



# CUDA RUNTIME API

v8.0 | February 2016

## 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. Modules.....	7
4.1. Device Management.....	8
cudaChooseDevice.....	8
cudaDeviceGetAttribute.....	9
cudaDeviceGetByPCIBusId.....	13
cudaDeviceGetCacheConfig.....	14
cudaDeviceGetLimit.....	15
cudaDeviceGetP2PAttribute.....	16
cudaDeviceGetPCIBusId.....	17
cudaDeviceGetSharedMemConfig.....	17
cudaDeviceGetStreamPriorityRange.....	18
cudaDeviceReset.....	19
cudaDeviceSetCacheConfig.....	20
cudaDeviceSetLimit.....	21
cudaDeviceSetSharedMemConfig.....	22
cudaDeviceSynchronize.....	23
cudaGetDevice.....	24
cudaGetDeviceCount.....	24
cudaGetDeviceFlags.....	25
cudaGetDeviceProperties.....	26
cudalpcCloseMemHandle.....	31
cudalpcGetEventHandle.....	31
cudalpcGetMemHandle.....	32
cudalpcOpenEventHandle.....	33
cudalpcOpenMemHandle.....	34
cudaSetDevice.....	35
cudaSetDeviceFlags.....	36
cudaSetValidDevices.....	37
4.2. Thread Management [DEPRECATED].....	38
cudaThreadExit.....	38
cudaThreadGetCacheConfig.....	39
cudaThreadGetLimit.....	40
cudaThreadSetCacheConfig.....	41
cudaThreadSetLimit.....	42
cudaThreadSynchronize.....	43
4.3. Error Handling.....	43
cudaGetErrorName.....	44

cudaGetErrorString.....	44
cudaGetLastError.....	45
cudaPeekAtLastError.....	45
4.4. Stream Management.....	46
cudaStreamCallback_t.....	46
cudaStreamAddCallback.....	46
cudaStreamAttachMemAsync.....	48
cudaStreamCreate.....	50
cudaStreamCreateWithFlags.....	50
cudaStreamCreateWithPriority.....	51
cudaStreamDestroy.....	52
cudaStreamGetFlags.....	53
cudaStreamGetPriority.....	54
cudaStreamQuery.....	54
cudaStreamSynchronize.....	55
cudaStreamWaitEvent.....	56
4.5. Event Management.....	57
cudaEventCreate.....	57
cudaEventCreateWithFlags.....	57
cudaEventDestroy.....	58
cudaEventElapsedTime.....	59
cudaEventQuery.....	60
cudaEventRecord.....	61
cudaEventSynchronize.....	62
4.6. Execution Control.....	63
cudaFuncGetAttributes.....	63
cudaFuncSetCacheConfig.....	64
cudaFuncSetSharedMemConfig.....	65
cudaGetParameterBuffer.....	66
cudaGetParameterBufferV2.....	67
cudaLaunchKernel.....	68
cudaSetDoubleForDevice.....	69
cudaSetDoubleForHost.....	70
4.7. Occupancy.....	70
cudaOccupancyMaxActiveBlocksPerMultiprocessor.....	71
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags.....	71
4.8. Execution Control [DEPRECATED].....	73
cudaConfigureCall.....	73
cudaLaunch.....	74
cudaSetupArgument.....	75
4.9. Memory Management.....	75
cudaArrayGetInfo.....	76
cudaFree.....	76

cudaFreeArray.....	77
cudaFreeHost.....	78
cudaFreeMipmappedArray.....	78
cudaGetMipmappedArrayLevel.....	79
cudaGetSymbolAddress.....	80
cudaGetSymbolSize.....	80
cudaHostAlloc.....	81
cudaHostGetDevicePointer.....	83
cudaHostGetFlags.....	84
cudaHostRegister.....	84
cudaHostUnregister.....	86
cudaMalloc.....	87
cudaMalloc3D.....	88
cudaMalloc3DArray.....	89
cudaMallocArray.....	91
cudaMallocHost.....	93
cudaMallocManaged.....	94
cudaMallocMipmappedArray.....	96
cudaMallocPitch.....	98
cudaMemAdvise.....	100
cudaMemcpy.....	102
cudaMemcpy2D.....	103
cudaMemcpy2DArrayToArray.....	104
cudaMemcpy2DAsync.....	106
cudaMemcpy2DFromArray.....	107
cudaMemcpy2DFromArrayAsync.....	109
cudaMemcpy2DToArray.....	110
cudaMemcpy2DToArrayAsync.....	112
cudaMemcpy3D.....	114
cudaMemcpy3DAsync.....	116
cudaMemcpy3DPeer.....	118
cudaMemcpy3DPeerAsync.....	119
cudaMemcpyArrayToArray.....	119
cudaMemcpyAsync.....	121
cudaMemcpyFromArray.....	122
cudaMemcpyFromArrayAsync.....	123
cudaMemcpyFromSymbol.....	125
cudaMemcpyFromSymbolAsync.....	126
cudaMemcpyPeer.....	127
cudaMemcpyPeerAsync.....	128
cudaMemcpyToArray.....	129
cudaMemcpyToArrayAsync.....	130
cudaMemcpyToSymbol.....	132

cudaMemcpyToSymbolAsync.....	133
cudaMemGetInfo.....	134
cudaMemPrefetchAsync.....	135
cudaMemRangeGetAttribute.....	137
cudaMemRangeGetAttributes.....	138
cudaMemset.....	140
cudaMemset2D.....	141
cudaMemset2DAsync.....	142
cudaMemset3D.....	143
cudaMemset3DAsync.....	144
cudaMemsetAsync.....	145
make_cudaExtent.....	146
make_cudaPitchedPtr.....	147
make_cudaPos.....	147
4.10. Unified Addressing.....	148
cudaPointerGetAttributes.....	149
4.11. Peer Device Memory Access.....	151
cudaDeviceCanAccessPeer.....	151
cudaDeviceDisablePeerAccess.....	152
cudaDeviceEnablePeerAccess.....	152
4.12. OpenGL Interoperability.....	153
cudaGLDeviceList.....	153
cudaGLGetDevices.....	154
cudaGraphicsGLRegisterBuffer.....	155
cudaGraphicsGLRegisterImage.....	156
cudaWGLGetDevice.....	157
4.13. OpenGL Interoperability [DEPRECATED].....	158
cudaGLMapFlags.....	158
cudaGLMapBufferObject.....	158
cudaGLMapBufferObjectAsync.....	159
cudaGLRegisterBufferObject.....	160
cudaGLSetBufferObjectMapFlags.....	161
cudaGLSetGLDevice.....	162
cudaGLUnmapBufferObject.....	162
cudaGLUnmapBufferObjectAsync.....	163
cudaGLUnregisterBufferObject.....	164
4.14. Direct3D 9 Interoperability.....	164
cudaD3D9DeviceList.....	165
cudaD3D9GetDevice.....	165
cudaD3D9GetDevices.....	166
cudaD3D9GetDirect3DDevice.....	167
cudaD3D9SetDirect3DDevice.....	167
cudaGraphicsD3D9RegisterResource.....	168

4.15. Direct3D 9 Interoperability [DEPRECATED].....	170
cudaD3D9MapFlags.....	170
cudaD3D9RegisterFlags.....	171
cudaD3D9MapResources.....	171
cudaD3D9RegisterResource.....	172
cudaD3D9ResourceGetMappedArray.....	174
cudaD3D9ResourceGetMappedPitch.....	175
cudaD3D9ResourceGetMappedPointer.....	176
cudaD3D9ResourceGetMappedSize.....	177
cudaD3D9ResourceGetSurfaceDimensions.....	178
cudaD3D9ResourceSetMapFlags.....	180
cudaD3D9UnmapResources.....	181
cudaD3D9UnregisterResource.....	182
4.16. Direct3D 10 Interoperability.....	182
cudaD3D10DeviceList.....	182
cudaD3D10GetDevice.....	183
cudaD3D10GetDevices.....	183
cudaGraphicsD3D10RegisterResource.....	185
4.17. Direct3D 10 Interoperability [DEPRECATED].....	187
cudaD3D10MapFlags.....	187
cudaD3D10RegisterFlags.....	187
cudaD3D10GetDirect3DDevice.....	187
cudaD3D10MapResources.....	188
cudaD3D10RegisterResource.....	189
cudaD3D10ResourceGetMappedArray.....	190
cudaD3D10ResourceGetMappedPitch.....	191
cudaD3D10ResourceGetMappedPointer.....	193
cudaD3D10ResourceGetMappedSize.....	194
cudaD3D10ResourceGetSurfaceDimensions.....	195
cudaD3D10ResourceSetMapFlags.....	196
cudaD3D10SetDirect3DDevice.....	197
cudaD3D10UnmapResources.....	198
cudaD3D10UnregisterResource.....	199
4.18. Direct3D 11 Interoperability.....	199
cudaD3D11DeviceList.....	199
cudaD3D11GetDevice.....	200
cudaD3D11GetDevices.....	200
cudaGraphicsD3D11RegisterResource.....	202
4.19. Direct3D 11 Interoperability [DEPRECATED].....	204
cudaD3D11GetDirect3DDevice.....	204
cudaD3D11SetDirect3DDevice.....	204
4.20. VDP AU Interoperability.....	205
cudaGraphicsVDP AURegisterOutputSurface.....	205

cudaGraphicsVDPAURegisterVideoSurface.....	206
cudaVDPAUGetDevice.....	207
cudaVDPAUSetVDPAUDevice.....	208
4.21. EGL Interoperability.....	209
cudaEGLStreamConsumerAcquireFrame.....	209
cudaEGLStreamConsumerConnect.....	210
cudaEGLStreamConsumerConnectWithFlags.....	210
cudaEGLStreamConsumerReleaseFrame.....	211
cudaEGLStreamProducerConnect.....	212
cudaEGLStreamProducerDisconnect.....	213
cudaEGLStreamProducerPresentFrame.....	213
cudaEGLStreamProducerReturnFrame.....	214
cudaGraphicsEGLRegisterImage.....	215
cudaGraphicsResourceGetMappedEglFrame.....	216
4.22. Graphics Interoperability.....	217
cudaGraphicsMapResources.....	217
cudaGraphicsResourceGetMappedMipmappedArray.....	218
cudaGraphicsResourceGetMappedPointer.....	219
cudaGraphicsResourceSetMapFlags.....	220
cudaGraphicsSubResourceGetMappedArray.....	221
cudaGraphicsUnmapResources.....	222
cudaGraphicsUnregisterResource.....	223
4.23. Texture Reference Management.....	223
cudaBindTexture.....	224
cudaBindTexture2D.....	225
cudaBindTextureToArray.....	226
cudaBindTextureToMipmappedArray.....	227
cudaCreateChannelDesc.....	228
cudaGetChannelDesc.....	229
cudaGetTextureAlignmentOffset.....	229
cudaGetTextureReference.....	230
cudaUnbindTexture.....	231
4.24. Surface Reference Management.....	231
cudaBindSurfaceToArray.....	232
cudaGetSurfaceReference.....	232
4.25. Texture Object Management.....	233
cudaCreateTextureObject.....	233
cudaDestroyTextureObject.....	238
cudaGetTextureObjectResourceDesc.....	239
cudaGetTextureObjectResourceViewDesc.....	239
cudaGetTextureObjectTextureDesc.....	240
4.26. Surface Object Management.....	240
cudaCreateSurfaceObject.....	241

cudaDestroySurfaceObject.....	241
cudaGetSurfaceObjectResourceDesc.....	242
4.27. Version Management.....	242
cudaDriverGetVersion.....	242
cudaRuntimeGetVersion.....	243
4.28. C++ API Routines.....	244
__cudaOccupancyB2DHelper.....	244
cudaBindSurfaceToArray.....	244
cudaBindSurfaceToArray.....	245
cudaBindTexture.....	245
cudaBindTexture.....	246
cudaBindTexture2D.....	247
cudaBindTexture2D.....	249
cudaBindTextureToArray.....	250
cudaBindTextureToArray.....	251
cudaBindTextureToMipmappedArray.....	252
cudaBindTextureToMipmappedArray.....	253
cudaCreateChannelDesc.....	254
cudaEventCreate.....	254
cudaFuncGetAttributes.....	255
cudaFuncSetCacheConfig.....	256
cudaGetSymbolAddress.....	257
cudaGetSymbolSize.....	258
cudaGetTextureAlignmentOffset.....	258
cudaLaunch.....	259
cudaLaunchKernel.....	260
cudaMallocHost.....	261
cudaMallocManaged.....	263
cudaMemcpyFromSymbol.....	265
cudaMemcpyFromSymbolAsync.....	266
cudaMemcpyToSymbol.....	267
cudaMemcpyToSymbolAsync.....	268
cudaOccupancyMaxActiveBlocksPerMultiprocessor.....	270
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags.....	271
cudaOccupancyMaxPotentialBlockSize.....	272
cudaOccupancyMaxPotentialBlockSizeVariableSMem.....	273
cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags.....	274
cudaOccupancyMaxPotentialBlockSizeWithFlags.....	276
cudaSetupArgument.....	278
cudaStreamAttachMemAsync.....	278
cudaUnbindTexture.....	280
4.29. Interactions with the CUDA Driver API.....	281
4.30. Profiler Control.....	283



cudaProfilerInitialize.....	283
cudaProfilerStart.....	284
cudaProfilerStop.....	284
4.31. Data types used by CUDA Runtime.....	285
cudaChannelFormatDesc.....	286
cudaDeviceProp.....	286
cudaEglFrame.....	286
cudaEglPlaneDesc.....	286
cudaExtent.....	286
cudaFuncAttributes.....	286
cudalpcEventHandle_t.....	286
cudalpcMemHandle_t.....	286
cudaMemcpy3DParms.....	286
cudaMemcpy3DPeerParms.....	286
cudaPitchedPtr.....	286
cudaPointerAttributes.....	286
cudaPos.....	286
cudaResourceDesc.....	286
cudaResourceViewDesc.....	286
cudaTextureDesc.....	286
surfaceReference.....	286
textureReference.....	286
cudaChannelFormatKind.....	287
cudaComputeMode.....	287
cudaDeviceAttr.....	287
cudaDeviceP2PAttr.....	292
cudaEglColorFormat.....	292
cudaEglFrameType.....	293
cudaEglResourceLocationFlags.....	293
cudaError.....	293
cudaFuncCache.....	300
cudaGraphicsCubeFace.....	301
cudaGraphicsMapFlags.....	301
cudaGraphicsRegisterFlags.....	301
cudaLimit.....	302
cudaMemcpyKind.....	302
cudaMemoryAdvise.....	303
cudaMemoryType.....	303
cudaMemRangeAttribute.....	303
cudaOutputMode.....	304
cudaResourceType.....	304
cudaResourceViewFormat.....	304
cudaSharedMemConfig.....	306

cudaSurfaceBoundaryMode.....	306
cudaSurfaceFormatMode.....	306
cudaTextureAddressMode.....	307
cudaTextureFilterMode.....	307
cudaTextureReadMode.....	307
cudaArray_const_t.....	307
cudaArray_t.....	308
cudaEglStreamConnection.....	308
cudaError_t.....	308
cudaEvent_t.....	308
cudaGraphicsResource_t.....	308
cudaMipmappedArray_const_t.....	308
cudaMipmappedArray_t.....	308
cudaOutputMode_t.....	308
cudaStream_t.....	308
cudaSurfaceObject_t.....	308
cudaTextureObject_t.....	309
cudaUUID_t.....	309
CUDA_EGL_MAX_PLANES.....	309
CUDA_IPC_HANDLE_SIZE.....	309
cudaArrayCubemap.....	309
cudaArrayDefault.....	309
cudaArrayLayered.....	309
cudaArraySurfaceLoadStore.....	309
cudaArrayTextureGather.....	309
cudaCpuDeviceld.....	309
cudaDeviceBlockingSync.....	310
cudaDeviceLmemResizeToMax.....	310
cudaDeviceMapHost.....	310
cudaDeviceMask.....	310
cudaDevicePropDontCare.....	310
cudaDeviceScheduleAuto.....	310
cudaDeviceScheduleBlockingSync.....	310
cudaDeviceScheduleMask.....	310
cudaDeviceScheduleSpin.....	310
cudaDeviceScheduleYield.....	310
cudaEventBlockingSync.....	311
cudaEventDefault.....	311
cudaEventDisableTiming.....	311
cudaEventInterprocess.....	311
cudaHostAllocDefault.....	311
cudaHostAllocMapped.....	311
cudaHostAllocPortable.....	311

cudaHostAllocWriteCombined.....	311
cudaHostRegisterDefault.....	311
cudaHostRegisterIoMemory.....	311
cudaHostRegisterMapped.....	311
cudaHostRegisterPortable.....	312
cudaInvalidDeviceld.....	312
cudaIpcMemLazyEnablePeerAccess.....	312
cudaMemAttachGlobal.....	312
cudaMemAttachHost.....	312
cudaMemAttachSingle.....	312
cudaOccupancyDefault.....	312
cudaOccupancyDisableCachingOverride.....	312
cudaPeerAccessDefault.....	312
cudaStreamDefault.....	312
cudaStreamLegacy.....	312
cudaStreamNonBlocking.....	313
cudaStreamPerThread.....	313
<b>Chapter 5. Data Structures.....</b>	<b>314</b>
__cudaOccupancyB2DHelper.....	314
cudaChannelFormatDesc.....	315
f.....	315
w.....	315
x.....	315
y.....	315
z.....	315
cudaDeviceProp.....	315
asyncEngineCount.....	315
canMapHostMemory.....	315
clockRate.....	315
computeMode.....	316
concurrentKernels.....	316
concurrentManagedAccess.....	316
deviceOverlap.....	316
ECCEnabled.....	316
globalL1CacheSupported.....	316
hostNativeAtomicSupported.....	316
integrated.....	316
isMultiGpuBoard.....	316
kernelExecTimeoutEnabled.....	316
l2CacheSize.....	316
localL1CacheSupported.....	317
major.....	317
managedMemory.....	317

maxGridSize.....	317
maxSurface1D.....	317
maxSurface1DLayered.....	317
maxSurface2D.....	317
maxSurface2DLayered.....	317
maxSurface3D.....	317
maxSurfaceCubemap.....	317
maxSurfaceCubemapLayered.....	317
maxTexture1D.....	318
maxTexture1DLayered.....	318
maxTexture1DLinear.....	318
maxTexture1DMipmap.....	318
maxTexture2D.....	318
maxTexture2DGather.....	318
maxTexture2DLayered.....	318
maxTexture2DLinear.....	318
maxTexture2DMipmap.....	318
maxTexture3D.....	318
maxTexture3DAlt.....	318
maxTextureCubemap.....	319
maxTextureCubemapLayered.....	319
maxThreadsDim.....	319
maxThreadsPerBlock.....	319
maxThreadsPerMultiProcessor.....	319
memoryBusWidth.....	319
memoryClockRate.....	319
memPitch.....	319
minor.....	319
multiGpuBoardGroupID.....	319
multiProcessorCount.....	319
name.....	320
pageableMemoryAccess.....	320
pciBusID.....	320
pciDeviceID.....	320
pciDomainID.....	320
regsPerBlock.....	320
regsPerMultiprocessor.....	320
sharedMemPerBlock.....	320
sharedMemPerMultiprocessor.....	320
singleToDoublePrecisionPerfRatio.....	320
streamPrioritiesSupported.....	320
surfaceAlignment.....	321
tccDriver.....	321

textureAlignment.....	321
texturePitchAlignment.....	321
totalConstMem.....	321
totalGlobalMem.....	321
unifiedAddressing.....	321
warpSize.....	321
cudaEglFrame.....	321
eglColorFormat.....	322
frameType.....	322
pArray.....	322
planeCount.....	322
planeDesc.....	322
pPitch.....	322
cudaEglPlaneDesc.....	322
channelDesc.....	322
depth.....	323
height.....	323
numChannels.....	323
pitch.....	323
reserved.....	323
width.....	323
cudaExtent.....	323
depth.....	323
height.....	323
width.....	323
cudaFuncAttributes.....	324
binaryVersion.....	324
cacheModeCA.....	324
constSizeBytes.....	324
localSizeBytes.....	324
maxThreadsPerBlock.....	324
numRegs.....	324
ptxVersion.....	324
sharedSizeBytes.....	325
cudaIpcEventHandle_t.....	325
cudaIpcMemHandle_t.....	325
cudaMemcpy3DParms.....	325
dstArray.....	325
dstPos.....	325
dstPtr.....	325
extent.....	325
kind.....	325
srcArray.....	325

srcPos.....	326
srcPtr.....	326
cudaMemcpy3DPeerParms.....	326
dstArray.....	326
dstDevice.....	326
dstPos.....	326
dstPtr.....	326
extent.....	326
srcArray.....	326
srcDevice.....	326
srcPos.....	326
srcPtr.....	327
cudaPitchedPtr.....	327
pitch.....	327
ptr.....	327
xsize.....	327
ysize.....	327
cudaPointerAttributes.....	327
device.....	327
devicePointer.....	328
hostPointer.....	328
isManaged.....	328
memoryType.....	328
cudaPos.....	328
x.....	328
y.....	328
z.....	328
cudaResourceDesc.....	329
array.....	329
desc.....	329
devPtr.....	329
height.....	329
mipmap.....	329
pitchInBytes.....	329
resType.....	329
sizeInBytes.....	329
width.....	329
cudaResourceViewDesc.....	330
depth.....	330
firstLayer.....	330
firstMipmapLevel.....	330
format.....	330
height.....	330

lastLayer.....	330
lastMipmapLevel.....	330
width.....	330
cudaTextureDesc.....	330
addressMode.....	331
borderColor.....	331
filterMode.....	331
maxAnisotropy.....	331
maxMipmapLevelClamp.....	331
minMipmapLevelClamp.....	331
mipmapFilterMode.....	331
mipmapLevelBias.....	331
normalizedCoords.....	331
readMode.....	331
sRGB.....	332
surfaceReference.....	332
channelDesc.....	332
textureReference.....	332
addressMode.....	332
channelDesc.....	332
filterMode.....	332
maxAnisotropy.....	332
maxMipmapLevelClamp.....	332
minMipmapLevelClamp.....	333
mipmapFilterMode.....	333
mipmapLevelBias.....	333
normalized.....	333
sRGB.....	333
<b>Chapter 6. Data Fields.....</b>	<b>334</b>
<b>Chapter 7. Deprecated List.....</b>	<b>343</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. In the reference documentation, each memcpy function is categorized as synchronous or asynchronous, corresponding to the definitions below.

### Memcpy

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. 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.

## MODULES

Here is a list of all modules:

- ▶ Device Management
- ▶ Thread Management [DEPRECATED]
- ▶ Error Handling
- ▶ Stream Management
- ▶ Event Management
- ▶ Execution Control
- ▶ Occupancy
- ▶ Execution Control [DEPRECATED]
- ▶ Memory Management
- ▶ 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
- ▶ Surface Reference Management
- ▶ Texture Object Management
- ▶ Surface Object Management

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

## 4.1. Device Management

### CUDART\_DEVICE

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.

#### 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;
- ▶ `cudaDevAttrGlobalL1CacheSupported`: 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;
- ▶ `cudaDevAttrManagedMemSupported`: 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.



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

#### See also:

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

## `__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] [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.

#### See also:

`cudaDeviceGetPCIBusId`

## `__host__ __device__ cudaError_t cudaDeviceGetCacheConfig (cudaFuncCache *pCacheConfig)`

Returns the preferred cache configuration for the current device.

### Parameters

#### `pCacheConfig`

- Returned cache configuration

### Returns

`cudaSuccess`, `cudaErrorInitializationError`

### 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.

### See also:

`cudaDeviceSetCacheConfig`, `cudaFuncSetCacheConfig` ( C API),  
`cudaFuncSetCacheConfig` ( C++ API)

## `__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()` and `fprintf()` device system calls.
- ▶ `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.



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

### See also:

`cudaDeviceSetLimit`

## `__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.

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.

### See also:

`cudaCtxEnablePeerAccess`, `cudaCtxDisablePeerAccess`, `cudaCtxCanAccessPeer`



## `__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.

### See also:

`cudaDeviceGetByPCIBusId`

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

Returns the shared memory configuration for the current device.

### Parameters

#### **pConfig**

- Returned cache configuration

## Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInitializationError`

## 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.

## See also:

`cudaDeviceSetCacheConfig`, `cudaDeviceGetCacheConfig`,  
`cudaDeviceSetSharedMemConfig`, `cudaFuncSetCacheConfig`

## `__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`, `cudaErrorInvalidValue`

## 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.

See also:

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

## `__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.

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`, `cudaErrorInitializationError`

### 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.

See also:

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

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

Set resource limits.

### Parameters

#### **limit**

- Limit to set

#### **value**

- Size of limit

### Returns

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

### Description

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

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

- ▶ `cudaLimitStackSize` controls the stack size in bytes of each GPU thread.
- ▶ `cudaLimitPrintfFifoSize` controls the size in bytes of the shared FIFO used by the `printf()` and `fprintf()` device system calls. Setting `cudaLimitPrintfFifoSize` must not be performed after launching any kernel that uses the `printf()` or `fprintf()` device system calls - 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.



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

See also:

`cudaDeviceGetLimit`

## `__host__ cudaError_t cudaDeviceSetSharedMemConfig(cudaSharedMemConfig config)`

Sets the shared memory configuration for the current device.

### Parameters

**config**

- Requested cache configuration

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInitializationError`

## 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.

See also:

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

## `__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.

See also:

`cudaDeviceReset`

`__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`

#### 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.

See also:

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

`__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

[cudaSuccess](#), [cudaErrorNoDevice](#), [cudaErrorInsufficientDriver](#)

## Description

Returns in `*count` the number of devices with compute capability greater or equal to 2.0 that are available for execution. If there is no such device then [cudaGetDeviceCount\(\)](#) will return [cudaErrorNoDevice](#). If no driver can be loaded to determine if any such devices exist then [cudaGetDeviceCount\(\)](#) will return [cudaErrorInsufficientDriver](#).



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

## See also:

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

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

Gets the flags for the current device.

## Parameters

### `flags`

- Pointer to store the device flags

## Returns

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

## Description

Returns in `flags` the flags for the current device. If there is a current device for the calling thread, 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.

**See also:**

`cudaGetDevice`, `cudaGetDeviceProperties`, `cudaSetDevice`, `cudaSetDeviceFlags`

## `__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];
    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 managedMemSupported;
    int isMultiGpuBoard;
    int multiGpuBoardGroupID;
    int singleToDoublePrecisionPerfRatio;
    int pageableMemoryAccess;
    int concurrentManagedAccess;
}

```

where:

- ▶ `name[256]` is an ASCII string identifying the device;
- ▶ `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.

See also:

[cudaGetDeviceCount](#), [cudaGetDevice](#), [cudaSetDevice](#), [cudaChooseDevice](#),  
[cudaDeviceGetAttribute](#)

## **\_\_host\_\_ cudaError\_t cudalpcCloseMemHandle (void \*devPtr)**

Close memory mapped with [cudaIpcOpenMemHandle](#).

### **Parameters**

#### **devPtr**

- Device pointer returned by [cudaIpcOpenMemHandle](#)

### **Returns**

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

### **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.

### **See also:**

[cudaMalloc](#), [cudaFree](#), [cudaIpcGetEventHandle](#), [cudaIpcOpenEventHandle](#),  
[cudaIpcGetMemHandle](#), [cudaIpcOpenMemHandle](#),

## **\_\_host\_\_ cudaError\_t cudalpcGetEventHandle (cudalpcEventHandle\_t \*handle, cudaEvent\_t event)**

Gets an interprocess handle for a previously allocated event.

### **Parameters**

#### **handle**

- Pointer to a user allocated [cudalpcEventHandle](#) in which to return the opaque event handle

#### **event**

- Event allocated with [cudaEventInterprocess](#) and [cudaEventDisableTiming](#) flags.

## Returns

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

## 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.

## See also:

[cudaEventCreate](#), [cudaEventDestroy](#), [cudaEventSynchronize](#), [cudaEventQuery](#), [cudaStreamWaitEvent](#), [cudaIpcOpenEventHandle](#), [cudaIpcGetMemHandle](#), [cudaIpcOpenMemHandle](#), [cudaIpcCloseMemHandle](#)

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

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

## Parameters

### **handle**

- Pointer to user allocated [cudaIpcMemHandle](#) to return the handle in.

### **devPtr**

- Base pointer to previously allocated device memory

## Returns

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

## 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.

**See also:**

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

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

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

### Parameters

**event**

- Returns the imported event

**handle**

- Interprocess handle to open

### Returns

`cudaSuccess`, `cudaErrorMapBufferObjectFailed`, `cudaErrorInvalidResourceHandle`

### 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.

**See also:**

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

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

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

### Parameters

#### **devPtr**

- Returned device pointer

#### **handle**

- `cudaIpcMemHandle` to open

#### **flags**

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

### Returns

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

### 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.

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.



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](#),

## **\_\_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.

**See also:**

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

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

Sets flags to be used for device executions.

### Parameters

#### `flags`

- Parameters for device operation

### Returns

`cudaSuccess`, `cudaErrorInvalidDevice`, `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.
- ▶ `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.

#### See also:

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

## `__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.

#### See also:

[cudaGetDeviceCount](#), [cudaSetDevice](#), [cudaGetDeviceProperties](#), [cudaSetDeviceFlags](#), [cudaChooseDevice](#)

## 4.2. 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.

#### 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`, `cudaErrorInitializationError`

### 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.

### 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()` and `fprintf()` device system calls.
- ▶ `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.

### 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`, `cudaErrorInitializationError`

### 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.

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()` and `fprintf()` device system calls. Setting [cudaLimitPrintfFifoSize](#) must be performed before launching any kernel that uses the `printf()` or `fprintf()` device system calls, 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.

**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.

**See also:**

[cudaDeviceSynchronize](#)

## **4.3. 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](#)

## `__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](#)

## `__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`, `cudaErrorInvalidHostPointer`, `cudaErrorInvalidDevicePointer`, `cudaErrorInvalidTexture`, `cudaErrorInvalidTextureBinding`, `cudaErrorInvalidChannelDescriptor`, `cudaErrorInvalidMemcpyDirection`, `cudaErrorInvalidFilterSetting`, `cudaErrorInvalidNormSetting`, `cudaErrorUnknown`, `cudaErrorInvalidResourceHandle`, `cudaErrorInsufficientDriver`, `cudaErrorSetOnActiveProcess`, `cudaErrorStartupFailure`,

### 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.

### 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`, `cudaErrorInvalidHostPointer`, `cudaErrorInvalidDevicePointer`, `cudaErrorInvalidTexture`, `cudaErrorInvalidTextureBinding`, `cudaErrorInvalidChannelDescriptor`,

`cudaErrorInvalidMemcpyDirection`, `cudaErrorInvalidFilterSetting`,  
`cudaErrorInvalidNormSetting`, `cudaErrorUnknown`, `cudaErrorInvalidResourceHandle`,  
`cudaErrorInsufficientDriver`, `cudaErrorSetOnActiveProcess`, `cudaErrorStartupFailure`,

### 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.

### See also:

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

## 4.4. 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](#), [cudaErrorNotSupported](#)

**Description**

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 will 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.

**See also:**

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

**`__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)

#### **length**

- Length of memory (must be zero, 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 address within managed memory space declared using the `__managed__` keyword or allocated with `cudaMallocManaged`.

`length` must be zero, to indicate that the entire allocation's stream association is being changed. Currently, it's not possible to change stream association for a portion of an allocation. The default value for `length` is 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.

#### See also:

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

## `__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.

### See also:

`cudaStreamCreateWithPriority`, `cudaStreamCreateWithFlags`, `cudaStreamGetPriority`, `cudaStreamGetFlags`, `cudaStreamQuery`, `cudaStreamSynchronize`, `cudaStreamWaitEvent`, `cudaStreamAddCallback`, `cudaStreamDestroy`

## `__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.

## See also:

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

## `__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.
- ▶ 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`

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

Destroys and cleans up an asynchronous stream.

### Parameters

#### **stream**

- Stream identifier

### Returns

`cudaSuccess`, `cudaErrorInvalidResourceHandle`

### Description

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

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



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

**See also:**

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

## **\_\_host\_\_ cudaError\_t cudaStreamGetFlags (cudaStream\_t hStream, unsigned int \*flags)**

Query the flags of a stream.

**Parameters****hStream**

- Handle to the stream to be queried

**flags**

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

**Returns**

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

**Description**

Query the flags of a stream. The flags are returned in `flags`. See [cudaStreamCreateWithFlags](#) for a list of valid flags.



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

**See also:**

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

## `__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.

### See also:

`cudaStreamCreateWithPriority`, `cudaDeviceGetStreamPriorityRange`,  
`cudaStreamGetFlags`

## `__host__ cudaError_t cudaStreamQuery(cudaStream_t stream)`

Queries an asynchronous stream for completion status.

### Parameters

#### **stream**

- Stream identifier

### Returns

`cudaSuccess`, `cudaErrorNotReady`, `cudaErrorInvalidResourceHandle`

## Description

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

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



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

## See also:

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

## `__host__ cudaError_t cudaStreamSynchronize(cudaStream_t stream)`

Waits for stream tasks to complete.

## Parameters

### `stream`

- Stream identifier

## Returns

`cudaSuccess`, `cudaErrorInvalidResourceHandle`

## Description

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



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

## See also:

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

## `__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`, `cudaErrorInvalidResourceHandle`

### Description

Makes all future work submitted to `stream` wait until `event` reports completion before beginning execution. This synchronization will be performed efficiently on the device. The event `event` may be from a different context than `stream`, in which case this function will perform cross-device synchronization.

The stream `stream` will wait only for the completion of the most recent host call to `cudaEventRecord()` on `event`. Once this call has returned, any functions (including `cudaEventRecord()` and `cudaEventDestroy()`) may be called on `event` again, and the subsequent calls will not have any effect on `stream`.

If `cudaEventRecord()` has not been called on `event`, this call acts as if the record has already completed, and so is a functional no-op.



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

### See also:

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



## 4.5. 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`, `cudaErrorInitializationError`, `cudaErrorInvalidValue`, `cudaErrorLaunchFailure`, `cudaErrorMemoryAllocation`

#### Description

Creates an event object using `cudaEventDefault`.



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

#### See also:

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

### `__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`, `cudaErrorInitializationError`, `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()`.
- ▶ `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.

**See also:**

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

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

Destroys an event object.

**Parameters****event**

- Event to destroy

**Returns**

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

**Description**

Destroys the event specified by `event`.

In case `event` has been recorded but has not yet been completed when [cudaEventDestroy\(\)](#) is called, the function will return immediately and the resources associated with `event` will be released automatically once the device has completed `event`.



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

**See also:**

[cudaEventCreate](#) ( C API), [cudaEventCreateWithFlags](#), [cudaEventQuery](#),  
[cudaEventSynchronize](#), [cudaEventRecord](#), [cudaEventElapsedTime](#)

## **\_\_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](#), [cudaErrorInitializationError](#),  
[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.

#### See also:

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

## `__host__ cudaError_t cudaEventQuery (cudaEvent_t event)`

Queries an event's status.

#### Parameters

##### **event**

- Event to query

#### Returns

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

#### Description

Query the status of all device work preceding the most recent call to `cudaEventRecord()` (in the appropriate compute streams, as specified by the arguments to `cudaEventRecord()`).

If this work has successfully been completed by the device, or if `cudaEventRecord()` has not been called on `event`, then `cudaSuccess` is returned. If this work has not yet been completed by the device then `cudaErrorNotReady` is returned.

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.

#### See also:

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

## `__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`, `cudaErrorInitializationError`, `cudaErrorInvalidResourceHandle`, `cudaErrorLaunchFailure`

#### Description

Records an event. See note about NULL stream behavior. Since operation is asynchronous, `cudaEventQuery()` or `cudaEventSynchronize()` must be used to determine when the event has actually been recorded.

If `cudaEventRecord()` has previously been called on `event`, then this call will overwrite any existing state in `event`. Any subsequent calls which examine the status of `event` will only examine the completion of this most recent call to `cudaEventRecord()`.



► This function uses standard `default stream` semantics.

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

#### See also:

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

## `__host__ cudaError_t cudaEventSynchronize(cudaEvent_t event)`

Waits for an event to complete.

#### Parameters

##### **event**

- Event to wait for

#### Returns

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

#### Description

Wait until the completion of all device work preceding the most recent call to [cudaEventRecord\(\)](#) (in the appropriate compute streams, as specified by the arguments to [cudaEventRecord\(\)](#)).

If [cudaEventRecord\(\)](#) has not been called on `event`, [cudaSuccess](#) is returned immediately.

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.

#### See also:

[cudaEventCreate](#) ( C API), [cudaEventCreateWithFlags](#), [cudaEventRecord](#), [cudaEventQuery](#), [cudaEventDestroy](#), [cudaEventElapsedTime](#)

## 4.6. 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](#), [cudaErrorInitializationError](#), [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.

See also:

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

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

Sets the preferred cache configuration for a device function.

### Parameters

#### **func**

- Device function symbol

#### **cacheConfig**

- Requested cache configuration

### Returns

`cudaSuccess`, `cudaErrorInitializationError`, `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.

#### See also:

[cudaConfigureCall](#), [cudaFuncSetCacheConfig](#) ( C++ API), [cudaFuncGetAttributes](#) ( C API), [cudaLaunchKernel](#) ( C API), [cudaSetDoubleForDevice](#), [cudaSetDoubleForHost](#), [cudaSetupArgument](#) ( C API), [cudaThreadGetCacheConfig](#), [cudaThreadSetCacheConfig](#)

## `__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](#), [cudaErrorInitializationError](#), [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.

**See also:**

`cudaConfigureCall`, `cudaDeviceSetSharedMemConfig`,  
`cudaDeviceGetSharedMemConfig`, `cudaDeviceSetCacheConfig`,  
`cudaDeviceGetCacheConfig`, `cudaFuncSetCacheConfig`

## `__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 blockDimension, unsigned int sharedMemSize)**

Launches a specified kernel.

### Parameters

#### **func**

- Pointer to the kernel to be launched

#### **gridDimension**

- Specifies grid dimensions

#### **blockDimension**

- Specifies block dimensions

#### **sharedMemSize**

- Specifies size of shared memory

### Returns

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

### Description

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

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



Note 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 cudaLaunchKernel (const void \*func, dim3 gridDim, dim3 blockDim, void \*\*args, size\_t sharedMem, cudaStream\_t stream)**

Launches a device function.

### Parameters

#### **func**

- Device function symbol

#### **gridDim**

- Grid dimentions

#### **blockDim**

- Block dimentions

#### **args**

- Arguments

#### **sharedMem**

- Shared memory

#### **stream**

- Stream identifier

### Returns

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

### 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.

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

## `__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.

[cudaLaunch](#) ( C API), [cudaFuncSetCacheConfig](#) ( C API), [cudaFuncGetAttributes](#) ( C API), [cudaSetDoubleForHost](#), [cudaSetupArgument](#) ( C API)

## `__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.

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

## 4.7. 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](#), [cudaErrorCudartUnloading](#), [cudaErrorInitializationError](#),  
[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.

### See also:

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

```
__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](#), [cudaErrorCudartUnloading](#), [cudaErrorInitializationError](#), [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.

See also:



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

## 4.8. Execution Control [DEPRECATED]

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

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

**`__host__ cudaError_t cudaConfigureCall (dim3 gridDim, dim3 blockDim, size_t sharedMem, cudaStream_t stream)`**

Configure a device-launch.

### Parameters

#### **gridDim**

- Grid dimensions

#### **blockDim**

- Block dimensions

#### **sharedMem**

- Shared memory

#### **stream**

- Stream identifier

### Returns

[cudaSuccess](#), [cudaErrorInvalidConfiguration](#)

### Description

**Deprecated** This function is deprecated as of CUDA 7.0

Specifies the grid and block dimensions for the device call to be executed similar to the execution configuration syntax. [cudaConfigureCall\(\)](#) is stack based. Each call pushes data on top of an execution stack. This data contains the dimension for the grid and thread blocks, together with any arguments for the call.



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

`cudaLaunchKernel` ( C API), `cudaFuncSetCacheConfig` ( C API), `cudaFuncGetAttributes` ( C API), `cudaLaunch` ( C API), `cudaSetDoubleForDevice`, `cudaSetDoubleForHost`, `cudaSetupArgument` ( C API),

## `__host__ cudaError_t cudaLaunch (const void *func)`

Launches a device function.

### Parameters

#### **func**

- Device function symbol

### Returns

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

### Description

**Deprecated** This function is deprecated as of CUDA 7.0

Launches the function `func` on the device. The parameter `func` must be a device function symbol. The parameter specified by `func` must be declared as a `__global__` function. For templated functions, pass the function symbol as follows: `func_name<template_arg_0,...,template_arg_N> cudaLaunch()` must be preceded by a call to `cudaConfigureCall()` since it pops the data that was pushed by `cudaConfigureCall()` from the execution stack.



- ▶ Note that this function may also return error codes from previous, asynchronous launches.
- ▶ Use of a string naming a variable as the `symbol` paramater was removed in CUDA 5.0.

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

**\_\_host\_\_ cudaError\_t cudaSetupArgument (const void \*arg, size\_t size, size\_t offset)**

Configure a device launch.

### Parameters

**arg**

- Argument to push for a kernel launch

**size**

- Size of argument

**offset**

- Offset in argument stack to push new arg

### Returns

[cudaSuccess](#)

### Description

**Deprecated** This function is deprecated as of CUDA 7.0

Pushes `size` bytes of the argument pointed to by `arg` at `offset` bytes from the start of the parameter passing area, which starts at offset 0. The arguments are stored in the top of the execution stack. [cudaSetupArgument\(\)](#) must be preceded by a call to [cudaConfigureCall\(\)](#).



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

[cudaLaunchKernel](#) ( C API), [cudaFuncSetCacheConfig](#) ( C API), [cudaFuncGetAttributes](#) ( C API), [cudaLaunch](#) ( C API), [cudaSetDoubleForDevice](#), [cudaSetDoubleForHost](#), [cudaSetupArgument](#) ( C++ API),

## 4.9. 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.

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

Frees memory on the device.

### Parameters

#### **devPtr**

- Device pointer to memory to free

### Returns

[cudaSuccess](#), [cudaErrorInvalidDevicePointer](#), [cudaErrorInitializationError](#)

## 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 `cudaErrorInvalidDevicePointer` 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.

## See also:

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

## `__host__ cudaError_t cudaFreeArray (cudaArray_t array)`

Frees an array on the device.

## Parameters

### `array`

- Pointer to array to free

## Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInitializationError`

## Description

Frees the CUDA array `array`, which must have been \* returned by a previous call to `cudaMallocArray()`. If `cudaFreeArray(array)` has already been called before, `cudaErrorInvalidValue` is returned. If `devPtr` is 0, no operation is performed.



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

## See also:

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

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

Frees page-locked memory.

### Parameters

#### **ptr**

- Pointer to memory to free

### Returns

`cudaSuccess`, `cudaErrorInitializationError`

### 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.

### See also:

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

## `__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`, `cudaErrorInitializationError`

### Description

Frees the CUDA mipmapped array `mipmappedArray`, which must have been returned by a previous call to `cudaMallocMipmappedArray()`. If `cudaFreeMipmappedArray(mipmappedArray)` has already been called before, `cudaErrorInvalidValue` is returned.



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

#### See also:

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

**`__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.

#### See also:

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

## `__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`

### 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.

### See also:

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

## `__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](#)

## 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.

## See also:

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

## `__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](#), [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.

`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.

**See also:**

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

## `__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.

See also:

[cudaSetDeviceFlags](#), [cudaHostAlloc](#)

**\_\_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.

See also:

[cudaHostAlloc](#)

**\_\_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`

**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.

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.

#### See also:

`cudaHostUnregister`, `cudaHostGetFlags`, `cudaHostGetDevicePointer`

## `__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`

### 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.

See also:

[cudaHostUnregister](#)

**\_\_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](#), [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.

See also:

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

## `__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`, `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.

### See also:

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



## `__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`, `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 <code>cudaArraySurfaceLoadStore</code> 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 }

CUDA array type	Valid extents that must always be met {(width range in elements), (height range), (depth range)}	Valid extents with cudaArraySurfaceLoadStore set {(width range in elements), (height range), (depth range)}
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.

See also:

[cudaMalloc3D](#), [cudaMalloc](#), [cudaMallocPitch](#), [cudaFree](#), [cudaFreeArray](#), [cudaMallocHost \( C API \)](#), [cudaFreeHost](#), [cudaHostAlloc](#), [make\\_cudaExtent](#)

**\_\_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](#), [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.

**See also:**

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

## `__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`, `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.

### See also:

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

## `__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.

#### See also:

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

**`__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`, `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  $[1, 1 + \text{floor}(\log_2(\max(\text{width}, \text{height}, \text{depth})))]$ .

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 {(width range in elements), (height range), (depth range)}
1D	{ (1,maxTexture1DMipmap), 0, 0 }
2D	{ (1,maxTexture2DMipmap[0]), (1,maxTexture2DMipmap[1]), 0 }
3D	{ (1,maxTexture3D[0]), (1,maxTexture3D[1]), (1,maxTexture3D[2]) }
1D Layered	{ (1,maxTexture1DLayered[0]), 0, (1,maxTexture1DLayered[1]) }
2D Layered	{ (1,maxTexture2DLayered[0]), (1,maxTexture2DLayered[1]), (1,maxTexture2DLayered[2]) }
Cubemap	{ (1,maxTextureCubemap), (1,maxTextureCubemap), 6 }
Cubemap Layered	{ (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[1]) }



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

See also:

[cudaMalloc3D](#), [cudaMalloc](#), [cudaMallocPitch](#), [cudaFree](#), [cudaFreeArray](#), [cudaMallocHost \( C API\)](#), [cudaFreeHost](#), [cudaHostAlloc](#), [make\\_cudaExtent](#)

**\_\_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](#), [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.

**See also:**

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

```
__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 `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.

- ▶ `cudaMemAdviceUnsetReadMostly`: Undoes the effect of `cudaMemAdviceReadMostly` and also prevents the Unified Memory driver from attempting heuristic read-duplication on the memory range. Any read-duplicated copies of the data will be collapsed into a single copy. The location for the collapsed copy will be the preferred location if the page has a preferred location and one of the read-duplicated copies was resident at that location. Otherwise, the location chosen is arbitrary.
- ▶ `cudaMemAdviceSetPreferredLocation`: 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 `cudaMemAdviceSetReadMostly` 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.
- ▶ `cudaMemAdviceUnsetPreferredLocation`: Undoes the effect of `cudaMemAdviceSetPreferredLocation` and changes the preferred location to none.
- ▶ `cudaMemAdviceSetAccessedBy`: 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.

- ▶ `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.



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

See also:

`cudaMemcpy`, `cudaMemcpyPeer`, `cudaMemcpyAsync`, `cudaMemcpy3DPeerAsync`, `cudaMemPrefetchAsync`

## `__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`, `cudaErrorInvalidDevicePointer`, `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.
- This function exhibits `synchronous` behavior for most use cases.

## See also:

`cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`,  
`cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyFromArrayToArray`,  
`cudaMemcpy2DFromArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`,  
`cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`,  
`cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`,  
`cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`,  
`cudaMemcpyFromSymbolAsync`

**\_\_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`,  
`cudaErrorInvalidDevicePointer`, `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.

**See also:**

`cudaMemcpy`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`,  
`cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyArrayToArray`,  
`cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`,  
`cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`,  
`cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`,  
`cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`,  
`cudaMemcpyFromSymbolAsync`

```
__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.

### See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`,  
`cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyArrayToArray`,  
`cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`,  
`cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`,  
`cudaMemcpyFromArrayAsync`, `cudaMemcpy2DFromArrayAsync`,  
`cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`

**\_\_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`,  
`cudaErrorInvalidDevicePointer`, `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.

## See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`, `cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyArrayToArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`

**`__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`, `cudaErrorInvalidDevicePointer`, `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.

### See also:

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpyToArray](#), [cudaMemcpy2DToArray](#),  
[cudaMemcpyFromArray](#), [cudaMemcpyArrayToArray](#), [cudaMemcpy2DArrayToArray](#),  
[cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyAsync](#),  
[cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpy2DToArrayAsync](#),  
[cudaMemcpyFromArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#),  
[cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

**\_\_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`, `cudaErrorInvalidDevicePointer`, `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.

## See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`, `cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyFromArrayToArray`, `cudaMemcpy2DFromArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`

**`__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](#), [cudaErrorInvalidDevicePointer](#),  
[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.

### See also:

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpyToArray](#), [cudaMemcpyFromArray](#),  
[cudaMemcpy2DFromArray](#), [cudaMemcpyArrayToArray](#),  
[cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#),  
[cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#),  
[cudaMemcpy2DToArrayAsync](#), [cudaMemcpyFromArrayAsync](#),  
[cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#),  
[cudaMemcpyFromSymbolAsync](#)

```
__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`, `cudaErrorInvalidDevicePointer`, `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.

## See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`, `cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyFromArrayToArray`, `cudaMemcpy2DFromArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpyFromArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`

## `__host__ cudaError_t cudaMemcpy3D (const cudaMemcpy3DParms *p)`

Copies data between 3D objects.

### Parameters

**p**

- 3D memory copy parameters

### Returns

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevicePointer`, `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**. 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.

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.

#### See also:

`cudaMalloc3D`, `cudaMalloc3DArray`, `cudaMemset3D`, `cudaMemcpy3DAsync`, `cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`, `cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyArrayToArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`,

cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync,  
 cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync,  
 cudaMemcpyFromSymbolAsync, make\_cudaExtent, make\_cudaPos

**\_\_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, cudaErrorInvalidDevicePointer,  
 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.

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.

**See also:**

[cudaMalloc3D](#), [cudaMalloc3DArray](#), [cudaMemset3D](#), [cudaMemcpy3D](#), [cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpyToArray](#), [cudaMemcpy2DToArray](#), [cudaMemcpyFromArray](#), [cudaMemcpy2DFromArray](#), [cudaMemcpyArrayToArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpyFromArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#), [make\\_cudaExtent](#), [make\\_cudaPos](#)

## **`__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.

**See also:**

[cudaMemcpy](#), [cudaMemcpyPeer](#), [cudaMemcpyAsync](#), [cudaMemcpyPeerAsync](#),  
[cudaMemcpy3DPeerAsync](#)

**\_\_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.

**See also:**

[cudaMemcpy](#), [cudaMemcpyPeer](#), [cudaMemcpyAsync](#), [cudaMemcpyPeerAsync](#),  
[cudaMemcpy3DPeerAsync](#)

**\_\_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

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.

See also:



`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`,  
`cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpy2DArrayToArray`,  
`cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`,  
`cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`,  
`cudaMemcpyFromArrayAsync`, `cudaMemcpy2DFromArrayAsync`,  
`cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`

**\_\_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`, `cudaErrorInvalidDevicePointer`,  
`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.

#### See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`, `cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyArrayToArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`

**`__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`, `cudaErrorInvalidDevicePointer`,  
`cudaErrorInvalidMemcpyDirection`

## Description

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.

## See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`,  
`cudaMemcpy2DFromArray`, `cudaMemcpyArrayToArray`,  
`cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`,  
`cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`,  
`cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`,  
`cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`,  
`cudaMemcpyFromSymbolAsync`

**`__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](#), [cudaErrorInvalidDevicePointer](#),  
[cudaErrorInvalidMemcpyDirection](#)

**Description**

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.

**See also:**

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpyToArray](#), [cudaMemcpy2DToArray](#),  
[cudaMemcpyFromArray](#), [cudaMemcpy2DFromArray](#), [cudaMemcpyFromArrayToArray](#),  
[cudaMemcpy2DFromArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#),  
[cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#),  
[cudaMemcpy2DToArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#),  
[cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

**`__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`, `cudaErrorInvalidDevicePointer`, `cudaErrorInvalidMemcpyDirection`

### 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.

See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`,  
`cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyArrayToArray`,  
`cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyAsync`,  
`cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`,  
`cudaMemcpyFromArrayAsync`, `cudaMemcpy2DFromArrayAsync`,  
`cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`

**`__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`,  
`cudaErrorInvalidDevicePointer`, `cudaErrorInvalidMemcpyDirection`

### 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.

#### See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`, `cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyFromArrayToArray`, `cudaMemcpy2DFromArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`

## `__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.

## See also:

[cudaMemcpy](#), [cudaMemcpyAsync](#), [cudaMemcpyPeerAsync](#), [cudaMemcpy3DPeerAsync](#)

**`__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.

## See also:

[cudaMemcpy](#), [cudaMemcpyPeer](#), [cudaMemcpyAsync](#), [cudaMemcpy3DPeerAsync](#)

**`__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](#), [cudaErrorInvalidDevicePointer](#), [cudaErrorInvalidMemcpyDirection](#)

## Description

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.

## See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy2DToArray`,  
`cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyArrayToArray`,  
`cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`,  
`cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`,  
`cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`,  
`cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`,  
`cudaMemcpyFromSymbolAsync`

**`__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`, `cudaErrorInvalidDevicePointer`,  
`cudaErrorInvalidMemcpyDirection`

**Description**

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.

**See also:**

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`,  
`cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyFromArrayToArray`,  
`cudaMemcpy2DFromArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`,  
`cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpy2DToArrayAsync`,  
`cudaMemcpyFromArrayAsync`, `cudaMemcpy2DFromArrayAsync`,  
`cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`

```
__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](#),  
[cudaErrorInvalidDevicePointer](#), [cudaErrorInvalidMemcpyDirection](#)

### 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.

See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`,  
`cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyArrayToArray`,  
`cudaMemcpy2DArrayToArray`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`,  
`cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`,  
`cudaMemcpyFromArrayAsync`, `cudaMemcpy2DFromArrayAsync`,  
`cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`

**`__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`,  
`cudaErrorInvalidDevicePointer`, `cudaErrorInvalidMemcpyDirection`

### 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.

#### See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`, `cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyArrayToArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyFromSymbolAsync`

## `__host__ cudaError_t cudaMemcpyGetInfo (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`, `cudaErrorInitializationError`, `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.

```
__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.

#### See also:

`cudaMemcpy`, `cudaMemcpyPeer`, `cudaMemcpyAsync`, `cudaMemcpy3DPeerAsync`, `cudaMemAdvise`



```
__host__ cudaError_t cudaMemRangeGetAttribute  
(void *data, size_t dataSize, cudaMemRangeAttribute  
attribute, const void *devPtr, size_t count)
```

Query an attribute of a given memory range.

### Parameters

#### **data**

- A pointers to a memory location where the result of each attribute query will be written to.

#### **dataSize**

- Array containing the size of data

#### **attribute**

- The attribute to query

#### **devPtr**

- Start of the range to query

#### **count**

- Size of the range to query

### Returns

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

### Description

Query an attribute about the memory range starting at `devPtr` with a size of `count` bytes. The memory range must refer to managed memory allocated via [cudaMallocManaged](#) or declared via `__managed__` variables.

The `attribute` parameter can take the following values:

- ▶ [cudaMemRangeAttributeReadMostly](#): If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. The result returned will be 1 if all pages in the given memory range have read-duplication enabled, or 0 otherwise.
- ▶ [cudaMemRangeAttributePreferredLocation](#): If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. The result returned will be a GPU device id if all pages in the memory range have that GPU as their preferred location, or it will be `cudaCpuDeviceId` if all pages in the memory range have the CPU as their preferred location, or it will be `cudaInvalidDeviceId` if either all the pages don't have the same preferred location or some of the pages don't have a preferred location at all. Note that the actual location of the pages in the memory range at the time of the query may be different from the preferred location.

- ▶ **cudaMemRangeAttributeAccessedBy**: If this attribute is specified, `data` will be interpreted as an array of 32-bit integers, and `dataSize` must be a non-zero multiple of 4. The result returned will be a list of device ids that had `cudaMemAdviceSetAccessedBy` set for that entire memory range. If any device does not have that advice set for the entire memory range, that device will not be included. If `data` is larger than the number of devices that have that advice set for that memory range, `cudaInvalidDeviceId` will be returned in all the extra space provided. For ex., if `dataSize` is 12 (i.e. `data` has 3 elements) and only device 0 has the advice set, then the result returned will be { 0, `cudaInvalidDeviceId`, `cudaInvalidDeviceId` }. If `data` is smaller than the number of devices that have that advice set, then only as many devices will be returned as can fit in the array. There is no guarantee on which specific devices will be returned, however.
- ▶ **cudaMemRangeAttributeLastPrefetchLocation**: If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. The result returned will be the last location to which all pages in the memory range were prefetched explicitly via `cudaMemPrefetchAsync`. This will either be a GPU id or `cudaCpuDeviceId` depending on whether the last location for prefetch was a GPU or the CPU respectively. If any page in the memory range was never explicitly prefetched or if all pages were not prefetched to the same location, `cudaInvalidDeviceId` will be returned. Note that this simply returns the last location that the 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.

See also:

`cudaMemRangeGetAttributes`, `cudaMemPrefetchAsync`, `cudaMemAdvise`

```
__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.

**See also:**

[cudaMemRangeGetAttribute](#), [cudaMemAdvise](#) [cudaMemPrefetchAsync](#)

## **\_\_host\_\_ cudaError\_t cudaMemset (void \*devPtr, int value, size\_t count)**

Initializes or sets device memory to a value.

**Parameters****devPtr**

- Pointer to device memory

**value**

- Value to set for each byte of specified memory

**count**

- Size in bytes to set

**Returns**

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

**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](#).

**See also:**

[cudaMemset2D](#), [cudaMemset3D](#), [cudaMemsetAsync](#), [cudaMemset2DAsync](#), [cudaMemset3DAsync](#)

## `__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`, `cudaErrorInvalidDevicePointer`

### 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](#).

### See also:

`cudaMemset`, `cudaMemset3D`, `cudaMemsetAsync`, `cudaMemset2DAsync`, `cudaMemset3DAsync`

```
__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](#), [cudaErrorInvalidDevicePointer](#)

### 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.

See also:

[cudaMemset](#), [cudaMemset2D](#), [cudaMemset3D](#), [cudaMemsetAsync](#),  
[cudaMemset3DAsync](#)

## **\_\_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](#), [cudaErrorInvalidDevicePointer](#)

### 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](#).

#### 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](#), [cudaErrorInvalidDevicePointer](#)

#### 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.

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`, `cudaErrorInvalidDevicePointer`

### 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.

See also:

[cudaMemset](#), [cudaMemset2D](#), [cudaMemset3D](#), [cudaMemset2DAsync](#),  
[cudaMemset3DAsync](#)

## `__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`

## 4.10. 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.

The `cudaPointerAttributes` structure is defined as:

```
struct cudaPointerAttributes {
    enum cudaMemoryType
        memoryType;
    int device;
    void *devicePointer;
    void *hostPointer;
    int isManaged;
}
```

In this structure, the individual fields mean

- ▶ `memoryType` identifies the physical location of the memory associated with pointer `ptr`. It can be `cudaMemoryTypeHost` for host memory or `cudaMemoryTypeDevice` for device 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.

## See also:

`cudaGetDeviceCount`, `cudaGetDevice`, `cudaSetDevice`, `cudaChooseDevice`

## 4.11. 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.

### See also:

`cudaDeviceEnablePeerAccess`, `cudaDeviceDisablePeerAccess`

## `__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.

### See also:

`cudaDeviceCanAccessPeer`, `cudaDeviceEnablePeerAccess`

## `__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.

Each device can support a system-wide maximum of eight peer connections.

Peer access is not supported in 32 bit applications.

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.

See also:

`cudaDeviceCanAccessPeer`, `cudaDeviceDisablePeerAccess`

## 4.12. 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 Interoperability](#).

### 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`

**`__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`

**`__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`

## `__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

## 4.13. 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`, `cudaErrorInvalidDevicePointer`, `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`, `cudaErrorInvalidDevicePointer`, `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](#)

## **4.14. 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](#),

**\_\_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](#)

## **\_\_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](#)

## **\_\_host\_\_ cudaError\_t cudaD3D9SetDirect3DDevice (IDirect3DDevice9 \*pD3D9Device, int device)**

Sets the Direct3D 9 device to use for interoperability with a CUDA device.

### Parameters

#### **pD3D9Device**

- Direct3D device to use for this thread

#### **device**

- The CUDA device to use. This device must be among the devices returned when querying [cudaD3D9DeviceListAll](#) from [cudaD3D9GetDevices](#), may be set to -1 to automatically select an appropriate CUDA device.

### Returns

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

## Description

Records `pD3D9Device` as the Direct3D 9 device to use for Direct3D 9 interoperability with the CUDA device `device` and sets `device` as the current device for the calling host thread.

If `device` has already been initialized then this call will fail with the error `cudaErrorSetOnActiveProcess`. In this case it is necessary to reset `device` using `cudaDeviceReset()` before Direct3D 9 interoperability on `device` may be enabled.

Successfully initializing CUDA interoperability with `pD3D9Device` will increase the internal reference count on `pD3D9Device`. This reference count will be decremented when `device` is reset using `cudaDeviceReset()`.

Note that this function is never required for correct functionality. Use of this function will result in accelerated interoperability only when the operating system is Windows Vista or Windows 7, and the device `pD3DDDevice` is not an `IDirect3DDevice9Ex`. In all other circumstances, this function is not necessary.



Note 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`

## 4.15. 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 `*ppArray` an array through which the subresource of the mapped Direct3D resource `pResource`, which corresponds to `face` and `level` may be accessed. The value set in `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 `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:

$$y * \text{pitch} + (\text{bytes per pixel}) * x$$

For a 3D surface, the byte offset of the sample at position *x*, *y*, *z* from the base pointer of the surface is:

$$z * \text{slicePitch} + y * \text{pitch} + (\text{bytes per pixel}) * x$$

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

If `pResource` is not of type `IDirect3DBaseTexture9` or one of its sub-types or if `pResource` has not been registered for use with CUDA, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` was not registered with usage flags `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 be 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`

## 4.16. 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](#),

**\_\_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](#)



```

__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.
- ▶ 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 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`

## 4.17. 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 (CUarray`

## **`**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`

## 4.18. 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](#)

**\_\_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](#)

```

__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.
- ▶ 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 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`

## 4.19. 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`

## 4.20. 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`

**`__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`

**`__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](#)

```
__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`

## 4.21. 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 Interoperability](#).

```
__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`

**Description**

Acquire an image frame from EGLStreamKHR.

`cudaGraphicsResourceGetMappedEglFrame` can be called on `pCudaResource` to get `cudaEglFrame`.

**See also:**

`cudaEGLStreamConsumerConnect`, `cudaEGLStreamConsumerDisconnect`,  
`cudaEGLStreamConsumerReleaseFrame`

**`__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`

**`__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](#)

## [\\_\\_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](#),

**`__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](#)

## **\_\_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](#)

## **\_\_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;
```

See also:

`cudaEGLStreamProducerConnect`, `cudaEGLStreamProducerDisconnect`,  
`cudaEGLStreamProducerReturnFrame`

**\_\_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`,

## `__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`

**`__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`

## 4.22. 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.

See also:

`cudaGraphicsResourceGetMappedPointer`, `cudaGraphicsSubResourceGetMappedArray`, `cudaGraphicsUnmapResources`

```
__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.

See also:

[cudaGraphicsResourceGetMappedPointer](#)

**\_\_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.

See also:

[cudaGraphicsMapResources](#), [cudaGraphicsSubResourceGetMappedArray](#)

## `__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.

### See also:

[cudaGraphicsMapResources](#)

**\_\_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.

### See also:

[cudaGraphicsResourceGetMappedPointer](#)

```
__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.

### See also:

[cudaGraphicsMapResources](#)

## `__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.

### See also:

`cudaGraphicsD3D9RegisterResource`, `cudaGraphicsD3D10RegisterResource`,  
`cudaGraphicsD3D11RegisterResource`, `cudaGraphicsGLRegisterBuffer`,  
`cudaGraphicsGLRegisterImage`

## 4.23. Texture Reference Management

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](#), [cudaErrorInvalidDevicePointer](#),  
[cudaErrorInvalidTexture](#)

### Description

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.



**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 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](#), [cudaErrorInvalidDevicePointer](#), [cudaErrorInvalidTexture](#)

**Description**

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.

#### 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)

**`__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`, `cudaErrorInvalidDevicePointer`, `cudaErrorInvalidTexture`

## Description

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.

## 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

**`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`, `cudaErrorInvalidDevicePointer`, `cudaErrorInvalidTexture`

## Description

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.

#### See also:

[cudaCreateChannelDesc](#) ( C API), [cudaGetChannelDesc](#), [cudaGetTextureReference](#), [cudaBindTexture](#) ( C API), [cudaBindTexture2D](#) ( C API), [cudaBindTextureToArray](#) ( C++ API), [cudaUnbindTexture](#) ( C API), [cudaGetTextureAlignmentOffset](#) ( C API)

## `__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`, `cudaGetTextureReference`, `cudaBindTexture` ( C API), `cudaBindTexture2D` ( C API), `cudaBindTextureToArray` ( C API), `cudaUnbindTexture` ( C API), `cudaGetTextureAlignmentOffset` ( C API)

## `__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.

### See also:

`cudaCreateChannelDesc` ( C API), `cudaGetTextureReference`, `cudaBindTexture` ( C API), `cudaBindTexture2D` ( C API), `cudaBindTextureToArray` ( C API), `cudaUnbindTexture` ( C API), `cudaGetTextureAlignmentOffset` ( C API)

## `__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

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.

## 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

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.

**See also:**

[cudaCreateChannelDesc](#) ( C API), [cudaGetChannelDesc](#),  
[cudaGetTextureAlignmentOffset](#) ( C API), [cudaBindTexture](#) ( C API),  
[cudaBindTexture2D](#) ( C API), [cudaBindTextureToArray](#) ( C API), [cudaUnbindTexture](#)  
( C API)

## **\_\_host\_\_ cudaError\_t cudaUnbindTexture (const textureReference \*texref)**

Unbinds a texture.

**Parameters****texref**

- Texture to unbind

**Returns**

[cudaSuccess](#)

**Description**

Unbinds the texture bound to `texref`.



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

**See also:**

[cudaCreateChannelDesc](#) ( C API), [cudaGetChannelDesc](#), [cudaGetTextureReference](#),  
[cudaBindTexture](#) ( C API), [cudaBindTexture2D](#) ( C API), [cudaBindTextureToArray](#) ( C  
API), [cudaUnbindTexture](#) ( C++ API), [cudaGetTextureAlignmentOffset](#) ( C API)

## 4.24. Surface Reference Management

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

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.

### See also:

[cudaBindSurfaceToArray \( C++ API\)](#), [cudaBindSurfaceToArray \( C++ API, inherited channel descriptor\)](#), [cudaGetSurfaceReference](#)

**\_\_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**

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` paramater was removed in CUDA 5.0.

**See also:**

`cudaBindSurfaceToArray` ( C API)

## 4.25. 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__ 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.

**See also:**

[`cudaDestroyTextureObject`](#)

## `__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`.

**See also:**

[`cudaCreateTextureObject`](#)

**\_\_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`.

#### See also:

[cudaCreateTextureObject](#)

**\_\_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.

**See also:**

`cudaCreateTextureObject`

```
__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`.

**See also:**

`cudaCreateTextureObject`

## 4.26. 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](#)

#### 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.

#### See also:

[cudaDestroySurfaceObject](#)

```
__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`.

**See also:**

[cudaCreateSurfaceObject](#)

**\_\_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****See also:**

[cudaCreateSurfaceObject](#)

## 4.27. Version Management

**\_\_host\_\_ cudaError\_t cudaDriverGetVersion (int  
\*driverVersion)**

Returns the CUDA driver version.

**Parameters**

**driverVersion**

- Returns the CUDA driver version.

## Returns

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

## Description

Returns in `*driverVersion` the version number of the installed CUDA driver. If no driver is installed, then 0 is returned as the driver version (via `driverVersion`). This function automatically returns [cudaErrorInvalidValue](#) if the `driverVersion` argument is NULL.



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

## See also:

[cudaRuntimeGetVersion](#)

**`__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 installed CUDA Runtime. This function automatically returns [cudaErrorInvalidValue](#) if the `runtimeVersion` argument is NULL.

## See also:

[cudaDriverGetVersion](#)

## 4.28. 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.

#### 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.

### 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`, `cudaErrorInvalidDevicePointer`,  
`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.

### 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](#), [cudaErrorInvalidDevicePointer](#),  
[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.

### 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](#), [cudaErrorInvalidDevicePointer](#), [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.

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`, `cudaErrorInvalidDevicePointer`, `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.

#### 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`, `cudaErrorInvalidDevicePointer`,  
`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.

#### 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`, `cudaErrorInvalidDevicePointer`,  
`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.

**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`, `cudaErrorInvalidDevicePointer`, `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.

**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`, `cudaErrorInvalidDevicePointer`,  
`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.

### 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`, `cudaErrorInitializationError`, `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.

## 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`, `cudaErrorInitializationError`, `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

fetches 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.

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

```
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`, `cudaErrorInitializationError`, `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.

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

### 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.

### 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](#)

### 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.

### 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.

**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 cudaLaunch (T \*func)

[C++ API] Launches a device function

**Parameters****func**

- Device function pointer to execute

**Returns**

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

**Description**

**Deprecated** This function is deprecated as of CUDA 7.0

Launches the function `func` on the device. The parameter `func` must be a function that executes on the device. The parameter specified by `func` must be declared as a `__global__` function. `cudaLaunch()` must be preceded by a call to `cudaConfigureCall()` since it pops the data that was pushed by `cudaConfigureCall()` from the execution stack.



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

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

```
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`

### 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.

[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.

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.

#### 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`,  
`cudaErrorInvalidDevicePointer`, `cudaErrorInvalidMemcpyDirection`

## 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`.



- 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.

## See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`, `cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyArrayToArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`, `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`,  
`cudaErrorInvalidDevicePointer`, `cudaErrorInvalidMemcpyDirection`

## 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.

## See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`,  
`cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyArrayToArray`,  
`cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`,  
`cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`,  
`cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`,  
`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`,  
`cudaErrorInvalidDevicePointer`, `cudaErrorInvalidMemcpyDirection`

**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.

**See also:**

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`,  
`cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyArrayToArray`,  
`cudaMemcpy2DArrayToArray`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`,  
`cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`,  
`cudaMemcpyFromArrayAsync`, `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](#), [cudaErrorInvalidDevicePointer](#), [cudaErrorInvalidMemcpyDirection](#)

### 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.

See also:

`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`,  
`cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyArrayToArray`,  
`cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`,  
`cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`,  
`cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`,  
`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`, `cudaErrorCudartUnloading`, `cudaErrorInitializationError`,  
`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.

### 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`, `cudaErrorCudartUnloading`, `cudaErrorInitializationError`,  
`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.

#### 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`, `cudaErrorCudartUnloading`, `cudaErrorInitializationError`,  
`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.

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`, `cudaErrorCudartUnloading`, `cudaErrorInitializationError`,  
`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.

**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`, `cudaErrorCudartUnloading`, `cudaErrorInitializationError`, `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.

#### 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](#), [cudaErrorCudartUnloading](#), [cudaErrorInitializationError](#),  
[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.

See also:

`cudaOccupancyMaxPotentialBlockSize`

`cudaOccupancyMaxActiveBlocksPerMultiprocessor`

`cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags`

`cudaOccupancyMaxPotentialBlockSizeVariableSMem`

`cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags`

## template < class T > \_\_host\_\_ cudaError\_t cudaSetupArgument (T arg, size\_t offset)

[C++ API] Configure a device launch

### Parameters

#### arg

- Argument to push for a kernel launch

#### offset

- Offset in argument stack to push new arg

### Returns

[cudaSuccess](#)

### Description

[Deprecated](#) This function is deprecated as of CUDA 7.0

Pushes `size` bytes of the argument pointed to by `arg` at `offset` bytes from the start of the parameter passing area, which starts at offset 0. The arguments are stored in the top of the execution stack. [cudaSetupArgument\(\)](#) must be preceded by a call to [cudaConfigureCall\(\)](#).



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

[cudaLaunchKernel](#) ( C++ API), [cudaFuncGetAttributes](#) ( C++ API), [cudaLaunch](#) ( C++ API), [cudaSetDoubleForDevice](#), [cudaSetDoubleForHost](#), [cudaSetupArgument](#) ( C API)

## 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)

#### length

- Length of memory (must be zero, 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 address within managed memory space declared using the `__managed__` keyword or allocated with `cudaMallocManaged`.

`length` must be zero, to indicate that the entire allocation's stream association is being changed. Currently, it's not possible to change stream association for a portion of an allocation. The default value for `length` is 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.

#### 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](#)

#### Description

Unbinds the texture bound to `tex`.



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

#### 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)

## 4.29. 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`.

## 4.30. 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 output file 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`

## `__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`

## `__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`

## 4.31. Data types used by CUDA Runtime

`struct cudaChannelFormatDesc`  
`struct cudaDeviceProp`  
`struct cudaEglFrame`  
`struct cudaEglPlaneDesc`  
`struct cudaExtent`  
`struct cudaFuncAttributes`  
`struct cudaIpcEventHandle_t`  
`struct cudaIpcMemHandle_t`  
`struct cudaMemcpy3DParms`  
`struct cudaMemcpy3DPeerParms`  
`struct cudaPitchedPtr`  
`struct cudaPointerAttributes`  
`struct cudaPos`  
`struct cudaResourceDesc`  
`struct cudaResourceViewDesc`  
`struct cudaTextureDesc`  
`struct surfaceReference`  
`struct textureReference`

## 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

## 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

## 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), 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, width, height ratio same as YUV422Planar.

**cudaEglColorFormatRGB = 4**

R/G/B three channels in one surface with RGB byte ordering.

**cudaEglColorFormatBGR = 5**

R/G/B three channels in one surface with BGR byte ordering.

**cudaEglColorFormatARGB = 6**

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

**cudaEglColorFormatRGBA = 7**

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

**cudaEglColorFormatL = 8**

single luminance channel in one surface.

**cudaEglColorFormatR = 9**

single color channel in one surface.

## enum cudaEglFrameType

CUDA EglFrame type - array or pointer

### Values

**cudaEglFrameTypeArray = 0**

Frame type CUDA array

**cudaEglFrameTypePitch = 1**

Frame type CUDA pointer

## enum cudaEglResourceLocationFlags

Resource location flags- sysmem or vidmem

For CUDA context on iGPU, since video and system memory are equivalent - these flags will not have an effect on the execution.

For CUDA context on dGPU, applications can use the flag [cudaEglResourceLocationFlags](#) to give a hint about the desired location.

[cudaEglResourceLocationSystemem](#) - 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 can also mean that the operation being queried is complete (see [cudaEventQuery\(\)](#) and [cudaStreamQuery\(\)](#)).

**cudaErrorMissingConfiguration = 1**

The device function being invoked (usually via `cudaLaunchKernel()`) was not previously configured via the `cudaConfigureCall()` function.

#### **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.

#### **cudaErrorLaunchFailure = 4**

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. The device cannot be used until `cudaThreadExit()` is called. All existing device memory allocations are invalid and must be reconstructed if the program is to continue using CUDA.

#### **cudaErrorPriorLaunchFailure = 5**

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.

#### **cudaErrorLaunchTimeout = 6**

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.

#### **cudaErrorLaunchOutOfResources = 7**

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.

#### **cudaErrorInvalidDeviceFunction = 8**

The requested device function does not exist or is not compiled for the proper device architecture.

#### **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.

#### **cudaErrorInvalidDevice = 10**

This indicates that the device ordinal supplied by the user does not correspond to a valid CUDA device.

#### **cudaErrorInvalidValue = 11**

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

**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.

**cudaErrorMapBufferObjectFailed = 14**

This indicates that the buffer object could not be mapped.

**cudaErrorUnmapBufferObjectFailed = 15**

This indicates that the buffer object could not be unmapped.

**cudaErrorInvalidHostPointer = 16**

This indicates that at least one host pointer passed to the API call is not a valid host pointer.

**cudaErrorInvalidDevicePointer = 17**

This indicates that at least one device pointer passed to the API call is not a valid device pointer.

**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.

#### **cudaErrorCudartUnloading = 29**

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.

#### **cudaErrorUnknown = 30**

This indicates that an unknown internal error has occurred.

#### **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.

#### **cudaErrorInvalidResourceHandle = 33**

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`.

#### **cudaErrorNotReady = 34**

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()`.

#### **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.

#### **cudaErrorSetOnActiveProcess = 36**

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.

**cudaErrorInvalidSurface = 37**

This indicates that the surface passed to the API call is not a valid surface.

**cudaErrorNoDevice = 38**

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

**cudaErrorECCUncorrectable = 39**

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

**cudaErrorSharedObjectSymbolNotFound = 40**

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

**cudaErrorSharedObjectInitFailed = 41**

This indicates that initialization of a shared object failed.

**cudaErrorUnsupportedLimit = 42**

This indicates that the `cudaLimit` passed to the API call is not supported by the active device.

**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.

**cudaErrorInvalidKernelImage = 47**

This indicates that the device kernel image is invalid.

**cudaErrorNoKernelImageForDevice = 48**

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.

**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.

**cudaErrorPeerAccessAlreadyEnabled = 50**

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 = 51**

This error indicates that [cudaDeviceDisablePeerAccess\(\)](#) is trying to disable peer addressing which has not been enabled yet via [cudaDeviceEnablePeerAccess\(\)](#).

**cudaErrorDeviceAlreadyInUse = 54**

This indicates that a call tried to access an exclusive-thread device that is already in use by a different thread.

**cudaErrorProfilerDisabled = 55**

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 = 56**

[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 = 57**

[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 = 58**

[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.

**cudaErrorAssert = 59**

An assert triggered in device code during kernel execution. The device cannot be used again until [cudaThreadExit\(\)](#) is called. All existing allocations are invalid and must be reconstructed if the program is to continue using CUDA.

**cudaErrorTooManyPeers = 60**

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 = 61**

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

**cudaErrorHostMemoryNotRegistered = 62**

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

**cudaErrorOperatingSystem = 63**

This error indicates that an OS call failed.

**cudaErrorPeerAccessUnsupported = 64**

This error indicates that P2P access is not supported across the given devices.

#### **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.

#### **cudaErrorNotPermitted = 70**

This error indicates the attempted operation is not permitted.

#### **cudaErrorNotSupported = 71**

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

#### **cudaErrorHardwareStackError = 72**

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 = 73**

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 = 74**

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 = 75**

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 = 76**

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.

#### **cudaErrorIllegalAddress = 77**

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.

#### **cudaErrorInvalidPtx = 78**

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 = 79**

This indicates an error with the OpenGL or DirectX context.

#### **cudaErrorNvlinkUncorrectable = 80**

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

#### **cudaErrorStartupFailure = 0x7f**

This indicates an internal startup failure in the CUDA runtime.

#### **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 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 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 cudaLimit

CUDA Limits

## Values

**cudaLimitStackSize = 0x00**

GPU thread stack size

**cudaLimitPrintfFifoSize = 0x01**

GPU printf/fprintf FIFO size

**cudaLimitMallocHeapSize = 0x02**

GPU malloc heap size

**cudaLimitDevRuntimeSyncDepth = 0x03**

GPU device runtime synchronize depth

**cudaLimitDevRuntimePendingLaunchCount = 0x04**

GPU device runtime pending launch count

## 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

**cudaMemoryTypeHost = 1**

Host memory

**cudaMemoryTypeDevice = 2**

Device 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 cudaSharedMemConfig

CUDA shared memory configuration

### Values

**cudaSharedMemBankSizeDefault = 0**

**cudaSharedMemBankSizeFourByte = 1**

**cudaSharedMemBankSizeEightByte = 2**

## 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 EGLStream Connection

```
typedef cudaError_t
```

CUDA Error types

```
typedef struct CUevent_st *cudaEvent_t
```

CUDA event types

```
typedef cudaGraphicsResource *cudaGraphicsResource_t
```

CUDA graphics resource types

```
typedef cudaMipmappedArray  
*cudaMipmappedArray_const_t
```

CUDA mipmapped array (as source argument)

```
typedef cudaMipmappedArray *cudaMipmappedArray_t
```

CUDA mipmapped array

```
typedef cudaOutputMode_t
```

CUDA output file modes

```
typedef struct CUstream_st *cudaStream_t
```

CUDA stream

```
typedef unsigned long long cudaSurfaceObject_t
```

An opaque value that represents a CUDA Surface object

```
typedef unsigned long long cudaTextureObject_t
```

An opaque value that represents a CUDA texture object

```
typedef struct CUuuid_st cudaUUID_t
```

CUDA UUID types

```
#define CUDA_EGL_MAX_PLANES 3
```

Maximum number of planes per frame

```
#define CUDA_IPC_HANDLE_SIZE 64
```

CUDA IPC Handle Size

```
#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 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 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 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 5.

## DATA STRUCTURES

Here are the data structures with brief descriptions:

`__cudaOccupancyB2DHelper`  
`cudaChannelFormatDesc`  
`cudaDeviceProp`  
`cudaEglFrame`  
`cudaEglPlaneDesc`  
`cudaExtent`  
`cudaFuncAttributes`  
`cudaIpcEventHandle_t`  
`cudaIpcMemHandle_t`  
`cudaMemcpy3DParms`  
`cudaMemcpy3DPeerParms`  
`cudaPitchedPtr`  
`cudaPointerAttributes`  
`cudaPos`  
`cudaResourceDesc`  
`cudaResourceViewDesc`  
`cudaTextureDesc`  
`surfaceReference`  
`textureReference`

### 5.1. `__cudaOccupancyB2DHelper`

C++ API Routines `cppClassifierVisibility: visibility=public` `cppClassifierTemplateModel:`  
`=`

Helper functor for `cudaOccupancyMaxPotentialBlockSize`

## 5.2. 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*

## 5.3. 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::clockRate`

Clock frequency in kilohertz

## `int cudaDeviceProp::computeMode`

Compute mode (See `cudaComputeMode`)

## `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::deviceOverlap`

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

## `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

## `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::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::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

## `int cudaDeviceProp::warpSize`

Warp size in threads

## 5.4. 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

## 5.5. `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

## 5.6. 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

## 5.7. 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::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::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.

## 5.8. `cudaIpcEventHandle_t` Struct Reference

CUDA IPC event handle

## 5.9. `cudaIpcMemHandle_t` Struct Reference

CUDA IPC memory handle

## 5.10. `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

## 5.11. 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

## 5.12. 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

## 5.13. 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](#) 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.

## `int cudaPointerAttributes::isManaged`

Indicates if this pointer points to managed memory

## `enumcudaMemoryType`

### `cudaPointerAttributes::memoryType`

The physical location of the memory, `cudaMemoryTypeHost` or `cudaMemoryTypeDevice`.

## 5.14. 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



## 5.15. cudaResourceDesc Struct Reference

CUDA resource descriptor

`cudaArray_t cudaResourceDesc::array`

CUDA array

`struct cudaChannelFormatDesc cudaResourceDesc::desc`

Channel descriptor

`void *cudaResourceDesc::devPtr`

Device pointer

`size_t cudaResourceDesc::height`

Height of the array in elements

`cudaMipmappedArray_t cudaResourceDesc::mipmap`

CUDA mipmapped array

`size_t cudaResourceDesc::pitchInBytes`

Pitch between two rows in bytes

`enum cudaResourceType cudaResourceDesc::resType`

Resource type

`size_t cudaResourceDesc::sizeInBytes`

Size in bytes

`size_t cudaResourceDesc::width`

Width of the array in elements

## 5.16. 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

## 5.17. 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

## 5.18. surfaceReference Struct Reference

CUDA Surface reference

```
struct cudaChannelFormatDesc
surfaceReference::channelDesc
```

Channel descriptor for surface reference

## 5.19. textureReference Struct Reference

CUDA texture reference

```
enum cudaTextureAddressMode
textureReference::addressMode
```

Texture address mode for up to 3 dimensions

```
struct cudaChannelFormatDesc
textureReference::channelDesc
```

Channel descriptor for the texture reference

```
enum cudaTextureFilterMode
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

## `enum cudaTextureFilterMode`

## `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 6.

## 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](#)

#### **array**

- [cudaResourceDesc](#)

#### **asyncEngineCount**

- [cudaDeviceProp](#)

### B

#### **binaryVersion**

- [cudaFuncAttributes](#)

#### **borderColor**

- [cudaTextureDesc](#)

### C

#### **cacheModeCA**

- [cudaFuncAttributes](#)

#### **canMapHostMemory**

- [cudaDeviceProp](#)

#### **channelDesc**

- [surfaceReference](#)
- [textureReference](#)
- [cudaEglPlaneDesc](#)

#### **clockRate**

- [cudaDeviceProp](#)

**computeMode**

cudaDeviceProp

**concurrentKernels**

cudaDeviceProp

**concurrentManagedAccess**

cudaDeviceProp

**constSizeBytes**

cudaFuncAttributes

**D****depth**

cudaEglPlaneDesc

cudaExtent

cudaResourceViewDesc

**desc**

cudaResourceDesc

**device**

cudaPointerAttributes

**deviceOverlap**

cudaDeviceProp

**devicePointer**

cudaPointerAttributes

**devPtr**

cudaResourceDesc

**dstArray**

cudaMemcpy3DParms

cudaMemcpy3DPeerParms

**dstDevice**

cudaMemcpy3DPeerParms

**dstPos**

cudaMemcpy3DPeerParms

cudaMemcpy3DParms

**dstPtr**

cudaMemcpy3DParms

cudaMemcpy3DPeerParms

**E****ECCEnabled**

cudaDeviceProp

**eglColorFormat**

cudaEglFrame

**extent**

cudaMemcpy3DParms

`cudaMemcpy3DPeerParms`

## F

### f

`cudaChannelFormatDesc`

### filterMode

`textureReference`

`cudaTextureDesc`

### firstLayer

`cudaResourceViewDesc`

### firstMipmapLevel

`cudaResourceViewDesc`

### format

`cudaResourceViewDesc`

### frameType

`cudaEglFrame`

## G

### globalL1CacheSupported

`cudaDeviceProp`

## H

### height

`cudaEglPlaneDesc`

`cudaExtent`

`cudaResourceViewDesc`

`cudaResourceDesc`

### hostNativeAtomicSupported

`cudaDeviceProp`

### hostPointer

`cudaPointerAttributes`

## I

### integrated

`cudaDeviceProp`

### isManaged

`cudaPointerAttributes`

### isMultiGpuBoard

`cudaDeviceProp`

## K

### kernelExecTimeoutEnabled

`cudaDeviceProp`



**kind**

[cudaMemcpy3DParms](#)

**L****l2CacheSize**

[cudaDeviceProp](#)

**lastLayer**

[cudaResourceViewDesc](#)

**lastMipmapLevel**

[cudaResourceViewDesc](#)

**localL1CacheSupported**

[cudaDeviceProp](#)

**localSizeBytes**

[cudaFuncAttributes](#)

**M****major**

[cudaDeviceProp](#)

**managedMemory**

[cudaDeviceProp](#)

**maxAnisotropy**

[cudaTextureDesc](#)  
[textureReference](#)

**maxGridSize**

[cudaDeviceProp](#)

**maxMipmapLevelClamp**

[textureReference](#)  
[cudaTextureDesc](#)

**maxSurface1D**

[cudaDeviceProp](#)

**maxSurface1DLayered**

[cudaDeviceProp](#)

**maxSurface2D**

[cudaDeviceProp](#)

**maxSurface2DLayered**

[cudaDeviceProp](#)

**maxSurface3D**

[cudaDeviceProp](#)

**maxSurfaceCubemap**

[cudaDeviceProp](#)

**maxSurfaceCubemapLayered**

[cudaDeviceProp](#)

**maxTexture1D**  
    cudaDeviceProp

**maxTexture1DLayered**  
    cudaDeviceProp

**maxTexture1DLinear**  
    cudaDeviceProp

**maxTexture1DMipmap**  
    cudaDeviceProp

**maxTexture2D**  
    cudaDeviceProp

**maxTexture2DGather**  
    cudaDeviceProp

**maxTexture2DLayered**  
    cudaDeviceProp

**maxTexture2DLinear**  
    cudaDeviceProp

**maxTexture2DMipmap**  
    cudaDeviceProp

**maxTexture3D**  
    cudaDeviceProp

**maxTexture3DAlt**  
    cudaDeviceProp

**maxTextureCubemap**  
    cudaDeviceProp

**maxTextureCubemapLayered**  
    cudaDeviceProp

**maxThreadsDim**  
    cudaDeviceProp

**maxThreadsPerBlock**  
    cudaFuncAttributes  
    cudaDeviceProp

**maxThreadsPerMultiProcessor**  
    cudaDeviceProp

**memoryBusWidth**  
    cudaDeviceProp

**memoryClockRate**  
    cudaDeviceProp

**memoryType**  
    cudaPointerAttributes

**memPitch**  
    cudaDeviceProp

**minMipmapLevelClamp**  
    textureReference

cudaTextureDesc  
**minor**  
 cudaDeviceProp  
**mipmap**  
 cudaResourceDesc  
**mipmapFilterMode**  
 cudaTextureDesc  
 textureReference  
**mipmapLevelBias**  
 textureReference  
 cudaTextureDesc  
**multiGpuBoardGroupID**  
 cudaDeviceProp  
**multiProcessorCount**  
 cudaDeviceProp

**N**  
**name**  
 cudaDeviceProp  
**normalized**  
 textureReference  
**normalizedCoords**  
 cudaTextureDesc  
**numChannels**  
 cudaEglPlaneDesc  
**numRegs**  
 cudaFuncAttributes

**P**  
**pageableMemoryAccess**  
 cudaDeviceProp  
**pArray**  
 cudaEglFrame  
**pciBusID**  
 cudaDeviceProp  
**pciDeviceID**  
 cudaDeviceProp  
**pciDomainID**  
 cudaDeviceProp  
**pitch**  
 cudaEglPlaneDesc  
 cudaPitchedPtr

**pitchInBytes**  
     [cudaResourceDesc](#)  
**planeCount**  
     [cudaEglFrame](#)  
**planeDesc**  
     [cudaEglFrame](#)  
**pPitch**  
     [cudaEglFrame](#)  
**ptr**  
     [cudaPitchedPtr](#)  
**ptxVersion**  
     [cudaFuncAttributes](#)

**R**  
**readMode**  
     [cudaTextureDesc](#)  
**regsPerBlock**  
     [cudaDeviceProp](#)  
**regsPerMultiprocessor**  
     [cudaDeviceProp](#)  
**reserved**  
     [cudaEglPlaneDesc](#)  
**resType**  
     [cudaResourceDesc](#)

**S**  
**sharedMemPerBlock**  
     [cudaDeviceProp](#)  
**sharedMemPerMultiprocessor**  
     [cudaDeviceProp](#)  
**sharedSizeBytes**  
     [cudaFuncAttributes](#)  
**singleToDoublePrecisionPerfRatio**  
     [cudaDeviceProp](#)  
**sizeInBytes**  
     [cudaResourceDesc](#)  
**srcArray**  
     [cudaMemcpy3DParms](#)  
     [cudaMemcpy3DPeerParms](#)  
**srcDevice**  
     [cudaMemcpy3DPeerParms](#)  
**srcPos**  
     [cudaMemcpy3DParms](#)

[cudaMemcpy3DPeerParms](#)

### **srcPtr**

[cudaMemcpy3DParms](#)

[cudaMemcpy3DPeerParms](#)

### **sRGB**

[textureReference](#)

[cudaTextureDesc](#)

### **streamPrioritiesSupported**

[cudaDeviceProp](#)

### **surfaceAlignment**

[cudaDeviceProp](#)

## **T**

### **tccDriver**

[cudaDeviceProp](#)

### **textureAlignment**

[cudaDeviceProp](#)

### **texturePitchAlignment**

[cudaDeviceProp](#)

### **totalConstMem**

[cudaDeviceProp](#)

### **totalGlobalMem**

[cudaDeviceProp](#)

## **U**

### **unifiedAddressing**

[cudaDeviceProp](#)

## **W**

### **w**

[cudaChannelFormatDesc](#)

### **warpSize**

[cudaDeviceProp](#)

### **width**

[cudaResourceDesc](#)

[cudaExtent](#)

[cudaEglPlaneDesc](#)

[cudaResourceViewDesc](#)

## **X**

### **x**

[cudaChannelFormatDesc](#)

[cudaPos](#)

**xsize**`cudaPitchedPtr`**Y****y**`cudaChannelFormatDesc``cudaPos`**ysize**`cudaPitchedPtr`**Z****z**`cudaChannelFormatDesc``cudaPos`

# Chapter 7.

## DEPRECATED LIST

**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 `cudaConfigureCall`**

This function is deprecated as of CUDA 7.0

**Global `cudaLaunch`**

This function is deprecated as of CUDA 7.0

**Global `cudaSetupArgument`**

This function is deprecated as of CUDA 7.0

**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 `cudaLaunch`**

This function is deprecated as of CUDA 7.0

**Global `cudaSetupArgument`**

This function is deprecated as of CUDA 7.0

**Global `cudaErrorPriorLaunchFailure`**

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

**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 `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 `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**

© 2007-2016 NVIDIA Corporation. All rights reserved.