA devices are busy or unavailable at the current time. Devices are often busy/unavailable due to use of **cudaComputeModeExclusive**, **cudaComputeModeProhibited** or when long running CUDA kernels have filled up the GPU and are blocking new work from starting. They can also be unavailable

due to memory constraints on a device that already has active CUDA work being performed.

#### **cudaErrorIncompatibleDriverContext = 49**

This indicates that the current context is not compatible with this the CUDA Runtime. This can only occur if you are using CUDA Runtime/Driver interoperability and have created an existing Driver context using the driver API. The Driver context may be incompatible either because the Driver context was created using an older version of the API, because the Runtime API call expects a primary driver context and the Driver context is not primary, or because the Driver context has been destroyed. Please see [Interactions](#) with the CUDA Driver API" for more information.

#### **cudaErrorMissingConfiguration = 52**

The device function being invoked (usually via [cudaLaunchKernel\(\)](#)) was not previously configured via the [cudaConfigureCall\(\)](#) function.

#### **cudaErrorPriorLaunchFailure = 53**

This indicated that a previous kernel launch failed. This was previously used for device emulation of kernel launches. [Deprecated](#) This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

#### **cudaErrorLaunchMaxDepthExceeded = 65**

This error indicates that a device runtime grid launch did not occur because the depth of the child grid would exceed the maximum supported number of nested grid launches.

#### **cudaErrorLaunchFileScopedTex = 66**

This error indicates that a grid launch did not occur because the kernel uses file-scoped textures which are unsupported by the device runtime. Kernels launched via the device runtime only support textures created with the Texture Object API's.

#### **cudaErrorLaunchFileScopedSurf = 67**

This error indicates that a grid launch did not occur because the kernel uses file-scoped surfaces which are unsupported by the device runtime. Kernels launched via the device runtime only support surfaces created with the Surface Object API's.

#### **cudaErrorSyncDepthExceeded = 68**

This error indicates that a call to [cudaDeviceSynchronize](#) made from the device runtime failed because the call was made at grid depth greater than either the default (2 levels of grids) or user specified device limit [cudaLimitDevRuntimeSyncDepth](#). To be able to synchronize on launched grids at a greater depth successfully, the maximum nested depth at which [cudaDeviceSynchronize](#) will be called must be specified with the [cudaLimitDevRuntimeSyncDepth](#) limit to the [cudaDeviceSetLimit](#) api before the host-side launch of a kernel using the device runtime. Keep in mind that additional levels of sync depth require the runtime to reserve large amounts of device memory that cannot be used for user allocations.

#### **cudaErrorLaunchPendingCountExceeded = 69**

This error indicates that a device runtime grid launch failed because the launch would exceed the limit [cudaLimitDevRuntimePendingLaunchCount](#). For this

launch to proceed successfully, `cudaDeviceSetLimit` must be called to set the `cudaLimitDevRuntimePendingLaunchCount` to be higher than the upper bound of outstanding launches that can be issued to the device runtime. Keep in mind that raising the limit of pending device runtime launches will require the runtime to reserve device memory that cannot be used for user allocations.

**`cudaErrorInvalidDeviceFunction = 98`**

The requested device function does not exist or is not compiled for the proper device architecture.

**`cudaErrorNoDevice = 100`**

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

**`cudaErrorInvalidDevice = 101`**

This indicates that the device ordinal supplied by the user does not correspond to a valid CUDA device.

**`cudaErrorStartupFailure = 127`**

This indicates an internal startup failure in the CUDA runtime.

**`cudaErrorInvalidKernelImage = 200`**

This indicates that the device kernel image is invalid.

**`cudaErrorDeviceUninitialized = 201`**

This most frequently indicates that there is no context bound to the current thread. This can also be returned if the context passed to an API call is not a valid handle (such as a context that has had `cuCtxDestroy()` invoked on it). This can also be returned if a user mixes different API versions (i.e. 3010 context with 3020 API calls). See `cuCtxGetApiVersion()` for more details.

**`cudaErrorMapBufferObjectFailed = 205`**

This indicates that the buffer object could not be mapped.

**`cudaErrorUnmapBufferObjectFailed = 206`**

This indicates that the buffer object could not be unmapped.

**`cudaErrorArrayIsMapped = 207`**

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

**`cudaErrorAlreadyMapped = 208`**

This indicates that the resource is already mapped.

**`cudaErrorNoKernelImageForDevice = 209`**

This indicates that there is no kernel image available that is suitable for the device. This can occur when a user specifies code generation options for a particular CUDA source file that do not include the corresponding device configuration.

**`cudaErrorAlreadyAcquired = 210`**

This indicates that a resource has already been acquired.

**`cudaErrorNotMapped = 211`**

This indicates that a resource is not mapped.

**`cudaErrorNotMappedAsArray = 212`**

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

**cudaErrorNotMappedAsPointer = 213**

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

**cudaErrorECCUncorrectable = 214**

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

**cudaErrorUnsupportedLimit = 215**

This indicates that the `cudaLimit` passed to the API call is not supported by the active device.

**cudaErrorDeviceAlreadyInUse = 216**

This indicates that a call tried to access an exclusive-thread device that is already in use by a different thread.

**cudaErrorPeerAccessUnsupported = 217**

This error indicates that P2P access is not supported across the given devices.

**cudaErrorInvalidPtx = 218**

A PTX compilation failed. The runtime may fall back to compiling PTX if an application does not contain a suitable binary for the current device.

**cudaErrorInvalidGraphicsContext = 219**

This indicates an error with the OpenGL or DirectX context.

**cudaErrorNvlinkUncorrectable = 220**

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

**cudaErrorJitCompilerNotFound = 221**

This indicates that the PTX JIT compiler library was not found. The JIT Compiler library is used for PTX compilation. The runtime may fall back to compiling PTX if an application does not contain a suitable binary for the current device.

**cudaErrorInvalidSource = 300**

This indicates that the device kernel source is invalid.

**cudaErrorFileNotFound = 301**

This indicates that the file specified was not found.

**cudaErrorSharedObjectSymbolNotFound = 302**

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

**cudaErrorSharedObjectInitFailed = 303**

This indicates that initialization of a shared object failed.

**cudaErrorOperatingSystem = 304**

This error indicates that an OS call failed.

**cudaErrorInvalidResourceHandle = 400**

This indicates that a resource handle passed to the API call was not valid. Resource handles are opaque types like `cudaStream_t` and `cudaEvent_t`.

**cudaErrorIllegalState = 401**

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

**cudaErrorSymbolNotFound = 500**

This indicates that a named symbol was not found. Examples of symbols are global/constant variable names, texture names, and surface names.

**cudaErrorNotReady = 600**

This indicates that asynchronous operations issued previously have not completed yet. This result is not actually an error, but must be indicated differently than `cudaSuccess` (which indicates completion). Calls that may return this value include `cudaEventQuery()` and `cudaStreamQuery()`.

#### **`cudaErrorIllegalAddress = 700`**

The device encountered a load or store instruction on an invalid memory address. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

#### **`cudaErrorLaunchOutOfResources = 701`**

This indicates that a launch did not occur because it did not have appropriate resources. Although this error is similar to `cudaErrorInvalidConfiguration`, this error usually indicates that the user has attempted to pass too many arguments to the device kernel, or the kernel launch specifies too many threads for the kernel's register count.

#### **`cudaErrorLaunchTimeout = 702`**

This indicates that the device kernel took too long to execute. This can only occur if timeouts are enabled - see the device property `kernelExecTimeoutEnabled` for more information. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

#### **`cudaErrorLaunchIncompatibleTexturing = 703`**

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

#### **`cudaErrorPeerAccessAlreadyEnabled = 704`**

This error indicates that a call to `cudaDeviceEnablePeerAccess()` is trying to re-enable peer addressing on from a context which has already had peer addressing enabled.

#### **`cudaErrorPeerAccessNotEnabled = 705`**

This error indicates that `cudaDeviceDisablePeerAccess()` is trying to disable peer addressing which has not been enabled yet via `cudaDeviceEnablePeerAccess()`.

#### **`cudaErrorSetOnActiveProcess = 708`**

This indicates that the user has called `cudaSetValidDevices()`, `cudaSetDeviceFlags()`, `cudaD3D9SetDirect3DDevice()`, `cudaD3D10SetDirect3DDevice()`, `cudaD3D11SetDirect3DDevice()`, or `cudaVDPAUSetVDPAUDevice()` after initializing the CUDA runtime by calling non-device management operations (allocating memory and launching kernels are examples of non-device management operations). This error can also be returned if using runtime/driver interoperability and there is an existing `CUcontext` active on the host thread.

#### **`cudaErrorContextIsDestroyed = 709`**

This error indicates that the context current to the calling thread has been destroyed using `cuCtxDestroy`, or is a primary context which has not yet been initialized.

#### **`cudaErrorAssert = 710`**



An assert triggered in device code during kernel execution. The device cannot be used again. All existing allocations are invalid. To continue using CUDA, the process must be terminated and relaunched.

**cudaErrorTooManyPeers = 711**

This error indicates that the hardware resources required to enable peer access have been exhausted for one or more of the devices passed to `cudaEnablePeerAccess()`.

**cudaErrorHostMemoryAlreadyRegistered = 712**

This error indicates that the memory range passed to `cudaHostRegister()` has already been registered.

**cudaErrorHostMemoryNotRegistered = 713**

This error indicates that the pointer passed to `cudaHostUnregister()` does not correspond to any currently registered memory region.

**cudaErrorHardwareStackError = 714**

Device encountered an error in the call stack during kernel execution, possibly due to stack corruption or exceeding the stack size limit. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

**cudaErrorIllegalInstruction = 715**

The device encountered an illegal instruction during kernel execution. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

**cudaErrorMisalignedAddress = 716**

The device encountered a load or store instruction on a memory address which is not aligned. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

**cudaErrorInvalidAddressSpace = 717**

While executing a kernel, the device encountered an instruction which can only operate on memory locations in certain address spaces (global, shared, or local), but was supplied a memory address not belonging to an allowed address space. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

**cudaErrorInvalidPc = 718**

The device encountered an invalid program counter. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

**cudaErrorLaunchFailure = 719**

An exception occurred on the device while executing a kernel. Common causes include dereferencing an invalid device pointer and accessing out of bounds shared memory. Less common cases can be system specific - more information about these cases can be found in the system specific user guide. This leaves the process in an

inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

**cudaErrorCooperativeLaunchTooLarge = 720**

This error indicates that the number of blocks launched per grid for a kernel that was launched via either `cudaLaunchCooperativeKernel` or `cudaLaunchCooperativeKernelMultiDevice` exceeds the maximum number of blocks as allowed by `cudaOccupancyMaxActiveBlocksPerMultiprocessor` or `cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags` times the number of multiprocessors as specified by the device attribute `cudaDevAttrMultiProcessorCount`.

**cudaErrorNotPermitted = 800**

This error indicates the attempted operation is not permitted.

**cudaErrorNotSupported = 801**

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

**cudaErrorSystemNotReady = 802**

This error indicates that the system is not yet ready to start any CUDA work. To continue using CUDA, verify the system configuration is in a valid state and all required driver daemons are actively running. More information about this error can be found in the system specific user guide.

**cudaErrorSystemDriverMismatch = 803**

This error indicates that there is a mismatch between the versions of the display driver and the CUDA driver. Refer to the compatibility documentation for supported versions.

**cudaErrorCompatNotSupportedOnDevice = 804**

This error indicates that the system was upgraded to run with forward compatibility but the visible hardware detected by CUDA does not support this configuration. Refer to the compatibility documentation for the supported hardware matrix or ensure that only supported hardware is visible during initialization via the `CUDA_VISIBLE_DEVICES` environment variable.

**cudaErrorStreamCaptureUnsupported = 900**

The operation is not permitted when the stream is capturing.

**cudaErrorStreamCaptureInvalidated = 901**

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

**cudaErrorStreamCaptureMerge = 902**

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

**cudaErrorStreamCaptureUnmatched = 903**

The capture was not initiated in this stream.

**cudaErrorStreamCaptureUnjoined = 904**

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

**cudaErrorStreamCaptureIsolation = 905**



A dependency would have been created which crosses the capture sequence boundary. Only implicit in-stream ordering dependencies are allowed to cross the boundary.

**cudaErrorStreamCaptureImplicit = 906**

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

**cudaErrorCapturedEvent = 907**

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

**cudaErrorStreamCaptureWrongThread = 908**

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

**cudaErrorTimeout = 909**

This indicates that the wait operation has timed out.

**cudaErrorGraphExecUpdateFailure = 910**

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

**cudaErrorUnknown = 999**

This indicates that an unknown internal error has occurred.

**cudaErrorApiFailureBase = 10000**

Any unhandled CUDA driver error is added to this value and returned via the runtime. Production releases of CUDA should not return such errors. *Deprecated*  
This error return is deprecated as of CUDA 4.1.

## enum cudaExternalMemoryHandleType

External memory handle types

### Values

**cudaExternalMemoryHandleTypeOpaqueFd = 1**

Handle is an opaque file descriptor

**cudaExternalMemoryHandleTypeOpaqueWin32 = 2**

Handle is an opaque shared NT handle

**cudaExternalMemoryHandleTypeOpaqueWin32Kmt = 3**

Handle is an opaque, globally shared handle

**cudaExternalMemoryHandleTypeD3D12Heap = 4**

Handle is a D3D12 heap object

**cudaExternalMemoryHandleTypeD3D12Resource = 5**

Handle is a D3D12 committed resource

**cudaExternalMemoryHandleTypeD3D11Resource = 6**

Handle is a shared NT handle to a D3D11 resource

**cudaExternalMemoryHandleTypeD3D11ResourceKmt = 7**

Handle is a globally shared handle to a D3D11 resource  
**cudaExternalMemoryHandleTypeNvSciBuf = 8**  
 Handle is an NvSciBuf object

## enum cudaExternalSemaphoreHandleType

External semaphore handle types

### Values

**cudaExternalSemaphoreHandleTypeOpaqueFd = 1**  
 Handle is an opaque file descriptor  
**cudaExternalSemaphoreHandleTypeOpaqueWin32 = 2**  
 Handle is an opaque shared NT handle  
**cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt = 3**  
 Handle is an opaque, globally shared handle  
**cudaExternalSemaphoreHandleTypeD3D12Fence = 4**  
 Handle is a shared NT handle referencing a D3D12 fence object  
**cudaExternalSemaphoreHandleTypeD3D11Fence = 5**  
 Handle is a shared NT handle referencing a D3D11 fence object  
**cudaExternalSemaphoreHandleTypeNvSciSync = 6**  
 Opaque handle to NvSciSync Object  
**cudaExternalSemaphoreHandleTypeKeyedMutex = 7**  
 Handle is a shared NT handle referencing a D3D11 keyed mutex object  
**cudaExternalSemaphoreHandleTypeKeyedMutexKmt = 8**  
 Handle is a shared KMT handle referencing a D3D11 keyed mutex object

## enum cudaFuncAttribute

CUDA function attributes that can be set using [cudaFuncSetAttribute](#)

### Values

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

## enum cudaFuncCache

CUDA function cache configurations

### Values

**cudaFuncCachePreferNone = 0**

Default function cache configuration, no preference

**cudaFuncCachePreferShared = 1**

Prefer larger shared memory and smaller L1 cache

**cudaFuncCachePreferL1 = 2**

Prefer larger L1 cache and smaller shared memory

**cudaFuncCachePreferEqual = 3**

Prefer equal size L1 cache and shared memory

## enum cudaGraphExecUpdateResult

CUDA Graph Update error types

### Values

**cudaGraphExecUpdateSuccess = 0x0**

The update succeeded

**cudaGraphExecUpdateError = 0x1**

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

**cudaGraphExecUpdateErrorTopologyChanged = 0x2**

The update failed because the topology changed

**cudaGraphExecUpdateErrorNodeTypeChanged = 0x3**

The update failed because a node type changed

**cudaGraphExecUpdateErrorFunctionChanged = 0x4**

The update failed because the function of a kernel node changed

**cudaGraphExecUpdateErrorParametersChanged = 0x5**

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

**cudaGraphExecUpdateErrorNotSupported = 0x6**

The update failed because something about the node is not supported

## enum cudaGraphicsCubeFace

CUDA graphics interop array indices for cube maps

### Values

**cudaGraphicsCubeFacePositiveX = 0x00**

Positive X face of cubemap

**cudaGraphicsCubeFaceNegativeX = 0x01**

Negative X face of cubemap

**cudaGraphicsCubeFacePositiveY = 0x02**

Positive Y face of cubemap

**cudaGraphicsCubeFaceNegativeY = 0x03**

Negative Y face of cubemap

**cudaGraphicsCubeFacePositiveZ = 0x04**

Positive Z face of cubemap

**cudaGraphicsCubeFaceNegativeZ = 0x05**

Negative Z face of cubemap

## enum cudaGraphicsMapFlags

CUDA graphics interop map flags

### Values

**cudaGraphicsMapFlagsNone = 0**

Default; Assume resource can be read/written

**cudaGraphicsMapFlagsReadOnly = 1**

CUDA will not write to this resource

**cudaGraphicsMapFlagsWriteDiscard = 2**

CUDA will only write to and will not read from this resource

## enum cudaGraphicsRegisterFlags

CUDA graphics interop register flags

### Values

**cudaGraphicsRegisterFlagsNone = 0**

Default

**cudaGraphicsRegisterFlagsReadOnly = 1**

CUDA will not write to this resource

**cudaGraphicsRegisterFlagsWriteDiscard = 2**

CUDA will only write to and will not read from this resource

**cudaGraphicsRegisterFlagsSurfaceLoadStore = 4**

CUDA will bind this resource to a surface reference

**cudaGraphicsRegisterFlagsTextureGather = 8**

CUDA will perform texture gather operations on this resource

## enum cudaGraphNodeType

CUDA Graph node types

### Values

**cudaGraphNodeTypeKernel = 0x00**

GPU kernel node

**cudaGraphNodeTypeMemcpy = 0x01**

Memcpy node

**cudaGraphNodeTypeMemset = 0x02**

Memset node

**cudaGraphNodeTypeHost = 0x03**

Host (executable) node

**cudaGraphNodeTypeGraph = 0x04**

Node which executes an embedded graph

**cudaGraphNodeTypeEmpty = 0x05**

Empty (no-op) node

**cudaGraphNodeTypeCount**

## enum cudaLimit

CUDA Limits

### Values

**cudaLimitStackSize = 0x00**

GPU thread stack size

**cudaLimitPrintfFifoSize = 0x01**

GPU printf FIFO size

**cudaLimitMallocHeapSize = 0x02**

GPU malloc heap size

**cudaLimitDevRuntimeSyncDepth = 0x03**

GPU device runtime synchronize depth

**cudaLimitDevRuntimePendingLaunchCount = 0x04**

GPU device runtime pending launch count

**cudaLimitMaxL2FetchGranularity = 0x05**

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

## enum cudaMemcpyKind

CUDA memory copy types

### Values

**cudaMemcpyHostToHost = 0**

Host -> Host

**cudaMemcpyHostToDevice = 1**

Host -> Device

**cudaMemcpyDeviceToHost = 2**

Device -> Host

**cudaMemcpyDeviceToDevice = 3**

Device -> Device

**cudaMemcpyDefault = 4**

Direction of the transfer is inferred from the pointer values. Requires unified virtual addressing

## enum cudaMemoryAdvise

CUDA Memory Advise values

### Values

**cudaMemAdviseSetReadMostly = 1**

Data will mostly be read and only occasionally be written to

**cudaMemAdviseUnsetReadMostly = 2**

Undo the effect of [cudaMemAdviseSetReadMostly](#)

**cudaMemAdviseSetPreferredLocation = 3**

Set the preferred location for the data as the specified device

**cudaMemAdviseUnsetPreferredLocation = 4**

Clear the preferred location for the data

**cudaMemAdviseSetAccessedBy = 5**

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

**cudaMemAdviseUnsetAccessedBy = 6**

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

## enum cudaMemoryType

CUDA memory types

### Values

**cudaMemoryTypeUnregistered = 0**

Unregistered memory

**cudaMemoryTypeHost = 1**

Host memory

**cudaMemoryTypeDevice = 2**

Device memory

**cudaMemoryTypeManaged = 3**

Managed memory

## enum cudaMemRangeAttribute

CUDA range attributes

### Values

**cudaMemRangeAttributeReadMostly = 1**

Whether the range will mostly be read and only occasionally be written to

**cudaMemRangeAttributePreferredLocation = 2**

The preferred location of the range

**cudaMemRangeAttributeAccessedBy = 3**

Memory range has [cudaMemAdviseSetAccessedBy](#) set for specified device

**cudaMemRangeAttributeLastPrefetchLocation = 4**

The last location to which the range was prefetched

## enum cudaOutputMode

CUDA Profiler Output modes

### Values

**cudaKeyValuePair = 0x00**

Output mode Key-Value pair format.

**cudaCSV = 0x01**

Output mode Comma separated values format.

## enum cudaResourceType

CUDA resource types

### Values

**cudaResourceTypeArray = 0x00**

Array resource

**cudaResourceTypeMipmappedArray = 0x01**

Mipmapped array resource

**cudaResourceTypeLinear = 0x02**

Linear resource

**cudaResourceTypePitch2D = 0x03**

Pitch 2D resource

## enum cudaResourceViewFormat

CUDA texture resource view formats

### Values

**cudaResViewFormatNone = 0x00**

No resource view format (use underlying resource format)

**cudaResViewFormatUnsignedChar1 = 0x01**

1 channel unsigned 8-bit integers

**cudaResViewFormatUnsignedChar2 = 0x02**

2 channel unsigned 8-bit integers

**cudaResViewFormatUnsignedChar4 = 0x03**

4 channel unsigned 8-bit integers

**cudaResViewFormatSignedChar1 = 0x04**  
 1 channel signed 8-bit integers  
**cudaResViewFormatSignedChar2 = 0x05**  
 2 channel signed 8-bit integers  
**cudaResViewFormatSignedChar4 = 0x06**  
 4 channel signed 8-bit integers  
**cudaResViewFormatUnsignedShort1 = 0x07**  
 1 channel unsigned 16-bit integers  
**cudaResViewFormatUnsignedShort2 = 0x08**  
 2 channel unsigned 16-bit integers  
**cudaResViewFormatUnsignedShort4 = 0x09**  
 4 channel unsigned 16-bit integers  
**cudaResViewFormatSignedShort1 = 0x0a**  
 1 channel signed 16-bit integers  
**cudaResViewFormatSignedShort2 = 0x0b**  
 2 channel signed 16-bit integers  
**cudaResViewFormatSignedShort4 = 0x0c**  
 4 channel signed 16-bit integers  
**cudaResViewFormatUnsignedInt1 = 0x0d**  
 1 channel unsigned 32-bit integers  
**cudaResViewFormatUnsignedInt2 = 0x0e**  
 2 channel unsigned 32-bit integers  
**cudaResViewFormatUnsignedInt4 = 0x0f**  
 4 channel unsigned 32-bit integers  
**cudaResViewFormatSignedInt1 = 0x10**  
 1 channel signed 32-bit integers  
**cudaResViewFormatSignedInt2 = 0x11**  
 2 channel signed 32-bit integers  
**cudaResViewFormatSignedInt4 = 0x12**  
 4 channel signed 32-bit integers  
**cudaResViewFormatHalf1 = 0x13**  
 1 channel 16-bit floating point  
**cudaResViewFormatHalf2 = 0x14**  
 2 channel 16-bit floating point  
**cudaResViewFormatHalf4 = 0x15**  
 4 channel 16-bit floating point  
**cudaResViewFormatFloat1 = 0x16**  
 1 channel 32-bit floating point  
**cudaResViewFormatFloat2 = 0x17**  
 2 channel 32-bit floating point  
**cudaResViewFormatFloat4 = 0x18**  
 4 channel 32-bit floating point  
**cudaResViewFormatUnsignedBlockCompressed1 = 0x19**



Block compressed 1  
**cudaResViewFormatUnsignedBlockCompressed2 = 0x1a**  
 Block compressed 2  
**cudaResViewFormatUnsignedBlockCompressed3 = 0x1b**  
 Block compressed 3  
**cudaResViewFormatUnsignedBlockCompressed4 = 0x1c**  
 Block compressed 4 unsigned  
**cudaResViewFormatSignedBlockCompressed4 = 0x1d**  
 Block compressed 4 signed  
**cudaResViewFormatUnsignedBlockCompressed5 = 0x1e**  
 Block compressed 5 unsigned  
**cudaResViewFormatSignedBlockCompressed5 = 0x1f**  
 Block compressed 5 signed  
**cudaResViewFormatUnsignedBlockCompressed6H = 0x20**  
 Block compressed 6 unsigned half-float  
**cudaResViewFormatSignedBlockCompressed6H = 0x21**  
 Block compressed 6 signed half-float  
**cudaResViewFormatUnsignedBlockCompressed7 = 0x22**  
 Block compressed 7

## enum cudaSharedCarveout

Shared memory carveout configurations. These may be passed to cudaFuncSetAttribute

### Values

**cudaSharedmemCarveoutDefault = -1**  
 No preference for shared memory or L1 (default)  
**cudaSharedmemCarveoutMaxShared = 100**  
 Prefer maximum available shared memory, minimum L1 cache  
**cudaSharedmemCarveoutMaxL1 = 0**  
 Prefer maximum available L1 cache, minimum shared memory

## enum cudaSharedMemConfig

CUDA shared memory configuration

### Values

**cudaSharedMemBankSizeDefault = 0**  
**cudaSharedMemBankSizeFourByte = 1**  
**cudaSharedMemBankSizeEightByte = 2**

## enum cudaStreamCaptureMode

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

### Values

**cudaStreamCaptureModeGlobal** = 0

**cudaStreamCaptureModeThreadLocal** = 1

**cudaStreamCaptureModeRelaxed** = 2

## enum cudaStreamCaptureStatus

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

### Values

**cudaStreamCaptureStatusNone** = 0

Stream is not capturing

**cudaStreamCaptureStatusActive** = 1

Stream is actively capturing

**cudaStreamCaptureStatusInvalidated** = 2

Stream is part of a capture sequence that has been invalidated, but not terminated

## enum cudaSurfaceBoundaryMode

CUDA Surface boundary modes

### Values

**cudaBoundaryModeZero** = 0

Zero boundary mode

**cudaBoundaryModeClamp** = 1

Clamp boundary mode

**cudaBoundaryModeTrap** = 2

Trap boundary mode

## enum cudaSurfaceFormatMode

CUDA Surface format modes

### Values

**cudaFormatModeForced** = 0

Forced format mode

**cudaFormatModeAuto** = 1

Auto format mode

## enum cudaTextureAddressMode

CUDA texture address modes

### Values

**cudaAddressModeWrap = 0**

Wrapping address mode

**cudaAddressModeClamp = 1**

Clamp to edge address mode

**cudaAddressModeMirror = 2**

Mirror address mode

**cudaAddressModeBorder = 3**

Border address mode

## enum cudaTextureFilterMode

CUDA texture filter modes

### Values

**cudaFilterModePoint = 0**

Point filter mode

**cudaFilterModeLinear = 1**

Linear filter mode

## enum cudaTextureReadMode

CUDA texture read modes

### Values

**cudaReadModeElementType = 0**

Read texture as specified element type

**cudaReadModeNormalizedFloat = 1**

Read texture as normalized float

## typedef cudaArray \*cudaArray\_const\_t

CUDA array (as source copy argument)

## typedef cudaArray \*cudaArray\_t

CUDA array

```
typedef struct CUeglStreamConnection_st  
*cudaEglStreamConnection
```

CUDA EGLSream Connection

```
typedef cudaError_t
```

CUDA Error types

```
typedef struct CUevent_st *cudaEvent_t
```

CUDA event types

```
typedef struct CUexternalMemory_st  
*cudaExternalMemory_t
```

CUDA external memory

```
typedef struct CUexternalSemaphore_st  
*cudaExternalSemaphore_t
```

CUDA external semaphore

```
typedef struct CUgraph_st *cudaGraph_t
```

CUDA graph

```
typedef struct CUgraphExec_st *cudaGraphExec_t
```

CUDA executable (launchable) graph

```
typedef cudaGraphicsResource *cudaGraphicsResource_t
```

CUDA graphics resource types

```
typedef struct CUgraphNode_st *cudaGraphNode_t
```

CUDA graph node.

```
typedef void (CUDART_CB *cudaHostFn_t) (void*  
userData)
```

CUDA host function

```
typedef cudaMipmappedArray
*cudaMipmappedArray_const_t
```

CUDA mipmapped array (as source argument)

```
typedef cudaMipmappedArray *cudaMipmappedArray_t
```

CUDA mipmapped array

```
typedef cudaOutputMode_t
```

CUDA output file modes

```
typedef struct CStream_st *cudaStream_t
```

CUDA stream

```
typedef unsigned long long cudaSurfaceObject_t
```

An opaque value that represents a CUDA Surface object

```
typedef unsigned long long cudaTextureObject_t
```

An opaque value that represents a CUDA texture object

```
#define CUDA_EGL_MAX_PLANES 3
```

Maximum number of planes per frame

```
#define CUDA_IPC_HANDLE_SIZE 64
```

CUDA IPC Handle Size

```
#define cudaArrayColorAttachment 0x20
```

Must be set in `cudaExternalMemoryGetMappedMipmappedArray` if the mipmapped array is used as a color target in a graphics API

```
#define cudaArrayCubemap 0x04
```

Must be set in `cudaMalloc3DArray` to create a cubemap CUDA array

## `#define cudaArrayDefault 0x00`

Default CUDA array allocation flag

## `#define cudaArrayLayered 0x01`

Must be set in `cudaMalloc3DArray` to create a layered CUDA array

## `#define cudaArraySurfaceLoadStore 0x02`

Must be set in `cudaMallocArray` or `cudaMalloc3DArray` in order to bind surfaces to the CUDA array

## `#define cudaArrayTextureGather 0x08`

Must be set in `cudaMallocArray` or `cudaMalloc3DArray` in order to perform texture gather operations on the CUDA array

## `#define cudaCooperativeLaunchMultiDeviceNoPostSync 0x02`

If set, any subsequent work pushed in a stream that participated in a call to `cudaLaunchCooperativeKernelMultiDevice` will only wait for the kernel launched on the GPU corresponding to that stream to complete before it begins execution.

## `#define cudaCooperativeLaunchMultiDeviceNoPreSync 0x01`

If set, each kernel launched as part of `cudaLaunchCooperativeKernelMultiDevice` only waits for prior work in the stream corresponding to that GPU to complete before the kernel begins execution.

## `#define cudaCpuDeviceId ((int)-1)`

Device id that represents the CPU

## `#define cudaDeviceBlockingSync 0x04`

**Deprecated** This flag was deprecated as of CUDA 4.0 and replaced with `cudaDeviceScheduleBlockingSync`.

Device flag - Use blocking synchronization

**#define cudaDeviceLmemResizeToMax 0x10**

Device flag - Keep local memory allocation after launch

**#define cudaDeviceMapHost 0x08**

Device flag - Support mapped pinned allocations

**#define cudaDeviceMask 0x1f**

Device flags mask

**#define cudaDevicePropDontCare**

Empty device properties

**#define cudaDeviceScheduleAuto 0x00**

Device flag - Automatic scheduling

**#define cudaDeviceScheduleBlockingSync 0x04**

Device flag - Use blocking synchronization

**#define cudaDeviceScheduleMask 0x07**

Device schedule flags mask

**#define cudaDeviceScheduleSpin 0x01**

Device flag - Spin default scheduling

**#define cudaDeviceScheduleYield 0x02**

Device flag - Yield default scheduling

**#define cudaEventBlockingSync 0x01**

Event uses blocking synchronization

**#define cudaEventDefault 0x00**

Default event flag

## **#define cudaEventDisableTiming 0x02**

Event will not record timing data

## **#define cudaEventInterprocess 0x04**

Event is suitable for interprocess use. `cudaEventDisableTiming` must be set

## **#define cudaExternalMemoryDedicated 0x1**

Indicates that the external memory object is a dedicated resource

## **#define cudaExternalSemaphoreSignalSkipNvSciBufMemSync 0x01**

When the `/p flags` parameter of `cudaExternalSemaphoreSignalParams` contains this flag, it indicates that signaling an external semaphore object should skip performing appropriate memory synchronization operations over all the external memory objects that are imported as `cudaExternalMemoryHandleTypeNvSciBuf`, which otherwise are performed by default to ensure data coherency with other importers of the same `NvSciBuf` memory objects.

## **#define cudaExternalSemaphoreWaitSkipNvSciBufMemSync 0x02**

When the `/p flags` parameter of `cudaExternalSemaphoreWaitParams` contains this flag, it indicates that waiting an external semaphore object should skip performing appropriate memory synchronization operations over all the external memory objects that are imported as `cudaExternalMemoryHandleTypeNvSciBuf`, which otherwise are performed by default to ensure data coherency with other importers of the same `NvSciBuf` memory objects.

## **#define cudaHostAllocDefault 0x00**

Default page-locked allocation flag

## **#define cudaHostAllocMapped 0x02**

Map allocation into device space



## **#define cudaHostAllocPortable 0x01**

Pinned memory accessible by all CUDA contexts

## **#define cudaHostAllocWriteCombined 0x04**

Write-combined memory

## **#define cudaHostRegisterDefault 0x00**

Default host memory registration flag

## **#define cudaHostRegisterIoMemory 0x04**

Memory-mapped I/O space

## **#define cudaHostRegisterMapped 0x02**

Map registered memory into device space

## **#define cudaHostRegisterPortable 0x01**

Pinned memory accessible by all CUDA contexts

## **#define cudaInvalidDeviceld ((int)-2)**

Device id that represents an invalid device

## **#define cudaIpcMemLazyEnablePeerAccess 0x01**

Automatically enable peer access between remote devices as needed

## **#define cudaMemAttachGlobal 0x01**

Memory can be accessed by any stream on any device

## **#define cudaMemAttachHost 0x02**

Memory cannot be accessed by any stream on any device

## **#define cudaMemAttachSingle 0x04**

Memory can only be accessed by a single stream on the associated device

## **#define cudaNvSciSyncAttrSignal 0x1**

When /p flags of `cudaDeviceGetNvSciSyncAttributes` is set to this, it indicates that application need signaler specific NvSciSyncAttr to be filled by `cudaDeviceGetNvSciSyncAttributes`.

## **#define cudaNvSciSyncAttrWait 0x2**

When /p flags of `cudaDeviceGetNvSciSyncAttributes` is set to this, it indicates that application need waiter specific NvSciSyncAttr to be filled by `cudaDeviceGetNvSciSyncAttributes`.

## **#define cudaOccupancyDefault 0x00**

Default behavior

## **#define cudaOccupancyDisableCachingOverride 0x01**

Assume global caching is enabled and cannot be automatically turned off

## **#define cudaPeerAccessDefault 0x00**

Default peer addressing enable flag

## **#define cudaStreamDefault 0x00**

Default stream flag

## **#define cudaStreamLegacy ((cudaStream\_t)0x1)**

Legacy stream handle

Stream handle that can be passed as a `cudaStream_t` to use an implicit stream with legacy synchronization behavior.

See details of the [synchronization behavior](#).

## **#define cudaStreamNonBlocking 0x01**

Stream does not synchronize with stream 0 (the NULL stream)

## **#define cudaStreamPerThread ((cudaStream\_t)0x2)**

Per-thread stream handle

Stream handle that can be passed as a `cudaStream_t` to use an implicit stream with per-thread synchronization behavior.

See details of the [synchronization behavior](#).

# Chapter 6.

## DATA STRUCTURES

Here are the data structures with brief descriptions:

`__cudaOccupancyB2DHelper`  
`cudaChannelFormatDesc`  
`cudaDeviceProp`  
`cudaEglFrame`  
`cudaEglPlaneDesc`  
`cudaExtent`  
`cudaExternalMemoryBufferDesc`  
`cudaExternalMemoryHandleDesc`  
`cudaExternalMemoryMipmappedArrayDesc`  
`cudaExternalSemaphoreHandleDesc`  
`cudaExternalSemaphoreSignalParams`  
`cudaExternalSemaphoreWaitParams`  
`cudaFuncAttributes`  
`cudaHostNodeParams`  
`cudaIpcEventHandle_t`  
`cudaIpcMemHandle_t`  
`cudaKernelNodeParams`  
`cudaLaunchParams`  
`cudaMemcpy3DParms`  
`cudaMemcpy3DPeerParms`  
`cudaMemsetParams`  
`cudaPitchedPtr`  
`cudaPointerAttributes`  
`cudaPos`  
`cudaResourceDesc`  
`cudaResourceViewDesc`  
`cudaTextureDesc`  
`cudaUUID_t`  
`surfaceReference`

`textureReference`

## 6.1. `__cudaOccupancyB2DHelper`

C++ API Routines `cppClassifierVisibility: visibility=public` `cppClassifierTemplateModel:`  
`=`

Helper functor for `cudaOccupancyMaxPotentialBlockSize`

## 6.2. `cudaChannelFormatDesc` Struct Reference

CUDA Channel format descriptor

`enum cudaChannelFormatKind`  
`cudaChannelFormatDesc::f`

Channel format kind

`int cudaChannelFormatDesc::w`

`w`

`int cudaChannelFormatDesc::x`

`x`

`int cudaChannelFormatDesc::y`

`y`

`int cudaChannelFormatDesc::z`

`z`

## 6.3. `cudaDeviceProp` Struct Reference

CUDA device properties

`int cudaDeviceProp::asyncEngineCount`

Number of asynchronous engines

## `int cudaDeviceProp::canMapHostMemory`

Device can map host memory with `cudaHostAlloc/cudaHostGetDevicePointer`

`int`

## `cudaDeviceProp::canUseHostPointerForRegisteredMem`

Device can access host registered memory at the same virtual address as the CPU

## `int cudaDeviceProp::clockRate`

Clock frequency in kilohertz

## `int cudaDeviceProp::computeMode`

Compute mode (See `cudaComputeMode`)

## `int cudaDeviceProp::computePreemptionSupported`

Device supports Compute Preemption

## `int cudaDeviceProp::concurrentKernels`

Device can possibly execute multiple kernels concurrently

## `int cudaDeviceProp::concurrentManagedAccess`

Device can coherently access managed memory concurrently with the CPU

## `int cudaDeviceProp::cooperativeLaunch`

Device supports launching cooperative kernels via `cudaLaunchCooperativeKernel`

## `int cudaDeviceProp::cooperativeMultiDeviceLaunch`

Device can participate in cooperative kernels launched via `cudaLaunchCooperativeKernelMultiDevice`

## `int cudaDeviceProp::deviceOverlap`

Device can concurrently copy memory and execute a kernel. Deprecated. Use instead `asyncEngineCount`.

## `int cudaDeviceProp::directManagedMemAccessFromHost`

Host can directly access managed memory on the device without migration.

## `int cudaDeviceProp::ECCEnabled`

Device has ECC support enabled

## `int cudaDeviceProp::globalL1CacheSupported`

Device supports caching globals in L1

## `int cudaDeviceProp::hostNativeAtomicSupported`

Link between the device and the host supports native atomic operations

## `int cudaDeviceProp::integrated`

Device is integrated as opposed to discrete

## `int cudaDeviceProp::isMultiGpuBoard`

Device is on a multi-GPU board

## `int cudaDeviceProp::kernelExecTimeoutEnabled`

Specified whether there is a run time limit on kernels

## `int cudaDeviceProp::l2CacheSize`

Size of L2 cache in bytes

## `int cudaDeviceProp::localL1CacheSupported`

Device supports caching locals in L1

## `char cudaDeviceProp::luid`

8-byte locally unique identifier. Value is undefined on TCC and non-Windows platforms

## `unsigned int cudaDeviceProp::luidDeviceNodeMask`

LUID device node mask. Value is undefined on TCC and non-Windows platforms

**int cudaDeviceProp::major**

Major compute capability

**int cudaDeviceProp::managedMemory**

Device supports allocating managed memory on this system

**int cudaDeviceProp::maxGridSize**

Maximum size of each dimension of a grid

**int cudaDeviceProp::maxSurface1D**

Maximum 1D surface size

**int cudaDeviceProp::maxSurface1DLayered**

Maximum 1D layered surface dimensions

**int cudaDeviceProp::maxSurface2D**

Maximum 2D surface dimensions

**int cudaDeviceProp::maxSurface2DLayered**

Maximum 2D layered surface dimensions

**int cudaDeviceProp::maxSurface3D**

Maximum 3D surface dimensions

**int cudaDeviceProp::maxSurfaceCubemap**

Maximum Cubemap surface dimensions

**int cudaDeviceProp::maxSurfaceCubemapLayered**

Maximum Cubemap layered surface dimensions

**int cudaDeviceProp::maxTexture1D**

Maximum 1D texture size



## `int cudaDeviceProp::maxTexture1DLayered`

Maximum 1D layered texture dimensions

## `int cudaDeviceProp::maxTexture1DLinear`

Maximum size for 1D textures bound to linear memory

## `int cudaDeviceProp::maxTexture1DMipmap`

Maximum 1D mipmapped texture size

## `int cudaDeviceProp::maxTexture2D`

Maximum 2D texture dimensions

## `int cudaDeviceProp::maxTexture2DGather`

Maximum 2D texture dimensions if texture gather operations have to be performed

## `int cudaDeviceProp::maxTexture2DLayered`

Maximum 2D layered texture dimensions

## `int cudaDeviceProp::maxTexture2DLinear`

Maximum dimensions (width, height, pitch) for 2D textures bound to pitched memory

## `int cudaDeviceProp::maxTexture2DMipmap`

Maximum 2D mipmapped texture dimensions

## `int cudaDeviceProp::maxTexture3D`

Maximum 3D texture dimensions

## `int cudaDeviceProp::maxTexture3DAlt`

Maximum alternate 3D texture dimensions

## `int cudaDeviceProp::maxTextureCubemap`

Maximum Cubemap texture dimensions

## `int cudaDeviceProp::maxTextureCubemapLayered`

Maximum Cubemap layered texture dimensions

## `int cudaDeviceProp::maxThreadsDim`

Maximum size of each dimension of a block

## `int cudaDeviceProp::maxThreadsPerBlock`

Maximum number of threads per block

## `int cudaDeviceProp::maxThreadsPerMultiProcessor`

Maximum resident threads per multiprocessor

## `int cudaDeviceProp::memoryBusWidth`

Global memory bus width in bits

## `int cudaDeviceProp::memoryClockRate`

Peak memory clock frequency in kilohertz

## `size_t cudaDeviceProp::memPitch`

Maximum pitch in bytes allowed by memory copies

## `int cudaDeviceProp::minor`

Minor compute capability

## `int cudaDeviceProp::multiGpuBoardGroupID`

Unique identifier for a group of devices on the same multi-GPU board

## `int cudaDeviceProp::multiProcessorCount`

Number of multiprocessors on device

## `char cudaDeviceProp::name`

ASCII string identifying device

## `int cudaDeviceProp::pageableMemoryAccess`

Device supports coherently accessing pageable memory without calling `cudaHostRegister` on it

`int`

## `cudaDeviceProp::pageableMemoryAccessUsesHostPageTables`

Device accesses pageable memory via the host's page tables

## `int cudaDeviceProp::pciBusID`

PCI bus ID of the device

## `int cudaDeviceProp::pciDeviceID`

PCI device ID of the device

## `int cudaDeviceProp::pciDomainID`

PCI domain ID of the device

## `int cudaDeviceProp::regsPerBlock`

32-bit registers available per block

## `int cudaDeviceProp::regsPerMultiprocessor`

32-bit registers available per multiprocessor

## `size_t cudaDeviceProp::sharedMemPerBlock`

Shared memory available per block in bytes

## `size_t cudaDeviceProp::sharedMemPerBlockOptin`

Per device maximum shared memory per block usable by special opt in

## `size_t cudaDeviceProp::sharedMemPerMultiprocessor`

Shared memory available per multiprocessor in bytes

## `int cudaDeviceProp::singleToDoublePrecisionPerfRatio`

Ratio of single precision performance (in floating-point operations per second) to double precision performance

## `int cudaDeviceProp::streamPrioritiesSupported`

Device supports stream priorities

## `size_t cudaDeviceProp::surfaceAlignment`

Alignment requirements for surfaces

## `int cudaDeviceProp::tccDriver`

1 if device is a Tesla device using TCC driver, 0 otherwise

## `size_t cudaDeviceProp::textureAlignment`

Alignment requirement for textures

## `size_t cudaDeviceProp::texturePitchAlignment`

Pitch alignment requirement for texture references bound to pitched memory

## `size_t cudaDeviceProp::totalConstMem`

Constant memory available on device in bytes

## `size_t cudaDeviceProp::totalGlobalMem`

Global memory available on device in bytes

## `int cudaDeviceProp::unifiedAddressing`

Device shares a unified address space with the host

## `cudaUUID_t cudaDeviceProp::uuid`

16-byte unique identifier

## `int cudaDeviceProp::warpSize`

Warp size in threads

## 6.4. cudaEglFrame Struct Reference

CUDA EglFrame Descriptor - structure defining one frame of EGL.

Each frame may contain one or more planes depending on whether the surface is Multiplanar or not. Each plane of EglFrame is represented by [cudaEglPlaneDesc](#) which is defined as:

```
typedef struct cudaEglPlaneDesc_st {
    unsigned int width;
    unsigned int height;
    unsigned int depth;
    unsigned int pitch;
    unsigned int numChannels;
    struct cudaChannelFormatDesc channelDesc;
    unsigned int reserved[4];
} cudaEglPlaneDesc;
```

### [cudaEglColorFormat](#) [cudaEglFrame::eglColorFormat](#)

CUDA EGL Color Format

### [cudaEglFrameType](#) [cudaEglFrame::frameType](#)

Array or Pitch

### [cudaArray\\_t](#) [cudaEglFrame::pArray](#)

Array of CUDA arrays corresponding to each plane

### [unsigned int](#) [cudaEglFrame::planeCount](#)

Number of planes

### [struct cudaEglPlaneDesc](#) [cudaEglFrame::planeDesc](#)

CUDA EGL Plane Descriptor [cudaEglPlaneDesc](#)

### [struct cudaPitchedPtr](#) [cudaEglFrame::pPitch](#)

Array of Pointers corresponding to each plane

## 6.5. cudaEglPlaneDesc Struct Reference

CUDA EGL Plane Descriptor - structure defining each plane of a CUDA EglFrame

```
struct cudaChannelFormatDesc
cudaEglPlaneDesc::channelDesc
```

Channel Format Descriptor

```
unsigned int cudaEglPlaneDesc::depth
```

Depth of plane

```
unsigned int cudaEglPlaneDesc::height
```

Height of plane

```
unsigned int cudaEglPlaneDesc::numChannels
```

Number of channels for the plane

```
unsigned int cudaEglPlaneDesc::pitch
```

Pitch of plane

```
unsigned int cudaEglPlaneDesc::reserved
```

Reserved for future use

```
unsigned int cudaEglPlaneDesc::width
```

Width of plane

## 6.6. cudaExtent Struct Reference

CUDA extent

**See also:**

[make\\_cudaExtent](#)

```
size_t cudaExtent::depth
```

Depth in elements

```
size_t cudaExtent::height
```

Height in elements

## `size_t cudaExtent::width`

Width in elements when referring to array memory, in bytes when referring to linear memory

## 6.7. `cudaExternalMemoryBufferDesc` Struct Reference

External memory buffer descriptor

### `unsigned int cudaExternalMemoryBufferDesc::flags`

Flags reserved for future use. Must be zero.

### `unsigned long long cudaExternalMemoryBufferDesc::offset`

Offset into the memory object where the buffer's base is

### `unsigned long long cudaExternalMemoryBufferDesc::size`

Size of the buffer

## 6.8. `cudaExternalMemoryHandleDesc` Struct Reference

External memory handle descriptor

### `int cudaExternalMemoryHandleDesc::fd`

File descriptor referencing the memory object. Valid when type is `cudaExternalMemoryHandleTypeOpaqueFd`

### `unsigned int cudaExternalMemoryHandleDesc::flags`

Flags must either be zero or `cudaExternalMemoryDedicated`

### `void *cudaExternalMemoryHandleDesc::handle`

Valid NT handle. Must be NULL if 'name' is non-NULL

**const void \*cudaExternalMemoryHandleDesc::name**

Name of a valid memory object. Must be NULL if 'handle' is non-NULL.

**const void**

**\*cudaExternalMemoryHandleDesc::nvSciBufObject**

A handle representing NvSciBuf Object. Valid when type is `cudaExternalMemoryHandleTypeNvSciBuf`

**unsigned long long**

**cudaExternalMemoryHandleDesc::size**

Size of the memory allocation

**enumcudaExternalMemoryHandleType**

**cudaExternalMemoryHandleDesc::type**

Type of the handle

**cudaExternalMemoryHandleDesc::@6::@7**

**cudaExternalMemoryHandleDesc::win32**

Win32 handle referencing the semaphore object. Valid when type is one of the following:

- ▶ `cudaExternalMemoryHandleTypeOpaqueWin32`
- ▶ `cudaExternalMemoryHandleTypeOpaqueWin32Kmt`
- ▶ `cudaExternalMemoryHandleTypeD3D12Heap`
- ▶ `cudaExternalMemoryHandleTypeD3D12Resource`
- ▶ `cudaExternalMemoryHandleTypeD3D11Resource`
- ▶ `cudaExternalMemoryHandleTypeD3D11ResourceKmt` Exactly one of 'handle' and 'name' must be non-NULL. If type is one of the following: `cudaExternalMemoryHandleTypeOpaqueWin32Kmt` `cudaExternalMemoryHandleTypeD3D11ResourceKmt` then 'name' must be NULL.

## 6.9. cudaExternalMemoryMipmappedArrayDesc Struct Reference

External memory mipmap descriptor



```
struct cudaExtent
cudaExternalMemoryMipmappedArrayDesc::extent
```

Dimensions of base level of the mipmap chain

```
unsigned int
cudaExternalMemoryMipmappedArrayDesc::flags
```

Flags associated with CUDA mipmapped arrays. See [cudaMallocMipmappedArray](#)

```
struct cudaChannelFormatDesc
cudaExternalMemoryMipmappedArrayDesc::formatDesc
```

Format of base level of the mipmap chain

```
unsigned int
cudaExternalMemoryMipmappedArrayDesc::numLevels
```

Total number of levels in the mipmap chain

```
unsigned long long
cudaExternalMemoryMipmappedArrayDesc::offset
```

Offset into the memory object where the base level of the mipmap chain is.

## 6.10. cudaExternalSemaphoreHandleDesc Struct Reference

External semaphore handle descriptor

```
int cudaExternalSemaphoreHandleDesc::fd
```

File descriptor referencing the semaphore object. Valid when type is [cudaExternalSemaphoreHandleTypeOpaqueFd](#)

```
unsigned int cudaExternalSemaphoreHandleDesc::flags
```

Flags reserved for the future. Must be zero.

**void \*cudaExternalSemaphoreHandleDesc::handle**

Valid NT handle. Must be NULL if 'name' is non-NULL

**const void \*cudaExternalSemaphoreHandleDesc::name**

Name of a valid synchronization primitive. Must be NULL if 'handle' is non-NULL.

**const void**

**\*cudaExternalSemaphoreHandleDesc::nvSciSyncObj**

Valid NvSciSyncObj. Must be non NULL

**enum cudaExternalSemaphoreHandleType  
cudaExternalSemaphoreHandleDesc::type**

Type of the handle

**cudaExternalSemaphoreHandleDesc::@8::@9  
cudaExternalSemaphoreHandleDesc::win32**

Win32 handle referencing the semaphore object. Valid when type is one of the following:

- ▶ `cudaExternalSemaphoreHandleTypeOpaqueWin32`
- ▶ `cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt`
- ▶ `cudaExternalSemaphoreHandleTypeD3D12Fence`
- ▶ `cudaExternalSemaphoreHandleTypeD3D11Fence`
- ▶ `cudaExternalSemaphoreHandleTypeKeyedMutex` Exactly one of 'handle' and 'name' must be non-NULL. If type is one of the following: `cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt` `cudaExternalSemaphoreHandleTypeKeyedMutexKmt` then 'name' must be NULL.

## 6.11. cudaExternalSemaphoreSignalParams Struct Reference

External semaphore signal parameters

**void \*cudaExternalSemaphoreSignalParams::fence**

Pointer to NvSciSyncFence. Valid if `cudaExternalSemaphoreHandleType` is of type `cudaExternalSemaphoreHandleTypeNvSciSync`.

**cudaExternalSemaphoreSignalParams::@10::@11**  
**cudaExternalSemaphoreSignalParams::fence**

Parameters for fence objects

**unsigned int cudaExternalSemaphoreSignalParams::flags**

Only when `cudaExternalSemaphoreSignalParams` is used to signal a `cudaExternalSemaphore_t` of type `cudaExternalSemaphoreHandleTypeNvSciSync`, the valid flag is `cudaExternalSemaphoreSignalSkipNvSciBufMemSync`: which indicates that while signaling the `cudaExternalSemaphore_t`, no memory synchronization operations should be performed for any external memory object imported as `cudaExternalMemoryHandleTypeNvSciBuf`. For all other types of `cudaExternalSemaphore_t`, flags must be zero.

**cudaExternalSemaphoreSignalParams::@10::@13**  
**cudaExternalSemaphoreSignalParams::keyedMutex**

Parameters for keyed mutex objects

**unsigned long long**  
**cudaExternalSemaphoreSignalParams::value**

Value of fence to be signaled

## 6.12. cudaExternalSemaphoreWaitParams Struct Reference

External semaphore wait parameters

**void \*cudaExternalSemaphoreWaitParams::fence**

Pointer to `NvSciSyncFence`. Valid if `cudaExternalSemaphoreHandleType` is of type `cudaExternalSemaphoreHandleTypeNvSciSync`.

**cudaExternalSemaphoreWaitParams::@14::@15**  
**cudaExternalSemaphoreWaitParams::fence**

Parameters for fence objects

## unsigned int cudaExternalSemaphoreWaitParams::flags

Only when `cudaExternalSemaphoreSignalParams` is used to signal a `cudaExternalSemaphore_t` of type `cudaExternalSemaphoreHandleTypeNvSciSync`, the valid flag is `cudaExternalSemaphoreSignalSkipNvSciBufMemSync`: which indicates that while waiting for the `cudaExternalSemaphore_t`, no memory synchronization operations should be performed for any external memory object imported as `cudaExternalMemoryHandleTypeNvSciBuf`. For all other types of `cudaExternalSemaphore_t`, flags must be zero.

## unsigned long long cudaExternalSemaphoreWaitParams::key

Value of key to acquire the mutex with

## cudaExternalSemaphoreWaitParams::@14::@17 cudaExternalSemaphoreWaitParams::keyedMutex

Parameters for keyed mutex objects

## unsigned int cudaExternalSemaphoreWaitParams::timeoutMs

Timeout in milliseconds to wait to acquire the mutex

## unsigned long long cudaExternalSemaphoreWaitParams::value

Value of fence to be waited on

## 6.13. cudaFuncAttributes Struct Reference

CUDA function attributes

### int cudaFuncAttributes::binaryVersion

The binary architecture version for which the function was compiled. This value is the major binary version \* 10 + the minor binary version, so a binary version 1.3 function would return the value 13.

## `int cudaFuncAttributes::cacheModeCA`

The attribute to indicate whether the function has been compiled with user specified option "-Xptxas --dlcm=ca" set.

## `size_t cudaFuncAttributes::constSizeBytes`

The size in bytes of user-allocated constant memory required by this function.

## `size_t cudaFuncAttributes::localSizeBytes`

The size in bytes of local memory used by each thread of this function.

## `int cudaFuncAttributes::maxDynamicSharedSizeBytes`

The maximum size in bytes of dynamic shared memory per block for this function. Any launch must have a dynamic shared memory size smaller than this value.

## `int cudaFuncAttributes::maxThreadsPerBlock`

The maximum number of threads per block, beyond which a launch of the function would fail. This number depends on both the function and the device on which the function is currently loaded.

## `int cudaFuncAttributes::numRegs`

The number of registers used by each thread of this function.

## `int cudaFuncAttributes::preferredShmemCarveout`

On devices where the L1 cache and shared memory use the same hardware resources, this sets the shared memory carveout preference, in percent of the maximum shared memory. Refer to [cudaDevAttrMaxSharedMemoryPerMultiprocessor](#). This is only a hint, and the driver can choose a different ratio if required to execute the function. See [cudaFuncSetAttribute](#)

## `int cudaFuncAttributes::ptxVersion`

The PTX virtual architecture version for which the function was compiled. This value is the major PTX version \* 10 + the minor PTX version, so a PTX version 1.3 function would return the value 13.

## `size_t cudaFuncAttributes::sharedSizeBytes`

The size in bytes of statically-allocated shared memory per block required by this function. This does not include dynamically-allocated shared memory requested by the user at runtime.

## 6.14. `cudaHostNodeParams` Struct Reference

CUDA host node parameters

### `cudaHostFn_t cudaHostNodeParams::fn`

The function to call when the node executes

### `void *cudaHostNodeParams::userData`

Argument to pass to the function

## 6.15. `cudaIpcEventHandle_t` Struct Reference

CUDA IPC event handle

## 6.16. `cudaIpcMemHandle_t` Struct Reference

CUDA IPC memory handle

## 6.17. `cudaKernelNodeParams` Struct Reference

CUDA GPU kernel node parameters

### `dim3 cudaKernelNodeParams::blockDim`

Block dimensions

### `**cudaKernelNodeParams::extra`

Pointer to kernel arguments in the "extra" format

**void \*cudaKernelNodeParams::func**

Kernel to launch

**dim3 cudaKernelNodeParams::gridDim**

Grid dimensions

**\*\*cudaKernelNodeParams::kernelParams**

Array of pointers to individual kernel arguments

**unsigned int cudaKernelNodeParams::sharedMemBytes**

Dynamic shared-memory size per thread block in bytes

## 6.18. cudaLaunchParams Struct Reference

CUDA launch parameters

**\*\*cudaLaunchParams::args**

Arguments

**dim3 cudaLaunchParams::blockDim**

Block dimensions

**void \*cudaLaunchParams::func**

Device function symbol

**dim3 cudaLaunchParams::gridDim**

Grid dimensions

**size\_t cudaLaunchParams::sharedMem**

Shared memory

**cudaStream\_t cudaLaunchParams::stream**

Stream identifier

## 6.19. cudaMemcpy3DParms Struct Reference

CUDA 3D memory copying parameters

`cudaArray_t cudaMemcpy3DParms::dstArray`

Destination memory address

`struct cudaPos cudaMemcpy3DParms::dstPos`

Destination position offset

`struct cudaPitchedPtr cudaMemcpy3DParms::dstPtr`

Pitched destination memory address

`struct cudaExtent cudaMemcpy3DParms::extent`

Requested memory copy size

`enum cudaMemcpyKind cudaMemcpy3DParms::kind`

Type of transfer

`cudaArray_t cudaMemcpy3DParms::srcArray`

Source memory address

`struct cudaPos cudaMemcpy3DParms::srcPos`

Source position offset

`struct cudaPitchedPtr cudaMemcpy3DParms::srcPtr`

Pitched source memory address

## 6.20. cudaMemcpy3DPeerParms Struct Reference

CUDA 3D cross-device memory copying parameters



**cudaArray\_t cudaMemcpy3DPeerParms::dstArray**

Destination memory address

**int cudaMemcpy3DPeerParms::dstDevice**

Destination device

**struct cudaPos cudaMemcpy3DPeerParms::dstPos**

Destination position offset

**struct cudaPitchedPtr cudaMemcpy3DPeerParms::dstPtr**

Pitched destination memory address

**struct cudaExtent cudaMemcpy3DPeerParms::extent**

Requested memory copy size

**cudaArray\_t cudaMemcpy3DPeerParms::srcArray**

Source memory address

**int cudaMemcpy3DPeerParms::srcDevice**

Source device

**struct cudaPos cudaMemcpy3DPeerParms::srcPos**

Source position offset

**struct cudaPitchedPtr cudaMemcpy3DPeerParms::srcPtr**

Pitched source memory address

## 6.21. cudaMemcpyParams Struct Reference

CUDA Memset node parameters

**void \*cudaMemcpyParams::dst**

Destination device pointer

## `unsigned int cudaMemsetParams::elementSize`

Size of each element in bytes. Must be 1, 2, or 4.

## `size_t cudaMemsetParams::height`

Number of rows

## `size_t cudaMemsetParams::pitch`

Pitch of destination device pointer. Unused if height is 1

## `unsigned int cudaMemsetParams::value`

Value to be set

## `size_t cudaMemsetParams::width`

Width in bytes, of the row

## 6.22. `cudaPitchedPtr` Struct Reference

CUDA Pitched memory pointer

**See also:**

`make_cudaPitchedPtr`

## `size_t cudaPitchedPtr::pitch`

Pitch of allocated memory in bytes

## `void *cudaPitchedPtr::ptr`

Pointer to allocated memory

## `size_t cudaPitchedPtr::xsize`

Logical width of allocation in elements

## `size_t cudaPitchedPtr::ysize`

Logical height of allocation in elements

## 6.23. cudaPointerAttributes Struct Reference

CUDA pointer attributes

### int cudaPointerAttributes::device

The device against which the memory was allocated or registered. If the memory type is `cudaMemoryTypeDevice` then this identifies the device on which the memory referred physically resides. If the memory type is `cudaMemoryTypeHost` or `cudaMemoryTypeManaged` then this identifies the device which was current when the memory was allocated or registered (and if that device is deinitialized then this allocation will vanish with that device's state).

### void \*cudaPointerAttributes::devicePointer

The address which may be dereferenced on the current device to access the memory or NULL if no such address exists.

### void \*cudaPointerAttributes::hostPointer

The address which may be dereferenced on the host to access the memory or NULL if no such address exists.



CUDA doesn't check if unregistered memory is allocated so this field may contain invalid pointer if an invalid pointer has been passed to CUDA.

### `__CUDA_DEPRECATED` int cudaPointerAttributes::isManaged

Deprecated

Indicates if this pointer points to managed memory

### `__CUDA_DEPRECATED` enumcudaMemoryType cudaPointerAttributes::memoryType

Deprecated

The physical location of the memory, `cudaMemoryTypeHost` or `cudaMemoryTypeDevice`. Note that managed memory can return either `cudaMemoryTypeDevice` or `cudaMemoryTypeHost` regardless of it's physical location.

## enum cudaMemoryType cudaPointerAttributes::type

The type of memory - `cudaMemoryTypeUnregistered`, `cudaMemoryTypeHost`, `cudaMemoryTypeDevice` or `cudaMemoryTypeManaged`.

## 6.24. cudaPos Struct Reference

CUDA 3D position

**See also:**

`make_cudaPos`

`size_t cudaPos::x`

x

`size_t cudaPos::y`

y

`size_t cudaPos::z`

z

## 6.25. cudaResourceDesc Struct Reference

CUDA resource descriptor

`cudaArray_t cudaResourceDesc::array`

CUDA array

`struct cudaChannelFormatDesc cudaResourceDesc::desc`

Channel descriptor

`void *cudaResourceDesc::devPtr`

Device pointer

**size\_t cudaResourceDesc::height**

Height of the array in elements

**cudaMipmappedArray\_t cudaResourceDesc::mipmap**

CUDA mipmapped array

**size\_t cudaResourceDesc::pitchInBytes**

Pitch between two rows in bytes

**enumcudaResourceType cudaResourceDesc::resType**

Resource type

**size\_t cudaResourceDesc::sizeInBytes**

Size in bytes

**size\_t cudaResourceDesc::width**

Width of the array in elements

## 6.26. cudaResourceViewDesc Struct Reference

CUDA resource view descriptor

**size\_t cudaResourceViewDesc::depth**

Depth of the resource view

**unsigned int cudaResourceViewDesc::firstLayer**

First layer index

**unsigned int cudaResourceViewDesc::firstMipmapLevel**

First defined mipmap level

`enum cudaResourceViewFormat`  
`cudaResourceViewDesc::format`

Resource view format

`size_t cudaResourceViewDesc::height`

Height of the resource view

`unsigned int cudaResourceViewDesc::lastLayer`

Last layer index

`unsigned int cudaResourceViewDesc::lastMipmapLevel`

Last defined mipmap level

`size_t cudaResourceViewDesc::width`

Width of the resource view

## 6.27. cudaTextureDesc Struct Reference

CUDA texture descriptor

`enum cudaTextureAddressMode`  
`cudaTextureDesc::addressMode`

Texture address mode for up to 3 dimensions

`float cudaTextureDesc::borderColor`

Texture Border Color

`enum cudaTextureFilterMode`  
`cudaTextureDesc::filterMode`

Texture filter mode

`unsigned int cudaTextureDesc::maxAnisotropy`

Limit to the anisotropy ratio

## float cudaTextureDesc::maxMipmapLevelClamp

Upper end of the mipmap level range to clamp access to

## float cudaTextureDesc::minMipmapLevelClamp

Lower end of the mipmap level range to clamp access to

## enum cudaTextureFilterMode cudaTextureDesc::mipmapFilterMode

Mipmap filter mode

## float cudaTextureDesc::mipmapLevelBias

Offset applied to the supplied mipmap level

## int cudaTextureDesc::normalizedCoords

Indicates whether texture reads are normalized or not

## enum cudaTextureReadMode cudaTextureDesc::readMode

Texture read mode

## int cudaTextureDesc::sRGB

Perform sRGB->linear conversion during texture read

## 6.28. CUuuid\_st Struct Reference

CUDA UUID types

### char CUuuid\_st::bytes

< CUDA definition of UUID

## 6.29. surfaceReference Struct Reference

CUDA Surface reference

```
struct cudaChannelFormatDesc  
surfaceReference::channelDesc
```

Channel descriptor for surface reference

## 6.30. textureReference Struct Reference

CUDA texture reference

```
enumcudaTextureAddressMode  
textureReference::addressMode
```

Texture address mode for up to 3 dimensions

```
struct cudaChannelFormatDesc  
textureReference::channelDesc
```

Channel descriptor for the texture reference

```
enumcudaTextureFilterMode  
textureReference::filterMode
```

Texture filter mode

```
unsigned int textureReference::maxAnisotropy
```

Limit to the anisotropy ratio

```
float textureReference::maxMipmapLevelClamp
```

Upper end of the mipmap level range to clamp access to

```
float textureReference::minMipmapLevelClamp
```

Lower end of the mipmap level range to clamp access to

```
enumcudaTextureFilterMode  
textureReference::mipmapFilterMode
```

Mipmap filter mode



## `float textureReference::mipmapLevelBias`

Offset applied to the supplied mipmap level

## `int textureReference::normalized`

Indicates whether texture reads are normalized or not

## `int textureReference::sRGB`

Perform sRGB->linear conversion during texture read

# Chapter 7.

## DATA FIELDS

Here is a list of all documented struct and union fields with links to the struct/union documentation for each field:

### A

#### **addressMode**

- [textureReference](#)
- [cudaTextureDesc](#)

#### **args**

- [cudaLaunchParams](#)

#### **array**

- [cudaResourceDesc](#)

#### **asyncEngineCount**

- [cudaDeviceProp](#)

### B

#### **binaryVersion**

- [cudaFuncAttributes](#)

#### **blockDim**

- [cudaLaunchParams](#)
- [cudaKernelNodeParams](#)

#### **borderColor**

- [cudaTextureDesc](#)

#### **bytes**

- [cudaUUID\\_t](#)

### C

#### **cacheModeCA**

- [cudaFuncAttributes](#)

#### **canMapHostMemory**

- [cudaDeviceProp](#)

**canUseHostPointerForRegisteredMem**

cudaDeviceProp

**channelDesc**

cudaEglPlaneDesc

surfaceReference

textureReference

**clockRate**

cudaDeviceProp

**computeMode**

cudaDeviceProp

**computePreemptionSupported**

cudaDeviceProp

**concurrentKernels**

cudaDeviceProp

**concurrentManagedAccess**

cudaDeviceProp

**constSizeBytes**

cudaFuncAttributes

**cooperativeLaunch**

cudaDeviceProp

**cooperativeMultiDeviceLaunch**

cudaDeviceProp

**D****depth**

cudaEglPlaneDesc

cudaExtent

cudaResourceViewDesc

**desc**

cudaResourceDesc

**device**

cudaPointerAttributes

**deviceOverlap**

cudaDeviceProp

**devicePointer**

cudaPointerAttributes

**devPtr**

cudaResourceDesc

**directManagedMemAccessFromHost**

cudaDeviceProp

**dst**

cudaMemsetParams

**dstArray**

cudaMemcpy3DParms  
 cudaMemcpy3DPeerParms

**dstDevice**

cudaMemcpy3DPeerParms

**dstPos**

cudaMemcpy3DPeerParms  
 cudaMemcpy3DParms

**dstPtr**

cudaMemcpy3DParms  
 cudaMemcpy3DPeerParms

**E****ECCEnabled**

cudaDeviceProp

**eglColorFormat**

cudaEglFrame

**elementSize**

cudaMemsetParams

**extent**

cudaMemcpy3DPeerParms  
 cudaMemcpy3DParms  
 cudaExternalMemoryMipmappedArrayDesc

**extra**

cudaKernelNodeParams

**F****f**

cudaChannelFormatDesc

**fd**

cudaExternalMemoryHandleDesc  
 cudaExternalSemaphoreHandleDesc

**fence**

cudaExternalSemaphoreSignalParams  
 cudaExternalSemaphoreWaitParams

**filterMode**

textureReference  
 cudaTextureDesc

**firstLayer**

cudaResourceViewDesc

**firstMipmapLevel**

cudaResourceViewDesc

**flags**

- cudaExternalSemaphoreWaitParams
- cudaExternalSemaphoreSignalParams
- cudaExternalSemaphoreHandleDesc
- cudaExternalMemoryHandleDesc
- cudaExternalMemoryBufferDesc
- cudaExternalMemoryMipmappedArrayDesc

**fn**

- cudaHostNodeParams

**format**

- cudaResourceViewDesc

**formatDesc**

- cudaExternalMemoryMipmappedArrayDesc

**frameType**

- cudaEglFrame

**func**

- cudaKernelNodeParams
- cudaLaunchParams

**G****globalL1CacheSupported**

- cudaDeviceProp

**gridDim**

- cudaKernelNodeParams
- cudaLaunchParams

**H****handle**

- cudaExternalMemoryHandleDesc
- cudaExternalSemaphoreHandleDesc

**height**

- cudaExtent
- cudaResourceViewDesc
- cudaResourceDesc
- cudaMemsetParams
- cudaEglPlaneDesc

**hostNativeAtomicSupported**

- cudaDeviceProp

**hostPointer**

- cudaPointerAttributes

**I****integrated**

cudaDeviceProp

**isManaged**

cudaPointerAttributes

**isMultiGpuBoard**

cudaDeviceProp

**K****kernelExecTimeoutEnabled**

cudaDeviceProp

**kernelParams**

cudaKernelNodeParams

**key**

cudaExternalSemaphoreWaitParams

**keyedMutex**

cudaExternalSemaphoreSignalParams

cudaExternalSemaphoreWaitParams

**kind**

cudaMemcpy3DParms

**L****l2CacheSize**

cudaDeviceProp

**lastLayer**

cudaResourceViewDesc

**lastMipmapLevel**

cudaResourceViewDesc

**localL1CacheSupported**

cudaDeviceProp

**localSizeBytes**

cudaFuncAttributes

**luid**

cudaDeviceProp

**luidDeviceNodeMask**

cudaDeviceProp

**M****major**

cudaDeviceProp

**managedMemory**

cudaDeviceProp

**maxAnisotropy**  
    cudaTextureDesc  
    textureReference

**maxDynamicSharedSizeBytes**  
    cudaFuncAttributes

**maxGridSize**  
    cudaDeviceProp

**maxMipmapLevelClamp**  
    cudaTextureDesc  
    textureReference

**maxSurface1D**  
    cudaDeviceProp

**maxSurface1DLayered**  
    cudaDeviceProp

**maxSurface2D**  
    cudaDeviceProp

**maxSurface2DLayered**  
    cudaDeviceProp

**maxSurface3D**  
    cudaDeviceProp

**maxSurfaceCubemap**  
    cudaDeviceProp

**maxSurfaceCubemapLayered**  
    cudaDeviceProp

**maxTexture1D**  
    cudaDeviceProp

**maxTexture1DLayered**  
    cudaDeviceProp

**maxTexture1DLinear**  
    cudaDeviceProp

**maxTexture1DMipmap**  
    cudaDeviceProp

**maxTexture2D**  
    cudaDeviceProp

**maxTexture2DGather**  
    cudaDeviceProp

**maxTexture2DLayered**  
    cudaDeviceProp

**maxTexture2DLinear**  
    cudaDeviceProp

**maxTexture2DMipmap**  
    cudaDeviceProp

**maxTexture3D**  
    cudaDeviceProp

**maxTexture3DAlt**  
    cudaDeviceProp

**maxTextureCubemap**  
    cudaDeviceProp

**maxTextureCubemapLayered**  
    cudaDeviceProp

**maxThreadsDim**  
    cudaDeviceProp

**maxThreadsPerBlock**  
    cudaDeviceProp  
    cudaFuncAttributes

**maxThreadsPerMultiProcessor**  
    cudaDeviceProp

**memoryBusWidth**  
    cudaDeviceProp

**memoryClockRate**  
    cudaDeviceProp

**memoryType**  
    cudaPointerAttributes

**memPitch**  
    cudaDeviceProp

**minMipmapLevelClamp**  
    cudaTextureDesc  
    textureReference

**minor**  
    cudaDeviceProp

**mipmap**  
    cudaResourceDesc

**mipmapFilterMode**  
    textureReference  
    cudaTextureDesc

**mipmapLevelBias**  
    textureReference  
    cudaTextureDesc

**multiGpuBoardGroupID**  
    cudaDeviceProp

**multiProcessorCount**  
    cudaDeviceProp



**N****name**

[cudaDeviceProp](#)  
[cudaExternalMemoryHandleDesc](#)  
[cudaExternalSemaphoreHandleDesc](#)

**normalized**

[textureReference](#)

**normalizedCoords**

[cudaTextureDesc](#)

**numChannels**

[cudaEglPlaneDesc](#)

**numLevels**

[cudaExternalMemoryMipmappedArrayDesc](#)

**numRegs**

[cudaFuncAttributes](#)

**nvSciBufObject**

[cudaExternalMemoryHandleDesc](#)

**nvSciSyncObj**

[cudaExternalSemaphoreHandleDesc](#)

**O****offset**

[cudaExternalMemoryBufferDesc](#)  
[cudaExternalMemoryMipmappedArrayDesc](#)

**P****pageableMemoryAccess**

[cudaDeviceProp](#)

**pageableMemoryAccessUsesHostPageTables**

[cudaDeviceProp](#)

**pArray**

[cudaEglFrame](#)

**pciBusID**

[cudaDeviceProp](#)

**pciDeviceID**

[cudaDeviceProp](#)

**pciDomainID**

[cudaDeviceProp](#)

**pitch**

[cudaPitchedPtr](#)  
[cudaMemsetParams](#)  
[cudaEglPlaneDesc](#)

**pitchInBytes**

[cudaResourceDesc](#)

**planeCount**

[cudaEglFrame](#)

**planeDesc**

[cudaEglFrame](#)

**pPitch**

[cudaEglFrame](#)

**preferredShmemCarveout**

[cudaFuncAttributes](#)

**ptr**

[cudaPitchedPtr](#)

**ptxVersion**

[cudaFuncAttributes](#)

**R****readMode**

[cudaTextureDesc](#)

**regsPerBlock**

[cudaDeviceProp](#)

**regsPerMultiprocessor**

[cudaDeviceProp](#)

**reserved**

[cudaEglPlaneDesc](#)

**resType**

[cudaResourceDesc](#)

**S****sharedMem**

[cudaLaunchParams](#)

**sharedMemBytes**

[cudaKernelNodeParams](#)

**sharedMemPerBlock**

[cudaDeviceProp](#)

**sharedMemPerBlockOptin**

[cudaDeviceProp](#)

**sharedMemPerMultiprocessor**

[cudaDeviceProp](#)

**sharedSizeBytes**

[cudaFuncAttributes](#)

**singleToDoublePrecisionPerfRatio**

[cudaDeviceProp](#)

**size**  
     [cudaExternalMemoryHandleDesc](#)  
     [cudaExternalMemoryBufferDesc](#)  
**sizeInBytes**  
     [cudaResourceDesc](#)  
**srcArray**  
     [cudaMemcpy3DParms](#)  
     [cudaMemcpy3DPeerParms](#)  
**srcDevice**  
     [cudaMemcpy3DPeerParms](#)  
**srcPos**  
     [cudaMemcpy3DPeerParms](#)  
     [cudaMemcpy3DParms](#)  
**srcPtr**  
     [cudaMemcpy3DParms](#)  
     [cudaMemcpy3DPeerParms](#)  
**sRGB**  
     [cudaTextureDesc](#)  
     [textureReference](#)  
**stream**  
     [cudaLaunchParams](#)  
**streamPrioritiesSupported**  
     [cudaDeviceProp](#)  
**surfaceAlignment**  
     [cudaDeviceProp](#)

**T**  
**tccDriver**  
     [cudaDeviceProp](#)  
**textureAlignment**  
     [cudaDeviceProp](#)  
**texturePitchAlignment**  
     [cudaDeviceProp](#)  
**timeoutMs**  
     [cudaExternalSemaphoreWaitParams](#)  
**totalConstMem**  
     [cudaDeviceProp](#)  
**totalGlobalMem**  
     [cudaDeviceProp](#)  
**type**  
     [cudaExternalSemaphoreHandleDesc](#)  
     [cudaPointerAttributes](#)  
     [cudaExternalMemoryHandleDesc](#)

**U****unifiedAddressing**

cudaDeviceProp

**userData**

cudaHostNodeParams

**uuid**

cudaDeviceProp

**V****value**

cudaMemsetParams

cudaExternalSemaphoreWaitParams

cudaExternalSemaphoreSignalParams

**W****w**

cudaChannelFormatDesc

**warpSize**

cudaDeviceProp

**width**

cudaExtent

cudaResourceViewDesc

cudaResourceDesc

cudaMemsetParams

cudaEglPlaneDesc

**win32**

cudaExternalSemaphoreHandleDesc

cudaExternalMemoryHandleDesc

**X****x**

cudaChannelFormatDesc

cudaPos

**xsize**

cudaPitchedPtr

**Y****y**

cudaChannelFormatDesc

cudaPos

**ysize**

cudaPitchedPtr

**Z****z**`cudaChannelFormatDesc``cudaPos`

## Chapter 8.

# DEPRECATED LIST

Global `cudaPointerAttributes::isManaged`

Global `cudaPointerAttributes::memoryType`

Global `cudaThreadExit`

Global `cudaThreadGetCacheConfig`

Global `cudaThreadGetLimit`

Global `cudaThreadSetCacheConfig`

Global `cudaThreadSetLimit`

Global `cudaThreadSynchronize`

Global `cudaSetDoubleForDevice`

This function is deprecated as of CUDA 7.5

**Global `cudaSetDoubleForHost`**

This function is deprecated as of CUDA 7.5

**Global `cudaMemcpyArrayToArray`****Global `cudaMemcpyFromArray`****Global `cudaMemcpyFromArrayAsync`****Global `cudaMemcpyToArray`****Global `cudaMemcpyToArrayAsync`****Global `cudaGLMapBufferObject`**

This function is deprecated as of CUDA 3.0.

**Global `cudaGLMapBufferObjectAsync`**

This function is deprecated as of CUDA 3.0.

**Global `cudaGLRegisterBufferObject`**

This function is deprecated as of CUDA 3.0.

**Global `cudaGLSetBufferObjectMapFlags`**

This function is deprecated as of CUDA 3.0.

**Global `cudaGLSetGLDevice`**

This function is deprecated as of CUDA 5.0.

**Global `cudaGLUnmapBufferObject`**

This function is deprecated as of CUDA 3.0.

**Global `cudaGLUnmapBufferObjectAsync`**

This function is deprecated as of CUDA 3.0.

**Global `cudaGLUnregisterBufferObject`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D9MapResources`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D9RegisterResource`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D9ResourceGetMappedArray`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D9ResourceGetMappedPitch`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D9ResourceGetMappedPointer`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D9ResourceGetMappedSize`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D9ResourceGetSurfaceDimensions`**

This function is deprecated as of CUDA 3.0.



**Global `cudaD3D9ResourceSetMapFlags`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D9UnmapResources`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D9UnregisterResource`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D10GetDirect3DDevice`**

This function is deprecated as of CUDA 5.0.

**Global `cudaD3D10MapResources`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D10RegisterResource`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D10ResourceGetMappedArray`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D10ResourceGetMappedPitch`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D10ResourceGetMappedPointer`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D10ResourceGetMappedSize`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D10ResourceGetSurfaceDimensions`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D10ResourceSetMapFlags`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D10SetDirect3DDevice`**

This function is deprecated as of CUDA 5.0.

**Global `cudaD3D10UnmapResources`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D10UnregisterResource`**

This function is deprecated as of CUDA 3.0.

**Global `cudaD3D11GetDirect3DDevice`**

This function is deprecated as of CUDA 5.0.

**Global `cudaD3D11SetDirect3DDevice`**

This function is deprecated as of CUDA 5.0.

**Global `cudaBindTexture`****Global `cudaBindTexture2D`****Global `cudaBindTextureToArray`****Global `cudaBindTextureToMipmappedArray`**

**Global `cudaGetTextureAlignmentOffset`****Global `cudaGetTextureReference`****Global `cudaUnbindTexture`****Global `cudaBindSurfaceToArray`****Global `cudaGetSurfaceReference`****Global `cudaErrorProfilerNotInitialized`**

This error return is deprecated as of CUDA 5.0. It is no longer an error to attempt to enable/disable the profiling via `cudaProfilerStart` or `cudaProfilerStop` without initialization.

**Global `cudaErrorProfilerAlreadyStarted`**

This error return is deprecated as of CUDA 5.0. It is no longer an error to call `cudaProfilerStart()` when profiling is already enabled.

**Global `cudaErrorProfilerAlreadyStopped`**

This error return is deprecated as of CUDA 5.0. It is no longer an error to call `cudaProfilerStop()` when profiling is already disabled.

**Global `cudaErrorInvalidHostPointer`**

This error return is deprecated as of CUDA 10.1.

**Global `cudaErrorInvalidDevicePointer`**

This error return is deprecated as of CUDA 10.1.

**Global `cudaErrorAddressOfConstant`**

This error return is deprecated as of CUDA 3.1. Variables in constant memory may now have their address taken by the runtime via `cudaGetSymbolAddress()`.

**Global `cudaErrorTextureFetchFailed`**

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

**Global `cudaErrorTextureNotBound`**

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

**Global `cudaErrorSynchronizationError`**

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

**Global `cudaErrorMixedDeviceExecution`**

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

**Global `cudaErrorNotYetImplemented`**

This error return is deprecated as of CUDA 4.1.

**Global `cudaErrorMemoryValueTooLarge`**

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

**Global `cudaErrorPriorLaunchFailure`**

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

**Global `cudaErrorApiFailureBase`**

This error return is deprecated as of CUDA 4.1.

**Global `cudaDeviceBlockingSync`**

This flag was deprecated as of CUDA 4.0 and replaced with `cudaDeviceScheduleBlockingSync`.

## **Notice**

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

## **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**

© 2007-2019 NVIDIA Corporation. All rights reserved.