



# NVIDIA CUDA

## Reference Manual

Version 2.2

---

April 2009



# Contents

<b>1</b>	<b>Module Index</b>	<b>1</b>
1.1	Modules . . . . .	1
<b>2</b>	<b>Data Structure Index</b>	<b>3</b>
2.1	Data Structures . . . . .	3
<b>3</b>	<b>Module Documentation</b>	<b>5</b>
3.1	CUDA Runtime API . . . . .	5
3.1.1	Detailed Description . . . . .	5
3.2	Thread Management . . . . .	6
3.2.1	Detailed Description . . . . .	6
3.2.2	Function Documentation . . . . .	6
3.2.2.1	cudaThreadExit . . . . .	6
3.2.2.2	cudaThreadSynchronize . . . . .	6
3.3	Error Handling . . . . .	7
3.3.1	Detailed Description . . . . .	7
3.3.2	Function Documentation . . . . .	7
3.3.2.1	cudaGetErrorString . . . . .	7
3.3.2.2	cudaGetLastError . . . . .	7
3.4	Device Management . . . . .	9
3.4.1	Detailed Description . . . . .	9
3.4.2	Function Documentation . . . . .	9
3.4.2.1	cudaChooseDevice . . . . .	9
3.4.2.2	cudaGetDevice . . . . .	10
3.4.2.3	cudaGetDeviceCount . . . . .	10
3.4.2.4	cudaGetDeviceProperties . . . . .	10
3.4.2.5	cudaSetDevice . . . . .	12
3.4.2.6	cudaSetDeviceFlags . . . . .	12
3.4.2.7	cudaSetValidDevices . . . . .	13

3.5	Stream Management	14
3.5.1	Detailed Description	14
3.5.2	Function Documentation	14
3.5.2.1	cudaStreamCreate	14
3.5.2.2	cudaStreamDestroy	14
3.5.2.3	cudaStreamQuery	15
3.5.2.4	cudaStreamSynchronize	15
3.6	Event Management	16
3.6.1	Detailed Description	16
3.6.2	Function Documentation	16
3.6.2.1	cudaEventCreate	16
3.6.2.2	cudaEventCreateWithFlags	17
3.6.2.3	cudaEventDestroy	17
3.6.2.4	cudaEventElapsedTime	17
3.6.2.5	cudaEventQuery	18
3.6.2.6	cudaEventRecord	18
3.6.2.7	cudaEventSynchronize	19
3.7	Execution Control	20
3.7.1	Detailed Description	20
3.7.2	Function Documentation	20
3.7.2.1	cudaConfigureCall	20
3.7.2.2	cudaFuncGetAttributes	21
3.7.2.3	cudaLaunch	21
3.7.2.4	cudaSetDoubleForDevice	21
3.7.2.5	cudaSetDoubleForHost	22
3.7.2.6	cudaSetupArgument	22
3.8	Memory Management	23
3.8.1	Detailed Description	25
3.8.2	Function Documentation	25
3.8.2.1	cudaFree	25
3.8.2.2	cudaFreeArray	26
3.8.2.3	cudaFreeHost	26
3.8.2.4	cudaGetSymbolAddress	27
3.8.2.5	cudaGetSymbolSize	27
3.8.2.6	cudaHostAlloc	27
3.8.2.7	cudaHostGetDevicePointer	28
3.8.2.8	cudaMalloc	29

---

3.8.2.9	<code>cudaMalloc3D</code>	29
3.8.2.10	<code>cudaMalloc3DArray</code>	30
3.8.2.11	<code>cudaMallocArray</code>	31
3.8.2.12	<code>cudaMallocHost</code>	31
3.8.2.13	<code>cudaMallocPitch</code>	32
3.8.2.14	<code>cudaMemcpy</code>	33
3.8.2.15	<code>cudaMemcpy2D</code>	33
3.8.2.16	<code>cudaMemcpy2DArrayToArray</code>	34
3.8.2.17	<code>cudaMemcpy2DAsync</code>	35
3.8.2.18	<code>cudaMemcpy2DFromArray</code>	35
3.8.2.19	<code>cudaMemcpy2DFromArrayAsync</code>	36
3.8.2.20	<code>cudaMemcpy2DToArray</code>	37
3.8.2.21	<code>cudaMemcpy2DToArrayAsync</code>	38
3.8.2.22	<code>cudaMemcpy3D</code>	39
3.8.2.23	<code>cudaMemcpy3DAsync</code>	40
3.8.2.24	<code>cudaMemcpyArrayToArray</code>	42
3.8.2.25	<code>cudaMemcpyAsync</code>	42
3.8.2.26	<code>cudaMemcpyFromArray</code>	43
3.8.2.27	<code>cudaMemcpyFromArrayAsync</code>	44
3.8.2.28	<code>cudaMemcpyFromSymbol</code>	44
3.8.2.29	<code>cudaMemcpyFromSymbolAsync</code>	45
3.8.2.30	<code>cudaMemcpyToArray</code>	46
3.8.2.31	<code>cudaMemcpyToArrayAsync</code>	46
3.8.2.32	<code>cudaMemcpyToSymbol</code>	47
3.8.2.33	<code>cudaMemcpyToSymbolAsync</code>	48
3.8.2.34	<code>cudaMemset</code>	48
3.8.2.35	<code>cudaMemset2D</code>	49
3.8.2.36	<code>cudaMemset3D</code>	49
3.9	OpenGL Interoperability	51
3.9.1	Detailed Description	51
3.9.2	Function Documentation	51
3.9.2.1	<code>cudaGLMapBufferObject</code>	51
3.9.2.2	<code>cudaGLRegisterBufferObject</code>	52
3.9.2.3	<code>cudaGLSetGLDevice</code>	52
3.9.2.4	<code>cudaGLUnmapBufferObject</code>	52
3.9.2.5	<code>cudaGLUnregisterBufferObject</code>	53
3.9.2.6	<code>cudaWGLGetDevice</code>	53

3.10 Direct3D 9 Interoperability . . . . .	54
3.10.1 Detailed Description . . . . .	55
3.10.2 Enumeration Type Documentation . . . . .	55
3.10.2.1 cudaD3D9MapFlags . . . . .	55
3.10.2.2 cudaD3D9RegisterFlags . . . . .	55
3.10.3 Function Documentation . . . . .	55
3.10.3.1 cudaD3D9GetDevice . . . . .	55
3.10.3.2 cudaD3D9GetDirect3DDevice . . . . .	56
3.10.3.3 cudaD3D9MapResources . . . . .	56
3.10.3.4 cudaD3D9RegisterResource . . . . .	57
3.10.3.5 cudaD3D9ResourceGetMappedArray . . . . .	58
3.10.3.6 cudaD3D9ResourceGetMappedPitch . . . . .	59
3.10.3.7 cudaD3D9ResourceGetMappedPointer . . . . .	60
3.10.3.8 cudaD3D9ResourceGetMappedSize . . . . .	60
3.10.3.9 cudaD3D9ResourceGetSurfaceDimensions . . . . .	61
3.10.3.10 cudaD3D9ResourceSetMapFlags . . . . .	62
3.10.3.11 cudaD3D9SetDirect3DDevice . . . . .	63
3.10.3.12 cudaD3D9UnmapResources . . . . .	63
3.10.3.13 cudaD3D9UnregisterResource . . . . .	64
3.11 Direct3D 10 Interoperability . . . . .	65
3.11.1 Detailed Description . . . . .	66
3.11.2 Enumeration Type Documentation . . . . .	66
3.11.2.1 cudaD3D10MapFlags . . . . .	66
3.11.2.2 cudaD3D10RegisterFlags . . . . .	66
3.11.3 Function Documentation . . . . .	66
3.11.3.1 cudaD3D10GetDevice . . . . .	66
3.11.3.2 cudaD3D10MapResources . . . . .	67
3.11.3.3 cudaD3D10RegisterResource . . . . .	67
3.11.3.4 cudaD3D10ResourceGetMappedArray . . . . .	69
3.11.3.5 cudaD3D10ResourceGetMappedPitch . . . . .	69
3.11.3.6 cudaD3D10ResourceGetMappedPointer . . . . .	70
3.11.3.7 cudaD3D10ResourceGetMappedSize . . . . .	71
3.11.3.8 cudaD3D10ResourceGetSurfaceDimensions . . . . .	71
3.11.3.9 cudaD3D10ResourceSetMapFlags . . . . .	72
3.11.3.10 cudaD3D10SetDirect3DDevice . . . . .	73
3.11.3.11 cudaD3D10UnmapResources . . . . .	73
3.11.3.12 cudaD3D10UnregisterResource . . . . .	74

3.12 Texture Reference Management	75
3.12.1 Detailed Description	75
3.12.2 Function Documentation	75
3.12.2.1 cudaBindTexture	75
3.12.2.2 cudaBindTexture2D	76
3.12.2.3 cudaBindTextureToArray	77
3.12.2.4 cudaCreateChannelDesc	77
3.12.2.5 cudaGetChannelDesc	78
3.12.2.6 cudaGetTextureAlignmentOffset	78
3.12.2.7 cudaGetTextureReference	79
3.12.2.8 cudaUnbindTexture	79
3.13 Version Management	80
3.13.1 Function Documentation	80
3.13.1.1 cudaDriverGetVersion	80
3.13.1.2 cudaRuntimeGetVersion	80
3.14 C++ API Routines	81
3.14.1 Detailed Description	82
3.14.2 Function Documentation	82
3.14.2.1 cudaBindTexture	82
3.14.2.2 cudaBindTexture	82
3.14.2.3 cudaBindTexture2D	83
3.14.2.4 cudaBindTextureToArray	84
3.14.2.5 cudaBindTextureToArray	84
3.14.2.6 cudaCreateChannelDesc	85
3.14.2.7 cudaGetSymbolAddress	85
3.14.2.8 cudaGetSymbolSize	86
3.14.2.9 cudaGetTextureAlignmentOffset	86
3.14.2.10 cudaLaunch	86
3.14.2.11 cudaSetupArgument	87
3.14.2.12 cudaUnbindTexture	87
3.15 Data types used by CUDA Runtime	89
3.15.1 Typedef Documentation	91
3.15.1.1 cudaError_t	91
3.15.1.2 cudaEvent_t	91
3.15.1.3 cudaStream_t	91
3.15.2 Enumeration Type Documentation	91
3.15.2.1 cudaChannelFormatKind	91

---

3.15.2.2	<code>cudaComputeMode</code>	92
3.15.2.3	<code>cudaError</code>	92
3.15.2.4	<code>cudaMemcpyKind</code>	93
3.16	CUDA Driver API	94
3.16.1	Detailed Description	94
3.17	Initialization	95
3.17.1	Detailed Description	95
3.17.2	Function Documentation	95
3.17.2.1	<code>cuInit</code>	95
3.18	Device Management	96
3.18.1	Detailed Description	96
3.18.2	Function Documentation	96
3.18.2.1	<code>cuDeviceComputeCapability</code>	96
3.18.2.2	<code>cuDeviceGet</code>	97
3.18.2.3	<code>cuDeviceGetAttribute</code>	97
3.18.2.4	<code>cuDeviceGetCount</code>	99
3.18.2.5	<code>cuDeviceGetName</code>	99
3.18.2.6	<code>cuDeviceGetProperties</code>	99
3.18.2.7	<code>cuDeviceTotalMem</code>	101
3.19	Version Management	102
3.19.1	Detailed Description	102
3.19.2	Function Documentation	102
3.19.2.1	<code>cuDriverGetVersion</code>	102
3.20	Context Management	103
3.20.1	Detailed Description	103
3.20.2	Function Documentation	103
3.20.2.1	<code>cuCtxAttach</code>	103
3.20.2.2	<code>cuCtxCreate</code>	104
3.20.2.3	<code>cuCtxDestroy</code>	105
3.20.2.4	<code>cuCtxDetach</code>	105
3.20.2.5	<code>cuCtxGetDevice</code>	106
3.20.2.6	<code>cuCtxPopCurrent</code>	106
3.20.2.7	<code>cuCtxPushCurrent</code>	107
3.20.2.8	<code>cuCtxSynchronize</code>	107
3.21	Module Management	108
3.21.1	Detailed Description	108
3.21.2	Function Documentation	108



3.21.2.1	<a href="#">cuModuleGetFunction</a>	108
3.21.2.2	<a href="#">cuModuleGetGlobal</a>	109
3.21.2.3	<a href="#">cuModuleGetTexRef</a>	109
3.21.2.4	<a href="#">cuModuleLoad</a>	110
3.21.2.5	<a href="#">cuModuleLoadData</a>	110
3.21.2.6	<a href="#">cuModuleLoadDataEx</a>	111
3.21.2.7	<a href="#">cuModuleLoadFatBinary</a>	112
3.21.2.8	<a href="#">cuModuleUnload</a>	112
3.22	<a href="#">Stream Management</a>	114
3.22.1	<a href="#">Detailed Description</a>	114
3.22.2	<a href="#">Function Documentation</a>	114
3.22.2.1	<a href="#">cuStreamCreate</a>	114
3.22.2.2	<a href="#">cuStreamDestroy</a>	114
3.22.2.3	<a href="#">cuStreamQuery</a>	115
3.22.2.4	<a href="#">cuStreamSynchronize</a>	115
3.23	<a href="#">Event Management</a>	116
3.23.1	<a href="#">Detailed Description</a>	116
3.23.2	<a href="#">Function Documentation</a>	116
3.23.2.1	<a href="#">cuEventCreate</a>	116
3.23.2.2	<a href="#">cuEventDestroy</a>	117
3.23.2.3	<a href="#">cuEventElapsedTime</a>	117
3.23.2.4	<a href="#">cuEventQuery</a>	118
3.23.2.5	<a href="#">cuEventRecord</a>	118
3.23.2.6	<a href="#">cuEventSynchronize</a>	118
3.24	<a href="#">Execution Control</a>	120
3.24.1	<a href="#">Detailed Description</a>	120
3.24.2	<a href="#">Function Documentation</a>	120
3.24.2.1	<a href="#">cuFuncGetAttribute</a>	120
3.24.2.2	<a href="#">cuFuncSetBlockShape</a>	121
3.24.2.3	<a href="#">cuFuncSetSharedSize</a>	122
3.24.2.4	<a href="#">cuLaunch</a>	122
3.24.2.5	<a href="#">cuLaunchGrid</a>	122
3.24.2.6	<a href="#">cuLaunchGridAsync</a>	123
3.24.2.7	<a href="#">cuParamSetf</a>	123
3.24.2.8	<a href="#">cuParamSeti</a>	124
3.24.2.9	<a href="#">cuParamSetSize</a>	124
3.24.2.10	<a href="#">cuParamSetTexRef</a>	125

---

3.24.2.11	cuParamSetv	125
3.25	Memory Management	127
3.25.1	Detailed Description	129
3.25.2	Function Documentation	129
3.25.2.1	cuArray3DCreate	129
3.25.2.2	cuArray3DGetDescriptor	131
3.25.2.3	cuArrayCreate	131
3.25.2.4	cuArrayDestroy	133
3.25.2.5	cuArrayGetDescriptor	133
3.25.2.6	cuMemAlloc	134
3.25.2.7	cuMemAllocHost	135
3.25.2.8	cuMemAllocPitch	135
3.25.2.9	cuMemcpy2D	136
3.25.2.10	cuMemcpy2DAsync	138
3.25.2.11	cuMemcpy2DUnaligned	140
3.25.2.12	cuMemcpy3D	143
3.25.2.13	cuMemcpy3DAsync	145
3.25.2.14	cuMemcpyAtoA	147
3.25.2.15	cuMemcpyAtoD	148
3.25.2.16	cuMemcpyAtoH	148
3.25.2.17	cuMemcpyAtoHAsync	149
3.25.2.18	cuMemcpyDtoA	150
3.25.2.19	cuMemcpyDtoD	150
3.25.2.20	cuMemcpyDtoH	151
3.25.2.21	cuMemcpyDtoHAsync	151
3.25.2.22	cuMemcpyHtoA	152
3.25.2.23	cuMemcpyHtoAAsync	153
3.25.2.24	cuMemcpyHtoD	153
3.25.2.25	cuMemcpyHtoDAsync	154
3.25.2.26	cuMemFree	154
3.25.2.27	cuMemFreeHost	155
3.25.2.28	cuMemGetAddressRange	155
3.25.2.29	cuMemGetInfo	156
3.25.2.30	cuMemHostAlloc	157
3.25.2.31	cuMemHostGetDevicePointer	158
3.25.2.32	cuMemsetD16	158
3.25.2.33	cuMemsetD2D16	159

3.25.2.34	cuMemsetD2D32	159
3.25.2.35	cuMemsetD2D8	160
3.25.2.36	cuMemsetD32	161
3.25.2.37	cuMemsetD8	161
3.26	Texture Reference Management	162
3.26.1	Detailed Description	163
3.26.2	Function Documentation	163
3.26.2.1	cuTexRefCreate	163
3.26.2.2	cuTexRefDestroy	163
3.26.2.3	cuTexRefGetAddress	163
3.26.2.4	cuTexRefGetAddressMode	164
3.26.2.5	cuTexRefGetArray	164
3.26.2.6	cuTexRefGetFilterMode	165
3.26.2.7	cuTexRefGetFlags	165
3.26.2.8	cuTexRefGetFormat	165
3.26.2.9	cuTexRefSetAddress	166
3.26.2.10	cuTexRefSetAddress2D	166
3.26.2.11	cuTexRefSetAddressMode	167
3.26.2.12	cuTexRefSetArray	168
3.26.2.13	cuTexRefSetFilterMode	168
3.26.2.14	cuTexRefSetFlags	169
3.26.2.15	cuTexRefSetFormat	169
3.27	OpenGL Interoperability	170
3.27.1	Detailed Description	170
3.27.2	Function Documentation	170
3.27.2.1	cuGLCtxCreate	170
3.27.2.2	cuGLInit	171
3.27.2.3	cuGLMapBufferObject	171
3.27.2.4	cuGLRegisterBufferObject	172
3.27.2.5	cuGLUnmapBufferObject	172
3.27.2.6	cuGLUnregisterBufferObject	172
3.27.2.7	cuWGLGetDevice	173
3.28	Direct3D 9 Interoperability	174
3.28.1	Detailed Description	175
3.28.2	Function Documentation	175
3.28.2.1	cuD3D9CtxCreate	175
3.28.2.2	cuD3D9GetDevice	175

3.28.2.3	cuD3D9GetDirect3DDevice	176
3.28.2.4	cuD3D9MapResources	176
3.28.2.5	cuD3D9RegisterResource	177
3.28.2.6	cuD3D9ResourceGetMappedArray	178
3.28.2.7	cuD3D9ResourceGetMappedPitch	179
3.28.2.8	cuD3D9ResourceGetMappedPointer	180
3.28.2.9	cuD3D9ResourceGetMappedSize	181
3.28.2.10	cuD3D9ResourceGetSurfaceDimensions	181
3.28.2.11	cuD3D9ResourceSetMapFlags	182
3.28.2.12	cuD3D9UnmapResources	183
3.28.2.13	cuD3D9UnregisterResource	183
3.29	Direct3D 10 Interoperability	185
3.29.1	Detailed Description	185
3.29.2	Function Documentation	186
3.29.2.1	cuD3D10CtxCreate	186
3.29.2.2	cuD3D10GetDevice	186
3.29.2.3	cuD3D10MapResources	187
3.29.2.4	cuD3D10RegisterResource	187
3.29.2.5	cuD3D10ResourceGetMappedArray	189
3.29.2.6	cuD3D10ResourceGetMappedPitch	189
3.29.2.7	cuD3D10ResourceGetMappedPointer	190
3.29.2.8	cuD3D10ResourceGetMappedSize	191
3.29.2.9	cuD3D10ResourceGetSurfaceDimensions	191
3.29.2.10	cuD3D10ResourceSetMapFlags	192
3.29.2.11	cuD3D10UnmapResources	193
3.29.2.12	cuD3D10UnregisterResource	193
3.30	Data types used by CUDA driver	195
3.30.1	Define Documentation	199
3.30.1.1	CU_MEMHOSTALLOC_DEVICEMAP	199
3.30.1.2	CU_MEMHOSTALLOC_PORTABLE	199
3.30.1.3	CU_MEMHOSTALLOC_WRITECOMBINED	199
3.30.1.4	CU_PARAM_TR_DEFAULT	199
3.30.1.5	CU_TRSA_OVERRIDE_FORMAT	199
3.30.1.6	CU_TRSF_NORMALIZED_COORDINATES	199
3.30.1.7	CU_TRSF_READ_AS_INTEGER	199
3.30.1.8	CUDA_VERSION	199
3.30.2	Typedef Documentation	199

---

3.30.2.1	CUaddress_mode	199
3.30.2.2	CUarray_format	199
3.30.2.3	CUcomputemode	200
3.30.2.4	CUctx_flags	200
3.30.2.5	CUDA_MEMCPY2D	200
3.30.2.6	CUDA_MEMCPY3D	200
3.30.2.7	CUdevice_attribute	200
3.30.2.8	CUdevprop	200
3.30.2.9	CUevent_flags	200
3.30.2.10	CUfilter_mode	200
3.30.2.11	CUfunction_attribute	200
3.30.2.12	CUjit_fallback	200
3.30.2.13	CUjit_option	200
3.30.2.14	CUjit_target	200
3.30.2.15	CUmemorytype	201
3.30.2.16	CUresult	201
3.30.3	Enumeration Type Documentation	201
3.30.3.1	CUaddress_mode_enum	201
3.30.3.2	CUarray_format_enum	201
3.30.3.3	CUcomputemode_enum	201
3.30.3.4	CUctx_flags_enum	202
3.30.3.5	cudaError_enum	202
3.30.3.6	CUdevice_attribute_enum	203
3.30.3.7	CUevent_flags_enum	203
3.30.3.8	CUfilter_mode_enum	204
3.30.3.9	CUfunction_attribute_enum	204
3.30.3.10	CUjit_fallback_enum	204
3.30.3.11	CUjit_option_enum	204
3.30.3.12	CUjit_target_enum	205
3.30.3.13	CUmemorytype_enum	205
<b>4</b>	<b>Data Structure Documentation</b>	<b>207</b>
4.1	CUDA_ARRAY3D_DESCRIPTOR Struct Reference	207
4.1.1	Detailed Description	207
4.2	CUDA_ARRAY_DESCRIPTOR Struct Reference	208
4.2.1	Detailed Description	208
4.3	CUDA_MEMCPY2D_st Struct Reference	209

---

4.3.1 Detailed Description . . . . .	210
4.4 CUDA_MEMCPY3D_st Struct Reference . . . . .	211
4.4.1 Detailed Description . . . . .	212
4.5 cudaChannelFormatDesc Struct Reference . . . . .	213
4.5.1 Detailed Description . . . . .	213
4.6 cudaDeviceProp Struct Reference . . . . .	214
4.6.1 Detailed Description . . . . .	215
4.7 cudaExtent Struct Reference . . . . .	216
4.7.1 Detailed Description . . . . .	216
4.8 cudaFuncAttributes Struct Reference . . . . .	217
4.8.1 Detailed Description . . . . .	217
4.9 cudaMemcpy3DParms Struct Reference . . . . .	218
4.9.1 Detailed Description . . . . .	218
4.10 cudaPitchedPtr Struct Reference . . . . .	219
4.10.1 Detailed Description . . . . .	219
4.11 cudaPos Struct Reference . . . . .	220
4.11.1 Detailed Description . . . . .	220
4.12 CUdevprop_st Struct Reference . . . . .	221
4.12.1 Detailed Description . . . . .	221

# Chapter 1

## Module Index

### 1.1 Modules

Here is a list of all modules:

CUDA Runtime API . . . . .	5
Thread Management . . . . .	6
Error Handling . . . . .	7
Device Management . . . . .	9
Stream Management . . . . .	14
Event Management . . . . .	16
Execution Control . . . . .	20
Memory Management . . . . .	23
OpenGL Interoperability . . . . .	51
Direct3D 9 Interoperability . . . . .	54
Direct3D 10 Interoperability . . . . .	65
Texture Reference Management . . . . .	75
Version Management . . . . .	80
C++ API Routines . . . . .	81
Data types used by CUDA Runtime . . . . .	89
CUDA Driver API . . . . .	94
Initialization . . . . .	95
Device Management . . . . .	96
Version Management . . . . .	102
Context Management . . . . .	103
Module Management . . . . .	108
Stream Management . . . . .	114
Event Management . . . . .	116
Execution Control . . . . .	120
Memory Management . . . . .	127
Texture Reference Management . . . . .	162
OpenGL Interoperability . . . . .	170
Direct3D 9 Interoperability . . . . .	174
Direct3D 10 Interoperability . . . . .	185
Data types used by CUDA driver . . . . .	195





# Chapter 2

## Data Structure Index

### 2.1 Data Structures

Here are the data structures with brief descriptions:

<a href="#">CUDA_ARRAY3D_DESCRIPTOR</a>	207
<a href="#">CUDA_ARRAY_DESCRIPTOR</a>	208
<a href="#">CUDA_MEMCPY2D_st</a>	209
<a href="#">CUDA_MEMCPY3D_st</a>	211
<a href="#">cudaChannelFormatDesc</a>	213
<a href="#">cudaDeviceProp</a>	214
<a href="#">cudaExtent</a>	216
<a href="#">cudaFuncAttributes</a>	217
<a href="#">cudaMemcpy3DParms</a>	218
<a href="#">cudaPitchedPtr</a>	219
<a href="#">cudaPos</a>	220
<a href="#">CUdevprop_st</a>	221



# Chapter 3

## Module Documentation

### 3.1 CUDA Runtime API

#### Modules

- [Thread Management](#)
- [Error Handling](#)
- [Device Management](#)
- [Stream Management](#)
- [Event Management](#)
- [Execution Control](#)
- [Memory Management](#)
- [OpenGL Interoperability](#)
- [Direct3D 9 Interoperability](#)
- [Direct3D 10 Interoperability](#)
- [Texture Reference Management](#)
- [Version Management](#)
- [C++ API Routines](#)

*C++-style interface built on top of CUDA runtime API.*

- [Data types used by CUDA Runtime](#)

#### 3.1.1 Detailed Description

There are two levels for the runtime API.

The C API (*cuda\_runtime\_api.h*) is a C-style interface that does not require compiling with `nvcc`.

The C++ API (*cuda\_runtime.h*) is a C++-style interface built on top of the C API. It wraps some of the C API routines, using overloading, references and default arguments. These wrappers can be used from C++ code and can be compiled with any C++ compiler. The C++ API also has some CUDA-specific wrappers that wrap C API routines that deal with symbols, textures, and device functions. These wrappers require the use of `nvcc` because they depend on code being generated by the compiler. For example, the execution configuration syntax to invoke kernels is only available in source code compiled with `nvcc`.

## 3.2 Thread Management

### Functions

- [cudaError\\_t cudaThreadExit](#) (void)  
*Exit and clean up from CUDA launches.*
- [cudaError\\_t cudaThreadSynchronize](#) (void)  
*Wait for compute device to finish.*

### 3.2.1 Detailed Description

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

### 3.2.2 Function Documentation

#### 3.2.2.1 [cudaError\\_t cudaThreadExit](#) (void)

Explicitly cleans up all runtime-related resources associated with the calling host thread. Any subsequent API call reinitializes the runtime. [cudaThreadExit\(\)](#) is implicitly called on host thread exit.

**Returns:**

[cudaSuccess](#)

**Note:**

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

**See also:**

[cudaThreadSynchronize](#)

#### 3.2.2.2 [cudaError\\_t cudaThreadSynchronize](#) (void)

Blocks until the device has completed all preceding requested tasks. [cudaThreadSynchronize\(\)](#) returns an error if one of the preceding tasks has failed.

**Returns:**

[cudaSuccess](#)

**Note:**

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

**See also:**

[cudaThreadExit](#)

## 3.3 Error Handling

### Functions

- `const char * cudaGetErrorString (cudaError_t error)`  
*Returns the message string from an error code.*
- `cudaError_t cudaGetLastError (void)`  
*Returns the last error from a runtime call.*

### 3.3.1 Detailed Description

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

### 3.3.2 Function Documentation

#### 3.3.2.1 `const char* cudaGetErrorString (cudaError_t error)`

Returns the message string from an error code.

#### Parameters:

*error* - Error code to convert to string

#### Returns:

`char*` pointer to a NULL-terminated string

#### See also:

[cudaGetLastError](#), [cudaError](#)

#### 3.3.2.2 `cudaError_t cudaGetLastError (void)`

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

#### Returns:

[cudaSuccess](#), [cudaErrorMissingConfiguration](#), [cudaErrorMemoryAllocation](#), [cudaErrorInitializationError](#), [cudaErrorLaunchFailure](#), [cudaErrorPriorLaunchFailure](#), [cudaErrorLaunchTimeout](#), [cudaErrorLaunchOutOfResources](#), [cudaErrorInvalidDeviceFunction](#), [cudaErrorInvalidConfiguration](#), [cudaErrorInvalidDevice](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidPitchValue](#), [cudaErrorInvalidSymbol](#), [cudaErrorMapBufferObjectFailed](#), [cudaErrorUnmapBufferObjectFailed](#), [cudaErrorInvalidHostPointer](#), [cudaErrorInvalidDevicePointer](#), [cudaErrorInvalidTexture](#), [cudaErrorInvalidTextureBinding](#), [cudaErrorInvalidChannelDescriptor](#), [cudaErrorInvalidMemcpyDirection](#), [cudaErrorAddressOfConstant](#), [cudaErrorTextureFetchFailed](#), [cudaErrorTextureNotBound](#), [cudaErrorSynchronizationError](#), [cudaErrorInvalidFilterSetting](#), [cudaErrorInvalidNormSetting](#), [cudaErrorMixedDeviceExecution](#), [cudaErrorCudartUnloading](#), [cudaErrorUnknown](#), [cudaErrorNotYetImplemented](#), [cudaErrorMemoryValueTooLarge](#), [cudaErrorInvalidResourceHandle](#), [cudaErrorNotReady](#), [cudaErrorInsufficientDriver](#), [cudaErrorSetOnActiveProcess](#), [cudaErrorStartupFailure](#), [cudaErrorApiFailureBase](#)

**Note:**

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

**See also:**

[cudaGetErrorString](#), [cudaError](#)

## 3.4 Device Management

### Functions

- [cudaError\\_t cudaChooseDevice](#) (int \*device, const struct [cudaDeviceProp](#) \*prop)  
*Select compute-device which best matches criteria.*
- [cudaError\\_t cudaGetDevice](#) (int \*device)  
*Returns which device is currently being used.*
- [cudaError\\_t cudaGetDeviceCount](#) (int \*count)  
*Returns the number of compute-capable devices.*
- [cudaError\\_t cudaGetDeviceProperties](#) (struct [cudaDeviceProp](#) \*prop, int device)  
*Returns information about the compute-device.*
- [cudaError\\_t cudaSetDevice](#) (int device)  
*Set device to be used for GPU executions.*
- [cudaError\\_t cudaSetDeviceFlags](#) (int flags)  
*Sets flags to be used for device executions.*
- [cudaError\\_t cudaSetValidDevices](#) (int \*device\_arr, int len)  
*Set a list of devices that can be used for CUDA.*

### 3.4.1 Detailed Description

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

### 3.4.2 Function Documentation

#### 3.4.2.1 [cudaError\\_t cudaChooseDevice](#) (int \* device, const struct [cudaDeviceProp](#) \* prop)

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

#### Parameters:

- device* - Device with best match
- prop* - Desired device properties

#### Returns:

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

#### Note:

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

#### See also:

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

### 3.4.2.2 `cudaError_t cudaGetDevice (int * device)`

Returns in `*device` the device on which the active host thread executes the device code.

**Parameters:**

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

**Returns:**

[cudaSuccess](#)

**Note:**

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

**See also:**

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

### 3.4.2.3 `cudaError_t cudaGetDeviceCount (int * count)`

Returns in `*count` the number of devices with compute capability greater or equal to 1.0 that are available for execution. If there is no such device, `cudaGetDeviceCount ()` returns 1 and device 0 only supports device emulation mode. Since this device will be able to emulate all hardware features, this device will report major and minor compute capability versions of 9999.

**Parameters:**

*count* - Returns the number of devices with compute capability greater or equal to 1.0

**Returns:**

[cudaSuccess](#)

**Note:**

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

**See also:**

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

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

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];
};
```



```
    size_t totalConstMem;
    int major;
    int minor;
    int clockRate;
    size_t textureAlignment;
    int deviceOverlap;
    int multiProcessorCount;
    int kernelExecTimeoutEnabled;
    int integrated;
    int canMapHostMemory;
    int computeMode;
}
```

where:

- `name` 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; this amount is shared by all thread blocks simultaneously resident on a multiprocessor;
- `regsPerBlock` is the maximum number of 32-bit registers available to a thread block; this number is shared by all thread blocks simultaneously resident on a multiprocessor;
- `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;
- `deviceOverlap` is 1 if the device can concurrently copy memory between host and device while executing a kernel, or 0 if not;
- `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. Any errors from calling `cudaSetDevice()` with an exclusive (and occupied) or prohibited device will only show up after a non-device management runtime function is called. At that time, `cudaErrorNoDevice` will be returned.

**Parameters:**

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

**Returns:**

`cudaSuccess`, `cudaErrorInvalidDevice`

**See also:**

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

### 3.4.2.5 `cudaError_t cudaSetDevice (int device)`

Records *device* as the device on which the active host thread executes the device code. If the host thread has already initialized the CUDA runtime by calling non-device management runtime functions, this call returns `cudaErrorSetActiveProcess`.

**Parameters:**

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

**Returns:**

`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorSetActiveProcess`

**Note:**

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

**See also:**

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

### 3.4.2.6 `cudaError_t cudaSetDeviceFlags (int flags)`

Records *flags* as the flags to use when the active host thread executes device code. If the host thread has already initialized the CUDA runtime by calling non-device management runtime functions, this call returns `cudaErrorSetActiveProcess`.

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.
- [cudaDeviceBlockingSync](#): Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the device to finish work.
- [cudaDeviceMapHost](#): This flag must be set in order to allocate pinned host memory that is accessible to the device. If this flag is not set, [cudaHostGetDevicePointer\(\)](#) will always return a failure code.

**Parameters:**

*flags* - Parameters for device operation

**Returns:**

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

**See also:**

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

**3.4.2.7 `cudaError_t cudaSetValidDevices (int * device_arr, int len)`**

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` is greater than the number of devices in the system, then [cudaErrorInvalidValue](#) is returned.

**Parameters:**

*device\_arr* - List of devices to try

*len* - Number of devices in specified list

**Returns:**

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

**Note:**

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

**See also:**

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

## 3.5 Stream Management

### Functions

- [cudaError\\_t cudaStreamCreate \(cudaStream\\_t \\*pStream\)](#)  
*Create an asynchronous stream.*
- [cudaError\\_t cudaStreamDestroy \(cudaStream\\_t stream\)](#)  
*Destroys and cleans up an asynchronous stream.*
- [cudaError\\_t cudaStreamQuery \(cudaStream\\_t stream\)](#)  
*Queries an asynchronous stream for completion status.*
- [cudaError\\_t cudaStreamSynchronize \(cudaStream\\_t stream\)](#)  
*Waits for stream tasks to complete.*

### 3.5.1 Detailed Description

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

### 3.5.2 Function Documentation

#### 3.5.2.1 [cudaError\\_t cudaStreamCreate \(cudaStream\\_t \\* pStream\)](#)

Creates a new asynchronous stream.

**Parameters:**

*pStream* - Pointer to new stream identifier

**Returns:**

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

**Note:**

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

**See also:**

[cudaStreamQuery](#), [cudaStreamSynchronize](#), [cudaStreamDestroy](#)

#### 3.5.2.2 [cudaError\\_t cudaStreamDestroy \(cudaStream\\_t stream\)](#)

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

**Parameters:**

*stream* - Stream identifier

**Returns:**

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

**Note:**

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

**See also:**

[cudaStreamCreate](#), [cudaStreamQuery](#), [cudaStreamSynchronize](#)

**3.5.2.3 `cudaError_t cudaStreamQuery (cudaStream_t stream)`**

Returns [cudaSuccess](#) if all operations in `stream` have completed, or [cudaErrorNotReady](#) if not.

**Parameters:**

*stream* - Stream identifier

**Returns:**

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

**Note:**

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

**See also:**

[cudaStreamCreate](#), [cudaStreamSynchronize](#), [cudaStreamDestroy](#)

**3.5.2.4 `cudaError_t cudaStreamSynchronize (cudaStream_t stream)`**

Blocks until `stream` has completed all operations.

**Parameters:**

*stream* - Stream identifier

**Returns:**

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

**Note:**

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

**See also:**

[cudaStreamCreate](#), [cudaStreamQuery](#), [cudaStreamDestroy](#)

## 3.6 Event Management

### Functions

- [cudaError\\_t cudaEventCreate \(cudaEvent\\_t \\*event\)](#)  
*Creates an event object.*
- [cudaError\\_t cudaEventCreateWithFlags \(cudaEvent\\_t \\*event, int flags\)](#)  
*Creates an event object with the specified flags.*
- [cudaError\\_t cudaEventDestroy \(cudaEvent\\_t event\)](#)  
*Destroys an event object.*
- [cudaError\\_t cudaEventElapsedTime \(float \\*ms, cudaEvent\\_t start, cudaEvent\\_t end\)](#)  
*Computes the elapsed time between events.*
- [cudaError\\_t cudaEventQuery \(cudaEvent\\_t event\)](#)  
*Query if an event has been recorded.*
- [cudaError\\_t cudaEventRecord \(cudaEvent\\_t event, cudaStream\\_t stream\)](#)  
*Records an event.*
- [cudaError\\_t cudaEventSynchronize \(cudaEvent\\_t event\)](#)  
*Wait for an event to be recorded.*

### 3.6.1 Detailed Description

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

### 3.6.2 Function Documentation

#### 3.6.2.1 [cudaError\\_t cudaEventCreate \(cudaEvent\\_t \\* event\)](#)

Creates an event object using [cudaEventDefault](#).

#### Parameters:

*event* - Newly created event

#### Returns:

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

#### Note:

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

#### See also:

[cudaEventCreateWithFlags](#), [cudaEventRecord](#), [cudaEventQuery](#), [cudaEventSynchronize](#), [cudaEventDestroy](#), [cudaEventElapsedTime](#)

### 3.6.2.2 `cudaError_t cudaEventCreateWithFlags (cudaEvent_t * event, int flags)`

Creates an event object with the specified flags.

Valid flags include: [cudaEventDefault](#), [cudaEventBlockingSync](#)

**Parameters:**

*event* - Newly created event

*flags* - Flags for new event

**Returns:**

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

**Note:**

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

**See also:**

[cudaEventCreate](#), [cudaEventRecord](#), [cudaEventQuery](#), [cudaEventSynchronize](#), [cudaEventDestroy](#), [cudaEventElapsedTime](#)

### 3.6.2.3 `cudaError_t cudaEventDestroy (cudaEvent_t event)`

Destroys the specified event object.

**Parameters:**

*event* - Event to destroy

**Returns:**

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

**Note:**

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

**See also:**

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

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

Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds). If either event has not been recorded yet, this function returns [cudaErrorInvalidValue](#). If either event has been recorded with a non-zero stream, the result is undefined.

**Parameters:**

*ms* - Time between `start` and `stop` in ms

*start* - Starting event

*end* - Stopping event

**Returns:**

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

**Note:**

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

**See also:**

[cudaEventCreate](#), [cudaEventCreateWithFlags](#), [cudaEventQuery](#), [cudaEventSynchronize](#), [cudaEventDestroy](#), [cudaEventRecord](#)

### 3.6.2.5 `cudaError_t cudaEventQuery (cudaEvent_t event)`

Returns [cudaSuccess](#) if the event has actually been recorded, or [cudaErrorNotReady](#) if not. If [cudaEventRecord\(\)](#) has not been called on this event, the function returns [cudaErrorInvalidValue](#).

**Parameters:**

*event* - Event to query

**Returns:**

[cudaSuccess](#), [cudaErrorNotReady](#), [cudaErrorInitializationError](#), [cudaErrorPriorLaunchFailure](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidResourceHandle](#)

**Note:**

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

**See also:**

[cudaEventCreate](#), [cudaEventCreateWithFlags](#), [cudaEventRecord](#), [cudaEventSynchronize](#), [cudaEventDestroy](#), [cudaEventElapsedTime](#)

### 3.6.2.6 `cudaError_t cudaEventRecord (cudaEvent_t event, cudaStream_t stream)`

Records an event. If `stream` is non-zero, the event is recorded after all preceding operations in the stream have been completed; otherwise, it is recorded after all preceding operations in the CUDA context have been completed. Since this operation is asynchronous, [cudaEventQuery\(\)](#) and/or [cudaEventSynchronize\(\)](#) must be used to determine when the event has actually been recorded.

If [cudaEventRecord\(\)](#) has previously been called and the event has not been recorded yet, this function returns [cudaErrorInvalidValue](#).

**Parameters:**

*event* - Event to record

*stream* - Stream in which to record event



**Returns:**

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

**Note:**

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

**See also:**

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

**3.6.2.7 `cudaError_t cudaEventSynchronize (cudaEvent_t event)`**

Blocks until the event has actually been recorded. If [cudaEventRecord\(\)](#) has not been called on this event, the function returns [cudaErrorInvalidValue](#).

**Parameters:**

*event* - Event to wait for

**Returns:**

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

**Note:**

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

**See also:**

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

## 3.7 Execution Control

### Functions

- [cudaError\\_t cudaConfigureCall](#) (dim3 gridDim, dim3 blockDim, size\_t sharedMem, [cudaStream\\_t](#) stream)  
*Configure a device-launch.*
- [cudaError\\_t cudaFuncGetAttributes](#) (struct [cudaFuncAttributes](#) \*attr, const char \*func)  
*Find out attributes for a given function.*
- [cudaError\\_t cudaLaunch](#) (const char \*entry)  
*Launches a device function.*
- [cudaError\\_t cudaSetDoubleForDevice](#) (double \*d)  
*Converts a double argument to be executed on a device.*
- [cudaError\\_t cudaSetDoubleForHost](#) (double \*d)  
*Converts a double argument after execution on a device.*
- [cudaError\\_t cudaSetupArgument](#) (const void \*arg, size\_t size, size\_t offset)  
*Configure a device launch.*

### 3.7.1 Detailed Description

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

### 3.7.2 Function Documentation

#### 3.7.2.1 [cudaError\\_t cudaConfigureCall](#) (dim3 *gridDim*, dim3 *blockDim*, size\_t *sharedMem*, [cudaStream\\_t](#) *stream*)

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.

#### Parameters:

- gridDim* - Grid dimensions
- blockDim* - Block dimensions
- sharedMem* - Shared memory
- stream* - Stream identifier

#### Returns:

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

#### Note:

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

See also:

[cudaLaunch \(C API\)](#), [cudaLaunch \(C++ API\)](#), [cudaSetupArgument \(C API\)](#)

### 3.7.2.2 `cudaError_t cudaFuncGetAttributes (struct cudaFuncAttributes * attr, const char * func)`

This function obtains the attributes of a function specified via `func`. The fetched attributes are placed in `attr`. If the specified function does not exist, then `:cudaErrorInvalidDeviceFunction` is returned.

**Parameters:**

*attr* - Return pointer to function's attributes

*func* - Function to get attributes of

**Returns:**

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

**Note:**

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

See also:

[cudaLaunch \(C API\)](#), [cudaLaunch \(C++ API\)](#)

### 3.7.2.3 `cudaError_t cudaLaunch (const char * entry)`

Launches the function `entry` on the device. `entry` can either be a function that executes on the device, or it can be a character string, naming a function that executes on the device. `entry` 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.

**Parameters:**

*entry* - Device function pointer or char string naming device function to execute

**Returns:**

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

**Note:**

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

See also:

[cudaLaunch \(C++ API\)](#), [cudaSetupArgument \(C API\)](#), [cudaConfigureCall](#)

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

**Parameters:**

*d* - Double to convert

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.

**Returns:**

[cudaSuccess](#)

**Note:**

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

**See also:**

[cudaLaunch \(C API\)](#), [cudaLaunch \(C++ API\)](#), [cudaSetupArgument \(C API\)](#), [cudaSetDoubleForHost](#)

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

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.

**Parameters:**

*d* - Double to convert

**Returns:**

[cudaSuccess](#)

**Note:**

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

**See also:**

[cudaLaunch \(C API\)](#), [cudaLaunch \(C++ API\)](#), [cudaSetupArgument \(C API\)](#), [cudaSetDoubleForDevice](#)

### 3.7.2.6 `cudaError_t cudaSetupArgument (const void * arg, size_t size, size_t offset)`

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()`.

**Parameters:**

*arg* - Argument to push for a kernel launch

*size* - Size of argument

*offset* - Offset in argument stack to push new arg

**Returns:**

[cudaSuccess](#)

**Note:**

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

**See also:**

[cudaLaunch \(C API\)](#), [cudaLaunch \(C++ API\)](#), [cudaSetupArgument \(C++ API\)](#), [cudaConfigureCall](#)

## 3.8 Memory Management

### Functions

- [cudaError\\_t cudaFree](#) (void \*devPtr)  
*Frees memory on the device.*
- [cudaError\\_t cudaFreeArray](#) (struct cudaArray \*array)  
*Frees an array on the device.*
- [cudaError\\_t cudaFreeHost](#) (void \*ptr)  
*Frees page-locked memory.*
- [cudaError\\_t cudaGetSymbolAddress](#) (void \*\*devPtr, const char \*symbol)  
*Finds the address associated with a CUDA symbol.*
- [cudaError\\_t cudaGetSymbolSize](#) (size\_t \*size, const char \*symbol)  
*Finds the size of the object associated with a CUDA symbol.*
- [cudaError\\_t cudaHostAlloc](#) (void \*\*ptr, size\_t size, unsigned int flags)  
*Allocates page-locked memory on the host.*
- [cudaError\\_t cudaHostGetDevicePointer](#) (void \*\*pDevice, void \*pHost, unsigned int flags)  
*Passes back device pointer of mapped host memory allocated by [cudaHostAlloc\(\)](#).*
- [cudaError\\_t cudaMalloc](#) (void \*\*devPtr, size\_t size)  
*Allocate memory on the device.*
- [cudaError\\_t cudaMalloc3D](#) (struct [cudaPitchedPtr](#) \*pitchedDevPtr, struct [cudaExtent](#) extent)  
*Allocates logical 1D, 2D, or 3D memory objects on the device.*
- [cudaError\\_t cudaMalloc3DArray](#) (struct cudaArray \*\*arrayPtr, const struct [cudaChannelFormatDesc](#) \*desc, struct [cudaExtent](#) extent)  
*Allocate an array on the device.*
- [cudaError\\_t cudaMallocArray](#) (struct cudaArray \*\*arrayPtr, const struct [cudaChannelFormatDesc](#) \*desc, size\_t width, size\_t height)  
*Allocate an array on the device.*
- [cudaError\\_t cudaMallocHost](#) (void \*\*ptr, size\_t size)  
*Allocates page-locked memory on the host.*
- [cudaError\\_t cudaMallocPitch](#) (void \*\*devPtr, size\_t \*pitch, size\_t width, size\_t height)  
*Allocates pitched memory on the device.*
- [cudaError\\_t cudaMemcpy](#) (void \*dst, const void \*src, size\_t count, enum [cudaMemcpyKind](#) kind)  
*Copies data between host and device.*
- [cudaError\\_t cudaMemcpy2D](#) (void \*dst, size\_t dpitch, const void \*src, size\_t spitch, size\_t width, size\_t height, enum [cudaMemcpyKind](#) kind)

*Copies data between host and device.*

- [cudaError\\_t cudaMemcpy2DArrayToArray](#) (struct cudaArray \*dst, size\_t wOffsetDst, size\_t hOffsetDst, const struct cudaArray \*src, size\_t wOffsetSrc, size\_t hOffsetSrc, size\_t width, size\_t height, enum [cudaMemcpyKind](#) kind)

*Copies data between host and device.*

- [cudaError\\_t cudaMemcpy2DAsync](#) (void \*dst, size\_t dpitch, const void \*src, size\_t spitch, size\_t width, size\_t height, enum [cudaMemcpyKind](#) kind, [cudaStream\\_t](#) stream)

*Copies data between host and device.*

- [cudaError\\_t cudaMemcpy2DFromArray](#) (void \*dst, size\_t dpitch, const struct cudaArray \*src, size\_t wOffset, size\_t hOffset, size\_t width, size\_t height, enum [cudaMemcpyKind](#) kind)

*Copies data between host and device.*

- [cudaError\\_t cudaMemcpy2DFromArrayAsync](#) (void \*dst, size\_t dpitch, const struct cudaArray \*src, size\_t wOffset, size\_t hOffset, size\_t width, size\_t height, enum [cudaMemcpyKind](#) kind, [cudaStream\\_t](#) stream)

*Copies data between host and device.*

- [cudaError\\_t cudaMemcpy2DToArray](#) (struct cudaArray \*dst, size\_t wOffset, size\_t hOffset, const void \*src, size\_t spitch, size\_t width, size\_t height, enum [cudaMemcpyKind](#) kind)

*Copies data between host and device.*

- [cudaError\\_t cudaMemcpy2DToArrayAsync](#) (struct cudaArray \*dst, size\_t wOffset, size\_t hOffset, const void \*src, size\_t spitch, size\_t width, size\_t height, enum [cudaMemcpyKind](#) kind, [cudaStream\\_t](#) stream)

*Copies data between host and device.*

- [cudaError\\_t cudaMemcpy3D](#) (const struct [cudaMemcpy3DParms](#) \*p)

*Copies data between 3D objects.*

- [cudaError\\_t cudaMemcpy3DAsync](#) (const struct [cudaMemcpy3DParms](#) \*p, [cudaStream\\_t](#) stream)

*Copies data between 3D objects.*

- [cudaError\\_t cudaMemcpyArrayToArray](#) (struct cudaArray \*dst, size\_t wOffsetDst, size\_t hOffsetDst, const struct cudaArray \*src, size\_t wOffsetSrc, size\_t hOffsetSrc, size\_t count, enum [cudaMemcpyKind](#) kind)

*Copies data between host and device.*

- [cudaError\\_t cudaMemcpyAsync](#) (void \*dst, const void \*src, size\_t count, enum [cudaMemcpyKind](#) kind, [cudaStream\\_t](#) stream)

*Copies data between host and device.*

- [cudaError\\_t cudaMemcpyFromArray](#) (void \*dst, const struct cudaArray \*src, size\_t wOffset, size\_t hOffset, size\_t count, enum [cudaMemcpyKind](#) kind)

*Copies data between host and device.*

- [cudaError\\_t cudaMemcpyFromArrayAsync](#) (void \*dst, const struct cudaArray \*src, size\_t wOffset, size\_t hOffset, size\_t count, enum [cudaMemcpyKind](#) kind, [cudaStream\\_t](#) stream)

*Copies data between host and device.*

- [cudaError\\_t cudaMemcpyFromSymbol](#) (void \*dst, const char \*symbol, size\_t count, size\_t offset, enum [cudaMemcpyKind](#) kind)

*Copies data from the given symbol on the device.*

- [cudaError\\_t cudaMemcpyFromSymbolAsync](#) (void \*dst, const char \*symbol, size\_t count, size\_t offset, enum cudaMemcpyKind kind, cudaStream\_t stream)

*Copies data from the given symbol on the device.*

- [cudaError\\_t cudaMemcpyToArray](#) (struct cudaArray \*dst, size\_t wOffset, size\_t hOffset, const void \*src, size\_t count, enum cudaMemcpyKind kind)

*Copies data between host and device.*

- [cudaError\\_t cudaMemcpyToArrayAsync](#) (struct cudaArray \*dst, size\_t wOffset, size\_t hOffset, const void \*src, size\_t count, enum cudaMemcpyKind kind, cudaStream\_t stream)

*Copies data between host and device.*

- [cudaError\\_t cudaMemcpyToSymbol](#) (const char \*symbol, const void \*src, size\_t count, size\_t offset, enum cudaMemcpyKind kind)

*Copies data to the given symbol on the device.*

- [cudaError\\_t cudaMemcpyToSymbolAsync](#) (const char \*symbol, const void \*src, size\_t count, size\_t offset, enum cudaMemcpyKind kind, cudaStream\_t stream)

*Copies data to the given symbol on the device.*

- [cudaError\\_t cudaMemset](#) (void \*devPtr, int value, size\_t count)

*Initializes or sets device memory to a value.*

- [cudaError\\_t cudaMemset2D](#) (void \*devPtr, size\_t pitch, int value, size\_t width, size\_t height)

*Initializes or sets device memory to a value.*

- [cudaError\\_t cudaMemset3D](#) (struct cudaPitchedPtr pitchedDevPtr, int value, struct cudaExtent extent)

*Initializes or sets device memory to a value.*

### 3.8.1 Detailed Description

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

### 3.8.2 Function Documentation

#### 3.8.2.1 [cudaError\\_t cudaFree](#) (void \* devPtr)

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.

#### Parameters:

*devPtr* - Device pointer to memory to free

#### Returns:

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

**Note:**

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

**See also:**

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

**3.8.2.2 `cudaError_t cudaFreeArray (struct cudaArray * array)`**

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.

**Parameters:**

*array* - Pointer to array to free

**Returns:**

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

**Note:**

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

**See also:**

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

**3.8.2.3 `cudaError_t cudaFreeHost (void * ptr)`**

Frees the memory space pointed to by `hostPtr`, which must have been returned by a previous call to [cudaMallocHost\(\)](#).

**Parameters:**

*ptr* - Pointer to memory to free

**Returns:**

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

**Note:**

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

**See also:**

[cudaMalloc](#), [cudaMallocPitch](#), [cudaFree](#), [cudaMallocArray](#), [cudaFreeArray](#), [cudaMallocHost](#), [cudaMalloc3D](#), [cudaMalloc3DArray](#), [cudaHostAlloc](#)



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

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

##### Parameters:

*devPtr* - Return device pointer associated with symbol

*symbol* - Global variable or string symbol to search for

##### Returns:

`cudaSuccess`, `cudaErrorInvalidSymbol`, `cudaErrorAddressOfConstant`

##### Note:

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

##### See also:

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

#### 3.8.2.5 `cudaError_t cudaGetSymbolSize (size_t * size, const char * symbol)`

Returns in `*size` the size of symbol `symbol`. `symbol` can either be a variable that resides in global or constant memory space, or it can be a character string, naming 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.

##### Parameters:

*size* - Size of object associated with symbol

*symbol* - Global variable or string symbol to find size of

##### Returns:

`cudaSuccess`, `cudaErrorInvalidSymbol`

##### Note:

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

##### See also:

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

#### 3.8.2.6 `cudaError_t cudaHostAlloc (void ** ptr, size_t size, unsigned int flags)`

Allocates `count` 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()`.

#### Parameters:

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

#### Returns:

`cudaSuccess`, `cudaErrorMemoryAllocation`

#### Note:

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

#### See also:

`cudaSetDeviceFlags`, `cudaMallocHost`, `cudaFreeHost`

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

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

`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.

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

**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](#)

**Note:**

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

**See also:**

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

**3.8.2.8 [cudaError\\_t cudaMalloc](#) (void \*\* *devPtr*, size\_t *size*)**

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.

**Parameters:**

*devPtr* - Pointer to allocated device memory

*size* - Requested allocation size in bytes

**Returns:**

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

**See also:**

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

**3.8.2.9 [cudaError\\_t cudaMalloc3D](#) (struct [cudaPitchedPtr](#) \* *pitchedDevPtr*, struct [cudaExtent](#) *extent*)**

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).

**Parameters:**

*pitchedDevPtr* - Pointer to allocated pitched device memory

*extent* - Requested allocation size

**Returns:**

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

**Note:**

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

**See also:**

[cudaMallocPitch](#), [cudaFree](#), [cudaMemcpy3D](#), [cudaMemset3D](#), [cudaMalloc3DArray](#), [cudaMallocArray](#), [cudaFreeArray](#), [cudaMallocHost](#), [cudaFreeHost](#), [cudaHostAlloc](#), [make\\_cudaPitchedPtr](#), [make\\_cudaExtent](#)

**3.8.2.10 `cudaError_t cudaMalloc3DArray (struct cudaArray ** arrayPtr, const struct cudaChannelFormatDesc * desc, struct cudaExtent extent)`**

Allocates a CUDA array according to the [cudaChannelFormatDesc](#) structure `desc` and returns a handle to the new CUDA array in `*arrayPtr`.

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()` is able to allocate 1D, 2D, or 3D arrays.

- A 1D array is allocated if the height and depth extent are both zero. For 1D arrays valid extent ranges are {(1, 8192), 0, 0}.
- A 2D array is allocated if only the depth extent is zero. For 2D arrays valid extent ranges are {(1, 65536), (1, 32768), 0}.
- A 3D array is allocated if all three extents are non-zero. For 3D arrays valid extent ranges are {(1, 2048), (1, 2048), (1, 2048)}.

**Note:**

Due to the differing extent limits, it may be advantageous to use a degenerate array (with unused dimensions set to one) of higher dimensionality. For instance, a degenerate 2D array allows for significantly more linear storage than a 1D array.

**Parameters:**

*arrayPtr* - Pointer to allocated array in device memory

*desc* - Requested channel format

*extent* - Requested allocation size

**Returns:**

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

**Note:**

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

**See also:**

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

### 3.8.2.11 `cudaError_t cudaMallocArray (struct cudaArray ** arrayPtr, const struct cudaChannelFormatDesc * desc, size_t width, size_t height)`

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](#).

**Parameters:**

*arrayPtr* - Pointer to allocated array in device memory  
*desc* - Requested channel format  
*width* - Requested array allocation width  
*height* - Requested array allocation height

**Returns:**

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

**Note:**

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

**See also:**

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

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

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.

**Parameters:**

*ptr* - Pointer to allocated host memory  
*size* - Requested allocation size in bytes

**Returns:**

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

**Note:**

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

**See also:**

[cudaMalloc](#), [cudaMallocPitch](#), [cudaMallocArray](#), [cudaMalloc3D](#), [cudaMalloc3DArray](#), [cudaHostAlloc](#), [cudaFree](#), [cudaFreeArray](#), [cudaFreeHost](#), [cudaHostAlloc](#)

**3.8.2.13 `cudaError_t cudaMallocPitch (void ** devPtr, size_t * pitch, size_t width, size_t height)`**

Allocates at least `widthInBytes * 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).

**Parameters:**

*devPtr* - Pointer to allocated pitched device memory  
*pitch* - Pitch for allocation  
*width* - Requested pitched allocation width  
*height* - Requested pitched allocation height

**Returns:**

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

**Note:**

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

**See also:**

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

**3.8.2.14 `cudaError_t cudaMemcpy (void * dst, const void * src, size_t count, enum cudaMemcpyKind kind)`**

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` is one of [`cudaMemcpyHostToHost`](#), [`cudaMemcpyHostToDevice`](#), [`cudaMemcpyDeviceToHost`](#), or [`cudaMemcpyDeviceToDevice`](#), and specifies the direction of the copy. The memory areas may not overlap. Calling [`cudaMemcpy\(\)`](#) with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior.

**Parameters:**

*dst* - Destination memory address

*src* - Source memory address

*count* - Size in bytes to copy

*kind* - Type of transfer

**Returns:**

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

**Note:**

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

**See also:**

[`cudaMemcpy2D`](#), [`cudaMemcpyToArray`](#), [`cudaMemcpy2DToArray`](#), [`cudaMemcpyFromArray`](#), [`cudaMemcpy2DFromArray`](#), [`cudaMemcpyArrayToArray`](#), [`cudaMemcpy2DArrayToArray`](#), [`cudaMemcpyToSymbol`](#), [`cudaMemcpyFromSymbol`](#), [`cudaMemcpyAsync`](#), [`cudaMemcpy2DAsync`](#), [`cudaMemcpyToArrayAsync`](#), [`cudaMemcpy2DToArrayAsync`](#), [`cudaMemcpyFromArrayAsync`](#), [`cudaMemcpy2DFromArrayAsync`](#), [`cudaMemcpyToSymbolAsync`](#), [`cudaMemcpyFromSymbolAsync`](#)

**3.8.2.15 `cudaError_t cudaMemcpy2D (void * dst, size_t dpitch, const void * src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind)`**

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` is one of [`cudaMemcpyHostToHost`](#), [`cudaMemcpyHostToDevice`](#), [`cudaMemcpyDeviceToHost`](#), or [`cudaMemcpyDeviceToDevice`](#), and specifies the direction of the copy. `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. 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` is greater than the maximum allowed.

**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](#)

**Note:**

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

**See also:**

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

### 3.8.2.16 `cudaError_t cudaMemcpy2DArrayToArray (struct cudaArray * dst, size_t wOffsetDst, size_t hOffsetDst, const struct cudaArray * src, size_t wOffsetSrc, size_t hOffsetSrc, size_t width, size_t height, enum cudaMemcpyKind kind)`

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` is one of [cudaMemcpyHostToHost](#), [cudaMemcpyHostToDevice](#), [cudaMemcpyDeviceToHost](#), or [cudaMemcpyDeviceToDevice](#), and specifies the direction of the copy.

**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](#)

**Note:**

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

**See also:**

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



### 3.8.2.17 `cudaError_t cudaMemcpy2DAsync` (`void *dst`, `size_t dpitch`, `const void *src`, `size_t spitch`, `size_t width`, `size_t height`, `enum cudaMemcpyKind kind`, `cudaStream_t stream`)

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` is one of [cudaMemcpyHostToHost](#), [cudaMemcpyHostToDevice](#), [cudaMemcpyDeviceToHost](#), or [cudaMemcpyDeviceToDevice](#), and specifies the direction of the copy. `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. 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. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input. 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.

**IMPORTANT NOTE:** Copies with `kind == cudaMemcpyDeviceToDevice` are asynchronous with respect to the host, but never overlap with kernel execution.

#### 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](#)

#### Note:

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

#### See also:

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

### 3.8.2.18 `cudaError_t cudaMemcpy2DFromArray` (`void *dst`, `size_t dpitch`, `const struct cudaArray *src`, `size_t wOffset`, `size_t hOffset`, `size_t width`, `size_t height`, `enum cudaMemcpyKind kind`)

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` is one of [cudaMemcpyHostToHost](#), [cudaMemcpyHostToDevice](#), [cudaMemcpyDeviceToHost](#), or [cudaMemcpyDeviceToDevice](#), and specifies the

direction of the copy. `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. `cudaMemcpy2DFromArray()` returns an error if `dpitch` is greater than the maximum allowed.

**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`

**Note:**

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

**See also:**

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

**3.8.2.19** `cudaError_t cudaMemcpy2DFromArrayAsync (void *dst, size_t dpitch, const struct cudaArray *src, size_t wOffset, size_t hOffset, size_t width, size_t height, enum cudaMemcpyKind kind, cudaStream_t stream)`

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` is one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, or `cudaMemcpyDeviceToDevice`, and specifies the direction of the copy. `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. `cudaMemcpy2DFromArrayAsync()` returns an error if `dpitch` is greater than the maximum allowed.

`cudaMemcpy2DFromArrayAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input. 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.

**IMPORTANT NOTE:** Copies with `kind == cudaMemcpyDeviceToDevice` are asynchronous with respect to the host, but never overlap with kernel execution.

**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](#)

**Note:**

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

**See also:**

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpyToArray](#), [cudaMemcpy2DToArray](#), [cudaMemcpyFromArray](#), [cudaMemcpy2DFromArray](#), [cudaMemcpyArrayToArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpyFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

**3.8.2.20 `cudaError_t cudaMemcpy2DToArray` (`struct cudaArray * dst`, `size_t wOffset`, `size_t hOffset`, `const void * src`, `size_t spitch`, `size_t width`, `size_t height`, `enum cudaMemcpyKind kind`)**

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` is one of [cudaMemcpyHostToHost](#), [cudaMemcpyHostToDevice](#), [cudaMemcpyDeviceToHost](#), or [cudaMemcpyDeviceToDevice](#), and specifies the direction of the copy. `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. [cudaMemcpy2DToArray\(\)](#) returns an error if `spitch` is greater than the maximum allowed.

**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](#)

**Note:**

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

**See also:**

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

### 3.8.2.21 `cudaError_t cudaMemcpy2DToArrayAsync(struct cudaArray * dst, size_t wOffset, size_t hOffset, const void * src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind, cudaStream_t stream)`

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* is one of [cudaMemcpyHostToHost](#), [cudaMemcpyHostToDevice](#), [cudaMemcpyDeviceToHost](#), or [cudaMemcpyDeviceToDevice](#), and specifies the direction of the copy. *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. [cudaMemcpy2DToArrayAsync\(\)](#) returns an error if *spitch* is greater than the maximum allowed.

[cudaMemcpy2DToArrayAsync\(\)](#) is asynchronous with respect to the host, so the call may return before the copy is complete. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input. 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.

**IMPORTANT NOTE:** Copies with *kind* == [cudaMemcpyDeviceToDevice](#) are asynchronous with respect to the host, but never overlap with kernel execution.

**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](#)

**Note:**

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

**See also:**

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpyToArray](#), [cudaMemcpy2DToArray](#), [cudaMemcpyFromArray](#), [cudaMemcpy2DFromArray](#), [cudaMemcpyToArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpyFromArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

**3.8.2.22 cudaMemcpy3D (const struct cudaMemcpy3DParms \*p)**

```

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 {
    struct cudaArray    *srcArray;
    struct cudaPos      srcPos;
    struct cudaPitchedPtr srcPtr;
    struct cudaArray    *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](#), or [cudaMemcpyDeviceToDevice](#).

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.

`cudaMemcpy3D()` returns an error if the pitch of `srcPtr` or `dstPtr` is greater than the maximum allowed. The pitch of a `cudaPitchedPtr` allocated with `cudaMalloc3D()` will always be valid.

**Parameters:**

*p* - 3D memory copy parameters

**Returns:**

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevicePointer`, `cudaErrorInvalidPitchValue`, `cudaErrorInvalidMemcpyDirection`

**Note:**

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

**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`

### 3.8.2.23 `cudaError_t cudaMemcpy3DAsync(const struct cudaMemcpy3DParms *p, cudaStream_t stream)`

```

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 {
    struct cudaArray    *srcArray;
    struct cudaPos      srcPos;
    struct cudaPitchedPtr srcPtr;
    struct cudaArray    *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`, or `cudaMemcpyDeviceToDevice`.

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.

`cudaMemcpy3DAsync()` returns an error if the pitch of `srcPtr` or `dstPtr` is greater than 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. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input. 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.

**IMPORTANT NOTE:** Copies with `kind == cudaMemcpyDeviceToDevice` are asynchronous with respect to the host, but never overlap with kernel execution.

#### Parameters:

*p* - 3D memory copy parameters

*stream* - Stream identifier

#### Returns:

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevicePointer`, `cudaErrorInvalidPitchValue`, `cudaErrorInvalidMemcpyDirection`

#### Note:

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

#### 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`

### 3.8.2.24 `cudaError_t cudaMemcpyToArray (struct cudaArray * dst, size_t wOffsetDst, size_t hOffsetDst, const struct cudaArray * src, size_t wOffsetSrc, size_t hOffsetSrc, size_t count, enum cudaMemcpyKind kind)`

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` is one of [cudaMemcpyHostToHost](#), [cudaMemcpyHostToDevice](#), [cudaMemcpyDeviceToHost](#), or [cudaMemcpyDeviceToDevice](#), and specifies the direction of the copy.

#### 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](#)

#### Note:

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](#)

### 3.8.2.25 `cudaError_t cudaMemcpyAsync (void * dst, const void * src, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream)`

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` is one of [cudaMemcpyHostToHost](#), [cudaMemcpyHostToDevice](#), [cudaMemcpyDeviceToHost](#), or [cudaMemcpyDeviceToDevice](#), and specifies the direction of the copy. 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. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input. 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.

**IMPORTANT NOTE:** Copies with `kind == cudaMemcpyDeviceToDevice` are asynchronous with respect to the host, but never overlap with kernel execution.



**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](#)

**Note:**

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

**See also:**

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpyToArray](#), [cudaMemcpy2DToArray](#), [cudaMemcpyFromArray](#), [cudaMemcpy2DFromArray](#), [cudaMemcpyArrayToArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpyFromArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

**3.8.2.26 `cudaError_t cudaMemcpyFromArray (void *dst, const struct cudaArray *src, size_t wOffset, size_t hOffset, size_t count, enum cudaMemcpyKind kind)`**

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` is one of [cudaMemcpyHostToHost](#), [cudaMemcpyHostToDevice](#), [cudaMemcpyDeviceToHost](#), or [cudaMemcpyDeviceToDevice](#), and specifies the direction of the copy.

**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](#)

**Note:**

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

**See also:**

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpyToArray](#), [cudaMemcpy2DToArray](#), [cudaMemcpy2DFromArray](#), [cudaMemcpyArrayToArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpyFromArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

### 3.8.2.27 `cudaError_t cudaMemcpyFromArrayAsync (void * dst, const struct cudaArray * src, size_t wOffset, size_t hOffset, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream)`

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` is one of [cudaMemcpyHostToHost](#), [cudaMemcpyHostToDevice](#), [cudaMemcpyDeviceToHost](#), or [cudaMemcpyDeviceToDevice](#), and specifies the direction of the copy.

`cudaMemcpyFromArrayAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input. 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.

**IMPORTANT NOTE:** Copies with `kind == cudaMemcpyDeviceToDevice` are asynchronous with respect to the host, but never overlap with kernel execution.

#### 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](#)

#### Note:

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

#### See also:

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpyToArray](#), [cudaMemcpy2DToArray](#), [cudaMemcpyFromArray](#), [cudaMemcpy2DFromArray](#), [cudaMemcpyArrayToArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

### 3.8.2.28 `cudaError_t cudaMemcpyFromSymbol (void * dst, const char * symbol, size_t count, size_t offset, enum cudaMemcpyKind kind)`

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` can either be a variable that resides in global or constant memory space, or it can be a character string, naming a variable that resides in global or constant memory space. `kind` can be either [cudaMemcpyDeviceToHost](#) or [cudaMemcpyDeviceToDevice](#).

#### Parameters:

*dst* - Destination memory address  
*symbol* - Symbol source from device

*count* - Size in bytes to copy  
*offset* - Offset from start of symbol in bytes  
*kind* - Type of transfer

**Returns:**

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

**Note:**

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

**See also:**

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpyToArray](#), [cudaMemcpy2DToArray](#), [cudaMemcpyFromArray](#), [cudaMemcpy2DFromArray](#), [cudaMemcpyArrayToArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpyFromArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

### 3.8.2.29 `cudaError_t cudaMemcpyFromSymbolAsync(void *dst, const char *symbol, size_t count, size_t offset, enum cudaMemcpyKind kind, cudaStream_t stream)`

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` can either be a variable that resides in global or constant memory space, or it can be a character string, naming 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. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input. 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.

**IMPORTANT NOTE:** Copies with `kind == cudaMemcpyDeviceToDevice` are asynchronous with respect to the host, but never overlap with kernel execution.

**Parameters:**

*dst* - Destination memory address  
*symbol* - Symbol source from device  
*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](#)

**Note:**

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

**See also:**

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpyToArray](#), [cudaMemcpy2DToArray](#), [cudaMemcpyFromArray](#), [cudaMemcpy2DFromArray](#), [cudaMemcpyArrayToArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpyFromArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#)

### 3.8.2.30 `cudaError_t cudaMemcpyToArray (struct cudaArray * dst, size_t wOffset, size_t hOffset, const void * src, size_t count, enum cudaMemcpyKind kind)`

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` is one of [cudaMemcpyHostToHost](#), [cudaMemcpyHostToDevice](#), [cudaMemcpyDeviceToHost](#), or [cudaMemcpyDeviceToDevice](#), and specifies the direction of the copy.

**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](#)

**Note:**

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

**See also:**

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpy2DToArray](#), [cudaMemcpyFromArray](#), [cudaMemcpy2DFromArray](#), [cudaMemcpyArrayToArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpyFromArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

### 3.8.2.31 `cudaError_t cudaMemcpyToArrayAsync (struct cudaArray * dst, size_t wOffset, size_t hOffset, const void * src, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream)`

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` is one of [cudaMemcpyHostToHost](#), [cudaMemcpyHostToDevice](#), [cudaMemcpyDeviceToHost](#), or [cudaMemcpyDeviceToDevice](#), and specifies the direction of the copy.

`cudaMemcpyToArrayAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input. 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.

**IMPORTANT NOTE:** Copies with `kind == cudaMemcpyDeviceToDevice` are asynchronous with respect to the host, but never overlap with kernel execution.

**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](#)

**Note:**

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

**See also:**

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpyToArray](#), [cudaMemcpy2DToArray](#), [cudaMemcpyFromArray](#), [cudaMemcpy2DFromArray](#), [cudaMemcpyArrayToArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpyFromArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

**3.8.2.32 `cudaError_t cudaMemcpyToSymbol(const char * symbol, const void * src, size_t count, size_t offset, enum cudaMemcpyKind kind)`**

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` can either be a variable that resides in global or constant memory space, or it can be a character string, naming a variable that resides in global or constant memory space. `kind` can be either [cudaMemcpyHostToDevice](#) or [cudaMemcpyDeviceToDevice](#).

**Parameters:**

*symbol* - Symbol destination on device  
*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](#)

**Note:**

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

**See also:**

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpyToArray](#), [cudaMemcpy2DToArray](#), [cudaMemcpyFromArray](#), [cudaMemcpy2DFromArray](#), [cudaMemcpyArrayToArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyFromSymbolAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpyFromArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

### 3.8.2.33 `cudaError_t cudaMemcpyToSymbolAsync(const char *symbol, const void *src, size_t count, size_t offset, enum cudaMemcpyKind kind, cudaStream_t stream)`

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` can either be a variable that resides in global or constant memory space, or it can be a character string, naming 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. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input. 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.

**IMPORTANT NOTE:** Copies with `kind == cudaMemcpyDeviceToDevice` are asynchronous with respect to the host, but never overlap with kernel execution.

**Parameters:**

*symbol* - Symbol destination on device  
*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](#)

**Note:**

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

**See also:**

[cudaMemcpy](#), [cudaMemcpy2D](#), [cudaMemcpyToArray](#), [cudaMemcpy2DToArray](#), [cudaMemcpyFromArray](#), [cudaMemcpy2DFromArray](#), [cudaMemcpyArrayToArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpyFromArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyFromSymbolAsync](#)

### 3.8.2.34 `cudaError_t cudaMemset(void *devPtr, int value, size_t count)`

Fills the first `count` bytes of the memory area pointed to by `devPtr` with the constant byte value `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](#)

**Note:**

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

**See also:**

[cudaMemset2D](#), [cudaMemset3D](#)

**3.8.2.35 `cudaError_t cudaMemset2D (void * devPtr, size_t pitch, int value, size_t width, size_t height)`**

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\(\)](#).

**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](#)

**Note:**

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

**See also:**

[cudaMemset](#), [cudaMemset3D](#)

**3.8.2.36 `cudaError_t cudaMemset3D (struct cudaPitchedPtr pitchedDevPtr, int value, struct cudaExtent extent)`**

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\(\)](#).

**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

**Returns:**

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

**Note:**

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

**See also:**

[cudaMemset](#), [cudaMemset2D](#), [cudaMalloc3D](#), [make\\_cudaPitchedPtr](#), [make\\_cudaExtent](#)



## 3.9 OpenGL Interoperability

### Functions

- [cudaError\\_t cudaGLMapBufferObject](#) (void \*\*devPtr, GLuint bufObj)  
*Maps a buffer object for access by CUDA.*
- [cudaError\\_t cudaGLRegisterBufferObject](#) (GLuint bufObj)  
*Registers a buffer object for access by CUDA.*
- [cudaError\\_t cudaGLSetGLDevice](#) (int device)  
*Sets the CUDA device for use with OpenGL interoperability.*
- [cudaError\\_t cudaGLUnmapBufferObject](#) (GLuint bufObj)  
*Unmaps a buffer object for access by CUDA.*
- [cudaError\\_t cudaGLUnregisterBufferObject](#) (GLuint bufObj)  
*Unregisters a buffer object for access by CUDA.*
- [cudaError\\_t cudaWGLGetDevice](#) (int \*device, HGPUNV hGpu)  
*Gets the CUDA device associated with hGpu.*

### 3.9.1 Detailed Description

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

### 3.9.2 Function Documentation

#### 3.9.2.1 [cudaError\\_t cudaGLMapBufferObject](#) (void \*\* devPtr, GLuint bufObj)

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.

#### Parameters:

- `devPtr` - Returned device pointer to CUDA object
- `bufObj` - Buffer object ID to map

#### Returns:

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

#### Note:

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

**See also:**

[cudaGLSetGLDevice](#), [cudaGLRegisterBufferObject](#), [cudaGLUnmapBufferObject](#), [cudaGLUnregisterBufferObject](#)

**3.9.2.2 `cudaError_t cudaGLRegisterBufferObject (GLuint bufObj)`**

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.

**Parameters:**

*bufObj* - Buffer object ID to register

**Returns:**

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

**Note:**

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

**See also:**

[cudaGLSetGLDevice](#), [cudaGLMapBufferObject](#), [cudaGLUnmapBufferObject](#), [cudaGLUnregisterBufferObject](#)

**3.9.2.3 `cudaError_t cudaGLSetGLDevice (int device)`**

Records *device* as the device on which the active host thread executes the device code. Records the thread as using OpenGL interoperability. If the host thread has already initialized the CUDA runtime by calling non-device management runtime functions, this call returns [cudaErrorSetOnActiveProcess](#).

**Parameters:**

*device* - Device to use for OpenGL interoperability

**Returns:**

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

**Note:**

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

**See also:**

[cudaGLRegisterBufferObject](#), [cudaGLMapBufferObject](#), [cudaGLUnmapBufferObject](#), [cudaGLUnregisterBufferObject](#)

**3.9.2.4 `cudaError_t cudaGLUnmapBufferObject (GLuint bufObj)`**

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.

**Parameters:**

*bufObj* - Buffer object to unmap

**Returns:**

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

**Note:**

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

**See also:**

[cudaGLSetGLDevice](#), [cudaGLRegisterBufferObject](#), [cudaGLMapBufferObject](#), [cudaGLUnregisterBufferObject](#)

**3.9.2.5 `cudaError_t cudaGLUnregisterBufferObject (GLuint bufObj)`**

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.

**Parameters:**

*bufObj* - Buffer object to unregister

**Returns:**

[cudaSuccess](#)

**Note:**

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

**See also:**

[cudaGLSetGLDevice](#), [cudaGLRegisterBufferObject](#), [cudaGLMapBufferObject](#), [cudaGLUnmapBufferObject](#)

**3.9.2.6 `cudaError_t cudaWGLGetDevice (int * device, HGPUNV hGpu)`**

Returns the CUDA device associated with a `hGpu`, if applicable.

**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](#)

**Note:**

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

**See also:**

`WGL_NV_gpu_affinity`, [cudaGLSetGLDevice](#)

## 3.10 Direct3D 9 Interoperability

### Direct3D 9 Interoperability with CUDA Runtime

Direct3D 9 Interoperability with CUDA Runtime

#### Author:

NVIDIA Corporation

- enum [cudaD3D9MapFlags](#) {  
[cudaD3D9MapFlagsNone](#),  
[cudaD3D9MapFlagsReadOnly](#),  
[cudaD3D9MapFlagsWriteDiscard](#) }
- enum [cudaD3D9RegisterFlags](#) {  
[cudaD3D9RegisterFlagsNone](#),  
[cudaD3D9RegisterFlagsArray](#) }
- [cudaError\\_t cudaD3D9GetDevice](#) (int \*device, const char \*pszAdapterName)  
*Gets the device number for an adapter.*
- [cudaError\\_t cudaD3D9GetDirect3DDevice](#) (IDirect3DDevice9 \*\*ppDxDevice)  
*Gets the Direct3D device against which the current CUDA context was created.*
- [cudaError\\_t cudaD3D9MapResources](#) (int count, IDirect3DResource9 \*\*ppResources)  
*Map Direct3D resources for access by CUDA.*
- [cudaError\\_t cudaD3D9RegisterResource](#) (IDirect3DResource9 \*pResource, unsigned int flags)  
*Registers a Direct3D resource for access by CUDA.*
- [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.*
- [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.*
- [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.*
- [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.*
- [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.*

- `cudaError_t cudaD3D9ResourceSetMapFlags` (`IDirect3DResource9 *pResource`, unsigned int flags)  
*Set usage flags for mapping a Direct3D resource.*
- `cudaError_t cudaD3D9SetDirect3DDevice` (`IDirect3DDevice9 *pDxDevice`)  
*Sets the Direct3D device to use for interoperability in this thread.*
- `cudaError_t cudaD3D9UnmapResources` (int count, `IDirect3DResource9 **ppResources`)  
*Unmap Direct3D resources for access by CUDA.*
- `cudaError_t cudaD3D9UnregisterResource` (`IDirect3DResource9 *pResource`)  
*Unregisters a Direct3D resource for access by CUDA.*

### 3.10.1 Detailed Description

This section describes the Direct3D 9 interoperability functions of the CUDA runtime application programming interface.

### 3.10.2 Enumeration Type Documentation

#### 3.10.2.1 enum cudaD3D9MapFlags

CUDA D3D9 Map Flags

##### Enumerator:

- `cudaD3D9MapFlagsNone` Default; Assume resource can be read/written.
- `cudaD3D9MapFlagsReadOnly` CUDA kernels will not write to this resource.
- `cudaD3D9MapFlagsWriteDiscard` CUDA kernels will only write to and will not read from this resource.

#### 3.10.2.2 enum cudaD3D9RegisterFlags

CUDA D3D9 Register Flags

##### Enumerator:

- `cudaD3D9RegisterFlagsNone` Default; Resource can be accessed through a void\*.
- `cudaD3D9RegisterFlagsArray` Resource can be accessed through a CUarray\*.

### 3.10.3 Function Documentation

#### 3.10.3.1 cudaError\_t cudaD3D9GetDevice (int \* device, const char \* pszAdapterName)

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.

##### Parameters:

`device` - Returns the device corresponding to `pszAdapterName`

*pszAdapterName* - D3D9 adapter to get device for

**Returns:**

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

**Note:**

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

**See also:**

[cudaD3D9SetDirect3DDevice](#), [cudaD3D9GetDirect3DDevice](#), [cudaD3D9RegisterResource](#),  
[cudaD3D9UnregisterResource](#), [cudaD3D9MapResources](#), [cudaD3D9UnmapResources](#),  
[cudaD3D9ResourceGetSurfaceDimensions](#), [cudaD3D9ResourceSetMapFlags](#), [cu-](#)  
[daD3D9ResourceGetMappedArray](#), [cudaD3D9ResourceGetMappedPointer](#), [cu-](#)  
[daD3D9ResourceGetMappedSize](#), [cudaD3D9ResourceGetMappedPitch](#)

### 3.10.3.2 `cudaError_t cudaD3D9GetDirect3DDevice (IDirect3DDevice9 ** ppDxDevice)`

Returns in *ppDxDevice* the Direct3D device against which this CUDA context was created in [cudaD3D9SetDirect3DDevice\(\)](#).

**Parameters:**

*ppDxDevice* - Returns the Direct3D device for this thread

**Returns:**

[cudaSuccess](#), [cudaErrorUnknown](#)

**Note:**

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

**See also:**

[cudaD3D9GetDevice](#), [cudaD3D9SetDirect3DDevice](#), [cudaD3D9RegisterResource](#), [cu-](#)  
[daD3D9UnregisterResource](#), [cudaD3D9MapResources](#), [cudaD3D9UnmapResources](#),  
[cudaD3D9ResourceGetSurfaceDimensions](#), [cudaD3D9ResourceSetMapFlags](#), [cu-](#)  
[daD3D9ResourceGetMappedArray](#), [cudaD3D9ResourceGetMappedPointer](#), [cu-](#)  
[daD3D9ResourceGetMappedSize](#), [cudaD3D9ResourceGetMappedPitch](#)

### 3.10.3.3 `cudaError_t cudaD3D9MapResources (int count, IDirect3DResource9 ** ppResources)`

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.

**Parameters:**

*count* - Number of resources to map for CUDA

*ppResources* - Resources to map for CUDA

**Returns:**

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

**Note:**

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

**See also:**

[cudaD3D9GetDevice](#), [cudaD3D9SetDirect3DDevice](#), [cudaD3D9GetDirect3DDevice](#), [cudaD3D9RegisterResource](#), [cudaD3D9UnregisterResource](#), [cudaD3D9UnmapResources](#), [cudaD3D9ResourceGetSurfaceDimensions](#), [cudaD3D9ResourceSetMapFlags](#), [cudaD3D9ResourceGetMappedArray](#), [cudaD3D9ResourceGetMappedPointer](#), [cudaD3D9ResourceGetMappedSize](#), [cudaD3D9ResourceGetMappedPitch](#)

**3.10.3.4 `cudaError_t cudaD3D9RegisterResource (IDirect3DResource9 * pResource, unsigned int flags)`**

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 is potentially high-overhead and should not be called every frame in interactive applications.

The type of `pResource` must be one of the following.

- `IDirect3DVertexBuffer9`: 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.

**Parameters:**

*pResource* - Resource to register  
*flags* - Parameters for resource registration

**Returns:**

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

**Note:**

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

**See also:**

[cudaD3D9GetDevice](#), [cudaD3D9SetDirect3DDevice](#), [cudaD3D9GetDirect3DDevice](#), [cudaD3D9UnregisterResource](#), [cudaD3D9MapResources](#), [cudaD3D9UnmapResources](#), [cudaD3D9ResourceGetSurfaceDimensions](#), [cudaD3D9ResourceSetMapFlags](#), [cudaD3D9ResourceGetMappedArray](#), [cudaD3D9ResourceGetMappedPointer](#), [cudaD3D9ResourceGetMappedSize](#), [cudaD3D9ResourceGetMappedPitch](#)

**3.10.3.5 `cudaError_t cudaD3D9ResourceGetMappedArray (cudaArray **ppArray, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)`**

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\(\)](#).

**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](#)

**Note:**

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

**See also:**

[cudaD3D9GetDevice](#), [cudaD3D9SetDirect3DDevice](#), [cudaD3D9GetDirect3DDevice](#), [cudaD3D9RegisterResource](#), [cudaD3D9UnregisterResource](#), [cudaD3D9MapResources](#), [cudaD3D9UnmapResources](#), [cudaD3D9ResourceGetSurfaceDimensions](#), [cudaD3D9ResourceSetMapFlags](#), [cudaD3D9ResourceGetMappedPointer](#), [cudaD3D9ResourceGetMappedSize](#), [cudaD3D9ResourceGetMappedPitch](#)

### 3.10.3.6 `cudaError_t cudaD3D9ResourceGetMappedPitch(size_t * pPitch, size_t * pPitchSlice, IDirect3DResource9 * pResource, unsigned int face, unsigned int level)`

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\(\)](#).

**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](#)

**Note:**

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

**See also:**

[cudaD3D9GetDevice](#), [cudaD3D9SetDirect3DDevice](#), [cudaD3D9GetDirect3DDevice](#), [cudaD3D9RegisterResource](#), [cudaD3D9UnregisterResource](#), [cudaD3D9MapResources](#), [cudaD3D9UnmapResources](#), [cudaD3D9ResourceGetSurfaceDimensions](#), [cudaD3D9ResourceSetMapFlags](#), [cudaD3D9ResourceGetMappedArray](#), [cudaD3D9ResourceGetMappedPointer](#), [cudaD3D9ResourceGetMappedSize](#)

### 3.10.3.7 `cudaError_t cudaD3D9ResourceGetMappedPointer (void ** pPointer, IDirect3DResource9 * pResource, unsigned int face, unsigned int level)`

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.

**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](#)

**Note:**

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

**See also:**

[cudaD3D9GetDevice](#), [cudaD3D9SetDirect3DDevice](#), [cudaD3D9GetDirect3DDevice](#), [cudaD3D9RegisterResource](#), [cudaD3D9UnregisterResource](#), [cudaD3D9MapResources](#), [cudaD3D9UnmapResources](#), [cudaD3D9ResourceGetSurfaceDimensions](#), [cudaD3D9ResourceSetMapFlags](#), [cudaD3D9ResourceGetMappedArray](#), [cudaD3D9ResourceGetMappedSize](#), [cudaD3D9ResourceGetMappedPitch](#)

### 3.10.3.8 `cudaError_t cudaD3D9ResourceGetMappedSize (size_t * pSize, IDirect3DResource9 * pResource, unsigned int face, unsigned int level)`

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()`.

**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`

**Note:**

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

**See also:**

`cudaD3D9GetDevice`, `cudaD3D9SetDirect3DDevice`, `cudaD3D9GetDirect3DDevice`, `cudaD3D9RegisterResource`, `cudaD3D9UnregisterResource`, `cudaD3D9MapResources`, `cudaD3D9UnmapResources`, `cudaD3D9ResourceGetSurfaceDimensions`, `cudaD3D9ResourceSetMapFlags`, `cudaD3D9ResourceGetMappedArray`, `cudaD3D9ResourceGetMappedPointer`, `cudaD3D9ResourceGetMappedPitch`

**3.10.3.9 `cudaError_t cudaD3D9ResourceGetSurfaceDimensions(size_t *pWidth, size_t *pHeight, size_t *pDepth, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)`**

Returns in `*pWidth`, `*pHeight`, and `*pDepth` the dimensions of the subresource of the mapped Direct3D resource `pResource` which corresponds to `face` and `level`.

Because anti-aliased surfaces may have multiple samples per pixel, it is possible that the dimensions of a resource will be an integer factor larger than the dimensions reported by the Direct3D runtime.

The parameters `pWidth`, `pHeight`, and `pDepth` are optional. For 2D surfaces, the value returned in `*pDepth` will be 0.

If `pResource` is not of type `IDirect3DBaseTexture9` or `IDirect3DSurface9` or if `pResource` has not been registered for use with CUDA, then `cudaErrorInvalidResourceHandle` is returned.

For usage requirements of `face` and `level` parameters, see `cudaD3D9ResourceGetMappedPointer`.

**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](#),

**Note:**

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

**See also:**

[cudaD3D9GetDevice](#), [cudaD3D9SetDirect3DDevice](#), [cudaD3D9GetDirect3DDevice](#), [cudaD3D9RegisterResource](#), [cudaD3D9UnregisterResource](#), [cudaD3D9MapResources](#), [cudaD3D9UnmapResources](#), [cudaD3D9ResourceSetMapFlags](#), [cudaD3D9ResourceGetMappedArray](#), [cudaD3D9ResourceGetMappedPointer](#), [cudaD3D9ResourceGetMappedSize](#), [cudaD3D9ResourceGetMappedPitch](#)

### 3.10.3.10 `cudaError_t cudaD3D9ResourceSetMapFlags (IDirect3DResource9 * pResource, unsigned int flags)`

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.

**Parameters:**

*pResource* - Registered resource to set flags for

*flags* - Parameters for resource mapping

**Returns:**

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

**Note:**

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

**See also:**

[cudaD3D9GetDevice](#), [cudaD3D9SetDirect3DDevice](#), [cudaD3D9GetDirect3DDevice](#), [cudaD3D9RegisterResource](#), [cudaD3D9UnregisterResource](#), [cudaD3D9MapResources](#), [cudaD3D9UnmapResources](#), [cudaD3D9ResourceGetSurfaceDimensions](#), [cudaD3D9ResourceGetMappedArray](#), [cudaD3D9ResourceGetMappedPointer](#), [cudaD3D9ResourceGetMappedSize](#), [cudaD3D9ResourceGetMappedPitch](#)

**3.10.3.11 `cudaError_t cudaD3D9SetDirect3DDevice (IDirect3DDevice9 * pDxDevice)`**

Records `pDxDevice` as the Direct3D device to use for Direct3D interoperability on this host thread. In order to use Direct3D interoperability, this call must be made before any non-device management CUDA runtime calls on this thread. In that case, this call will return [`cudaErrorSetOnActiveProcess`](#).

Successful context creation on `pDxDevice` will increase the internal reference count on `pDxDevice`. This reference count will be decremented upon destruction of this context through [`cudaThreadExit\(\)`](#).

**Parameters:**

*pDxDevice* - Direct3D device to use for this thread

**Returns:**

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

**Note:**

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

**See also:**

[`cudaD3D9GetDevice`](#), [`cudaD3D9GetDirect3DDevice`](#), [`cudaD3D9RegisterResource`](#), [`cudaD3D9UnregisterResource`](#),  
[`cudaD3D9MapResources`](#), [`cudaD3D9UnmapResources`](#),  
[`cudaD3D9ResourceGetSurfaceDimensions`](#), [`cudaD3D9ResourceSetMapFlags`](#),  
[`cudaD3D9ResourceGetMappedArray`](#), [`cudaD3D9ResourceGetMappedPointer`](#),  
[`cudaD3D9ResourceGetMappedSize`](#), [`cudaD3D9ResourceGetMappedPitch`](#)

**3.10.3.12 `cudaError_t cudaD3D9UnmapResources (int count, IDirect3DResource9 ** ppResources)`**

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.

**Parameters:**

*count* - Number of resources to unmap for CUDA

*ppResources* - Resources to unmap for CUDA

**Returns:**

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

**Note:**

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

**See also:**

[`cudaD3D9GetDevice`](#), [`cudaD3D9SetDirect3DDevice`](#), [`cudaD3D9GetDirect3DDevice`](#), [`cudaD3D9RegisterResource`](#),  
[`cudaD3D9UnregisterResource`](#), [`cudaD3D9MapResources`](#),

[cudaD3D9ResourceGetSurfaceDimensions](#), [cudaD3D9ResourceSetMapFlags](#), [cu-](#)  
[daD3D9ResourceGetMappedArray](#), [cudaD3D9ResourceGetMappedPointer](#), [cu-](#)  
[daD3D9ResourceGetMappedSize](#), [cudaD3D9ResourceGetMappedPitch](#)

### 3.10.3.13 `cudaError_t cudaD3D9UnregisterResource (IDirect3DResource9 * pResource)`

Unregisters the Direct3D resource `pResource` so it is not accessible by CUDA unless registered again.

If `pResource` is not registered, then [cudaErrorInvalidResourceHandle](#) is returned.

#### Parameters:

*pResource* - Resource to unregister

#### Returns:

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

#### Note:

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

#### See also:

[cudaD3D9GetDevice](#), [cudaD3D9SetDirect3DDevice](#), [cudaD3D9GetDirect3DDevice](#), [cu-](#)  
[daD3D9RegisterResource](#), [cudaD3D9MapResources](#), [cudaD3D9UnmapResources](#),  
[cudaD3D9ResourceGetSurfaceDimensions](#), [cudaD3D9ResourceSetMapFlags](#), [cu-](#)  
[daD3D9ResourceGetMappedArray](#), [cudaD3D9ResourceGetMappedPointer](#), [cu-](#)  
[daD3D9ResourceGetMappedSize](#), [cudaD3D9ResourceGetMappedPitch](#)

## 3.11 Direct3D 10 Interoperability

### Direct3D 10 Interoperability with CUDA Runtime

Direct3D 10 Interoperability with CUDA Runtime

**Author:**

NVIDIA Corporation

- enum [cudaD3D10MapFlags](#) {  
[cudaD3D10MapFlagsNone](#),  
[cudaD3D10MapFlagsReadOnly](#),  
[cudaD3D10MapFlagsWriteDiscard](#) }
- enum [cudaD3D10RegisterFlags](#) {  
[cudaD3D10RegisterFlagsNone](#),  
[cudaD3D10RegisterFlagsArray](#) }
- [cudaError\\_t cudaD3D10GetDevice](#) (int \*device, IDXGIAdapter \*pAdapter)  
*Gets the device number for an adapter.*
- [cudaError\\_t cudaD3D10MapResources](#) (int count, ID3D10Resource \*\*ppResources)  
*Map Direct3D Resources for access by CUDA.*
- [cudaError\\_t cudaD3D10RegisterResource](#) (ID3D10Resource \*pResource, unsigned int flags)  
*Register a Direct3D resource for access by CUDA.*
- [cudaError\\_t cudaD3D10ResourceGetMappedArray](#) (cudaArray \*\*ppArray, ID3D10Resource \*pResource, unsigned int subResource)  
*Get an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- [cudaError\\_t cudaD3D10ResourceGetMappedPitch](#) (size\_t \*pPitch, size\_t \*pPitchSlice, ID3D10Resource \*pResource, unsigned int subResource)  
*Get the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- [cudaError\\_t cudaD3D10ResourceGetMappedPointer](#) (void \*\*ppPointer, ID3D10Resource \*pResource, unsigned int subResource)  
*Get a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- [cudaError\\_t cudaD3D10ResourceGetMappedSize](#) (size\_t \*pSize, ID3D10Resource \*pResource, unsigned int subResource)  
*Get the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- [cudaError\\_t cudaD3D10ResourceGetSurfaceDimensions](#) (size\_t \*pWidth, size\_t \*pHeight, size\_t \*pDepth, ID3D10Resource \*pResource, unsigned int subResource)  
*Get the dimensions of a registered Direct3D surface.*
- [cudaError\\_t cudaD3D10ResourceSetMapFlags](#) (ID3D10Resource \*pResource, unsigned int flags)  
*Set usage flags for mapping a Direct3D resource.*

- [cudaError\\_t cudaD3D10SetDirect3DDevice](#) (ID3D10Device \*pDxDevice)  
*Sets the Direct3D device to use for interoperability in this thread.*
- [cudaError\\_t cudaD3D10UnmapResources](#) (int count, ID3D10Resource \*\*ppResources)  
*Unmaps Direct3D resources.*
- [cudaError\\_t cudaD3D10UnregisterResource](#) (ID3D10Resource \*pResource)  
*Unregisters a Direct3D resource.*

### 3.11.1 Detailed Description

This section describes the Direct3D 10 interoperability functions of the CUDA runtime application programming interface.

### 3.11.2 Enumeration Type Documentation

#### 3.11.2.1 enum cudaD3D10MapFlags

CUDA D3D10 Map Flags

**Enumerator:**

- cudaD3D10MapFlagsNone* Default; Assume resource can be read/written.
- cudaD3D10MapFlagsReadOnly* CUDA kernels will not write to this resource.
- cudaD3D10MapFlagsWriteDiscard* CUDA kernels will only write to and will not read from this resource.

#### 3.11.2.2 enum cudaD3D10RegisterFlags

CUDA D3D10 Register Flags

**Enumerator:**

- cudaD3D10RegisterFlagsNone* Default; Resource can be accessed through a void\*.
- cudaD3D10RegisterFlagsArray* Resource can be accessed through a CUarray\*.

### 3.11.3 Function Documentation

#### 3.11.3.1 cudaError\_t cudaD3D10GetDevice (int \* device, IDXGIAdapter \* pAdapter)

Returns in *device* the CUDA-compatible device corresponding to the adapter *pAdapter* obtained from IDXGI-Factory::EnumAdapters. This call will succeed only if a device on adapter *pAdapter* is Cuda-compatible.

**Parameters:**

- device* - Returns the device corresponding to *pAdapter*
- pAdapter* - D3D10 adapter to get device for

**Returns:**

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



**Note:**

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

**See also:**

[cudaD3D10SetDirect3DDevice](#), [cudaD3D10RegisterResource](#), [cudaD3D10UnregisterResource](#), [cudaD3D10MapResources](#), [cudaD3D10UnmapResources](#), [cudaD3D10ResourceGetSurfaceDimensions](#), [cudaD3D10ResourceSetMapFlags](#), [cudaD3D10ResourceGetMappedArray](#), [cudaD3D10ResourceGetMappedPointer](#), [cudaD3D10ResourceGetMappedSize](#), [cudaD3D10ResourceGetMappedPitch](#)

**3.11.3.2 `cudaError_t cudaD3D10MapResources (int count, ID3D10Resource ** ppResources)`**

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.

**Parameters:**

*count* - Number of resources to map for CUDA

*ppResources* - Resources to map for CUDA

**Returns:**

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

**Note:**

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

**See also:**

[cudaD3D10GetDevice](#), [cudaD3D10SetDirect3DDevice](#), [cudaD3D10RegisterResource](#), [cudaD3D10UnregisterResource](#), [cudaD3D10UnmapResources](#), [cudaD3D10ResourceGetSurfaceDimensions](#), [cudaD3D10ResourceSetMapFlags](#), [cudaD3D10ResourceGetMappedArray](#), [cudaD3D10ResourceGetMappedPointer](#), [cudaD3D10ResourceGetMappedSize](#), [cudaD3D10ResourceGetMappedPitch](#)

**3.11.3.3 `cudaError_t cudaD3D10RegisterResource (ID3D10Resource * pResource, unsigned int flags)`**

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 is potentially high-overhead and should not be called every frame in interactive applications.

The type of `pResource` must be one of the following:

- `ID3D10Buffer`: Cannot be used with `flags` set to `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 `cuD3D10ResourceGetMappedArray()`. This option is only valid for resources of type `ID3D10Texture1D`, `ID3D10Texture2D`, and `ID3D10Texture3D`.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized on this context then `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.

#### Parameters:

*pResource* - Resource to register  
*flags* - Parameters for resource registration

#### Returns:

`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

#### Note:

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

#### See also:

`cudaD3D10GetDevice`, `cudaD3D10SetDirect3DDevice`, `cudaD3D10UnregisterResource`, `cudaD3D10MapResources`, `cudaD3D10UnmapResources`, `cudaD3D10ResourceGetSurfaceDimensions`, `cudaD3D10ResourceSetMapFlags`, `cudaD3D10ResourceGetMappedArray`, `cudaD3D10ResourceGetMappedPointer`, `cudaD3D10ResourceGetMappedSize`, `cudaD3D10ResourceGetMappedPitch`

### 3.11.3.4 `cudaError_t cudaD3D10ResourceGetMappedArray (cudaArray ** ppArray, ID3D10Resource * pResource, unsigned int subResource)`

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()`.

#### Parameters:

`ppArray` - Returned array corresponding to subresource  
`pResource` - Mapped resource to access  
`subResource` - Subresource of `pResource` to access

#### Returns:

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

#### Note:

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

#### See also:

`cudaD3D10GetDevice`, `cudaD3D10SetDirect3DDevice`, `cudaD3D10RegisterResource`, `cudaD3D10UnregisterResource`, `cudaD3D10MapResources`, `cudaD3D10UnmapResources`, `cudaD3D10ResourceGetSurfaceDimensions`, `cudaD3D10ResourceSetMapFlags`, `cudaD3D10ResourceGetMappedPointer`, `cudaD3D10ResourceGetMappedSize`, `cudaD3D10ResourceGetMappedPitch`

### 3.11.3.5 `cudaError_t cudaD3D10ResourceGetMappedPitch (size_t * pPitch, size_t * pPitchSlice, ID3D10Resource * pResource, unsigned int subResource)`

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()`.

**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](#)

**Note:**

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

**See also:**

[cudaD3D10GetDevice](#), [cudaD3D10SetDirect3DDevice](#), [cudaD3D10RegisterResource](#), [cu-daD3D10UnregisterResource](#), [cudaD3D10MapResources](#), [cudaD3D10UnmapResources](#), [cudaD3D10ResourceGetSurfaceDimensions](#), [cudaD3D10ResourceSetMapFlags](#), [cu-daD3D10ResourceGetMappedArray](#), [cudaD3D10ResourceGetMappedPointer](#), [cu-daD3D10ResourceGetMappedSize](#)

### 3.11.3.6 `cudaError_t cudaD3D10ResourceGetMappedPointer (void ** pPointer, ID3D10Resource * pResource, unsigned int subResource)`

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\(\)](#).

**Parameters:**

*pPointer* - Returned pointer corresponding to subresource  
*pResource* - Mapped resource to access  
*subResource* - Subresource of pResource to access

**Returns:**

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

**Note:**

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

**See also:**

[cudaD3D10GetDevice](#), [cudaD3D10SetDirect3DDevice](#), [cudaD3D10RegisterResource](#), [cu-daD3D10UnregisterResource](#), [cudaD3D10MapResources](#), [cudaD3D10UnmapResources](#), [cudaD3D10ResourceGetSurfaceDimensions](#), [cudaD3D10ResourceSetMapFlags](#), [cu-daD3D10ResourceGetMappedArray](#), [cudaD3D10ResourceGetMappedSize](#), [cu-daD3D10ResourceGetMappedPitch](#)

### 3.11.3.7 `cudaError_t cudaD3D10ResourceGetMappedSize (size_t * pSize, ID3D10Resource * pResource, unsigned int subResource)`

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 `cudaErrorInvalidHandle` is returned. If `pResource` is not mapped for access by CUDA then `cudaErrorUnknown` is returned.

For usage requirements of the `subResource` parameter see `cudaD3D10ResourceGetMappedPointer()`.

#### Parameters:

- `pSize` - Returned size of subresource
- `pResource` - Mapped resource to access
- `subResource` - Subresource of `pResource` to access

#### Returns:

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

#### Note:

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

#### See also:

`cudaD3D10GetDevice`, `cudaD3D10SetDirect3DDevice`, `cudaD3D10RegisterResource`, `cudaD3D10UnregisterResource`, `cudaD3D10MapResources`, `cudaD3D10UnmapResources`, `cudaD3D10ResourceGetSurfaceDimensions`, `cudaD3D10ResourceSetMapFlags`, `cudaD3D10ResourceGetMappedPointer`, `cudaD3D10ResourceGetMappedArray`, `cudaD3D10ResourceGetMappedPitch`

### 3.11.3.8 `cudaError_t cudaD3D10ResourceGetSurfaceDimensions (size_t * pWidth, size_t * pHeight, size_t * pDepth, ID3D10Resource * pResource, unsigned int subResource)`

Returns in `*pWidth`, `*pHeight`, and `*pDepth` the dimensions of the subresource of the mapped Direct3D resource `pResource` which corresponds to `subResource`.

Because anti-aliased surfaces may have multiple samples per pixel, it is possible that the dimensions of a resource will be an integer factor larger than the dimensions reported by the Direct3D runtime.

The parameters `pWidth`, `pHeight`, and `pDepth` are optional. For 2D surfaces, the value returned in `*pDepth` will be 0.

If `pResource` is not of type `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()`.

#### 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](#),

**Note:**

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

**See also:**

[cudaD3D10GetDevice](#), [cudaD3D10SetDirect3DDevice](#), [cudaD3D10RegisterResource](#), [cu-](#)  
[daD3D10UnregisterResource](#), [cudaD3D10MapResources](#), [cudaD3D10UnmapResources](#),  
[cudaD3D10ResourceSetMapFlags](#), [cudaD3D10ResourceGetMappedArray](#), [cu-](#)  
[daD3D10ResourceGetMappedPointer](#), [cudaD3D10ResourceGetMappedSize](#), [cu-](#)  
[daD3D10ResourceGetMappedPitch](#)

**3.11.3.9 `cudaError_t cudaD3D10ResourceSetMapFlags (ID3D10Resource * pResource, unsigned int flags)`**

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.

**Parameters:**

*pResource* - Registered resource to set flags for  
*flags* - Parameters for resource mapping

**Returns:**

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

**Note:**

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

**See also:**

[cudaD3D10GetDevice](#), [cudaD3D10SetDirect3DDevice](#), [cudaD3D10RegisterResource](#), [cu-](#)  
[daD3D10UnregisterResource](#), [cudaD3D10MapResources](#), [cudaD3D10UnmapResources](#),  
[cudaD3D10ResourceGetSurfaceDimensions](#), [cudaD3D10ResourceGetMappedArray](#), [cu-](#)  
[daD3D10ResourceGetMappedPointer](#), [cudaD3D10ResourceGetMappedSize](#), [cu-](#)  
[daD3D10ResourceGetMappedPitch](#)

### 3.11.3.10 `cudaError_t cudaD3D10SetDirect3DDevice (ID3D10Device * pDxDevice)`

Records `pDxDevice` as the Direct3D device to use for Direct3D interoperability on this host thread. In order to use Direct3D interoperability, this call must be made before any non-device management CUDA runtime calls on this thread. In that case, this call will return `cudaErrorSetActiveProcess`.

Successful context creation on `pDxDevice` will increase the internal reference count on `pDxDevice`. This reference count will be decremented upon destruction of this context through `cudaThreadExit()`.

#### Parameters:

*pDxDevice* - Direct3D device to use for interoperability

#### Returns:

`cudaSuccess`, `cudaErrorInitializationError`, `cudaErrorInvalidValue`, `cudaErrorSetActiveProcess`

#### Note:

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

#### See also:

`cudaD3D10GetDevice`, `cudaD3D10RegisterResource`, `cudaD3D10UnregisterResource`, `cu-`  
`daD3D10MapResources`, `cudaD3D10UnmapResources`, `cudaD3D10ResourceGetSurfaceDimensions`,  
`cudaD3D10ResourceSetMapFlags`, `cudaD3D10ResourceGetMappedArray`, `cu-`  
`daD3D10ResourceGetMappedPointer`, `cudaD3D10ResourceGetMappedSize`, `cu-`  
`daD3D10ResourceGetMappedPitch`

### 3.11.3.11 `cudaError_t cudaD3D10UnmapResources (int count, ID3D10Resource ** ppResources)`

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.

#### Parameters:

*count* - Number of resources to unmap for CUDA

*ppResources* - Resources to unmap for CUDA

#### Returns:

`cudaSuccess`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

#### Note:

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

#### See also:

`cudaD3D10GetDevice`, `cudaD3D10SetDirect3DDevice`, `cudaD3D10RegisterResource`, `cu-`  
`daD3D10UnregisterResource`, `cudaD3D10MapResources`, `cudaD3D10ResourceGetSurfaceDimensions`,

<a href="#">cudaD3D10ResourceSetMapFlags,</a>	<a href="#">cudaD3D10ResourceGetMappedArray,</a>	<a href="#">cu-</a>
<a href="#">daD3D10ResourceGetMappedPointer,</a>	<a href="#">cudaD3D10ResourceGetMappedSize,</a>	<a href="#">cu-</a>
<a href="#">daD3D10ResourceGetMappedPitch</a>		

### 3.11.3.12 `cudaError_t cudaD3D10UnregisterResource (ID3D10Resource * pResource)`

Unregisters the Direct3D resource `resource` so it is not accessible by CUDA unless registered again.

If `pResource` is not registered, then [cudaErrorInvalidResourceHandle](#) is returned.

#### Parameters:

*pResource* - Resource to unregister

#### Returns:

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

#### Note:

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

#### See also:

<a href="#">cudaD3D10GetDevice,</a>	<a href="#">cudaD3D10SetDirect3DDevice,</a>	<a href="#">cudaD3D10RegisterResource,</a>	<a href="#">cu-</a>
<a href="#">daD3D10MapResources,</a>	<a href="#">cudaD3D10UnmapResources,</a>	<a href="#">cudaD3D10ResourceGetSurfaceDimensions,</a>	
<a href="#">cudaD3D10ResourceSetMapFlags,</a>	<a href="#">cudaD3D10ResourceGetMappedArray,</a>	<a href="#">cu-</a>	
<a href="#">daD3D10ResourceGetMappedPointer,</a>	<a href="#">cudaD3D10ResourceGetMappedSize,</a>	<a href="#">cu-</a>	
<a href="#">daD3D10ResourceGetMappedPitch</a>			



## 3.12 Texture Reference Management

### Functions

- `cudaError_t cudaBindTexture` (`size_t *offset`, `const struct textureReference *texref`, `const void *devPtr`, `const struct cudaChannelFormatDesc *desc`, `size_t size`)  
*Binds a memory area to a texture.*
- `cudaError_t cudaBindTexture2D` (`size_t *offset`, `const struct textureReference *texref`, `const void *devPtr`, `const struct cudaChannelFormatDesc *desc`, `size_t width`, `size_t height`, `size_t pitch`)  
*Binds a 2D memory area to a texture.*
- `cudaError_t cudaBindTextureToArray` (`const struct textureReference *texref`, `const struct cudaArray *array`, `const struct cudaChannelFormatDesc *desc`)  
*Binds an array to a texture.*
- `struct cudaChannelFormatDesc cudaCreateChannelDesc` (`int x`, `int y`, `int z`, `int w`, `enum cudaChannelFormatKind f`)  
*Returns a channel descriptor using the specified format.*
- `cudaError_t cudaGetChannelDesc` (`struct cudaChannelFormatDesc *desc`, `const struct cudaArray *array`)  
*Get the channel descriptor of an array.*
- `cudaError_t cudaGetTextureAlignmentOffset` (`size_t *offset`, `const struct textureReference *texref`)  
*Get the alignment offset of a texture.*
- `cudaError_t cudaGetTextureReference` (`const struct textureReference **texref`, `const char *symbol`)  
*Get the texture reference associated with a symbol.*
- `cudaError_t cudaUnbindTexture` (`const struct textureReference *texref`)  
*Unbinds a texture.*

### 3.12.1 Detailed Description

This section describes the low level texture reference management functions of the CUDA runtime application programming interface.

### 3.12.2 Function Documentation

#### 3.12.2.1 `cudaError_t cudaBindTexture` (`size_t *offset`, `const struct textureReference *texref`, `const void *devPtr`, `const struct cudaChannelFormatDesc *desc`, `size_t size`)

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.

**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](#)

**Note:**

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)

**3.12.2.2 `cudaError_t cudaBindTexture2D` (`size_t * offset`, `const struct textureReference * texref`, `const void * devPtr`, `const struct cudaChannelFormatDesc * desc`, `size_t width`, `size_t height`, `size_t pitch`)**

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.

**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](#)

**Note:**

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), [cudaBindTextureToArray](#) (C API), [cudaGetTextureAlignmentOffset](#) (C API)

### 3.12.2.3 `cudaError_t cudaBindTextureToArray` (`const struct textureReference * texref`, `const struct cudaArray * array`, `const struct cudaChannelFormatDesc * desc`)

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.

**Parameters:**

*texref* - Texture to bind

*array* - Memory array on device

*desc* - Channel format

**Returns:**

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

**Note:**

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)

### 3.12.2.4 `struct cudaChannelFormatDesc cudaCreateChannelDesc` (`int x`, `int y`, `int z`, `int w`, `enum cudaChannelFormatKind f`) [read]

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](#).

**Parameters:**

*x* - X component

*y* - Y component

*z* - Z component  
*w* - W component  
*f* - Channel format

**Returns:**

Channel descriptor with format *f*

**See also:**

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

**3.12.2.5 `cudaError_t cudaGetChannelDesc (struct cudaChannelFormatDesc * desc, const struct cudaArray * array)`**

Returns in *\*desc* the channel descriptor of the CUDA array *array*.

**Parameters:**

*desc* - Channel format  
*array* - Memory array on device

**Returns:**

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

**Note:**

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 \(C API\)](#)

**3.12.2.6 `cudaError_t cudaGetTextureAlignmentOffset (size_t * offset, const struct textureReference * texref)`**

Returns in *\*offset* the offset that was returned when texture reference *texref* was bound.

**Parameters:**

*offset* - Offset of texture reference in bytes  
*texref* - Texture to get offset of

**Returns:**

[cudaSuccess](#), [cudaErrorInvalidTexture](#), [cudaErrorInvalidTextureBinding](#)

**Note:**

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)

**3.12.2.7 `cudaError_t cudaGetTextureReference` (`const struct textureReference ** texref`, `const char * symbol`)**

Returns in `*texref` the structure associated to the texture reference defined by symbol `symbol`.

**Parameters:**

*texref* - Texture associated with symbol  
*symbol* - Symbol to find texture reference for

**Returns:**

[cudaSuccess](#), [cudaErrorInvalidTexture](#),

**Note:**

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

**See also:**

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

**3.12.2.8 `cudaError_t cudaUnbindTexture` (`const struct textureReference * texref`)**

Unbinds the texture bound to `texref`.

**Parameters:**

*texref* - Texture to unbind

**Returns:**

[cudaSuccess](#)

**Note:**

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)

## 3.13 Version Management

### Functions

- [cudaError\\_t cudaDriverGetVersion](#) (int \*driverVersion)  
*Returns the CUDA driver version.*
- [cudaError\\_t cudaRuntimeGetVersion](#) (int \*runtimeVersion)  
*Returns the CUDA Runtime version.*

### 3.13.1 Function Documentation

#### 3.13.1.1 [cudaError\\_t cudaDriverGetVersion](#) (int \* *driverVersion*)

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.

#### Parameters:

*driverVersion* - Returns the CUDA driver version.

#### Returns:

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

#### Note:

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

#### See also:

[cudaRuntimeGetVersion](#)

#### 3.13.1.2 [cudaError\\_t cudaRuntimeGetVersion](#) (int \* *runtimeVersion*)

Returns in \**runtimeVersion* the version number of the installed CUDA Runtime. This function automatically returns [cudaErrorInvalidValue](#) if the *runtimeVersion* argument is NULL.

#### Parameters:

*runtimeVersion* - Returns the CUDA Runtime version.

#### Returns:

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

#### See also:

[cudaDriverGetVersion](#)

## 3.14 C++ API Routines

C++-style interface built on top of CUDA runtime API.

### Functions

- `template<class T, int dim, enum cudaTextureReadMode readMode>`  
`cudaError_t cudaBindTexture` (`size_t *offset`, `const struct texture< T, dim, readMode > &tex`, `const void *devPtr`, `size_t size=UINT_MAX`)  
*[C++ API] Binds a memory area to a texture*
- `template<class T, int dim, enum cudaTextureReadMode readMode>`  
`cudaError_t cudaBindTexture` (`size_t *offset`, `const struct texture< T, dim, readMode > &tex`, `const void *devPtr`, `const struct cudaChannelFormatDesc &desc`, `size_t size=UINT_MAX`)  
*[C++ API] Binds a memory area to a texture*
- `template<class T, int dim, enum cudaTextureReadMode readMode>`  
`cudaError_t cudaBindTexture2D` (`size_t *offset`, `const struct texture< T, dim, readMode > &tex`, `const void *devPtr`, `const struct cudaChannelFormatDesc &desc`, `size_t width`, `size_t height`, `size_t pitch`)  
*[C++ API] Binds a 2D memory area to a texture*
- `template<class T, int dim, enum cudaTextureReadMode readMode>`  
`cudaError_t cudaBindTextureToArray` (`const struct texture< T, dim, readMode > &tex`, `const struct cudaArray *array`)  
*[C++ API] Binds an array to a texture*
- `template<class T, int dim, enum cudaTextureReadMode readMode>`  
`cudaError_t cudaBindTextureToArray` (`const struct texture< T, dim, readMode > &tex`, `const struct cudaArray *array`, `const struct cudaChannelFormatDesc &desc`)  
*[C++ API] Binds an array to a texture*
- `template<class T >`  
`cudaChannelFormatDesc cudaCreateChannelDesc` (`void`)  
*[C++ API] Returns a channel descriptor using the specified format*
- `template<class T >`  
`cudaError_t cudaGetSymbolAddress` (`void **devPtr`, `const T &symbol`)  
*[C++ API] Finds the address associated with a CUDA symbol*
- `template<class T >`  
`cudaError_t cudaGetSymbolSize` (`size_t *size`, `const T &symbol`)  
*[C++ API] Finds the size of the object associated with a CUDA symbol*
- `template<class T, int dim, enum cudaTextureReadMode readMode>`  
`cudaError_t cudaGetTextureAlignmentOffset` (`size_t *offset`, `const struct texture< T, dim, readMode > &tex`)  
*[C++ API] Get the alignment offset of a texture*
- `template<class T >`  
`cudaError_t cudaLaunch` (`T *entry`)  
*[C++ API] Launches a device function*

- `template<class T >`  
[`cudaError\_t cudaSetupArgument`](#) (T arg, size\_t offset)  
*[C++ API] Configure a device launch*
- `template<class T, int dim, enum cudaTextureReadMode readMode>`  
[`cudaError\_t cudaUnbindTexture`](#) (const struct texture< T, dim, readMode > &tex)  
*[C++ API] Unbinds a texture*

### 3.14.1 Detailed Description

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.

### 3.14.2 Function Documentation

#### 3.14.2.1 `template<class T, int dim, enum cudaTextureReadMode readMode> cudaError_t cudaBindTexture (size_t * offset, const struct texture< T, dim, readMode > & tex, const void * devPtr, size_t size = UINT_MAX)`

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.

#### 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`](#)

#### Note:

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), [`cudaBindTextureToArray`](#) (C++ API), [`cudaBindTextureToArray`](#) (C++ API, inherited channel descriptor), [`cudaUnbindTexture`](#) (C++ API), [`cudaGetTextureAlignmentOffset`](#) (C++ API)

#### 3.14.2.2 `template<class T, int dim, enum cudaTextureReadMode readMode> cudaError_t cudaBindTexture (size_t * offset, const struct texture< T, dim, readMode > & tex, const void * devPtr, const struct cudaChannelFormatDesc & desc, size_t size = UINT_MAX)`

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.



**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](#)

**Note:**

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), [cudaBindTextureToArray](#) (C++ API), [cudaBindTextureToArray](#) (C++ API, inherited channel descriptor), [cudaUnbindTexture](#) (C++ API), [cudaGetTextureAlignmentOffset](#) (C++ API)

### 3.14.2.3 `template<class T, int dim, enum cudaTextureReadMode readMode> cudaError_t cudaBindTexture2D(size_t * offset, const struct texture< T, dim, readMode > & tex, const void * devPtr, const struct cudaChannelFormatDesc & desc, size_t width, size_t height, size_t pitch)`

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.

**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](#)

**Note:**

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), [cudaBindTextureToArray](#) (C++ API), [cudaBindTextureToArray](#) (C++ API, inherited channel descriptor), [cudaUnbindTexture](#) (C++ API), [cudaGetTextureAlignmentOffset](#) (C++ API)

### 3.14.2.4 `template<class T , int dim, enum cudaTextureReadMode readMode> cudaError_t cudaBindTextureToArray (const struct texture< T, dim, readMode > & tex, const struct cudaArray * array)`

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.

**Parameters:**

*tex* - Texture to bind  
*array* - Memory array on device

**Returns:**

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

**Note:**

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), [cudaBindTextureToArray](#) (C API), [cudaBindTextureToArray](#) (C++ API), [cudaUnbindTexture](#) (C++ API), [cudaGetTextureAlignmentOffset](#) (C++ API)

### 3.14.2.5 `template<class T , int dim, enum cudaTextureReadMode readMode> cudaError_t cudaBindTextureToArray (const struct texture< T, dim, readMode > & tex, const struct cudaArray * array, const struct cudaChannelFormatDesc & desc)`

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.

**Parameters:**

*tex* - Texture to bind  
*array* - Memory array on device  
*desc* - Channel format

**Returns:**

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

**Note:**

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), [cudaBindTextureToArray](#) (C API), [cudaBindTextureToArray](#) (C++ API, inherited channel descriptor), [cudaUnbindTexture](#) (C++ API), [cudaGetTextureAlignmentOffset](#) (C++ API)

**3.14.2.6 `template<class T > cudaChannelFormatDesc cudaCreateChannelDesc (void)`**

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](#).

**Returns:**

Channel descriptor with format `f`

**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)

**3.14.2.7 `template<class T > cudaError_t cudaGetSymbolAddress (void ** devPtr, const T & symbol)`**

Returns in `*devPtr` the address of symbol `symbol` on the device. `symbol` can either be a variable that resides in global memory space, or it can be a character string, naming a variable that resides in global memory space. If `symbol` cannot be found, or if `symbol` is not declared in the global memory space, `*devPtr` is unchanged and the error [cudaErrorInvalidSymbol](#) is returned.

**Parameters:**

*devPtr* - Return device pointer associated with symbol

*symbol* - Global variable or string symbol to search for

**Returns:**

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

**Note:**

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

**See also:**

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

### 3.14.2.8 `template<class T > cudaError_t cudaGetSymbolSize (size_t * size, const T & symbol)`

Returns in `*size` the size of symbol `symbol`. `symbol` can either be a variable that resides in global or constant memory space, or it can be a character string, naming 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.

#### Parameters:

- `size` - Size of object associated with symbol
- `symbol` - Global variable or string symbol to find size of

#### Returns:

`cudaSuccess`, `cudaErrorInvalidSymbol`

#### Note:

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

#### See also:

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

### 3.14.2.9 `template<class T , int dim, enum cudaTextureReadMode readMode> cudaError_t cudaGetTextureAlignmentOffset (size_t * offset, const struct texture< T, dim, readMode > & tex)`

Returns in `*offset` the offset that was returned when texture reference `tex` was bound.

#### Parameters:

- `offset` - Offset of texture reference in bytes
- `tex` - Texture to get offset of

#### Returns:

`cudaSuccess`, `cudaErrorInvalidTexture`, `cudaErrorInvalidTextureBinding`

#### Note:

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 \(C++ API\)](#), [cudaBindTextureToArray \(C++ API\)](#), [cudaBindTextureToArray \(C++ API, inherited channel descriptor\)](#), [cudaUnbindTexture \(C++ API\)](#), [cudaGetTextureAlignmentOffset \(C API\)](#)

### 3.14.2.10 `template<class T > cudaError_t cudaLaunch (T * entry)`

Launches the function `entry` on the device. `entry` can either be a function that executes on the device, or it can be a character string, naming a function that executes on the device. `entry` 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.

**Parameters:**

*entry* - Device function pointer or char string naming device function to execute

**Returns:**

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

**Note:**

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

**See also:**

[cudaConfigureCall](#), [cudaSetupArgument \(C++ API\)](#), [cudaLaunch \(C API\)](#)

**3.14.2.11 `template<class T > cudaError_t cudaSetupArgument (T arg, size_t offset)`**

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\(\)](#).

**Parameters:**

*arg* - Argument to push for a kernel launch

*offset* - Offset in argument stack to push new *arg*

**Returns:**

[cudaSuccess](#)

**Note:**

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

**See also:**

[cudaLaunch \(C++ API\)](#), [cudaSetupArgument \(C API\)](#) [cudaConfigureCall](#)

**3.14.2.12 `template<class T , int dim, enum cudaTextureReadMode readMode> cudaError_t cudaUnbindTexture (const struct texture< T, dim, readMode > & tex)`**

Unbinds the texture bound to *tex*.

**Parameters:**

*tex* - Texture to unbind

**Returns:**

[cudaSuccess](#)

**Note:**

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), [cudaBindTextureToArray](#) (C++ API), [cudaBindTextureToArray](#) (C++ API, inherited channel descriptor), [cudaUnbindTexture](#) (C API), [cudaGetTextureAlignmentOffset](#) (C++ API)

## 3.15 Data types used by CUDA Runtime

### Data Structures

- struct [cudaChannelFormatDesc](#)
- struct [cudaDeviceProp](#)
- struct [cudaExtent](#)
- struct [cudaFuncAttributes](#)
- struct [cudaMemcpy3DParms](#)
- struct [cudaPitchedPtr](#)
- struct [cudaPos](#)

### Data types used by CUDA Runtime

Data types used by CUDA Runtime

#### Author:

NVIDIA Corporation

- enum [cudaChannelFormatKind](#) {  
    [cudaChannelFormatKindSigned](#),  
    [cudaChannelFormatKindUnsigned](#),  
    [cudaChannelFormatKindFloat](#),  
    [cudaChannelFormatKindNone](#) }
- enum [cudaComputeMode](#) {  
    [cudaComputeModeDefault](#),  
    [cudaComputeModeExclusive](#),  
    [cudaComputeModeProhibited](#) }
- enum [cudaError](#) {  
    [cudaSuccess](#),  
    [cudaErrorMissingConfiguration](#),  
    [cudaErrorMemoryAllocation](#),  
    [cudaErrorInitializationError](#),  
    [cudaErrorLaunchFailure](#),  
    [cudaErrorPriorLaunchFailure](#),  
    [cudaErrorLaunchTimeout](#),  
    [cudaErrorLaunchOutOfResources](#),  
    [cudaErrorInvalidDeviceFunction](#),  
    [cudaErrorInvalidConfiguration](#),  
    [cudaErrorInvalidDevice](#),  
    [cudaErrorInvalidValue](#),  
    [cudaErrorInvalidPitchValue](#),  
    [cudaErrorInvalidSymbol](#),  
    [cudaErrorMapBufferObjectFailed](#),

```

cudaErrorUnmapBufferObjectFailed,
cudaErrorInvalidHostPointer,
cudaErrorInvalidDevicePointer,
cudaErrorInvalidTexture,
cudaErrorInvalidTextureBinding,
cudaErrorInvalidChannelDescriptor,
cudaErrorInvalidMemcpyDirection,
cudaErrorAddressOfConstant,
cudaErrorTextureFetchFailed,
cudaErrorTextureNotBound,
cudaErrorSynchronizationError,
cudaErrorInvalidFilterSetting,
cudaErrorInvalidNormSetting,
cudaErrorMixedDeviceExecution,
cudaErrorCudartUnloading,
cudaErrorUnknown,
cudaErrorNotYetImplemented,
cudaErrorMemoryValueTooLarge,
cudaErrorInvalidResourceHandle,
cudaErrorNotReady,
cudaErrorInsufficientDriver,
cudaErrorSetOnActiveProcess,
cudaErrorNoDevice,
cudaErrorStartupFailure,
cudaErrorApiFailureBase }
• enum cudaMemcpyKind {
  cudaMemcpyHostToHost,
  cudaMemcpyHostToDevice,
  cudaMemcpyDeviceToHost,
  cudaMemcpyDeviceToDevice }
• typedef enum cudaError cudaError_t
• typedef int cudaEvent_t
• typedef int cudaStream_t
• #define cudaDeviceBlockingSync
  Device flag - Use blocking synchronization.

• #define cudaDeviceMapHost
  Device flag - Support mapped pinned allocations.

• #define cudaDeviceMask
  Device flags mask.

• #define cudaDevicePropDontCare
  Empty device properties.

```



- #define `cudaDeviceScheduleAuto`  
*Device flag - Automatic scheduling.*
- #define `cudaDeviceScheduleSpin`  
*Device flag - Spin default scheduling.*
- #define `cudaDeviceScheduleYield`  
*Device flag - Yield default scheduling.*
- #define `cudaEventBlockingSync`  
*Event uses blocking synchronization.*
- #define `cudaEventDefault`  
*Default event flag.*
- #define `cudaHostAllocDefault`  
*Default page-locked allocation flag.*
- #define `cudaHostAllocMapped`  
*Map allocation into device space.*
- #define `cudaHostAllocPortable`  
*Pinned memory accessible by all CUDA contexts.*
- #define `cudaHostAllocWriteCombined`  
*Write-combined memory.*

### 3.15.1 Typedef Documentation

#### 3.15.1.1 typedef enum `cudaError` `cudaError_t`

CUDA Error types

#### 3.15.1.2 typedef int `cudaEvent_t`

CUDA event types

#### 3.15.1.3 typedef int `cudaStream_t`

CUDA stream

### 3.15.2 Enumeration Type Documentation

#### 3.15.2.1 enum `cudaChannelFormatKind`

Channel format kind

**Enumerator:**

- cudaChannelFormatKindSigned* Signed channel format.
- cudaChannelFormatKindUnsigned* Unsigned channel format.
- cudaChannelFormatKindFloat* Float channel format.
- cudaChannelFormatKindNone* No channel format.

**3.15.2.2 enum cudaComputeMode**

CUDA device compute modes

**Enumerator:**

- cudaComputeModeDefault* Default compute mode (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).

**3.15.2.3 enum cudaError**

CUDA error types

**Enumerator:**

- cudaSuccess* No errors.
- cudaErrorMissingConfiguration* Missing configuration error.
- cudaErrorMemoryAllocation* Memory allocation error.
- cudaErrorInitializationError* Initialization error.
- cudaErrorLaunchFailure* Launch failure.
- cudaErrorPriorLaunchFailure* Prior launch failure.
- cudaErrorLaunchTimeout* Launch timeout error.
- cudaErrorLaunchOutOfResources* Launch out of resources error.
- cudaErrorInvalidDeviceFunction* Invalid device function.
- cudaErrorInvalidConfiguration* Invalid configuration.
- cudaErrorInvalidDevice* Invalid device.
- cudaErrorInvalidValue* Invalid value.
- cudaErrorInvalidPitchValue* Invalid pitch value.
- cudaErrorInvalidSymbol* Invalid symbol.
- cudaErrorMapBufferObjectFailed* Map buffer object failed.
- cudaErrorUnmapBufferObjectFailed* Unmap buffer object failed.
- cudaErrorInvalidHostPointer* Invalid host pointer.
- cudaErrorInvalidDevicePointer* Invalid device pointer.
- cudaErrorInvalidTexture* Invalid texture.
- cudaErrorInvalidTextureBinding* Invalid texture binding.

*cudaErrorInvalidChannelDescriptor* Invalid channel descriptor.  
*cudaErrorInvalidMemcpyDirection* Invalid memcpy direction.  
*cudaErrorAddressOfConstant* Address of constant error.  
*cudaErrorTextureFetchFailed* Texture fetch failed.  
*cudaErrorTextureNotBound* Texture not bound error.  
*cudaErrorSynchronizationError* Synchronization error.  
*cudaErrorInvalidFilterSetting* Invalid filter setting.  
*cudaErrorInvalidNormSetting* Invalid norm setting.  
*cudaErrorMixedDeviceExecution* Mixed device execution.  
*cudaErrorCudartUnloading* CUDA runtime unloading.  
*cudaErrorUnknown* Unknown error condition.  
*cudaErrorNotYetImplemented* Function not yet implemented.  
*cudaErrorMemoryValueTooLarge* Memory value too large.  
*cudaErrorInvalidResourceHandle* Invalid resource handle.  
*cudaErrorNotReady* Not ready error.  
*cudaErrorInsufficientDriver* CUDA runtime is newer than driver.  
*cudaErrorSetOnActiveProcess* Set on active process error.  
*cudaErrorNoDevice* No available CUDA device.  
*cudaErrorStartupFailure* Startup failure.  
*cudaErrorApiFailureBase* API failure base.

#### 3.15.2.4 enum cudaMemcpyKind

CUDA memory copy types

Enumerator:

*cudaMemcpyHostToHost* Host -> Host.  
*cudaMemcpyHostToDevice* Host -> Device.  
*cudaMemcpyDeviceToHost* Device -> Host.  
*cudaMemcpyDeviceToDevice* Device -> Device.

## 3.16 CUDA Driver API

### Modules

- [Initialization](#)
- [Device Management](#)
- [Version Management](#)
- [Context Management](#)
- [Module Management](#)
- [Stream Management](#)
- [Event Management](#)
- [Execution Control](#)
- [Memory Management](#)
- [Texture Reference Management](#)
- [OpenGL Interoperability](#)
- [Direct3D 9 Interoperability](#)
- [Direct3D 10 Interoperability](#)
- [Data types used by CUDA driver](#)

### 3.16.1 Detailed Description

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

## 3.17 Initialization

### Functions

- [CUresult cuInit](#) (unsigned int *Flags*)  
*Initialize the CUDA driver API.*

### 3.17.1 Detailed Description

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

### 3.17.2 Function Documentation

#### 3.17.2.1 CUresult cuInit (unsigned int *Flags*)

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

#### Parameters:

*Flags* - Initialization flag for CUDA.

#### Returns:

`CUDA_SUCCESS`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_INVALID_DEVICE`

#### Note:

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

## 3.18 Device Management

### Functions

- [CUresult cuDeviceComputeCapability](#) (int \*major, int \*minor, [CUdevice](#) dev)  
*Returns the compute capability of the device.*
- [CUresult cuDeviceGet](#) ([CUdevice](#) \*device, int ordinal)  
*Returns a handle to a compute device.*
- [CUresult cuDeviceGetAttribute](#) (int \*pi, [CUdevice\\_attribute](#) attrib, [CUdevice](#) dev)  
*Returns information about the device.*
- [CUresult cuDeviceGetCount](#) (int \*count)  
*Returns the number of compute-capable devices.*
- [CUresult cuDeviceGetName](#) (char \*name, int len, [CUdevice](#) dev)  
*Returns an identifier string for the device.*
- [CUresult cuDeviceGetProperties](#) ([CUdevprop](#) \*prop, [CUdevice](#) dev)  
*Returns properties for a selected device.*
- [CUresult cuDeviceTotalMem](#) (unsigned int \*bytes, [CUdevice](#) dev)  
*Returns the total amount of memory on the device.*

### 3.18.1 Detailed Description

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

### 3.18.2 Function Documentation

#### 3.18.2.1 [CUresult cuDeviceComputeCapability](#) (int \*major, int \*minor, [CUdevice](#) dev)

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

#### Parameters:

- major* - Major revision number
- minor* - Minor revision number
- dev* - Device handle

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_INVALID\\_DEVICE](#)

#### Note:

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

See also:

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

### 3.18.2.2 CUresult cuDeviceGet (CUdevice \* device, int ordinal)

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

Parameters:

*device* - Returned device handle

*ordinal* - Device number to get handle for

Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_INVALID\\_DEVICE](#)

Note:

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

See also:

[cuDeviceComputeCapability](#), [cuDeviceGetAttribute](#), [cuDeviceGetCount](#), [cuDeviceGetName](#), [cuDeviceGetProperties](#), [cuDeviceTotalMem](#)

### 3.18.2.3 CUresult cuDeviceGetAttribute (int \* pi, CUdevice\_attribute attrib, CUdevice dev)

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

- [CU\\_DEVICE\\_ATTRIBUTE\\_MAX\\_THREADS\\_PER\\_BLOCK](#): Maximum number of threads per block;
- [CU\\_DEVICE\\_ATTRIBUTE\\_MAX\\_BLOCK\\_DIM\\_X](#): Maximum x-dimension of a block;
- [CU\\_DEVICE\\_ATTRIBUTE\\_MAX\\_BLOCK\\_DIM\\_Y](#): Maximum y-dimension of a block;
- [CU\\_DEVICE\\_ATTRIBUTE\\_MAX\\_BLOCK\\_DIM\\_Z](#): Maximum z-dimension of a block;
- [CU\\_DEVICE\\_ATTRIBUTE\\_MAX\\_GRID\\_DIM\\_X](#): Maximum x-dimension of a grid;
- [CU\\_DEVICE\\_ATTRIBUTE\\_MAX\\_GRID\\_DIM\\_Y](#): Maximum y-dimension of a grid;
- [CU\\_DEVICE\\_ATTRIBUTE\\_MAX\\_GRID\\_DIM\\_Z](#): Maximum z-dimension of a grid;
- [CU\\_DEVICE\\_ATTRIBUTE\\_MAX\\_SHARED\\_MEMORY\\_PER\\_BLOCK](#): Maximum amount of shared memory available to a thread block in bytes; this amount is shared by all thread blocks simultaneously resident on a multiprocessor;
- [CU\\_DEVICE\\_ATTRIBUTE\\_TOTAL\\_CONSTANT\\_MEMORY](#): Memory available on device for `__constant__` variables in a CUDA C kernel in bytes;
- [CU\\_DEVICE\\_ATTRIBUTE\\_WARP\\_SIZE](#): Warp size in threads;
- [CU\\_DEVICE\\_ATTRIBUTE\\_MAX\\_PITCH](#): Maximum pitch in bytes allowed by the memory copy functions that involve memory regions allocated through [cuMemAllocPitch\(\)](#);

- [CU\\_DEVICE\\_ATTRIBUTE\\_MAX\\_REGISTERS\\_PER\\_BLOCK](#): Maximum number of 32-bit registers available to a thread block; this number is shared by all thread blocks simultaneously resident on a multiprocessor;
- [CU\\_DEVICE\\_ATTRIBUTE\\_CLOCK\\_RATE](#): Peak clock frequency in kilohertz;
- [CU\\_DEVICE\\_ATTRIBUTE\\_TEXTURE\\_ALIGNMENT](#): Alignment requirement; texture base addresses aligned to textureAlign bytes do not need an offset applied to texture fetches;
- [CU\\_DEVICE\\_ATTRIBUTE\\_GPU\\_OVERLAP](#): 1 if the device can concurrently copy memory between host and device while executing a kernel, or 0 if not;
- [CU\\_DEVICE\\_ATTRIBUTE\\_MULTIPROCESSOR\\_COUNT](#): Number of multiprocessors on the device;
- [CU\\_DEVICE\\_ATTRIBUTE\\_KERNEL\\_EXEC\\_TIMEOUT](#): 1 if there is a run time limit for kernels executed on the device, or 0 if not;
- [CU\\_DEVICE\\_ATTRIBUTE\\_INTEGRATED](#): 1 if the device is integrated with the memory subsystem, or 0 if not;
- [CU\\_DEVICE\\_ATTRIBUTE\\_CAN\\_MAP\\_HOST\\_MEMORY](#): 1 if the device can map host memory into the CUDA address space, or 0 if not;
- [CU\\_DEVICE\\_ATTRIBUTE\\_COMPUTE\\_MODE](#): Compute mode that device is currently in. Available modes are as follows:
  - [CU\\_COMPUTEMODE\\_DEFAULT](#): Default mode - Device is not restricted and can have multiple CUDA contexts present at a single time.
  - [CU\\_COMPUTEMODE\\_EXCLUSIVE](#): Compute-exclusive mode - Device can have only one CUDA context present on it at a time.
  - [CU\\_COMPUTEMODE\\_PROHIBITED](#): Compute-prohibited mode - Device is prohibited from creating new CUDA contexts.

**Parameters:**

*pi* - Returned device attribute value

*attrib* - Device attribute to query

*dev* - Device handle

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_INVALID\\_DEVICE](#)

**Note:**

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

**See also:**

[cuDeviceComputeCapability](#), [cuDeviceGetCount](#), [cuDeviceGetName](#), [cuDeviceGet](#), [cuDeviceGetProperties](#), [cuDeviceTotalMem](#)



#### 3.18.2.4 CUresult cuDeviceGetCount (int \* count)

Returns in \*count the number of devices with compute capability greater than or equal to 1.0 that are available for execution. If there is no such device, cuDeviceGetCount() returns 0.

##### Parameters:

*count* - Returned number of compute-capable devices

##### Returns:

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

##### Note:

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

##### See also:

cuDeviceComputeCapability, cuDeviceGetAttribute, cuDeviceGetName, cuDeviceGet, cuDeviceGetProperties, cuDeviceTotalMem

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

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

##### Parameters:

*name* - Returned identifier string for the device

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

*dev* - Device to get identifier string for

##### Returns:

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_INVALID\_DEVICE

##### Note:

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

##### See also:

cuDeviceComputeCapability, cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGet, cuDeviceGetProperties, cuDeviceTotalMem

#### 3.18.2.6 CUresult cuDeviceGetProperties (CUdevprop \* prop, CUdevice dev)

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

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

where:

- `maxThreadsPerBlock` is the maximum number of threads per block;
- `maxThreadsDim[3]` is the maximum sizes of each dimension of a block;
- `maxGridSize[3]` is the maximum sizes of each dimension of a grid;
- `sharedMemPerBlock` is the total amount of shared memory available per block in bytes;
- `totalConstantMemory` is the total amount of constant memory available on the device in bytes;
- `SIMDWidth` is the warp size;
- `memPitch` is the maximum pitch allowed by the memory copy functions that involve memory regions allocated through [cuMemAllocPitch\(\)](#);
- `regsPerBlock` is the total number of registers available per block;
- `clockRate` is the clock frequency in kilohertz;
- `textureAlign` is the alignment requirement; texture base addresses that are aligned to `textureAlign` bytes do not need an offset applied to texture fetches.

#### Parameters:

*prop* - Returned properties of device

*dev* - Device to get properties for

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_INVALID\\_DEVICE](#)

#### Note:

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

#### See also:

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

### 3.18.2.7 CUresult cuDeviceTotalMem (unsigned int \* *bytes*, CUdevice *dev*)

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

**Parameters:**

*bytes* - Returned memory available on device in bytes

*dev* - Device handle

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_INVALID\\_DEVICE](#)

**Note:**

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

**See also:**

[cuDeviceComputeCapability](#), [cuDeviceGetAttribute](#), [cuDeviceGetCount](#), [cuDeviceGetName](#), [cuDeviceGet](#), [cuDeviceGetProperties](#),

## 3.19 Version Management

### Functions

- [CUresult cuDriverGetVersion](#) (int \*driverVersion)

*Returns the CUDA driver version.*

### 3.19.1 Detailed Description

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

### 3.19.2 Function Documentation

#### 3.19.2.1 CUresult cuDriverGetVersion (int \* driverVersion)

Returns in \*driverVersion the version number of the installed CUDA driver. This function automatically returns [CUDA\\_ERROR\\_INVALID\\_VALUE](#) if the driverVersion argument is NULL.

#### Parameters:

*driverVersion* - Returns the CUDA driver version

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

#### Note:

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

## 3.20 Context Management

### Functions

- **CUresult cuCtxAttach** (CUcontext \*pCtx, unsigned int Flags)  
*Increment a context's usage-count.*
- **CUresult cuCtxCreate** (CUcontext \*pCtx, unsigned int Flags, CUdevice dev)  
*Create a CUDA context.*
- **CUresult cuCtxDestroy** (CUcontext ctx)  
*Destroy the current context or a floating CUDA context.*
- **CUresult cuCtxDetach** (CUcontext pCtx)  
*Decrement a context's usage-count.*
- **CUresult cuCtxGetDevice** (CUdevice \*device)  
*Returns the device ID for the current context.*
- **CUresult cuCtxPopCurrent** (CUcontext \*pCtx)  
*Pops the current CUDA context from the current CPU thread.*
- **CUresult cuCtxPushCurrent** (CUcontext NewCtx)  
*Pushes a floating context on the current CPU thread.*
- **CUresult cuCtxSynchronize** (void)  
*Block for a context's tasks to complete.*

### 3.20.1 Detailed Description

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

### 3.20.2 Function Documentation

#### 3.20.2.1 CUresult cuCtxAttach (CUcontext \*pCtx, unsigned int Flags)

Increments the usage count of the context and passes back a context handle in \*pCtx that must be passed to **cuCtxDetach()** when the application is done with the context. **cuCtxAttach()** fails if there is no context current to the thread.

Currently, the `Flags` parameter must be 0.

#### Parameters:

- pCtx** - Returned context handle of the current context
- Flags** - Context attach flags (must be 0)

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**Note:**

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

**See also:**

[cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxDetach](#), [cuCtxGetDevice](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSynchronize](#)

**3.20.2.2 CUresult cuCtxCreate (CUcontext \* *pCtx*, unsigned int *Flags*, CUdevice *dev*)**

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

The two LSBs of the `Flags` parameter can be used to control how the OS thread, which owns the CUDA context at the time of an API call, interacts with the OS scheduler when waiting for results from the GPU.

- [CU\\_CTX\\_SCHED\\_AUTO](#): The default value if the `Flags` parameter is zero, uses a heuristic based on the number of active CUDA contexts in the process  $C$  and the number of logical processors in the system  $P$ . If  $C > P$ , then CUDA will yield to other OS threads when waiting for the GPU, otherwise CUDA will not yield while waiting for results and actively spin on the processor.
- [CU\\_CTX\\_SCHED\\_SPIN](#): Instruct CUDA to actively spin when waiting for results from the GPU. This can decrease latency when waiting for the GPU, but may lower the performance of CPU threads if they are performing work in parallel with the CUDA thread.
- [CU\\_CTX\\_SCHED\\_YIELD](#): Instruct CUDA to yield its thread when waiting for results from the GPU. This can increase latency when waiting for the GPU, but can increase the performance of CPU threads performing work in parallel with the GPU.
- [CU\\_CTX\\_BLOCKING\\_SYNC](#): Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the GPU to finish work.
- [CU\\_CTX\\_MAP\\_HOST](#): Instruct CUDA to support mapped pinned allocations. This flag must be set in order to allocate pinned host memory that is accessible to the GPU.

**Note to Linux users:**

Context creation will fail with [CUDA\\_ERROR\\_UNKNOWN](#) if the compute mode of the device is [CU\\_COMPUTEMODE\\_PROHIBITED](#). Similarly, context creation will also fail with [CUDA\\_ERROR\\_UNKNOWN](#) if the compute mode for the device is set to [CU\\_COMPUTEMODE\\_EXCLUSIVE](#) and there is already an active context on the device. The function `cuDeviceGetAttribute()` can be used with [CU\\_DEVICE\\_ATTRIBUTE\\_COMPUTE\\_MODE](#) to determine the compute mode of the device. The `nvidia-smi` tool can be used to set the compute mode for devices. Documentation for `nvidia-smi` can be obtained by passing a `-h` option to it.

**Parameters:**

*pCtx* - Returned context handle of the new context

*Flags* - Context creation flags

*dev* - Device to create context on

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_DEVICE, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_OUT\_OF\_MEMORY, CUDA\_ERROR\_UNKNOWN

**Note:**

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

**See also:**

[cuCtxAttach](#), [cuCtxDestroy](#), [cuCtxDetach](#), [cuCtxGetDevice](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSynchronize](#)

### 3.20.2.3 CUresult cuCtxDestroy (CUcontext *ctx*)

Destroys the CUDA context specified by `ctx`. If the context usage count is not equal to 1, or the context is current to any CPU thread other than the current one, this function fails. Floating contexts (detached from a CPU thread via [cuCtxPopCurrent\(\)](#)) may be destroyed by this function.

**Parameters:**

*ctx* - Context to destroy

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

[cuCtxAttach](#), [cuCtxCreate](#), [cuCtxDetach](#), [cuCtxGetDevice](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSynchronize](#)

### 3.20.2.4 CUresult cuCtxDetach (CUcontext *pCtx*)

Decrements the usage count of the context `pCtx`, and destroys the context if the usage count goes to 0. The context must be a handle that was passed back by [cuCtxCreate\(\)](#) or [cuCtxAttach\(\)](#), and must be current to the calling thread.

**Parameters:**

*pCtx* - Context to destroy

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT

**Note:**

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

**See also:**

[cuCtxAttach](#), [cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxGetDevice](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSynchronize](#)

**3.20.2.5 CUresult cuCtxGetDevice (CUdevice \* device)**

Returns in *\*device* the ordinal of the current context's device.

**Parameters:**

*device* - Returned device ID for the current context

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#),

**Note:**

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

**See also:**

[cuCtxAttach](#), [cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxDetach](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSynchronize](#)

**3.20.2.6 CUresult cuCtxPopCurrent (CUcontext \* pCtx)**

Pops the current CUDA context from the CPU thread. The CUDA context must have a usage count of 1. CUDA contexts have a usage count of 1 upon creation; the usage count may be incremented with [cuCtxAttach\(\)](#) and decremented with [cuCtxDetach\(\)](#).

If successful, [cuCtxPopCurrent\(\)](#) passes back the new context handle in *\*pCtx*. The old context may then be made current to a different CPU thread by calling [cuCtxPushCurrent\(\)](#).

Floating contexts may be destroyed by calling [cuCtxDestroy\(\)](#).

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

**Parameters:**

*pCtx* - Returned new context handle

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#)

**Note:**

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

**See also:**

[cuCtxAttach](#), [cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxDetach](#), [cuCtxGetDevice](#), [cuCtxPushCurrent](#), [cuCtxSynchronize](#)



### 3.20.2.7 CUresult cuCtxPushCurrent (CUcontext NewCtx)

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

The previous current context may be made current again by calling `cuCtxDestroy()` or `cuCtxPopCurrent()`.

The context must be "floating," i.e. not attached to any thread. Contexts are made to float by calling `cuCtxPopCurrent()`.

#### Parameters:

`NewCtx` - Floating context to attach

#### Returns:

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

#### Note:

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

#### See also:

`cuCtxAttach`, `cuCtxCreate`, `cuCtxDestroy`, `cuCtxDetach`, `cuCtxGetDevice`, `cuCtxPopCurrent`, `cuCtxSynchronize`

### 3.20.2.8 CUresult cuCtxSynchronize (void)

Blocks until the device has completed all preceding requested tasks. `cuCtxSynchronize()` returns an error if one of the preceding tasks failed.

#### Returns:

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

#### Note:

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

#### See also:

`cuCtxAttach`, `cuCtxCreate`, `cuCtxDestroy`, `cuCtxDetach`, `cuCtxGetDevice`, `cuCtxPopCurrent`, `cuCtxSynchronize`

## 3.21 Module Management

### Functions

- **CUresult cuModuleGetFunction** (CUfunction \*func, CUmodule mod, const char \*name)  
*Returns a function handle.*
- **CUresult cuModuleGetGlobal** (CUdeviceptr \*ret\_dptr, unsigned int \*ret\_bytes, CUmodule mod, const char \*name)  
*Returns a global pointer from a module.*
- **CUresult cuModuleGetTexRef** (CUTexref \*ppTexRef, CUmodule mod, const char \*name)  
*Returns a handle to a texture-reference.*
- **CUresult cuModuleLoad** (CUmodule \*phMod, const char \*fname)  
*Loads a compute module.*
- **CUresult cuModuleLoadData** (CUmodule \*phMod, const void \*p)  
*Load a module's data.*
- **CUresult cuModuleLoadDataEx** (CUmodule \*phMod, const void \*p, unsigned int numOptions, CUjit\_option \*options, void \*\*optionValues)  
*Load a module's data with options.*
- **CUresult cuModuleLoadFatBinary** (CUmodule \*phMod, const void \*vfatCubin)  
*Load a module's data.*
- **CUresult cuModuleUnload** (CUmodule mod)  
*Unloads a module.*

### 3.21.1 Detailed Description

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

### 3.21.2 Function Documentation

#### 3.21.2.1 CUresult cuModuleGetFunction (CUfunction \*func, CUmodule mod, const char \* name)

Returns in \*func the handle of the function of name name located in module mod. If no function of that name exists, cuModuleGetFunction() returns [CUDA\\_ERROR\\_NOT\\_FOUND](#).

#### Parameters:

- func* - Returned function handle
- mod* - Module to retrieve function from
- name* - Name of function to retrieve

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_NOT\_FOUND

**Note:**

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

**See also:**

[cuModuleGetGlobal](#), [cuModuleGetTexRef](#), [cuModuleLoad](#), [cuModuleLoadData](#), [cuModuleLoadDataEx](#), [cuModuleLoadFatBinary](#), [cuModuleUnload](#)

**3.21.2.2 CUresult cuModuleGetGlobal (CUdeviceptr \* *ret\_dptr*, unsigned int \* *ret\_bytes*, CUmodule *mod*, const char \* *name*)**

Returns in *ret\_dptr* and *ret\_bytes* the base pointer and size of the global of name *name* located in module *mod*. If no variable of that name exists, [cuModuleGetGlobal\(\)](#) returns [CUDA\\_ERROR\\_NOT\\_FOUND](#). Both parameters *ret\_dptr* and *ret\_bytes* are optional. If one of them is NULL, it is ignored.

**Parameters:**

*ret\_dptr* - Returned global device pointer

*ret\_bytes* - Returned global size in bytes

*mod* - Module to retrieve function from

*name* - Name of global to retrieve

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_NOT\_FOUND

**Note:**

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

**See also:**

[cuModuleGetFunction](#), [cuModuleGetTexRef](#), [cuModuleLoad](#), [cuModuleLoadData](#), [cuModuleLoadDataEx](#), [cuModuleLoadFatBinary](#), [cuModuleUnload](#)

**3.21.2.3 CUresult cuModuleGetTexRef (CUTexref \* *ppTexRef*, CUmodule *mod*, const char \* *name*)**

Returns in *ppTexRef* the handle of the texture reference of name *name* in the module *mod*. If no texture reference of that name exists, [cuModuleGetTexRef\(\)](#) returns [CUDA\\_ERROR\\_NOT\\_FOUND](#). This texture reference handle should not be destroyed, since it will be destroyed when the module is unloaded.

**Parameters:**

*ppTexRef* - Returned global device pointer

*mod* - Module to retrieve texture-reference from

*name* - Name of texture-reference to retrieve

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_NOT\_FOUND

**Note:**

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

**See also:**

[cuModuleGetFunction](#), [cuModuleGetGlobal](#), [cuModuleLoad](#), [cuModuleLoadData](#), [cuModuleLoadDataEx](#), [cuModuleLoadFatBinary](#), [cuModuleUnload](#)

**3.21.2.4 CUresult cuModuleLoad (CUmodule \* *phMod*, const char \* *fname*)**

Takes a filename *fname* and loads the corresponding module *phMod* into the current context. The CUDA driver API does not attempt to lazily allocate the resources needed by a module; if the memory for functions and data (constant and global) needed by the module cannot be allocated, [cuModuleLoad\(\)](#) fails. The file should be a *cubin* file as output by **nvcc** or a *PTX* file, either as output by **nvcc** or handwritten.

**Parameters:**

*phMod* - Returned module  
*fname* - Filename of module to load

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_NOT\_FOUND, CUDA\_ERROR\_OUT\_OF\_MEMORY, CUDA\_ERROR\_FILE\_NOT\_FOUND

**Note:**

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

**See also:**

[cuModuleGetFunction](#), [cuModuleGetGlobal](#), [cuModuleGetTexRef](#), [cuModuleLoadData](#), [cuModuleLoadDataEx](#), [cuModuleLoadFatBinary](#), [cuModuleUnload](#)

**3.21.2.5 CUresult cuModuleLoadData (CUmodule \* *phMod*, const void \* *p*)**

Takes a pointer *p* and loads the corresponding module *phMod* into the current context. The pointer may be obtained by mapping a *cubin* or *PTX* file, passing a *cubin* or *PTX* file as a text string, or incorporating a *cubin* object into the executable resources and using operating system calls such as Windows `FindResource()` to obtain the pointer.

**Parameters:**

*phMod* - Returned module  
*p* - Module data to load

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_OUT\_OF\_MEMORY

**Note:**

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

**See also:**

[cuModuleGetFunction](#), [cuModuleGetGlobal](#), [cuModuleGetTexRef](#), [cuModuleLoad](#), [cuModuleLoadDataEx](#), [cuModuleLoadFatBinary](#), [cuModuleUnload](#)

**3.21.2.6 CUresult cuModuleLoadDataEx (CUmodule \* *phMod*, const void \* *p*, unsigned int *numOptions*, CUjit\_option \* *options*, void \*\* *optionValues*)**

Takes a pointer *p* and loads the corresponding module *phMod* into the current context. The pointer may be obtained by mapping a *cubin* or *PTX* file, passing a *cubin* or *PTX* file as a text string, or incorporating a *cubin* object into the executable resources and using operating system calls such as Windows `FindResource()` to obtain the pointer. Options are passed as an array via *options* and any corresponding parameters are passed in *optionValues*. The number of total options is supplied via *numOptions*. Any outputs will be returned via *optionValues*. Supported options are:

- [CU\\_JIT\\_MAX\\_REGISTERS](#): input specifies the maximum number of registers per thread;
- [CU\\_JIT\\_THREADS\\_PER\\_BLOCK](#): input specifies number of threads per block to target compilation for; output returns the number of threads the compiler actually targeted;
- [CU\\_JIT\\_WALL\\_TIME](#): output returns the float value of wall clock time, in milliseconds, spent compiling the *PTX* code;
- [CU\\_JIT\\_INFO\\_LOG\\_BUFFER](#): input is a pointer to a buffer in which to print any informational log messages from *PTX* assembly;
- [CU\\_JIT\\_INFO\\_LOG\\_BUFFER\\_SIZE\\_BYTES](#): input is the size in bytes of the buffer; output is the number of bytes filled with messages;
- [CU\\_JIT\\_ERROR\\_LOG\\_BUFFER](#): input is a pointer to a buffer in which to print any error log messages from *PTX* assembly;
- [CU\\_JIT\\_ERROR\\_LOG\\_BUFFER\\_SIZE\\_BYTES](#): input is the size in bytes of the buffer; output is the number of bytes filled with messages;
- [CU\\_JIT\\_OPTIMIZATION\\_LEVEL](#): input is the level of optimization to apply to generated code (0 - 4), with 4 being the default and highest level;
- [CU\\_JIT\\_TARGET\\_FROM\\_CUCONTEXT](#): causes compilation target to be determined based on current attached context (default);
- [CU\\_JIT\\_TARGET](#): input is the compilation target based on supplied [CUjit\\_target\\_enum](#); possible values are:
  - [CU\\_TARGET\\_COMPUTE\\_10](#)
  - [CU\\_TARGET\\_COMPUTE\\_11](#)
  - [CU\\_TARGET\\_COMPUTE\\_12](#)
  - [CU\\_TARGET\\_COMPUTE\\_13](#)

**Parameters:**

*phMod* - Returned module

*p* - Module data to load

*numOptions* - Number of options

*options* - Options for JIT

*optionValues* - Option values for JIT

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_OUT\\_OF\\_MEMORY](#), [CUDA\\_ERROR\\_NO\\_BINARY\\_FOR\\_GPU](#)

**Note:**

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

**See also:**

[cuModuleGetFunction](#), [cuModuleGetGlobal](#), [cuModuleGetTexRef](#), [cuModuleLoad](#), [cuModuleLoadData](#), [cuModuleLoadFatBinary](#), [cuModuleUnload](#)

### 3.21.2.7 CUresult cuModuleLoadFatBinary (CUmodule \* *phMod*, const void \* *vfatCubin*)

Takes a pointer *vfatCubin* and loads the corresponding module *phMod* into the current context. The pointer represents a *fat binary* object, which is a collection of different *cubin* files, all representing the same device code, but compiled and optimized for different architectures. There is currently no documented API for constructing and using fat binary objects by programmers, and therefore this function is an internal function in this version of CUDA. More information can be found in the **nvcc** document.

**Parameters:**

*phMod* - Returned module

*vfatCubin* - Fat binary to load

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_NOT\\_FOUND](#), [CUDA\\_ERROR\\_OUT\\_OF\\_MEMORY](#), [CUDA\\_ERROR\\_NO\\_BINARY\\_FOR\\_GPU](#)

**Note:**

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

**See also:**

[cuModuleGetFunction](#), [cuModuleGetGlobal](#), [cuModuleGetTexRef](#), [cuModuleLoad](#), [cuModuleLoadData](#), [cuModuleLoadDataEx](#), [cuModuleUnload](#)

### 3.21.2.8 CUresult cuModuleUnload (CUmodule *mod*)

Unloads a module *mod* from the current context.

**Parameters:**

*mod* - Module to unload

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**Note:**

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

**See also:**

[cuModuleGetFunction](#), [cuModuleGetGlobal](#), [cuModuleGetTexRef](#), [cuModuleLoad](#), [cuModuleLoadData](#), [cuModuleLoadDataEx](#), [cuModuleLoadFatBinary](#)

## 3.22 Stream Management

### Functions

- [CUresult cuStreamCreate](#) ([CUstream](#) \*phStream, unsigned int Flags)  
*Create a stream.*
- [CUresult cuStreamDestroy](#) ([CUstream](#) hStream)  
*Destroys a stream.*
- [CUresult cuStreamQuery](#) ([CUstream](#) hStream)  
*Determine status of a compute stream.*
- [CUresult cuStreamSynchronize](#) ([CUstream](#) hStream)  
*Wait until a stream's tasks are completed.*

### 3.22.1 Detailed Description

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

### 3.22.2 Function Documentation

#### 3.22.2.1 [CUresult cuStreamCreate](#) ([CUstream](#) \* *phStream*, unsigned int *Flags*)

Creates a stream and returns a handle in `phStream`. `Flags` is required to be 0.

#### Parameters:

- phStream* - Returned newly created stream
- Flags* - Parameters for stream creation (must be 0)

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_OUT\\_OF\\_MEMORY](#)

#### Note:

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

#### See also:

[cuStreamDestroy](#), [cuStreamQuery](#), [cuStreamSynchronize](#)

#### 3.22.2.2 [CUresult cuStreamDestroy](#) ([CUstream](#) *hStream*)

Destroys the stream specified by `hStream`.



**Parameters:**

*hStream* - Stream to destroy

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**Note:**

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

**See also:**

[cuStreamCreate](#), [cuStreamQuery](#), [cuStreamSynchronize](#)

**3.22.2.3 CUresult cuStreamQuery (CUstream *hStream*)**

Returns [CUDA\\_SUCCESS](#) if all operations in the stream specified by `hStream` have completed, or [CUDA\\_ERROR\\_NOT\\_READY](#) if not.

**Parameters:**

*hStream* - Stream to query status of

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#), [CUDA\\_ERROR\\_NOT\\_READY](#)

**Note:**

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

**See also:**

[cuStreamCreate](#), [cuStreamDestroy](#), [cuStreamSynchronize](#)

**3.22.2.4 CUresult cuStreamSynchronize (CUstream *hStream*)**

Waits until the device has completed all operations in the stream specified by `hStream`.

**Parameters:**

*hStream* - Stream to wait for

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#)

**Note:**

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

**See also:**

[cuStreamCreate](#), [cuStreamDestroy](#), [cuStreamQuery](#)

## 3.23 Event Management

### Functions

- [CUresult cuEventCreate](#) (CUevent \*ppEvent, unsigned int Flags)  
*Creates an event.*
- [CUresult cuEventDestroy](#) (CUevent hEvent)  
*Destroys an event.*
- [CUresult cuEventElapsedTime](#) (float \*pMilliseconds, CUevent hStart, CUevent hEnd)  
*Computes the elapsed time between two events.*
- [CUresult cuEventQuery](#) (CUevent hEvent)  
*Queries an event's status.*
- [CUresult cuEventRecord](#) (CUevent hEvent, CUstream hStream)  
*Records an event.*
- [CUresult cuEventSynchronize](#) (CUevent hEvent)  
*Waits for an event to complete.*

### 3.23.1 Detailed Description

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

### 3.23.2 Function Documentation

#### 3.23.2.1 CUresult cuEventCreate (CUevent \* ppEvent, unsigned int Flags)

Creates an event \*ppEvent with the flags specified via `Flags`. Valid flags include:

- [CU\\_EVENT\\_DEFAULT](#): Default event creation flag
- [CU\\_EVENT\\_BLOCKING\\_SYNC](#): Specifies that event should use blocking synchronization

#### Parameters:

*ppEvent* - Returns newly created event  
*Flags* - Event creation flags

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_OUT\\_OF\\_MEMORY](#)

#### Note:

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

**See also:**

[cuEventRecord](#), [cuEventQuery](#), [cuEventSynchronize](#), [cuEventDestroy](#), [cuEventElapsedTime](#)

**3.23.2.2 CUresult cuEventDestroy (CUevent *hEvent*)**

Destroys the event specified by `event`.

**Parameters:**

*hEvent* - Event to destroy

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#)

**Note:**

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

**See also:**

[cuEventCreate](#), [cuEventRecord](#), [cuEventQuery](#), [cuEventSynchronize](#), [cuEventElapsedTime](#)

**3.23.2.3 CUresult cuEventElapsedTime (float \* *pMilliseconds*, CUevent *hStart*, CUevent *hEnd*)**

Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds). If either event has not been recorded yet, this function returns [CUDA\\_ERROR\\_NOT\\_READY](#). If either event has been recorded with a non-zero stream, the result is undefined.

**Parameters:**

*pMilliseconds* - Returned elapsed time in milliseconds

*hStart* - Starting event

*hEnd* - Ending event

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#), [CUDA\\_ERROR\\_NOT\\_READY](#)

**Note:**

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

**See also:**

[cuEventCreate](#), [cuEventRecord](#), [cuEventQuery](#), [cuEventSynchronize](#), [cuEventDestroy](#)

#### 3.23.2.4 CUresult cuEventQuery (CUevent *hEvent*)

Returns [CUDA\\_SUCCESS](#) if the event has actually been recorded, or [CUDA\\_ERROR\\_NOT\\_READY](#) if not. If [cuEventRecord\(\)](#) has not been called on this event, the function returns [CUDA\\_ERROR\\_INVALID\\_VALUE](#).

**Parameters:**

*hEvent* - Event to query

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_NOT\\_READY](#)

**Note:**

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

**See also:**

[cuEventCreate](#), [cuEventRecord](#), [cuEventSynchronize](#), [cuEventDestroy](#), [cuEventElapsedTime](#)

#### 3.23.2.5 CUresult cuEventRecord (CUevent *hEvent*, CUstream *hStream*)

Records an event. If `stream` is non-zero, the event is recorded after all preceding operations in the stream have been completed; otherwise, it is recorded after all preceding operations in the CUDA context have been completed. Since operation is asynchronous, [cuEventQuery\(\)](#) and/or [cuEventSynchronize\(\)](#) must be used to determine when the event has actually been recorded.

If [cuEventRecord\(\)](#) has previously been called and the event has not been recorded yet, this function returns [CUDA\\_ERROR\\_INVALID\\_VALUE](#).

**Parameters:**

*hEvent* - Event to record

*hStream* - Stream to record event for

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**Note:**

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

**See also:**

[cuEventCreate](#), [cuEventQuery](#), [cuEventSynchronize](#), [cuEventDestroy](#), [cuEventElapsedTime](#)

#### 3.23.2.6 CUresult cuEventSynchronize (CUevent *hEvent*)

Waits until the event has actually been recorded. If [cuEventRecord\(\)](#) has been called on this event, the function returns [CUDA\\_ERROR\\_INVALID\\_VALUE](#).

If [cuEventRecord\(\)](#) has previously been called and the event has not been recorded yet, this function returns [CUDA\\_ERROR\\_INVALID\\_VALUE](#).

**Parameters:**

*hEvent* - Event to wait for

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#)

**Note:**

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

**See also:**

[cuEventCreate](#), [cuEventRecord](#), [cuEventQuery](#), [cuEventDestroy](#), [cuEventElapsedTime](#)

## 3.24 Execution Control

### Functions

- **CUresult cuFuncGetAttribute** (int \*pi, CUfunction\_attribute attrib, CUfunction func)  
*Returns information about a function.*
- **CUresult cuFuncSetBlockShape** (CUfunction func, int x, int y, int z)  
*Sets the block-dimensions for the function.*
- **CUresult cuFuncSetSharedSize** (CUfunction func, unsigned int bytes)  
*Sets the dynamic shared-memory size for the function.*
- **CUresult cuLaunch** (CUfunction func)  
*Launches a CUDA function.*
- **CUresult cuLaunchGrid** (CUfunction func, int grid\_width, int grid\_height)  
*Launches a CUDA function.*
- **CUresult cuLaunchGridAsync** (CUfunction func, int grid\_width, int grid\_height, CUstream hStream)  
*Launches a CUDA function.*
- **CUresult cuParamSetf** (CUfunction func, int offset, float value)  
*Adds a floating-point parameter to the function's argument list.*
- **CUresult cuParamSeti** (CUfunction func, int offset, unsigned int value)  
*Adds an integer parameter to the function's argument list.*
- **CUresult cuParamSetSize** (CUfunction func, unsigned int numbytes)  
*Sets the parameter size for the function.*
- **CUresult cuParamSetTexRef** (CUfunction func, int texunit, CUtexref pTexRef)  
*Adds a texture-reference to the function's argument list.*
- **CUresult cuParamSetv** (CUfunction func, int offset, void \*ptr, unsigned int numbytes)  
*Adds arbitrary data to the function's argument list.*

### 3.24.1 Detailed Description

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

### 3.24.2 Function Documentation

#### 3.24.2.1 CUresult cuFuncGetAttribute (int \*pi, CUfunction\_attribute attrib, CUfunction func)

Returns in \*pi the integer value of the attribute attrib on the kernel given by func. The supported attributes are:

- [CU\\_FUNC\\_ATTRIBUTE\\_MAX\\_THREADS\\_PER\\_BLOCK](#): The number of threads beyond which a launch of the function would fail. This number depends on both the function and the device on which the function is currently loaded.
- [CU\\_FUNC\\_ATTRIBUTE\\_SHARED\\_SIZE\\_BYTES](#): The size in bytes of statically-allocated shared memory required by this function. This does not include dynamically-allocated shared memory requested by the user at runtime.
- [CU\\_FUNC\\_ATTRIBUTE\\_CONST\\_SIZE\\_BYTES](#): The size in bytes of user-allocated constant memory required by this function.
- [CU\\_FUNC\\_ATTRIBUTE\\_LOCAL\\_SIZE\\_BYTES](#): The size in bytes of thread local memory used by this function.
- [CU\\_FUNC\\_ATTRIBUTE\\_NUM\\_REGS](#): The number of registers used by each thread of this function.

**Parameters:**

*pi* - Returned attribute value  
*attrib* - Attribute requested  
*func* - Function to query attribute of

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**Note:**

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

**See also:**

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuParamSetSize](#), [cuParamSeti](#), [cuParamSetf](#), [cuParamSetv](#), [cuParamSetTexRef](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#)

**3.24.2.2 CUresult cuFuncSetBlockShape (CUfunction *func*, int *x*, int *y*, int *z*)**

Specifies the *x*, *y*, and *z* dimensions of the thread blocks that are created when the kernel given by *func* is launched.

**Parameters:**

*func* - Kernel to specify dimensions of  
*x* - X dimension  
*y* - Y dimension  
*z* - Z dimension

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**Note:**

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

**See also:**

[cuFuncSetSharedSize](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSeti](#), [cuParamSetf](#), [cuParamSetv](#), [cuParamSetTexRef](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#)

### 3.24.2.3 CUresult cuFuncSetSharedSize (CUfunction *func*, unsigned int *bytes*)

Sets through *bytes* the amount of dynamic shared memory that will be available to each thread block when the kernel given by *func* is launched.

**Parameters:**

*func* - Kernel to specify dynamic shared-memory size for

*bytes* - Dynamic shared-memory size per thread in bytes

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_HANDLE, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

cuFuncSetBlockShape, cuFuncGetAttribute, cuParamSetSize, cuParamSeti, cuParamSetf, cuParamSetv, cuParamSetTexRef, cuLaunch, cuLaunchGrid, cuLaunchGridAsync

### 3.24.2.4 CUresult cuLaunch (CUfunction *func*)

Invokes the kernel *func* on a 1 x 1 x 1 grid of blocks. The block contains the number of threads specified by a previous call to [cuFuncSetBlockShape\(\)](#).

**Parameters:**

*func* - Kernel to launch

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_LAUNCH\_INCOMPATIBLE\_TEXTURING

**Note:**

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

**See also:**

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSeti, cuParamSetv, cuParamSetTexRef, cuLaunchGrid, cuLaunchGridAsync

### 3.24.2.5 CUresult cuLaunchGrid (CUfunction *func*, int *grid\_width*, int *grid\_height*)

Invokes the kernel *func* on a *grid\_width* x *grid\_height* grid of blocks. Each block contains the number of threads specified by a previous call to [cuFuncSetBlockShape\(\)](#).

**Parameters:**

*func* - Kernel to launch



*grid\_width* - Width of grid in blocks

*grid\_height* - Height of grid in blocks

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_LAUNCH\_INCOMPATIBLE\_TEXTURING

**Note:**

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

**See also:**

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSeti, cuParamSetv, cuParamSetTexRef, cuLaunch, cuLaunchGridAsync

### 3.24.2.6 CUresult cuLaunchGridAsync (CUfunction *func*, int *grid\_width*, int *grid\_height*, CUstream *hStream*)

Invokes the kernel *func* on a *grid\_width* x *grid\_height* grid of blocks. Each block contains the number of threads specified by a previous call to [cuFuncSetBlockShape\(\)](#).

[cuLaunchGridAsync\(\)](#) can optionally be associated to a stream by passing a non-zero *hStream* argument.

**Parameters:**

*func* - Kernel to launch

*grid\_width* - Width of grid in blocks

*grid\_height* - Height of grid in blocks

*hStream* - Stream identifier

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_LAUNCH\_INCOMPATIBLE\_TEXTURING

**Note:**

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

**See also:**

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSeti, cuParamSetv, cuParamSetTexRef, cuLaunch, cuLaunchGrid

### 3.24.2.7 CUresult cuParamSetf (CUfunction *func*, int *offset*, float *value*)

Sets a floating-point parameter that will be specified the next time the kernel corresponding to *func* will be invoked. *offset* is a byte offset.

**Parameters:**

*func* - Kernel to add parameter to  
*offset* - Offset to add parameter to argument list  
*value* - Value of parameter

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**Note:**

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

**See also:**

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSeti](#), [cuParamSetv](#), [cuParamSetTexRef](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#)

**3.24.2.8 CUresult cuParamSeti (CUfunction *func*, int *offset*, unsigned int *value*)**

Sets an integer parameter that will be specified the next time the kernel corresponding to *func* will be invoked. *offset* is a byte offset.

**Parameters:**

*func* - Kernel to add parameter to  
*offset* - Offset to add parameter to argument list  
*value* - Value of parameter

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**Note:**

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

**See also:**

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSetf](#), [cuParamSetv](#), [cuParamSetTexRef](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#)

**3.24.2.9 CUresult cuParamSetSize (CUfunction *func*, unsigned int *numbytes*)**

Sets through *numbytes* the total size in bytes needed by the function parameters of the kernel corresponding to *func*.

**Parameters:**

*func* - Kernel to set parameter size for  
*numbytes* - Size of parameter list in bytes

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetf, cuParamSeti, cuParamSetv, cuParamSetTexRef, cuLaunch, cuLaunchGrid, cuLaunchGridAsync

**3.24.2.10 CUresult cuParamSetTexRef (CUfunction *func*, int *texunit*, CUtexref *pTexRef*)**

Makes the CUDA array or linear memory bound to the texture reference `pTexRef` available to a device program as a texture. In this version of CUDA, the texture-reference must be obtained via `cuModuleGetTexRef()` and the `texunit` parameter must be set to `CU_PARAM_TR_DEFAULT`.

**Parameters:**

*func* - Kernel to add texture-reference to  
*texunit* - Texture unit (must be `CU_PARAM_TR_DEFAULT`)  
*pTexRef* - Texture-reference to add to argument list

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSeti, cuParamSetv, cuLaunch, cuLaunchGrid, cuLaunchGridAsync

**3.24.2.11 CUresult cuParamSetv (CUfunction *func*, int *offset*, void \* *ptr*, unsigned int *numbytes*)**

Copies an arbitrary amount of data (specified in `numbytes`) from `ptr` into the parameter space of the kernel corresponding to `func`. `offset` is a byte offset.

**Parameters:**

*func* - Kernel to add data to  
*offset* - Offset to add data to argument list  
*ptr* - Pointer to arbitrary data  
*numbytes* - Size of data to copy in bytes

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**Note:**

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

**See also:**

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSetf](#), [cuParamSeti](#), [cuParamSetTexRef](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#)

## 3.25 Memory Management

### Functions

- **CUresult cuArray3DCreate** (CUarray \*pHandle, const CUDA\_ARRAY3D\_DESCRIPTOR \*pDesc)  
*Creates a 3D CUDA array.*
- **CUresult cuArray3DGetDescriptor** (CUDA\_ARRAY3D\_DESCRIPTOR \*pDesc, CUarray hArray)  
*Get a 3D CUDA array descriptor.*
- **CUresult cuArrayCreate** (CUarray \*pHandle, const CUDA\_ARRAY\_DESCRIPTOR \*pDesc)  
*Creates a 1D or 2D CUDA array.*
- **CUresult cuArrayDestroy** (CUarray pArray)  
*Destroys a CUDA array.*
- **CUresult cuArrayGetDescriptor** (CUDA\_ARRAY\_DESCRIPTOR \*pDesc, CUarray hArray)  
*Get a 1D or 2D CUDA array descriptor.*
- **CUresult cuMemAlloc** (CUdeviceptr \*dptr, unsigned int bytesize)  
*Allocates device memory.*
- **CUresult cuMemAllocHost** (void \*\*pp, unsigned int bytesize)  
*Allocates page-locked host memory.*
- **CUresult cuMemAllocPitch** (CUdeviceptr \*dptr, unsigned int \*pPitch, unsigned int WidthInBytes, unsigned int Height, unsigned int ElementSizeBytes)  
*Allocates pitched device memory.*
- **CUresult cuMemcpy2D** (const CUDA\_MEMCPY2D \*pCopy)  
*Copies memory for 2D arrays.*
- **CUresult cuMemcpy2DAsync** (const CUDA\_MEMCPY2D \*pCopy, CUstream hStream)  
*Copies memory for 2D arrays.*
- **CUresult cuMemcpy2DUnaligned** (const CUDA\_MEMCPY2D \*pCopy)  
*Copies memory for 2D arrays.*
- **CUresult cuMemcpy3D** (const CUDA\_MEMCPY3D \*pCopy)  
*Copies memory for 3D arrays.*
- **CUresult cuMemcpy3DAsync** (const CUDA\_MEMCPY3D \*pCopy, CUstream hStream)  
*Copies memory for 3D arrays.*
- **CUresult cuMemcpyAtoA** (CUarray hDst, unsigned int DstOffset, CUarray hSrc, unsigned int SrcOffset, unsigned int NumBytes)  
*Copies memory from Array to Array.*
- **CUresult cuMemcpyAtoD** (CUdeviceptr dptr, CUarray hSrc, unsigned int SrcOffset, unsigned int NumBytes)  
*Copies memory from Array to Device.*

- **CUresult cuMemcpyAtoH** (void \*pDst, **CUarray** hSrc, unsigned int srcOffset, unsigned int bytes)  
*Copies memory from Array to Host.*
- **CUresult cuMemcpyAtoHAsync** (void \*pDst, **CUarray** hSrc, unsigned int srcOffset, unsigned int bytes, **CUstream** hStream)  
*Copies memory from Array to Host.*
- **CUresult cuMemcpyDtoA** (**CUarray** hDst, unsigned int DstOffset, **CUdeviceptr** dptr, unsigned int NumBytes)  
*Copies memory from Device to Array.*
- **CUresult cuMemcpyDtoD** (**CUdeviceptr** dst, **CUdeviceptr** src, unsigned int bytes)  
*Copies memory from Device to Device.*
- **CUresult cuMemcpyDtoH** (void \*dst, **CUdeviceptr** dptr, unsigned int bytes)  
*Copies memory from Device to Host.*
- **CUresult cuMemcpyDtoHAsync** (void \*dst, **CUdeviceptr** dptr, unsigned int bytes, **CUstream** hStream)  
*Copies memory from Device to Host.*
- **CUresult cuMemcpyHtoA** (**CUarray** hDst, unsigned int dstOffset, const void \*pSrc, unsigned int bytes)  
*Copies memory from Host to Array.*
- **CUresult cuMemcpyHtoAAsync** (**CUarray** hDst, unsigned int dstOffset, const void \*pSrc, unsigned int bytes, **CUstream** hStream)  
*Copies memory from Host to Array.*
- **CUresult cuMemcpyHtoD** (**CUdeviceptr** dptr, const void \*src, unsigned int bytes)  
*Copies memory from Host to Device.*
- **CUresult cuMemcpyHtoDAsync** (**CUdeviceptr** dptr, const void \*src, unsigned int bytes, **CUstream** hStream)  
*Copies memory from Host to Device.*
- **CUresult cuMemFree** (**CUdeviceptr** dptr)  
*Frees device memory.*
- **CUresult cuMemFreeHost** (void \*p)  
*Frees page-locked host memory.*
- **CUresult cuMemGetAddressRange** (**CUdeviceptr** \*pdptr, unsigned int \*psize, **CUdeviceptr** dptr)  
*Get information on memory allocations.*
- **CUresult cuMemGetInfo** (unsigned int \*free, unsigned int \*total)  
*Gets free and total memory.*
- **CUresult cuMemHostAlloc** (void \*\*pp, size\_t bytes, unsigned int Flags)  
*Allocates page-locked host memory.*
- **CUresult cuMemHostGetDevicePointer** (**CUdeviceptr** \*ret, void \*p, unsigned int Flags)  
*Passes back device pointer of mapped pinned memory.*

- [CUresult cuMemsetD16](#) ([CUdeviceptr](#) dstDevice, unsigned short us, unsigned int N)  
*Initializes device memory.*
- [CUresult cuMemsetD2D16](#) ([CUdeviceptr](#) dstDevice, unsigned int dstPitch, unsigned short us, unsigned int Width, unsigned int Height)  
*Initializes device memory.*
- [CUresult cuMemsetD2D32](#) ([CUdeviceptr](#) dstDevice, unsigned int dstPitch, unsigned int ui, unsigned int Width, unsigned int Height)  
*Initializes device memory.*
- [CUresult cuMemsetD2D8](#) ([CUdeviceptr](#) dstDevice, unsigned int dstPitch, unsigned char uc, unsigned int Width, unsigned int Height)  
*Initializes device memory.*
- [CUresult cuMemsetD32](#) ([CUdeviceptr](#) dstDevice, unsigned int ui, unsigned int N)  
*Initializes device memory.*
- [CUresult cuMemsetD8](#) ([CUdeviceptr](#) dstDevice, unsigned char c, unsigned int N)  
*Initializes device memory.*

### 3.25.1 Detailed Description

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

### 3.25.2 Function Documentation

#### 3.25.2.1 [CUresult cuArray3DCreate](#) ([CUarray](#) \*pHandle, const [CUDA\\_ARRAY3D\\_DESCRIPTOR](#) \*pDesc)

Creates a CUDA array according to the [CUDA\\_ARRAY3D\\_DESCRIPTOR](#) structure pDesc and returns a handle to the new CUDA array in \*pHandle. The [CUDA\\_ARRAY3D\\_DESCRIPTOR](#) is defined as:

```
typedef struct {
    unsigned int Width;
    unsigned int Height;
    unsigned int Depth;
    CUarray_format Format;
    unsigned int NumChannels;
    unsigned int Flags;
} CUDA_ARRAY3D_DESCRIPTOR;
```

where:

- Width, Height, and Depth are the width, height, and depth of the CUDA array (in elements); the CUDA array is one-dimensional if height and depth are 0, two-dimensional if depth is 0, and three-dimensional otherwise;
- Format specifies the format of the elements; [CUarray\\_format](#) is defined as:

```
typedef enum CUarray_format_enum {
    CU_AD_FORMAT_UNSIGNED_INT8 = 0x01,
    CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
    CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
    CU_AD_FORMAT_SIGNED_INT8 = 0x08,
    CU_AD_FORMAT_SIGNED_INT16 = 0x09,
    CU_AD_FORMAT_SIGNED_INT32 = 0x0a,
    CU_AD_FORMAT_HALF = 0x10,
    CU_AD_FORMAT_FLOAT = 0x20
} CUarray_format;
```

- NumChannels specifies the number of packed components per CUDA array element; it may be 1, 2, or 4;
- Flags provides for future features. For now, it must be set to 0.

Here are examples of CUDA array descriptions:

Description for a CUDA array of 2048 floats:

```
CUDA_ARRAY3D_DESCRIPTOR desc;
desc.Format = CU_AD_FORMAT_FLOAT;
desc.NumChannels = 1;
desc.Width = 2048;
desc.Height = 0;
desc.Depth = 0;
```

Description for a 64 x 64 CUDA array of floats:

```
CUDA_ARRAY3D_DESCRIPTOR desc;
desc.Format = CU_AD_FORMAT_FLOAT;
desc.NumChannels = 1;
desc.Width = 64;
desc.Height = 64;
desc.Depth = 0;
```

Description for a width x height x depth CUDA array of 64-bit, 4x16-bit float16's:

```
CUDA_ARRAY3D_DESCRIPTOR desc;
desc.FormatFlags = CU_AD_FORMAT_HALF;
desc.NumChannels = 4;
desc.Width = width;
desc.Height = height;
desc.Depth = depth;
```

#### Parameters:

- pHandle* - Returned array
- pDesc* - 3D array descriptor

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_OUT\\_OF\\_MEMORY](#), [CUDA\\_ERROR\\_UNKNOWN](#)

#### Note:

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



**See also:**

[cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

**3.25.2.2 CUresult cuArray3DGetDescriptor (CUDA\_ARRAY3D\_DESCRIPTOR \*pDesc, CUarray hArray)**

Returns in *pDesc* a descriptor containing information on the format and dimensions of the CUDA array *hArray*. It is useful for subroutines that have been passed a CUDA array, but need to know the CUDA array parameters for validation or other purposes.

This function may be called on 1D and 2D arrays, in which case the *Height* and/or *Depth* members of the descriptor struct will be set to 0.

**Parameters:**

*pDesc* - Returned 3D array descriptor

*hArray* - 3D array to get descriptor of

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#)

**Note:**

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

**See also:**

[cuArray3DCreate](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

**3.25.2.3 CUresult cuArrayCreate (CUarray \*pHandle, const CUDA\_ARRAY\_DESCRIPTOR \*pDesc)**

Creates a CUDA array according to the [CUDA\\_ARRAY\\_DESCRIPTOR](#) structure *pDesc* and returns a handle to the new CUDA array in *pHandle*. The [CUDA\\_ARRAY\\_DESCRIPTOR](#) is defined as:

```
typedef struct {
    unsigned int Width;
    unsigned int Height;
    CUarray_format Format;
    unsigned int NumChannels;
} CUDA_ARRAY_DESCRIPTOR;
```

where:

- `Width`, and `Height` are the width, and height of the CUDA array (in elements); the CUDA array is one-dimensional if height is 0, two-dimensional otherwise;
- `Format` specifies the format of the elements; `CUarray_format` is defined as:

```
typedef enum CUarray_format_enum {
    CU_AD_FORMAT_UNSIGNED_INT8 = 0x01,
    CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
    CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
    CU_AD_FORMAT_SIGNED_INT8 = 0x08,
    CU_AD_FORMAT_SIGNED_INT16 = 0x09,
    CU_AD_FORMAT_SIGNED_INT32 = 0x0a,
    CU_AD_FORMAT_HALF = 0x10,
    CU_AD_FORMAT_FLOAT = 0x20
} CUarray_format;
```

- `NumChannels` specifies the number of packed components per CUDA array element; it may be 1, 2, or 4;

Here are examples of CUDA array descriptions:

Description for a CUDA array of 2048 floats:

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.Format = CU_AD_FORMAT_FLOAT;
desc.NumChannels = 1;
desc.Width = 2048;
desc.Height = 1;
```

Description for a 64 x 64 CUDA array of floats:

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.Format = CU_AD_FORMAT_FLOAT;
desc.NumChannels = 1;
desc.Width = 64;
desc.Height = 64;
```

Description for a width x height CUDA array of 64-bit, 4x16-bit float16's:

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.FormatFlags = CU_AD_FORMAT_HALF;
desc.NumChannels = 4;
desc.Width = width;
desc.Height = height;
```

Description for a width x height CUDA array of 16-bit elements, each of which is two 8-bit unsigned chars:

```
CUDA_ARRAY_DESCRIPTOR arrayDesc;
desc.FormatFlags = CU_AD_FORMAT_UNSIGNED_INTS;
desc.NumChannels = 2;
desc.Width = width;
desc.Height = height;
```

#### Parameters:

*pHandle* - Returned array

*pDesc* - Array descriptor

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_OUT\_OF\_MEMORY, CUDA\_ERROR\_UNKNOWN

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

**3.25.2.4 CUresult cuArrayDestroy (CUarray pArray)**

Destroys the CUDA array *pArray*.

**Parameters:**

*pArray* - Array to destroy

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_HANDLE, CUDA\_ERROR\_ARRAY\_IS\_MAPPED

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

**3.25.2.5 CUresult cuArrayGetDescriptor (CUDA\_ARRAY\_DESCRIPTOR \*pDesc, CUarray hArray)**

Returns in *pDesc* a descriptor containing information on the format and dimensions of the CUDA array *hArray*. It is useful for subroutines that have been passed a CUDA array, but need to know the CUDA array parameters for validation or other purposes.

**Parameters:**

*pDesc* - Returned array descriptor  
*hArray* - Array to get descriptor of

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_INVALID\_HANDLE

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

**3.25.2.6 CUresult cuMemAlloc (CUdeviceptr \* *dptr*, unsigned int *bytesize*)**

Allocates *bytesize* bytes of linear memory on the device and returns in *\*dptr* a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. If *bytesize* is 0, `cuMemAlloc()` returns `CUDA_ERROR_INVALID_VALUE`.

**Parameters:**

*dptr* - Returned device pointer  
*bytesize* - Requested allocation size in bytes

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_OUT\_OF\_MEMORY

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

### 3.25.2.7 CUresult cuMemAllocHost (void \*\* pp, unsigned int bytesize)

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

#### Parameters:

`pp` - Returned host pointer to page-locked memory  
`bytesize` - Requested allocation size in bytes

#### Returns:

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_OUT_OF_MEMORY`

#### Note:

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

#### See also:

`cuArray3DCreate`, `cuArray3DGetDescriptor`, `cuArrayCreate`, `cuArrayDestroy`, `cuArrayGetDescriptor`, `cuMemAlloc`, `cuMemAllocPitch`, `cuMemcpy2D`, `cuMemcpy2DAsync`, `cuMemcpy2DUnaligned`, `cuMemcpy3D`, `cuMemcpy3DAsync`, `cuMemcpyAtoA`, `cuMemcpyAtoD`, `cuMemcpyAtoH`, `cuMemcpyAtoHAsync`, `cuMemcpyDtoA`, `cuMemcpyDtoD`, `cuMemcpyDtoH`, `cuMemcpyDtoHAsync`, `cuMemcpyHtoA`, `cuMemcpyHtoAAsync`, `cuMemcpyHtoD`, `cuMemcpyHtoDAsync`, `cuMemFree`, `cuMemFreeHost`, `cuMemGetAddressRange`, `cuMemGetInfo`, `cuMemHostAlloc`, `cuMemHostGetDevicePointer`, `cuMemsetD2D8`, `cuMemsetD2D16`, `cuMemsetD2D32`, `cuMemsetD8`, `cuMemsetD16`, `cuMemsetD32`

### 3.25.2.8 CUresult cuMemAllocPitch (CUdeviceptr \* dptr, unsigned int \* pPitch, unsigned int WidthInBytes, unsigned int Height, unsigned int ElementSizeBytes)

Allocates at least `WidthInBytes * Height` bytes of linear memory on the device and returns in `*dptr` a pointer to the allocated memory. The function may pad the allocation to ensure that corresponding pointers in any given row will continue to meet the alignment requirements for coalescing as the address is updated from row to row. `ElementSizeBytes` specifies the size of the largest reads and writes that will be performed on the memory range. `ElementSizeBytes` may be 4, 8 or 16 (since coalesced memory transactions are not possible on other data sizes). If `ElementSizeBytes` is smaller than the actual read/write size of a kernel, the kernel will run correctly, but possibly at reduced speed. The pitch returned in `*pPitch` by `cuMemAllocPitch()` is the width in bytes of the allocation. The intended usage of pitch is as a separate parameter of the allocation, used to compute addresses within the 2D array. Given the row and column of an array element of type `T`, the address is computed as:

```
T* pElement = (T*)((char*)BaseAddress + Row * Pitch) + Column;
```

The pitch returned by `cuMemAllocPitch()` is guaranteed to work with `cuMemcpy2D()` under all circumstances. For allocations of 2D arrays, it is recommended that programmers consider performing pitch allocations using `cuMemAllocPitch()`. Due to alignment restrictions in the hardware, this is especially true if the application will be performing 2D memory copies between different regions of device memory (whether linear memory or CUDA arrays).

**Parameters:**

- dptr* - Returned device pointer
- pPitch* - Returned pitch of allocation in bytes
- WidthInBytes* - Requested allocation width in bytes
- Height* - Requested allocation height in rows
- ElementSizeBytes* - Size of largest reads/writes for range

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_OUT\\_OF\\_MEMORY](#)

**Note:**

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

**3.25.2.9 CUresult cuMemcpy2D (const CUDA\_MEMCPY2D \*pCopy)**

Perform a 2D memory copy according to the parameters specified in `pCopy`. The [CUDA\\_MEMCPY2D](#) structure is defined as:

```
typedef struct CUDA_MEMCPY2D_st {
    unsigned int srcXInBytes, srcY;
    CUmemorytype srcMemoryType;
    const void *srcHost;
    CUdeviceptr srcDevice;
    CUarray srcArray;
    unsigned int srcPitch;

    unsigned int dstXInBytes, dstY;
    CUmemorytype dstMemoryType;
    void *dstHost;
    CUdeviceptr dstDevice;
    CUarray dstArray;
    unsigned int dstPitch;

    unsigned int WidthInBytes;
    unsigned int Height;
} CUDA_MEMCPY2D;
```

where:

- `srcMemoryType` and `dstMemoryType` specify the type of memory of the source and destination, respectively; [CUmemorytype\\_enum](#) is defined as:

```
typedef enum CUMemoryType_enum {
    CU_MEMORYTYPE_HOST = 0x01,
    CU_MEMORYTYPE_DEVICE = 0x02,
    CU_MEMORYTYPE_ARRAY = 0x03
} CUMemoryType;
```

If srcMemoryType is [CU\\_MEMORYTYPE\\_HOST](#), srcHost and srcPitch specify the (host) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is [CU\\_MEMORYTYPE\\_DEVICE](#), srcDevice and srcPitch specify the (device) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is [CU\\_MEMORYTYPE\\_ARRAY](#), srcArray specifies the handle of the source data. srcHost, srcDevice and srcPitch are ignored.

If dstMemoryType is [CU\\_MEMORYTYPE\\_HOST](#), dstHost and dstPitch specify the (host) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is [CU\\_MEMORYTYPE\\_DEVICE](#), dstDevice and dstPitch specify the (device) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is [CU\\_MEMORYTYPE\\_ARRAY](#), dstArray specifies the handle of the destination data. dstHost, dstDevice and dstPitch are ignored.

- srcXInBytes and srcY specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*)((char*)srcHost+srcY*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
```

For CUDA arrays, srcXInBytes must be evenly divisible by the array element size.

- dstXInBytes and dstY specify the base address of the destination data for the copy.

For host pointers, the base address is

```
void* dstStart = (void*)((char*)dstHost+dstY*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
```

For CUDA arrays, `dstXInBytes` must be evenly divisible by the array element size.

- `WidthInBytes` and `Height` specify the width (in bytes) and height of the 2D copy being performed. Any pitches must be greater than or equal to `WidthInBytes`.

`cuMemcpy2D()` returns an error if any pitch is greater than the maximum allowed (`CUDA_DEVICE_ATTRIBUTE_MAX_PITCH`). `cuMemAllocPitch()` passes back pitches that always work with `cuMemcpy2D()`. On intra-device memory copies (device ? device, CUDA array ? device, CUDA array ? CUDA array), `cuMemcpy2D()` may fail for pitches not computed by `cuMemAllocPitch()`. `cuMemcpy2DUnaligned()` does not have this restriction, but may run significantly slower in the cases where `cuMemcpy2D()` would have returned an error code.

#### Parameters:

*pCopy* - Parameters for the memory copy

#### Returns:

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

#### Note:

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

#### See also:

`cuArray3DCreate`, `cuArray3DGetDescriptor`, `cuArrayCreate`, `cuArrayDestroy`, `cuArrayGetDescriptor`, `cuMemAlloc`, `cuMemAllocHost`, `cuMemAllocPitch`, `cuMemcpy2DAsync`, `cuMemcpy2DUnaligned`, `cuMemcpy3D`, `cuMemcpy3DAsync`, `cuMemcpyAtoA`, `cuMemcpyAtoD`, `cuMemcpyAtoH`, `cuMemcpyAtoHAsync`, `cuMemcpyDtoA`, `cuMemcpyDtoD`, `cuMemcpyDtoH`, `cuMemcpyDtoHAsync`, `cuMemcpyHtoA`, `cuMemcpyHtoAAsync`, `cuMemcpyHtoD`, `cuMemcpyHtoDAsync`, `cuMemFree`, `cuMemFreeHost`, `cuMemGetAddressRange`, `cuMemGetInfo`, `cuMemHostAlloc`, `cuMemHostGetDevicePointer`, `cuMemsetD2D8`, `cuMemsetD2D16`, `cuMemsetD2D32`, `cuMemsetD8`, `cuMemsetD16`, `cuMemsetD32`

#### 3.25.2.10 CUresult cuMemcpy2DAsync (const CUDA\_MEMCPY2D \*pCopy, CUstream hStream)

Perform a 2D memory copy according to the parameters specified in `pCopy`. The `CUDA_MEMCPY2D` structure is defined as:

```
typedef struct CUDA_MEMCPY2D_st {
    unsigned int srcXInBytes, srcY;
    CUmemorytype srcMemoryType;
    const void *srcHost;
    CUdeviceptr srcDevice;
    CUarray srcArray;
    unsigned int srcPitch;
    unsigned int dstXInBytes, dstY;
    CUmemorytype dstMemoryType;
    void *dstHost;
    CUdeviceptr dstDevice;
    CUarray dstArray;
    unsigned int dstPitch;
    unsigned int WidthInBytes;
    unsigned int Height;
} CUDA_MEMCPY2D;
```



where:

- `srcMemoryType` and `dstMemoryType` specify the type of memory of the source and destination, respectively; `CUmemorytype_enum` is defined as:

```
typedef enum CUmemorytype_enum {
    CU_MEMORYTYPE_HOST = 0x01,
    CU_MEMORYTYPE_DEVICE = 0x02,
    CU_MEMORYTYPE_ARRAY = 0x03
} CUmemorytype;
```

If `srcMemoryType` is `CU_MEMORYTYPE_HOST`, `srcHost` and `srcPitch` specify the (host) base address of the source data and the bytes per row to apply. `srcArray` is ignored.

If `srcMemoryType` is `CU_MEMORYTYPE_DEVICE`, `srcDevice` and `srcPitch` specify the (device) base address of the source data and the bytes per row to apply. `srcArray` is ignored.

If `srcMemoryType` is `CU_MEMORYTYPE_ARRAY`, `srcArray` specifies the handle of the source data. `srcHost`, `srcDevice` and `srcPitch` are ignored.

If `dstMemoryType` is `CU_MEMORYTYPE_HOST`, `dstHost` and `dstPitch` specify the (host) base address of the destination data and the bytes per row to apply. `dstArray` is ignored.

If `dstMemoryType` is `CU_MEMORYTYPE_DEVICE`, `dstDevice` and `dstPitch` specify the (device) base address of the destination data and the bytes per row to apply. `dstArray` is ignored.

If `dstMemoryType` is `CU_MEMORYTYPE_ARRAY`, `dstArray` specifies the handle of the destination data. `dstHost`, `dstDevice` and `dstPitch` are ignored.

- `srcXInBytes` and `srcY` specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*)((char*)srcHost+srcY*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
```

For CUDA arrays, `srcXInBytes` must be evenly divisible by the array element size.

- `dstXInBytes` and `dstY` specify the base address of the destination data for the copy.

For host pointers, the base address is

```
void* dstStart = (void*)((char*)dstHost+dstY*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
```

For CUDA arrays, `dstXInBytes` must be evenly divisible by the array element size.

- `WidthInBytes` and `Height` specify the width (in bytes) and height of the 2D copy being performed. Any pitches must be greater than or equal to `WidthInBytes`.

`cuMemcpy2D()` returns an error if any pitch is greater than the maximum allowed (`CU_DEVICE_ATTRIBUTE_MAX_PITCH`). `cuMemAllocPitch()` passes back pitches that always work with `cuMemcpy2D()`. On intra-device memory copies (device ? device, CUDA array ? device, CUDA array ? CUDA array), `cuMemcpy2D()` may fail for pitches not computed by `cuMemAllocPitch()`. `cuMemcpy2DUnaligned()` does not have this restriction, but may run significantly slower in the cases where `cuMemcpy2D()` would have returned an error code.

`cuMemcpy2DAsync()` is asynchronous and can optionally be associated to a stream by passing a non-zero `hStream` argument. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input.

#### Parameters:

*pCopy* - Parameters for the memory copy

*hStream* - Stream identifier

#### Returns:

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

#### Note:

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

#### See also:

`cuArray3DCreate`, `cuArray3DGetDescriptor`, `cuArrayCreate`, `cuArrayDestroy`, `cuArrayGetDescriptor`, `cuMemAlloc`, `cuMemAllocHost`, `cuMemAllocPitch`, `cuMemcpy2D`, `cuMemcpy2DUnaligned`, `cuMemcpy3D`, `cuMemcpy3DAsync`, `cuMemcpyAtoA`, `cuMemcpyAtoD`, `cuMemcpyAtoH`, `cuMemcpyAtoHAsync`, `cuMemcpyDtoA`, `cuMemcpyDtoD`, `cuMemcpyDtoH`, `cuMemcpyDtoHAsync`, `cuMemcpyHtoA`, `cuMemcpyHtoAAsync`, `cuMemcpyHtoD`, `cuMemcpyHtoDAsync`, `cuMemFree`, `cuMemFreeHost`, `cuMemGetAddressRange`, `cuMemGetInfo`, `cuMemHostAlloc`, `cuMemHostGetDevicePointer`, `cuMemsetD2D8`, `cuMemsetD2D16`, `cuMemsetD2D32`, `cuMemsetD8`, `cuMemsetD16`, `cuMemsetD32`

### 3.25.2.11 CUresult cuMemcpy2DUnaligned (const CUDA\_MEMCPY2D \* pCopy)

Perform a 2D memory copy according to the parameters specified in `pCopy`. The `CUDA_MEMCPY2D` structure is defined as:

```
typedef struct CUDA_MEMCPY2D_st {
    unsigned int srcXInBytes, srcY;
    CUmemorytype srcMemoryType;
    const void *srcHost;
```

```

CUdeviceptr srcDevice;
CUarray srcArray;
unsigned int srcPitch;
unsigned int dstXInBytes, dstY;
CUMemorytype dstMemoryType;
void *dstHost;
CUdeviceptr dstDevice;
CUarray dstArray;
unsigned int dstPitch;
unsigned int WidthInBytes;
unsigned int Height;
} CUDA_MEMCPY2D;

```

where:

- srcMemoryType and dstMemoryType specify the type of memory of the source and destination, respectively; [CUMemorytype\\_enum](#) is defined as:

```

typedef enum CUMemorytype_enum {
    CU_MEMORYTYPE_HOST = 0x01,
    CU_MEMORYTYPE_DEVICE = 0x02,
    CU_MEMORYTYPE_ARRAY = 0x03
} CUMemorytype;

```

If srcMemoryType is [CU\\_MEMORYTYPE\\_HOST](#), srcHost and srcPitch specify the (host) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is [CU\\_MEMORYTYPE\\_DEVICE](#), srcDevice and srcPitch specify the (device) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is [CU\\_MEMORYTYPE\\_ARRAY](#), srcArray specifies the handle of the source data. srcHost, srcDevice and srcPitch are ignored.

If dstMemoryType is [CU\\_MEMORYTYPE\\_HOST](#), dstHost and dstPitch specify the (host) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is [CU\\_MEMORYTYPE\\_DEVICE](#), dstDevice and dstPitch specify the (device) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is [CU\\_MEMORYTYPE\\_ARRAY](#), dstArray specifies the handle of the destination data. dstHost, dstDevice and dstPitch are ignored.

- srcXInBytes and srcY specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*)((char*)srcHost+srcY*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
```

For CUDA arrays, `srcXInBytes` must be evenly divisible by the array element size.

- `dstXInBytes` and `dstY` specify the base address of the destination data for the copy.

For host pointers, the base address is

```
void* dstStart = (void*)((char*)dstHost+dstY*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
```

For CUDA arrays, `dstXInBytes` must be evenly divisible by the array element size.

- `WidthInBytes` and `Height` specify the width (in bytes) and height of the 2D copy being performed. Any pitches must be greater than or equal to `WidthInBytes`.

[cuMemcpy2D\(\)](#) returns an error if any pitch is greater than the maximum allowed (`CU_DEVICE_ATTRIBUTE_MAX_PITCH`). [cuMemAllocPitch\(\)](#) passes back pitches that always work with [cuMemcpy2D\(\)](#). On intra-device memory copies (device ? device, CUDA array ? device, CUDA array ? CUDA array), [cuMemcpy2D\(\)](#) may fail for pitches not computed by [cuMemAllocPitch\(\)](#). [cuMemcpy2DUnaligned\(\)](#) does not have this restriction, but may run significantly slower in the cases where [cuMemcpy2D\(\)](#) would have returned an error code.

#### Parameters:

*pCopy* - Parameters for the memory copy

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

#### Note:

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

#### See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

### 3.25.2.12 CUresult cuMemcpy3D (const CUDA\_MEMCPY3D \*pCopy)

Perform a 3D memory copy according to the parameters specified in `pCopy`. The `CUDA_MEMCPY3D` structure is defined as:

```
typedef struct CUDA_MEMCPY3D_st {
    unsigned int srcXInBytes, srcY, srcZ;
    unsigned int srcLOD;
    CUmemorytype srcMemoryType;
    const void *srcHost;
    CUdeviceptr srcDevice;
    CUarray srcArray;
    unsigned int srcPitch; // ignored when src is array
    unsigned int srcHeight; // ignored when src is array; may be 0 if Depth==1

    unsigned int dstXInBytes, dstY, dstZ;
    unsigned int dstLOD;
    CUmemorytype dstMemoryType;
    void *dstHost;
    CUdeviceptr dstDevice;
    CUarray dstArray;
    unsigned int dstPitch; // ignored when dst is array
    unsigned int dstHeight; // ignored when dst is array; may be 0 if Depth==1

    unsigned int WidthInBytes;
    unsigned int Height;
    unsigned int Depth;
} CUDA_MEMCPY3D;
```

where:

- `srcMemoryType` and `dstMemoryType` specify the type of memory of the source and destination, respectively; `CUmemorytype_enum` is defined as:

```
typedef enum CUmemorytype_enum {
    CU_MEMORYTYPE_HOST = 0x01,
    CU_MEMORYTYPE_DEVICE = 0x02,
    CU_MEMORYTYPE_ARRAY = 0x03
} CUmemorytype;
```

If `srcMemoryType` is `CU_MEMORYTYPE_HOST`, `srcHost`, `srcPitch` and `srcHeight` specify the (host) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. `srcArray` is ignored.

If `srcMemoryType` is `CU_MEMORYTYPE_DEVICE`, `srcDevice`, `srcPitch` and `srcHeight` specify the (device) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. `srcArray` is ignored.

If `srcMemoryType` is `CU_MEMORYTYPE_ARRAY`, `srcArray` specifies the handle of the source data. `srcHost`, `srcDevice`, `srcPitch` and `srcHeight` are ignored.

If `dstMemoryType` is `CU_MEMORYTYPE_HOST`, `dstHost` and `dstPitch` specify the (host) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. `dstArray` is ignored.

If `dstMemoryType` is `CU_MEMORYTYPE_DEVICE`, `dstDevice` and `dstPitch` specify the (device) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. `dstArray` is ignored.

If `dstMemoryType` is [CU\\_MEMORYTYPE\\_ARRAY](#), `dstArray` specifies the handle of the destination data. `dstHost`, `dstDevice`, `dstPitch` and `dstHeight` are ignored.

- `srcXInBytes`, `srcY` and `srcZ` specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*)((char*)srcHost+(srcZ*srcHeight+srcY)*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice+(srcZ*srcHeight+srcY)*srcPitch+srcXInBytes;
```

For CUDA arrays, `srcXInBytes` must be evenly divisible by the array element size.

- `dstXInBytes`, `dstY` and `dstZ` specify the base address of the destination data for the copy.

For host pointers, the base address is

```
void* dstStart = (void*)((char*)dstHost+(dstZ*dstHeight+dstY)*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr dstStart = dstDevice+(dstZ*dstHeight+dstY)*dstPitch+dstXInBytes;
```

For CUDA arrays, `dstXInBytes` must be evenly divisible by the array element size.

- `WidthInBytes`, `Height` and `Depth` specify the width (in bytes), height and depth of the 3D copy being performed. Any pitches must be greater than or equal to `WidthInBytes`.

[cuMemcpy3D\(\)](#) returns an error if any pitch is greater than the maximum allowed ([CU\\_DEVICE\\_ATTRIBUTE\\_MAX\\_PITCH](#)).

The `srcLOD` and `dstLOD` members of the [CUDA\\_MEMCPY3D](#) structure must be set to 0.

#### Parameters:

*pCopy* - Parameters for the memory copy

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

#### Note:

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

**3.25.2.13 CUresult cuMemcpy3DAsync (const CUDA\_MEMCPY3D \*pCopy, CUstream hStream)**

Perform a 3D memory copy according to the parameters specified in pCopy. The [CUDA\\_MEMCPY3D](#) structure is defined as:

```
typedef struct CUDA_MEMCPY3D_st {
    unsigned int srcXInBytes, srcY, srcZ;
    unsigned int srcLOD;
    CUMemorytype srcMemoryType;
    const void *srcHost;
    CUdeviceptr srcDevice;
    CUarray srcArray;
    unsigned int srcPitch; // ignored when src is array
    unsigned int srcHeight; // ignored when src is array; may be 0 if Depth==1

    unsigned int dstXInBytes, dstY, dstZ;
    unsigned int dstLOD;
    CUMemorytype dstMemoryType;
    void *dstHost;
    CUdeviceptr dstDevice;
    CUarray dstArray;
    unsigned int dstPitch; // ignored when dst is array
    unsigned int dstHeight; // ignored when dst is array; may be 0 if Depth==1

    unsigned int WidthInBytes;
    unsigned int Height;
    unsigned int Depth;
} CUDA_MEMCPY3D;
```

where:

- srcMemoryType and dstMemoryType specify the type of memory of the source and destination, respectively; [CUMemorytype\\_enum](#) is defined as:

```
typedef enum CUMemorytype_enum {
    CU_MEMORYTYPE_HOST = 0x01,
    CU_MEMORYTYPE_DEVICE = 0x02,
    CU_MEMORYTYPE_ARRAY = 0x03
} CUMemorytype;
```

If srcMemoryType is [CU\\_MEMORYTYPE\\_HOST](#), srcHost, srcPitch and srcHeight specify the (host) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. srcArray is ignored.

If srcMemoryType is [CU\\_MEMORYTYPE\\_DEVICE](#), srcDevice, srcPitch and srcHeight specify the (device) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. srcArray is ignored.

If `srcMemoryType` is `CU_MEMORYTYPE_ARRAY`, `srcArray` specifies the handle of the source data. `srcHost`, `srcDevice`, `srcPitch` and `srcHeight` are ignored.

If `dstMemoryType` is `CU_MEMORYTYPE_HOST`, `dstHost` and `dstPitch` specify the (host) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. `dstArray` is ignored.

If `dstMemoryType` is `CU_MEMORYTYPE_DEVICE`, `dstDevice` and `dstPitch` specify the (device) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. `dstArray` is ignored.

If `dstMemoryType` is `CU_MEMORYTYPE_ARRAY`, `dstArray` specifies the handle of the destination data. `dstHost`, `dstDevice`, `dstPitch` and `dstHeight` are ignored.

- `srcXInBytes`, `srcY` and `srcZ` specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*)((char*)srcHost+(srcZ*srcHeight+srcY)*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice+(srcZ*srcHeight+srcY)*srcPitch+srcXInBytes;
```

For CUDA arrays, `srcXInBytes` must be evenly divisible by the array element size.

- `dstXInBytes`, `dstY` and `dstZ` specify the base address of the destination data for the copy.

For host pointers, the base address is

```
void* dstStart = (void*)((char*)dstHost+(dstZ*dstHeight+dstY)*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr dstStart = dstDevice+(dstZ*dstHeight+dstY)*dstPitch+dstXInBytes;
```

For CUDA arrays, `dstXInBytes` must be evenly divisible by the array element size.

- `WidthInBytes`, `Height` and `Depth` specify the width (in bytes), height and depth of the 3D copy being performed. Any pitches must be greater than or equal to `WidthInBytes`.

`cuMemcpy3D()` returns an error if any pitch is greater than the maximum allowed (`CU_DEVICE_ATTRIBUTE_MAX_PITCH`).

`cuMemcpy3DAsync()` is asynchronous and can optionally be associated to a stream by passing a non-zero `hStream` argument. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input.

The `srcLOD` and `dstLOD` members of the `CUDA_MEMCPY3D` structure must be set to 0.



**Parameters:**

*pCopy* - Parameters for the memory copy

*hStream* - Stream identifier

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

**3.25.2.14 CUresult cuMemcpyAtoA (CUarray *hDst*, unsigned int *DstOffset*, CUarray *hSrc*, unsigned int *SrcOffset*, unsigned int *NumBytes*)**

Copies from one 1D CUDA array to another. *hDst* and *hSrc* specify the handles of the destination and source CUDA arrays for the copy, respectively. *DstOffset* and *SrcOffset* specify the destination and source indices into the CUDA array. These values are in the range **[0, Width-1]** for the CUDA array; they are not byte offsets. *NumBytes* is the number of bytes to be copied. The size of the elements in the CUDA arrays need not be the same format, but the elements must be the same size; and count must be evenly divisible by that size.

**Parameters:**

*hDst* - Destination array

*DstOffset* - Offset of destination array

*hSrc* - Source array

*SrcOffset* - Offset of source array

*NumBytes* - Size of memory copy in bytes

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned,

[cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

### 3.25.2.15 CUresult cuMemcpyAtoD (CUdeviceptr *dptr*, CUarray *hSrc*, unsigned int *SrcOffset*, unsigned int *NumBytes*)

Copies from one 1D CUDA array to device memory. *dptr* specifies the base pointer of the destination and must be naturally aligned with the CUDA array elements. *hSrc* and *SrcOffset* specify the CUDA array handle and the index (in array elements) of the array element where the copy is to begin. *NumBytes* specifies the number of bytes to copy and must be evenly divisible by the array element size.

#### Parameters:

*dptr* - Destination device pointer  
*hSrc* - Source array  
*SrcOffset* - Offset of source array  
*NumBytes* - Size of memory copy in bytes

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

#### Note:

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

#### See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

### 3.25.2.16 CUresult cuMemcpyAtoH (void \* *pDst*, CUarray *hSrc*, unsigned int *srcOffset*, unsigned int *bytes*)

Copies from one 1D CUDA array to host memory. *pDst* specifies the base pointer of the destination. *hSrc* and *srcOffset* specify the CUDA array handle and starting index of the source data. *bytes* specifies the number of bytes to copy.

#### Parameters:

*pDst* - Destination device pointer  
*hSrc* - Source array  
*srcOffset* - Offset of source array

*bytes* - Size of memory copy in bytes

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

### 3.25.2.17 CUresult cuMemcpyAtoHAsync (void \* *pDst*, CUarray *hSrc*, unsigned int *srcOffset*, unsigned int *bytes*, CUstream *hStream*)

Copies from one 1D CUDA array to host memory. *pDst* specifies the base pointer of the destination. *hSrc* and *srcOffset* specify the CUDA array handle and starting index of the source data. *bytes* specifies the number of bytes to copy.

[cuMemcpyAtoHAsync\(\)](#) is asynchronous and can optionally be associated to a stream by passing a non-zero *stream* argument. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input.

**Parameters:**

*pDst* - Destination device pointer  
*hSrc* - Source array  
*srcOffset* - Offset of source array  
*bytes* - Size of memory copy in bytes  
*hStream* - Stream identifier

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

### 3.25.2.18 CUresult cuMemcpyDtoA (CUarray *hDst*, unsigned int *DstOffset*, CUdeviceptr *dptr*, unsigned int *NumBytes*)

Copies from device memory to a 1D CUDA array. *hDst* and *DstOffset* specify the CUDA array handle and starting index of the destination data. *dptr* specifies the base pointer of the source. *NumBytes* specifies the number of bytes to copy.

#### Parameters:

- hDst* - Destination array
- DstOffset* - Offset of destination array
- dptr* - Source device pointer
- NumBytes* - Size of memory copy in bytes

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

#### Note:

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

#### See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

### 3.25.2.19 CUresult cuMemcpyDtoD (CUdeviceptr *dst*, CUdeviceptr *src*, unsigned int *bytes*)

Copies from device memory to device memory. *dst* and *src* are the base pointers of the destination and source, respectively. *bytes* specifies the number of bytes to copy. Note that this function is asynchronous.

#### Parameters:

- dst* - Destination device pointer
- src* - Source device pointer
- bytes* - Size of memory copy in bytes

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

#### Note:

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

**3.25.2.20 CUresult cuMemcpyDtoH (void \* *dst*, CUdeviceptr *dptr*, unsigned int *bytes*)**

Copies from device to host memory. *dst* and *dptr* specify the base pointers of the destination and source, respectively. *bytes* specifies the number of bytes to copy. Note that this function is synchronous.

**Parameters:**

*dst* - Destination host pointer  
*dptr* - Source device pointer  
*bytes* - Size of memory copy in bytes

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**Note:**

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

**3.25.2.21 CUresult cuMemcpyDtoHAsync (void \* *dst*, CUdeviceptr *dptr*, unsigned int *bytes*, CUstream *hStream*)**

Copies from device to host memory. *dst* and *dptr* specify the base pointers of the destination and source, respectively. *bytes* specifies the number of bytes to copy.

[cuMemcpyDtoHAsync\(\)](#) is asynchronous and can optionally be associated to a stream by passing a non-zero *hStream* argument. It only works on page-locked memory and returns an error if a pointer to pageable memory is passed as input.

**Parameters:**

*dst* - Destination host pointer  
*dptr* - Source device pointer

*bytes* - Size of memory copy in bytes

*hStream* - Stream identifier

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

**3.25.2.22 CUresult cuMemcpyHtoA (CUarray *hDst*, unsigned int *dstOffset*, const void \* *pSrc*, unsigned int *bytes*)**

Copies from host memory to a 1D CUDA array. *hDst* and *dstOffset* specify the CUDA array handle and starting index of the destination data. *pSrc* specifies the base address of the source. *bytes* specifies the number of bytes to copy.

**Parameters:**

*hDst* - Destination array

*dstOffset* - Offset of destination array

*pSrc* - Source host pointer

*bytes* - Size of memory copy in bytes

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

**3.25.2.23 CUresult cuMemcpyHtoAAsync (CUarray *hDst*, unsigned int *dstOffset*, const void \* *pSrc*, unsigned int *bytes*, CUstream *hStream*)**

Copies from host memory to a 1D CUDA array. *hDst* and *dstOffset* specify the CUDA array handle and starting index of the destination data. *pSrc* specifies the base address of the source. *bytes* specifies the number of bytes to copy.

[cuMemcpyHtoAAsync\(\)](#) is asynchronous and can optionally be associated to a stream by passing a non-zero *hStream* argument. It only works on page-locked memory and returns an error if a pointer to pageable memory is passed as input.

**Parameters:**

*hDst* - Destination array  
*dstOffset* - Offset of destination array  
*pSrc* - Source host pointer  
*bytes* - Size of memory copy in bytes  
*hStream* - Stream identifier

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**Note:**

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

**3.25.2.24 CUresult cuMemcpyHtoD (CUdeviceptr *dptr*, const void \* *src*, unsigned int *bytes*)**

Copies from host memory to device memory. *dptr* and *src* are the base addresses of the destination and source, respectively. *bytes* specifies the number of bytes to copy. Note that this function is synchronous.

**Parameters:**

*dptr* - Destination device pointer  
*src* - Source host pointer  
*bytes* - Size of memory copy in bytes

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**Note:**

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

### 3.25.2.25 CUresult cuMemcpyHtoDAsync (CUdeviceptr *dptr*, const void \**src*, unsigned int *bytes*, CUstream *hStream*)

Copies from host memory to device memory. *dst* and *src* are the base addresses of the destination and source, respectively. *bytes* specifies the number of bytes to copy.

[cuMemcpyHtoDAsync\(\)](#) is asynchronous and can optionally be associated to a stream by passing a non-zero *hStream* argument. It only works on page-locked memory and returns an error if a pointer to pageable memory is passed as input.

**Parameters:**

*dptr* - Destination device pointer  
*src* - Source host pointer  
*bytes* - Size of memory copy in bytes  
*hStream* - Stream identifier

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**Note:**

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

### 3.25.2.26 CUresult cuMemFree (CUdeviceptr *dptr*)

Frees the memory space pointed to by *dptr*, which must have been returned by a previous call to [cuMemAlloc\(\)](#) or [cuMemAllocPitch\(\)](#).



**Parameters:**

*dptr* - Pointer to memory to free

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

**3.25.2.27 CUresult cuMemFreeHost (void \* p)**

Frees the memory space pointed to by *p*, which must have been returned by a previous call to [cuMemAllocHost\(\)](#).

**Parameters:**

*p* - Pointer to memory to free

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

**3.25.2.28 CUresult cuMemGetAddressRange (CUdeviceptr \* pdptr, unsigned int \* psize, CUdeviceptr dptr)**

Returns the base address in *\*pdptr* and size in *\*psize* of the allocation by [cuMemAlloc\(\)](#) or [cuMemAllocPitch\(\)](#) that contains the input pointer *dptr*. Both parameters *pdptr* and *psize* are optional. If one of them is NULL, it is ignored.

**Parameters:**

*pdptr* - Returned base address  
*psize* - Returned size of device memory allocation  
*dptr* - Device pointer to query

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

**3.25.2.29 CUresult cuMemGetInfo (unsigned int \*free, unsigned int \*total)**

Returns in *\*free* and *\*total* respectively, the free and total amount of memory available for allocation by the CUDA context, in bytes.

**Parameters:**

*free* - Returned free memory in bytes  
*total* - Returned total memory in bytes

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

### 3.25.2.30 CUresult cuMemHostAlloc (void \*\* *pp*, size\_t *bytes*, unsigned int *Flags*)

Allocates *bytes* bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as [cuMemcpyHtoD\(\)](#). Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as `malloc()`. 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.

- [CU\\_MEMHOSTALLOC\\_PORTABLE](#): The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
- [CU\\_MEMHOSTALLOC\\_DEVICEMAP](#): Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling [cuMemHostGetDevicePointer\(\)](#). This feature is available only on GPUs with compute capability greater than or equal to 1.1.
- [CU\\_MEMHOSTALLOC\\_WRITECOMBINED](#): Allocates the memory as write-combined (WC). WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs. WC memory is a good option for buffers that will be written by the CPU and read by the GPU via mapped pinned memory or host->device transfers.

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

The CUDA context must have been created with the [CU\\_CTX\\_MAP\\_HOST](#) flag in order for the [CU\\_MEMHOSTALLOC\\_MAPPED](#) flag to have any effect.

The [CU\\_MEMHOSTALLOC\\_MAPPED](#) flag may be specified on CUDA contexts for devices that do not support mapped pinned memory. The failure is deferred to [cuMemHostGetDevicePointer\(\)](#) because the memory may be mapped into other CUDA contexts via the [CU\\_MEMHOSTALLOC\\_PORTABLE](#) flag.

The memory allocated by this function must be freed with [cuMemFreeHost\(\)](#).

#### Parameters:

- pp* - Returned host pointer to page-locked memory
- bytes* - Requested allocation size in bytes
- Flags* - Flags for allocation request

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_OUT\\_OF\\_MEMORY](#)

#### Note:

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

#### See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

### 3.25.2.31 CUresult cuMemHostGetDevicePointer (CUdeviceptr \* *ret*, void \* *p*, unsigned int *Flags*)

Passes back the device pointer `ret` corresponding to the mapped, pinned host buffer `p` allocated by `cuMemHostAlloc`. `cuMemHostGetDevicePointer()` will fail if the `CU_MEMALLOCHOST_DEVICEMAP` flag was not specified at the time the memory was allocated, or if the function is called on a GPU that does not support mapped pinned memory.

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

#### Parameters:

*ret* - Returned device pointer

*p* - Host pointer

*Flags* - Options (must be 0)

#### Returns:

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

#### Note:

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

#### See also:

`cuArray3DCreate`, `cuArray3DGetDescriptor`, `cuArrayCreate`, `cuArrayDestroy`, `cuArrayGetDescriptor`, `cuMemAlloc`, `cuMemAllocHost`, `cuMemAllocPitch`, `cuMemcpy2D`, `cuMemcpy2DAsync`, `cuMemcpy2DUnaligned`, `cuMemcpy3D`, `cuMemcpy3DAsync`, `cuMemcpyAtoA`, `cuMemcpyAtoD`, `cuMemcpyAtoH`, `cuMemcpyAtoHAsync`, `cuMemcpyDtoA`, `cuMemcpyDtoD`, `cuMemcpyDtoH`, `cuMemcpyDtoHAsync`, `cuMemcpyHtoA`, `cuMemcpyHtoAAsync`, `cuMemcpyHtoD`, `cuMemcpyHtoDAsync`, `cuMemFree`, `cuMemFreeHost`, `cuMemGetAddressRange`, `cuMemGetInfo`, `cuMemHostAlloc`, `cuMemsetD2D8`, `cuMemsetD2D16`, `cuMemsetD2D32`, `cuMemsetD8`, `cuMemsetD16`, `cuMemsetD32`

### 3.25.2.32 CUresult cuMemsetD16 (CUdeviceptr *dstDevice*, unsigned short *us*, unsigned int *N*)

Sets the memory range of `N` 16-bit values to the specified value `us`.

#### Parameters:

*dstDevice* - Destination device pointer

*us* - Value to set

*N* - Number of elements

#### Returns:

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

#### Note:

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

#### See also:

`cuArray3DCreate`, `cuArray3DGetDescriptor`, `cuArrayCreate`, `cuArrayDestroy`, `cuArrayGetDescriptor`, `cuMemAlloc`, `cuMemAllocHost`, `cuMemAllocPitch`, `cuMemcpy2D`, `cuMemcpy2DAsync`, `cuMemcpy2DUnaligned`,

[cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD32](#)

### 3.25.2.33 CUresult cuMemsetD2D16 (CUdeviceptr *dstDevice*, unsigned int *dstPitch*, unsigned short *us*, unsigned int *Width*, unsigned int *Height*)

Sets the 2D memory range of *Width* 16-bit values to the specified value *us*. *Height* specifies the number of rows to set, and *dstPitch* specifies the number of bytes between each row. This function performs fastest when the pitch is one that has been passed back by [cuMemAllocPitch\(\)](#).

#### Parameters:

*dstDevice* - Destination device pointer  
*dstPitch* - Pitch of destination device pointer  
*us* - Value to set  
*Width* - Width of row  
*Height* - Number of rows

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

#### Note:

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

#### See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

### 3.25.2.34 CUresult cuMemsetD2D32 (CUdeviceptr *dstDevice*, unsigned int *dstPitch*, unsigned int *ui*, unsigned int *Width*, unsigned int *Height*)

Sets the 2D memory range of *Width* 32-bit values to the specified value *ui*. *Height* specifies the number of rows to set, and *dstPitch* specifies the number of bytes between each row. This function performs fastest when the pitch is one that has been passed back by [cuMemAllocPitch\(\)](#).

#### Parameters:

*dstDevice* - Destination device pointer  
*dstPitch* - Pitch of destination device pointer

*ui* - Value to set  
*Width* - Width of row  
*Height* - Number of rows

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**Note:**

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

### 3.25.2.35 CUresult cuMemsetD2D8 (CUdeviceptr *dstDevice*, unsigned int *dstPitch*, unsigned char *uc*, unsigned int *Width*, unsigned int *Height*)

Sets the 2D memory range of *Width* 8-bit values to the specified value *uc*. *Height* specifies the number of rows to set, and *dstPitch* specifies the number of bytes between each row. This function performs fastest when the pitch is one that has been passed back by [cuMemAllocPitch\(\)](#).

**Parameters:**

*dstDevice* - Destination device pointer  
*dstPitch* - Pitch of destination device pointer  
*uc* - Value to set  
*Width* - Width of row  
*Height* - Number of rows

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**Note:**

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

**3.25.2.36 CUresult cuMemsetD32 (CUdeviceptr *dstDevice*, unsigned int *ui*, unsigned int *N*)**

Sets the memory range of  $N$  32-bit values to the specified value *ui*.

**Parameters:**

*dstDevice* - Destination device pointer

*ui* - Value to set

*N* - Number of elements

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16

**3.25.2.37 CUresult cuMemsetD8 (CUdeviceptr *dstDevice*, unsigned char *c*, unsigned int *N*)**

Sets the memory range of  $N$  8-bit values to the specified value *c*.

**Parameters:**

*dstDevice* - Destination device pointer

*c* - Value to set

*N* - Number of elements

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD16, cuMemsetD32

## 3.26 Texture Reference Management

### Functions

- [CUresult cuTexRefCreate](#) ([CUtexref](#) \*phTexRef)  
*Creates a texture reference.*
- [CUresult cuTexRefDestroy](#) ([CUtexref](#) hTexRef)  
*Destroys a texture reference.*
- [CUresult cuTexRefGetAddress](#) ([CUdeviceptr](#) \*pdptr, [CUtexref](#) hTexRef)  
*Gets the address associated with a texture reference.*
- [CUresult cuTexRefGetAddressMode](#) ([CUaddress\\_mode](#) \*pAddressMode, [CUtexref](#) hTexRef, int Dim)  
*Gets the addressing mode used by a texture reference.*
- [CUresult cuTexRefGetArray](#) ([CUarray](#) \*phArray, [CUtexref](#) hTexRef)  
*Gets the array bound to a texture reference.*
- [CUresult cuTexRefGetFilterMode](#) ([CUfilter\\_mode](#) \*pFilterMode, [CUtexref](#) hTexRef)  
*Gets the filter-mode used by a texture reference.*
- [CUresult cuTexRefGetFlags](#) (unsigned int \*pFlags, [CUtexref](#) hTexRef)  
*Gets the flags used by a texture reference.*
- [CUresult cuTexRefGetFormat](#) ([CUarray\\_format](#) \*pFormat, int \*pNumPackedComponents, [CUtexref](#) hTexRef)  
*Gets the format used by a texture reference.*
- [CUresult cuTexRefSetAddress](#) (unsigned int \*pOffset, [CUtexref](#) hTexRef, [CUdeviceptr](#) dptr, unsigned int bytes)  
*Binds an address as a texture reference.*
- [CUresult cuTexRefSetAddress2D](#) ([CUtexref](#) hTexRef, const [CUDA\\_ARRAY\\_DESCRIPTOR](#) \*desc, [CUdeviceptr](#) dptr, unsigned int PitchInBytes)  
*Binds an address as a 2D texture reference.*
- [CUresult cuTexRefSetAddressMode](#) ([CUtexref](#) hTexRef, int Dim, [CUaddress\\_mode](#) am)  
*Sets the addressing mode for a texture reference.*
- [CUresult cuTexRefSetArray](#) ([CUtexref](#) hTexRef, [CUarray](#) hArray, unsigned int Flags)  
*Binds an address as a texture reference.*
- [CUresult cuTexRefSetFilterMode](#) ([CUtexref](#) hTexRef, [CUfilter\\_mode](#) fm)  
*Sets the filtering mode for a texture reference.*
- [CUresult cuTexRefSetFlags](#) ([CUtexref](#) hTexRef, unsigned int Flags)  
*Sets the flags for a texture reference.*
- [CUresult cuTexRefSetFormat](#) ([CUtexref](#) hTexRef, [CUarray\\_format](#) Format, int NumChannels)  
*Sets the format for a texture reference.*



### 3.26.1 Detailed Description

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

### 3.26.2 Function Documentation

#### 3.26.2.1 CUresult cuTexRefCreate (CUtexref \* *phTexRef*)

Creates a texture reference and returns its handle in \**phTexRef*. Once created, the application must call [cuTexRefSetArray\(\)](#) or [cuTexRefSetAddress\(\)](#) to associate the reference with allocated memory. Other texture reference functions are used to specify the format and interpretation (addressing, filtering, etc.) to be used when the memory is read through this texture reference. To associate the texture reference with a texture ordinal for a given function, the application should call [cuParamSetTexRef\(\)](#).

**Parameters:**

*phTexRef* - Returned texture reference

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**See also:**

[cuTexRefDestroy](#), [cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

#### 3.26.2.2 CUresult cuTexRefDestroy (CUtexref *hTexRef*)

Destroys the texture reference specified by *hTexRef*.

**Parameters:**

*hTexRef* - Texture reference to destroy

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**See also:**

[cuTexRefCreate](#), [cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

#### 3.26.2.3 CUresult cuTexRefGetAddress (CUdeviceptr \* *pdptr*, CUtexref *hTexRef*)

Returns in \**pdptr* the base address bound to the texture reference *hTexRef*, or returns [CUDA\\_ERROR\\_INVALID\\_VALUE](#) if the texture reference is not bound to any device memory range.

**Parameters:**

*pdptr* - Returned device address  
*hTexRef* - Texture reference

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**See also:**

[cuTexRefCreate](#), [cuTexRefDestroy](#), [cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

**3.26.2.4 CUresult cuTexRefGetAddressMode (CUaddress\_mode \*pAddressMode, CUtexref hTexRef, int Dim)**

Returns in \*pAddressMode the addressing mode corresponding to the dimension Dim of the texture reference hTexRef. Currently, the only valid value for Dim are 0 and 1.

**Parameters:**

*pAddressMode* - Returned addressing mode  
*hTexRef* - Texture reference  
*Dim* - Dimension

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**See also:**

[cuTexRefCreate](#), [cuTexRefDestroy](#), [cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

**3.26.2.5 CUresult cuTexRefGetArray (CUarray \*phArray, CUtexref hTexRef)**

Returns in \*phArray the CUDA array bound to the texture reference hTexRef, or returns CUDA\_ERROR\_INVALID\_VALUE if the texture reference is not bound to any CUDA array.

**Parameters:**

*phArray* - Returned array  
*hTexRef* - Texture reference

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

See also:

[cuTexRefCreate](#), [cuTexRefDestroy](#), [cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

### 3.26.2.6 CUresult cuTexRefGetFilterMode (CUfilter\_mode \* *pFilterMode*, CUtexref *hTexRef*)

Returns in \**pFilterMode* the filtering mode of the texture reference *hTexRef*.

Parameters:

*pFilterMode* - Returned filtering mode

*hTexRef* - Texture reference

Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

See also:

[cuTexRefCreate](#), [cuTexRefDestroy](#), [cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

### 3.26.2.7 CUresult cuTexRefGetFlags (unsigned int \* *pFlags*, CUtexref *hTexRef*)

Returns in \**pFlags* the flags of the texture reference *hTexRef*.

Parameters:

*pFlags* - Returned flags

*hTexRef* - Texture reference

Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

See also:

[cuTexRefCreate](#), [cuTexRefDestroy](#), [cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFormat](#)

### 3.26.2.8 CUresult cuTexRefGetFormat (CUarray\_format \* *pFormat*, int \* *pNumPackedComponents*, CUtexref *hTexRef*)

Returns in \**pFormat* and \**pNumPackedComponents* the format and number of components of the CUDA array bound to the texture reference *hTexRef*. If *pFormat* or *pNumPackedComponents* is NULL, it will be ignored.

**Parameters:**

*pFormat* - Returned format  
*pNumPackedComponents* - Returned number of components  
*hTexRef* - Texture reference

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**See also:**

[cuTexRefCreate](#), [cuTexRefDestroy](#), [cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#)

### 3.26.2.9 CUresult cuTexRefSetAddress (unsigned int \*pOffset, CUtexref hTexRef, CUdeviceptr dptr, unsigned int bytes)

Binds a linear address range to the texture reference `hTexRef`. Any previous address or CUDA array state associated with the texture reference is superseded by this function. Any memory previously bound to `hTexRef` is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, [cuTexRefSetAddress\(\)](#) passes back a byte offset in `*pOffset` that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the `tex1Dfetch()` function.

If the device memory pointer was returned from [cuMemAlloc\(\)](#), the offset is guaranteed to be 0 and NULL may be passed as the `pOffset` parameter.

**Parameters:**

*pOffset* - Returned byte offset  
*hTexRef* - Texture reference to bind  
*dptr* - Device pointer to bind  
*bytes* - Size of memory to bind in bytes

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**See also:**

[cuTexRefCreate](#), [cuTexRefDestroy](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

### 3.26.2.10 CUresult cuTexRefSetAddress2D (CUtexref hTexRef, const CUDA\_ARRAY\_DESCRIPTOR \*desc, CUdeviceptr dptr, unsigned int PitchInBytes)

Binds a linear address range to the texture reference `hTexRef`. Any previous address or CUDA array state associated with the texture reference is superseded by this function. Any memory previously bound to `hTexRef` is unbound.

When using a `tex2D()` function inside a kernel, we have to either call `cuTexRefSetArray()` to bind the corresponding texture reference to an array, or `cuTexRefSetAddress2D()` to bind the texture reference to linear memory.

Function calls to `cuTexRefSetFormat()` cannot follow calls to `cuTexRefSetAddress2D()` for the same texture reference.

**Parameters:**

*hTexRef* - Texture reference to bind

*desc* - Descriptor of CUDA array

*dptr* - Device pointer to bind

*PitchInBytes* - Line pitch in bytes

**Returns:**

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

**See also:**

`cuTexRefCreate`, `cuTexRefDestroy`, `cuTexRefSetAddress`, `cuTexRefSetAddressMode`, `cuTexRefSetArray`, `cuTexRefSetFilterMode`, `cuTexRefSetFlags`, `cuTexRefSetFormat`, `cuTexRefGetAddress`, `cuTexRefGetAddressMode`, `cuTexRefGetArray`, `cuTexRefGetFilterMode`, `cuTexRefGetFlags`, `cuTexRefGetFormat`

### 3.26.2.11 CUresult cuTexRefSetAddressMode (CUtexref hTexRef, int Dim, CUaddress\_mode am)

Specifies the addressing mode `am` for the given dimension `Dim` of the texture reference `hTexRef`. If `Dim` is zero, the addressing mode is applied to the first parameter of the functions used to fetch from the texture; if `Dim` is 1, the second, and so on. `CUaddress_mode` is defined as:

```
typedef enum CUaddress_mode_enum {
    CU_TR_ADDRESS_MODE_WRAP = 0,
    CU_TR_ADDRESS_MODE_CLAMP = 1,
    CU_TR_ADDRESS_MODE_MIRROR = 2,
} CUaddress_mode;
```

Note that this call has no effect if `hTexRef` is bound to linear memory.

**Parameters:**

*hTexRef* - Texture reference

*Dim* - Dimension

*am* - Addressing mode to set

**Returns:**

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

**See also:**

`cuTexRefCreate`, `cuTexRefDestroy`, `cuTexRefSetAddress`, `cuTexRefSetAddress2D`, `cuTexRefSetArray`, `cuTexRefSetFilterMode`, `cuTexRefSetFlags`, `cuTexRefSetFormat`, `cuTexRefGetAddress`, `cuTexRefGetAddressMode`, `cuTexRefGetArray`, `cuTexRefGetFilterMode`, `cuTexRefGetFlags`, `cuTexRefGetFormat`

### 3.26.2.12 CUresult cuTexRefSetArray (CUtexref *hTexRef*, CUarray *hArray*, unsigned int *Flags*)

Binds the CUDA array `hArray` to the texture reference `hTexRef`. Any previous address or CUDA array state associated with the texture reference is superseded by this function. `Flags` must be set to `CU_TRSA_OVERRIDE_FORMAT`. Any CUDA array previously bound to `hTexRef` is unbound.

#### Parameters:

- hTexRef* - Texture reference to bind
- hArray* - Array to bind
- Flags* - Options (must be `CU_TRSA_OVERRIDE_FORMAT`)

#### Returns:

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

#### See also:

`cuTexRefCreate`, `cuTexRefDestroy`, `cuTexRefSetAddress`, `cuTexRefSetAddress2D`, `cuTexRefSetAddressMode`, `cuTexRefSetFilterMode`, `cuTexRefSetFlags`, `cuTexRefSetFormat`, `cuTexRefGetAddress`, `cuTexRefGetAddressMode`, `cuTexRefGetArray`, `cuTexRefGetFilterMode`, `cuTexRefGetFlags`, `cuTexRefGetFormat`

### 3.26.2.13 CUresult cuTexRefSetFilterMode (CUtexref *hTexRef*, CUfilter\_mode *fm*)

Specifies the filtering mode `fm` to be used when reading memory through the texture reference `hTexRef`. `CUfilter_mode_enum` is defined as:

```
typedef enum CUfilter_mode_enum {
    CU_TR_FILTER_MODE_POINT = 0,
    CU_TR_FILTER_MODE_LINEAR = 1
} CUfilter_mode;
```

Note that this call has no effect if `hTexRef` is bound to linear memory.

#### Parameters:

- hTexRef* - Texture reference
- fm* - Filtering mode to set

#### Returns:

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

#### See also:

`cuTexRefCreate`, `cuTexRefDestroy`, `cuTexRefSetAddress`, `cuTexRefSetAddress2D`, `cuTexRefSetAddressMode`, `cuTexRefSetArray`, `cuTexRefSetFlags`, `cuTexRefSetFormat`, `cuTexRefGetAddress`, `cuTexRefGetAddressMode`, `cuTexRefGetArray`, `cuTexRefGetFilterMode`, `cuTexRefGetFlags`, `cuTexRefGetFormat`

### 3.26.2.14 CUresult cuTexRefSetFlags (CUtexref *hTexRef*, unsigned int *Flags*)

Specifies optional flags via `Flags` to specify the behavior of data returned through the texture reference `hTexRef`. The valid flags are:

- [CU\\_TRSF\\_READ\\_AS\\_INTEGER](#), which suppresses the default behavior of having the texture promote integer data to floating point data in the range [0, 1];
- [CU\\_TRSF\\_NORMALIZED\\_COORDINATES](#), which suppresses the default behavior of having the texture coordinates range from [0, Dim) where Dim is the width or height of the CUDA array. Instead, the texture coordinates [0, 1.0) reference the entire breadth of the array dimension;

#### Parameters:

*hTexRef* - Texture reference

*Flags* - Optional flags to set

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

#### See also:

[cuTexRefCreate](#), [cuTexRefDestroy](#), [cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

### 3.26.2.15 CUresult cuTexRefSetFormat (CUtexref *hTexRef*, CUarray\_format *Format*, int *NumChannels*)

Specifies the format of the data to be read by the texture reference `hTexRef`. `Format` and `NumChannels` are exactly analogous to the `Format` and `NumChannels` members of the [CUDA\\_ARRAY\\_DESCRIPTOR](#) structure: They specify the format of each component and the number of components per array element.

#### Parameters:

*hTexRef* - Texture reference

*Format* - Format to set

*NumChannels* - Number of components per array element

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

#### See also:

[cuTexRefCreate](#), [cuTexRefDestroy](#), [cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

## 3.27 OpenGL Interoperability

### Functions

- [CUresult cuGLCtxCreate](#) ([CUcontext](#) \*pCtx, unsigned int Flags, [CUdevice](#) device)  
*Create a CUDA context for interoperability with OpenGL.*
- [CUresult cuGLInit](#) (void)  
*Initializes OpenGL interoperability.*
- [CUresult cuGLMapBufferObject](#) ([CUdeviceptr](#) \*dptr, unsigned int \*size, GLuint bufferObj)  
*Maps an OpenGL buffer object.*
- [CUresult cuGLRegisterBufferObject](#) (GLuint bufferObj)  
*Registers an OpenGL buffer object.*
- [CUresult cuGLUnmapBufferObject](#) (GLuint bufferObj)  
*Unmaps an OpenGL buffer object.*
- [CUresult cuGLUnregisterBufferObject](#) (GLuint bufferObj)  
*Unregister an OpenGL buffer object.*
- [CUresult cuWGLGetDevice](#) ([CUdevice](#) \*pDevice, HGPUNV hGpu)  
*Gets the CUDA device associated with hGpu.*

### 3.27.1 Detailed Description

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

### 3.27.2 Function Documentation

#### 3.27.2.1 [CUresult cuGLCtxCreate](#) ([CUcontext](#) \*pCtx, unsigned int *Flags*, [CUdevice](#) *device*)

Creates a new CUDA context, initializes OpenGL interoperability, and associates the CUDA context with the calling thread. It must be called before performing any other OpenGL interoperability operations. It may fail if the needed OpenGL driver facilities are not available. For usage of the `Flags` parameter, see [cuCtxCreate\(\)](#).

#### Parameters:

- pCtx* - Returned CUDA context
- Flags* - Options for CUDA context creation
- device* - Device on which to create the context

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_OUT\\_OF\\_MEMORY](#)



**Note:**

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

**See also:**

[cuCtxCreate](#), [cuGLInit](#), [cuGLMapBufferObject](#), [cuGLRegisterBufferObject](#), [cuGLUnmapBufferObject](#), [cuGLUnregisterBufferObject](#), [cuWGLGetDevice](#)

**3.27.2.2 CUresult cuGLInit (void)**

Initializes OpenGL interoperability. It must be called before performing any other OpenGL interoperability operations. It may fail if the needed OpenGL driver facilities are not available.

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_UNKNOWN](#)

**Note:**

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

**See also:**

[cuGLCtxCreate](#), [cuGLMapBufferObject](#), [cuGLRegisterBufferObject](#), [cuGLUnmapBufferObject](#), [cuGLUnregisterBufferObject](#), [cuWGLGetDevice](#)

**3.27.2.3 CUresult cuGLMapBufferObject (CUdeviceptr \* *dptr*, unsigned int \* *size*, GLuint *bufferObj*)**

Maps the buffer object of ID `bufferObj` into the address space of the current CUDA context and returns in `*dptr` and `*size` the base pointer and size of the resulting mapping.

**Parameters:**

*dptr* - Returned mapped base pointer

*size* - Returned size of mapping

*bufferObj* - Buffer object to map

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_MAP\\_FAILED](#)

**Note:**

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

**See also:**

[cuGLCtxCreate](#), [cuGLInit](#), [cuGLRegisterBufferObject](#), [cuGLUnmapBufferObject](#), [cuGLUnregisterBufferObject](#), [cuWGLGetDevice](#)

### 3.27.2.4 CUresult cuGLRegisterBufferObject (GLuint *bufferObj*)

Registers the buffer object of ID `bufferObj` for access by CUDA. This function must be called before CUDA can map the buffer object. While it is registered, the buffer object cannot be used by any OpenGL commands, except as a data source for OpenGL drawing commands.

**Parameters:**

*bufferObj* - Buffer object to register

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_ALREADY\\_MAPPED](#)

**Note:**

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

**See also:**

[cuGLCtxCreate](#), [cuGLInit](#), [cuGLMapBufferObject](#), [cuGLUnmapBufferObject](#), [cuGLUnregisterBufferObject](#), [cuWGLGetDevice](#)

### 3.27.2.5 CUresult cuGLUnmapBufferObject (GLuint *bufferObj*)

Unmaps the buffer object of ID `bufferObj` for access by CUDA.

**Parameters:**

*bufferObj* - Buffer object to unmap

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#)

**Note:**

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

**See also:**

[cuGLCtxCreate](#), [cuGLInit](#), [cuGLMapBufferObject](#), [cuGLRegisterBufferObject](#), [cuGLUnregisterBufferObject](#), [cuWGLGetDevice](#)

### 3.27.2.6 CUresult cuGLUnregisterBufferObject (GLuint *bufferObj*)

Unregisters the buffer object of ID `bufferObj` for access by CUDA.

**Parameters:**

*bufferObj* - Buffer object to unmap

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

cuGLCtxCreate, cuGLInit, cuGLMapBufferObject, cuGLRegisterBufferObject, cuGLUnmapBufferObject, cuWGLGetDevice

**3.27.2.7 CUresult cuWGLGetDevice (CUdevice \* *pDevice*, HGPUNV *hGpu*)**

Returns in \**pDevice* the CUDA device associated with a *hGpu*, if applicable.

**Parameters:**

*pDevice* - Device associated with *hGpu*

*hGpu* - Handle to a GPU, as queried via WGL\_NV\_gpu\_affinity()

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

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

**See also:**

cuGLCtxCreate, cuGLInit, cuGLMapBufferObject, cuGLRegisterBufferObject, cuGLUnmapBufferObject, cuGLUnregisterBufferObject

## 3.28 Direct3D 9 Interoperability

### Functions

- **CUresult cuD3D9CtxCreate** (CUcontext \*pCtx, CUdevice \*pCuDevice, unsigned int flags, IDirect3DDevice9 \*pDxDevice)  
*Create a CUDA context for interoperability with Direct3D.*
- **CUresult cuD3D9GetDevice** (CUdevice \*pDevice, const char \*pszAdapterName)  
*Gets the device number for an adapter.*
- **CUresult cuD3D9GetDirect3DDevice** (IDirect3DDevice9 \*\*ppDxDevice)  
*Get the Direct3D device against which the current CUDA context was created.*
- **CUresult cuD3D9MapResources** (unsigned int count, IDirect3DResource9 \*\*ppResources)  
*Map Direct3D resources for access by CUDA.*
- **CUresult cuD3D9RegisterResource** (IDirect3DResource9 \*pResource, unsigned int flags)  
*Register a Direct3D resource for access by CUDA.*
- **CUresult cuD3D9ResourceGetMappedArray** (CUarray \*pArray, IDirect3DResource9 \*pResource, U32 face, U32 level)  
*Get an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- **CUresult cuD3D9ResourceGetMappedPitch** (unsigned int \*pPitch, unsigned int \*pPitchSlice, IDirect3DResource9 \*pResource, U32 face, U32 level)  
*Get the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- **CUresult cuD3D9ResourceGetMappedPointer** (CUdeviceptr \*pDevPtr, IDirect3DResource9 \*pResource, U32 face, U32 level)  
*Get the pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- **CUresult cuD3D9ResourceGetMappedSize** (unsigned int \*pSize, IDirect3DResource9 \*pResource, U32 face, U32 level)  
*Get the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- **CUresult cuD3D9ResourceGetSurfaceDimensions** (unsigned int \*pWidth, unsigned int \*pHeight, unsigned int \*pDepth, IDirect3DResource9 \*pResource, unsigned int face, unsigned int level)  
*Get the dimensions of a registered surface.*
- **CUresult cuD3D9ResourceSetMapFlags** (IDirect3DResource9 \*pResource, unsigned int flags)  
*Set usage flags for mapping a Direct3D resource.*
- **CUresult cuD3D9UnmapResources** (unsigned int count, IDirect3DResource9 \*\*ppResources)  
*Unmaps Direct3D resources.*
- **CUresult cuD3D9UnregisterResource** (IDirect3DResource9 \*pResource)  
*Unregister a Direct3D resource.*

### 3.28.1 Detailed Description

This section describes the Direct3D 9 interoperability functions of the low-level CUDA driver application programming interface.

### 3.28.2 Function Documentation

#### 3.28.2.1 **CUresult cuD3D9CtxCreate (CUcontext \*pCtx, CUdevice \*pCuDevice, unsigned int flags, IDirect3DDevice9 \*pDxDevice)**

Creates a new CUDA context, enables interoperability for that context with the Direct3D device `pDxDevice`, and associates the created CUDA context with the calling thread. The **CUcontext** will be returned in `*pCtx`. If `pCuDevice` is non-NULL, then the **CUdevice** on which this CUDA context was created will be returned in `*pCuDevice`. For usage of the `flags` parameter, see [cuCtxCreate\(\)](#). Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context.

This context will function only until its Direct3D device is destroyed. On success, this call will increase the internal reference count on `pDxDevice`. This reference count will be decremented upon destruction of this context through [cuCtxDestroy\(\)](#).

#### Parameters:

- pCtx* - Returned newly created CUDA context
- pCuDevice* - Returned pointer to device on which context was created
- flags* - Context creation flags (see [cuCtxCreate\(\)](#) for details)
- pDxDevice* - Direct3D device to create interoperability context with

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_OUT\\_OF\\_MEMORY](#), [CUDA\\_ERROR\\_UNKNOWN](#)

#### Note:

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

#### See also:

[cuD3D9GetDevice](#), [cuD3D9GetDirect3DDevice](#), [cuD3D9MapResources](#), [cuD3D9RegisterResource](#), [cuD3D9ResourceGetMappedArray](#), [cuD3D9ResourceGetMappedPitch](#), [cuD3D9ResourceGetMappedPointer](#), [cuD3D9ResourceGetMappedSize](#), [cuD3D9ResourceGetSurfaceDimensions](#), [cuD3D9ResourceSetMapFlags](#), [cuD3D9UnmapResources](#), [cuD3D9UnregisterResource](#)

#### 3.28.2.2 **CUresult cuD3D9GetDevice (CUdevice \*pDevice, const char \*pszAdapterName)**

Returns in `*pDevice` 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.

#### Parameters:

- pDevice* - Returned CUDA device corresponding to `pszAdapterName`
- pszAdapterName* - Adapter name to query for device

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_UNKNOWN](#)

**Note:**

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

**See also:**

[cuD3D9CtxCreate](#), [cuD3D9GetDirect3DDevice](#), [cuD3D9MapResources](#), [cuD3D9RegisterResource](#), [cuD3D9ResourceGetMappedArray](#), [cuD3D9ResourceGetMappedPitch](#), [cuD3D9ResourceGetMappedPointer](#), [cuD3D9ResourceGetMappedSize](#), [cuD3D9ResourceGetSurfaceDimensions](#), [cuD3D9ResourceSetMapFlags](#), [cuD3D9UnmapResources](#), [cuD3D9UnregisterResource](#)

**3.28.2.3 CUresult cuD3D9GetDirect3DDevice (IDirect3DDevice9 \*\* ppDxDevice)**

Returns in \*ppDxDevice the Direct3D device against which this CUDA context was created in [cuD3D9CtxCreate\(\)](#).

**Parameters:**

*ppDxDevice* - Returned Direct3D device corresponding to CUDA context

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#)

**Note:**

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

**See also:**

[cuD3D9CtxCreate](#), [cuD3D9GetDevice](#), [cuD3D9MapResources](#), [cuD3D9RegisterResource](#), [cuD3D9ResourceGetMappedArray](#), [cuD3D9ResourceGetMappedPitch](#), [cuD3D9ResourceGetMappedPointer](#), [cuD3D9ResourceGetMappedSize](#), [cuD3D9ResourceGetSurfaceDimensions](#), [cuD3D9ResourceSetMapFlags](#), [cuD3D9UnmapResources](#), [cuD3D9UnregisterResource](#)

**3.28.2.4 CUresult cuD3D9MapResources (unsigned int count, IDirect3DResource9 \*\* ppResources)**

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 [cuD3D9MapResources\(\)](#) will complete before any CUDA kernels issued after [cuD3D9MapResources\(\)](#) begin.

If any of *ppResources* have not been registered for use with CUDA or if *ppResources* contains any duplicate entries, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If any of *ppResources* are presently mapped for access by CUDA, then [CUDA\\_ERROR\\_ALREADY\\_MAPPED](#) is returned.

**Parameters:**

*count* - Number of resources in *ppResources*

*ppResources* - Resources to map for CUDA usage

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_HANDLE, CUDA\_ERROR\_ALREADY\_MAPPED, CUDA\_ERROR\_UNKNOWN

**Note:**

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

**See also:**

cuD3D9CtxCreate, cuD3D9GetDevice, cuD3D9GetDirect3DDevice, cuD3D9RegisterResource, cuD3D9ResourceGetMappedArray, cuD3D9ResourceGetMappedPitch, cuD3D9ResourceGetMappedPointer, cuD3D9ResourceGetMappedSize, cuD3D9ResourceGetSurfaceDimensions, cuD3D9ResourceSetMapFlags, cuD3D9UnmapResources, cuD3D9UnregisterResource

### 3.28.2.5 CUresult cuD3D9RegisterResource (IDirect3DResource9 \*pResource, unsigned int flags)

Registers the Direct3D resource *pResource* for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through [cuD3D9UnregisterResource\(\)](#). Also on success, this call will increase the internal reference count on *pResource*. This reference count will be decremented when this resource is unregistered through [cuD3D9UnregisterResource\(\)](#).

This call is potentially high-overhead and should not be called every frame in interactive applications.

The type of *pResource* must be one of the following.

- IDirect3DVertexBuffer9: Cannot be used with *flags* set to CU\_D3D9\_REGISTER\_FLAGS\_ARRAY.
- IDirect3DIndexBuffer9: Cannot be used with *flags* set to CU\_D3D9\_REGISTER\_FLAGS\_ARRAY.
- IDirect3DSurface9: Only stand-alone objects of type IDirect3DSurface9 may be explicitly shared. In particular, individual mipmap levels and faces of cube maps may not be registered directly. To access individual surfaces associated with a texture, one must register the base texture object. For restrictions on the *flags* parameter, see type IDirect3DBaseTexture9.
- IDirect3DBaseTexture9: When a texture is registered, all surfaces associated with the all mipmap levels of all faces of the texture will be accessible to CUDA.

The *flags* argument specifies the mechanism through which CUDA will access the Direct3D resource. The following values are allowed.

- CU\_D3D9\_REGISTER\_FLAGS\_NONE: Specifies that CUDA will access this resource through a [CUdeviceptr](#). The pointer, size, and (for textures), pitch for each subresource of this allocation may be queried through [cuD3D9ResourceGetMappedPointer\(\)](#), [cuD3D9ResourceGetMappedSize\(\)](#), and [cuD3D9ResourceGetMappedPitch\(\)](#) respectively. This option is valid for all resource types.
- CU\_D3D9\_REGISTER\_FLAGS\_ARRAY: Specifies that CUDA will access this resource through a [CUarray](#) queried on a sub-resource basis through [cuD3D9ResourceGetMappedArray\(\)](#). This option is only valid for resources of type IDirect3DSurface9 and subtypes of IDirect3DBaseTexture9.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Any resources allocated in D3DPOOL\_SYSTEMMEM or D3DPOOL\_MANAGED may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized on this context, then [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#) is returned. If `pResource` is of incorrect type (e.g. is a non-stand-alone IDirect3DSurface9) or is already registered, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If `pResource` cannot be registered then [CUDA\\_ERROR\\_UNKNOWN](#) is returned.

**Parameters:**

*pResource* - Resource to register for CUDA access

*flags* - Flags for resource registration

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#), [CUDA\\_ERROR\\_OUT\\_OF\\_MEMORY](#), [CUDA\\_ERROR\\_UNKNOWN](#)

**Note:**

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

**See also:**

[cuD3D9CtxCreate](#), [cuD3D9GetDevice](#), [cuD3D9GetDirect3DDevice](#), [cuD3D9MapResources](#), [cuD3D9ResourceGetMappedArray](#), [cuD3D9ResourceGetMappedPitch](#), [cuD3D9ResourceGetMappedPointer](#), [cuD3D9ResourceGetMappedSize](#), [cuD3D9ResourceGetSurfaceDimensions](#), [cuD3D9ResourceSetMapFlags](#), [cuD3D9UnmapResources](#), [cuD3D9UnregisterResource](#)

**3.28.2.6 CUresult cuD3D9ResourceGetMappedArray (CUarray \*pArray, IDirect3DResource9 \*pResource, U32 face, U32 level)**

Returns in `*pArray` an array through which the subresource of the mapped Direct3D resource `pResource` which corresponds to `face` and `level` may be accessed. The value set in `pArray` may change every time that `pResource` is mapped.

If `pResource` is not registered then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If `pResource` was not registered with usage flags `CU_D3D9_REGISTER_FLAGS_ARRAY` then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If `pResource` is not mapped then [CUDA\\_ERROR\\_NOT\\_MAPPED](#) is returned.

For usage requirements of `face` and `level` parameters, see [cuD3D9ResourceGetMappedPointer\(\)](#).

**Parameters:**

*pArray* - Returned array corresponding to subresource

*pResource* - Mapped resource to access



*face* - Face of resource to access

*level* - Level of resource to access

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_INVALID\_HANDLE, CUDA\_ERROR\_NOT\_MAPPED

**Note:**

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

**See also:**

cuD3D9CtxCreate, cuD3D9GetDevice, cuD3D9GetDirect3DDevice, cuD3D9MapResources, cuD3D9RegisterResource, cuD3D9ResourceGetMappedPitch, cuD3D9ResourceGetMappedPointer, cuD3D9ResourceGetMappedSize, cuD3D9ResourceGetSurfaceDimensions, cuD3D9ResourceSetMapFlags, cuD3D9UnmapResources, cuD3D9UnregisterResource

**3.28.2.7 CUresult cuD3D9ResourceGetMappedPitch (unsigned int \*pPitch, unsigned int \*pPitchSlice, IDirect3DResource9 \*pResource, U32 face, U32 level)**

Returns in \*pPitch and \*pPitchSlice the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource pResource, which corresponds to face and level. The values set in pPitch and pPitchSlice may change every time that pResource is mapped.

The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position *x*, *y* from the base pointer of the surface is:

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

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

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

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

If pResource is not of type IDirect3DBaseTexture9 or one of its sub-types or if pResource has not been registered for use with CUDA, then cudaErrorInvalidResourceHandle is returned. If pResource was not registered with usage flags CU\_D3D9\_REGISTER\_FLAGS\_NONE, then CUDA\_ERROR\_INVALID\_HANDLE is returned. If pResource is not mapped for access by CUDA then CUDA\_ERROR\_NOT\_MAPPED is returned.

For usage requirements of face and level parameters, see cuD3D9ResourceGetMappedPointer().

**Parameters:**

*pPitch* - Returned pitch of subresource

*pPitchSlice* - Returned Z-slice pitch of subresource

*pResource* - Mapped resource to access

*face* - Face of resource to access

*level* - Level of resource to access

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_INVALID\_HANDLE, CUDA\_ERROR\_NOT\_MAPPED

**Note:**

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

**See also:**

`cuD3D9CtxCreate`, `cuD3D9GetDevice`, `cuD3D9GetDirect3DDevice`, `cuD3D9MapResources`,  
`cuD3D9RegisterResource`, `cuD3D9ResourceGetMappedArray`, `cuD3D9ResourceGetMappedPointer`,  
`cuD3D9ResourceGetMappedSize`, `cuD3D9ResourceGetSurfaceDimensions`, `cuD3D9ResourceSetMapFlags`,  
`cuD3D9UnmapResources`, `cuD3D9UnregisterResource`

### 3.28.2.8 CUresult cuD3D9ResourceGetMappedPointer (CUdeviceptr \* *pDevPtr*, IDirect3DResource9 \* *pResource*, U32 *face*, U32 *level*)

Returns in *pDevPtr* the base pointer of the subresource of the mapped Direct3D resource *pResource*, which corresponds to *face* and *level*. The value set in *pDevPtr* may change every time that *pResource* is mapped.

If *pResource* is not registered, then `CUDA_ERROR_INVALID_HANDLE` is returned. If *pResource* was not registered with usage flags `CU_D3D9_REGISTER_FLAGS_NONE`, then `CUDA_ERROR_INVALID_HANDLE` is returned. If *pResource* is not mapped, then `CUDA_ERROR_NOT_MAPPED` is returned.

If *pResource* is of type `IDirect3DCubeTexture9`, then *face* must be one of the values enumerated by type `D3DCUBEMAP_FACES`. For all other types *face* must be 0. If *face* is invalid, then `CUDA_ERROR_INVALID_VALUE` is returned.

If *pResource* is of type `IDirect3DBaseTexture9`, then *level* must correspond to a valid mipmap level. At present only mipmap level 0 is supported. For all other types *level* must be 0. If *level* is invalid, then `CUDA_ERROR_INVALID_VALUE` is returned.

**Parameters:**

*pDevPtr* - Returned pointer corresponding to subresource

*pResource* - Mapped resource to access

*face* - Face of resource to access

*level* - Level of resource to access

**Returns:**

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_INVALID_HANDLE`, `CUDA_ERROR_NOT_MAPPED`

**Note:**

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

**See also:**

`cuD3D9CtxCreate`, `cuD3D9GetDevice`, `cuD3D9GetDirect3DDevice`, `cuD3D9MapResources`,  
`cuD3D9RegisterResource`, `cuD3D9ResourceGetMappedArray`, `cuD3D9ResourceGetMappedPitch`,  
`cuD3D9ResourceGetMappedSize`, `cuD3D9ResourceGetSurfaceDimensions`, `cuD3D9ResourceSetMapFlags`,  
`cuD3D9UnmapResources`, `cuD3D9UnregisterResource`

### 3.28.2.9 CUresult cuD3D9ResourceGetMappedSize (unsigned int \* *pSize*, IDirect3DResource9 \* *pResource*, U32 *face*, U32 *level*)

Returns in *pSize* the size of the subresource of the mapped Direct3D resource *pResource*, which corresponds to *face* and *level*. The value set in *pSize* may change every time that *pResource* is mapped.

If *pResource* has not been registered for use with CUDA, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If *pResource* was not registered with usage flags `CU_D3D9_REGISTER_FLAGS_NONE`, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If *pResource* is not mapped for access by CUDA, then [CUDA\\_ERROR\\_NOT\\_MAPPED](#) is returned.

For usage requirements of *face* and *level* parameters, see [cuD3D9ResourceGetMappedPointer](#).

#### Parameters:

- pSize* - Returned size of subresource
- pResource* - Mapped resource to access
- face* - Face of resource to access
- level* - Level of resource to access

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#), [CUDA\\_ERROR\\_NOT\\_MAPPED](#)

#### Note:

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

#### See also:

[cuD3D9CtxCreate](#), [cuD3D9GetDevice](#), [cuD3D9GetDirect3DDevice](#), [cuD3D9MapResources](#), [cuD3D9RegisterResource](#), [cuD3D9ResourceGetMappedArray](#), [cuD3D9ResourceGetMappedPitch](#), [cuD3D9ResourceGetMappedPointer](#), [cuD3D9ResourceGetSurfaceDimensions](#), [cuD3D9ResourceSetMapFlags](#), [cuD3D9UnmapResources](#), [cuD3D9UnregisterResource](#)

### 3.28.2.10 CUresult cuD3D9ResourceGetSurfaceDimensions (unsigned int \* *pWidth*, unsigned int \* *pHeight*, unsigned int \* *pDepth*, IDirect3DResource9 \* *pResource*, unsigned int *face*, unsigned int *level*)

Returns in *pWidth*, *pHeight*, and *pDepth* the dimensions of the subresource of the mapped Direct3D resource *pResource*, which corresponds to *face* and *level*.

Because anti-aliased surfaces may have multiple samples per pixel, it is possible that the dimensions of a resource will be an integer factor larger than the dimensions reported by the Direct3D runtime.

The parameters *pWidth*, *pHeight*, and *pDepth* are optional. For 2D surfaces, the value returned in *pDepth* will be 0.

If *pResource* is not of type `IDirect3DBaseTexture9` or `IDirect3DSurface9` or if *pResource* has not been registered for use with CUDA, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned.

For usage requirements of *face* and *level* parameters, see [cuD3D9ResourceGetMappedPointer\(\)](#).

#### Parameters:

- pWidth* - Returned width of surface

*pHeight* - Returned height of surface  
*pDepth* - Returned depth of surface  
*pResource* - Registered resource to access  
*face* - Face of resource to access  
*level* - Level of resource to access

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_INVALID\_HANDLE

**Note:**

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

**See also:**

cuD3D9CtxCreate, cuD3D9GetDevice, cuD3D9GetDirect3DDevice, cuD3D9MapResources, cuD3D9RegisterResource, cuD3D9ResourceGetMappedArray, cuD3D9ResourceGetMappedPitch, cuD3D9ResourceGetMappedPointer, cuD3D9ResourceGetMappedSize, cuD3D9ResourceSetMapFlags, cuD3D9UnmapResources, cuD3D9UnregisterResource

**3.28.2.11 CUresult cuD3D9ResourceSetMapFlags (IDirect3DResource9 \* pResource, unsigned int flags)**

Set flags for mapping the Direct3D resource pResource.

Changes to flags will take effect the next time pResource is mapped. The flags argument may be any of the following:

- CU\_D3D9\_MAPRESOURCE\_FLAGS\_NONE: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- CU\_D3D9\_MAPRESOURCE\_FLAGS\_READONLY: Specifies that CUDA kernels which access this resource will not write to this resource.
- CU\_D3D9\_MAPRESOURCE\_FLAGS\_WRITEDISCARD: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If pResource has not been registered for use with CUDA, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If pResource is presently mapped for access by CUDA, then [CUDA\\_ERROR\\_ALREADY\\_MAPPED](#) is returned.

**Parameters:**

*pResource* - Registered resource to set flags for  
*flags* - Parameters for resource mapping

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_INVALID\_HANDLE, CUDA\_ERROR\_ALREADY\_MAPPED

**Note:**

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

**See also:**

[cuD3D9CtxCreate](#), [cuD3D9GetDevice](#), [cuD3D9GetDirect3DDevice](#), [cuD3D9MapResources](#),  
[cuD3D9RegisterResource](#), [cuD3D9ResourceGetMappedArray](#), [cuD3D9ResourceGetMappedPitch](#),  
[cuD3D9ResourceGetMappedPointer](#), [cuD3D9ResourceGetMappedSize](#), [cuD3D9ResourceGetSurfaceDimensions](#),  
[cuD3D9UnmapResources](#), [cuD3D9UnregisterResource](#)

**3.28.2.12 CUresult cuD3D9UnmapResources (unsigned int *count*, IDirect3DResource9 \*\* *ppResources*)**

Unmaps the `count` Direct3D resources in `ppResources`.

This function provides the synchronization guarantee that any CUDA kernels issued before `cuD3D9UnmapResources()` will complete before any Direct3D calls issued after `cuD3D9UnmapResources()` begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries, then `CUDA_ERROR_INVALID_HANDLE` is returned. If any of `ppResources` are not presently mapped for access by CUDA, then `CUDA_ERROR_NOT_MAPPED` is returned.

**Parameters:**

*count* - Number of resources to unmap for CUDA  
*ppResources* - Resources to unmap for CUDA

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#), [CUDA\\_ERROR\\_NOT\\_MAPPED](#),  
[CUDA\\_ERROR\\_UNKNOWN](#)

**Note:**

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

**See also:**

[cuD3D9CtxCreate](#), [cuD3D9GetDevice](#), [cuD3D9GetDirect3DDevice](#), [cuD3D9MapResources](#),  
[cuD3D9RegisterResource](#), [cuD3D9ResourceGetMappedArray](#), [cuD3D9ResourceGetMappedPitch](#),  
[cuD3D9ResourceGetMappedPointer](#), [cuD3D9ResourceGetMappedSize](#), [cuD3D9ResourceGetSurfaceDimensions](#),  
[cuD3D9ResourceSetMapFlags](#), [cuD3D9UnregisterResource](#)

**3.28.2.13 CUresult cuD3D9UnregisterResource (IDirect3DResource9 \* *pResource*)**

Unregisters the Direct3D resource `pResource` so it is not accessible by CUDA unless registered again.

If `pResource` is not registered, then `CUDA_ERROR_INVALID_HANDLE` is returned.

**Parameters:**

*pResource* - Resource to unregister

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#), [CUDA\\_ERROR\\_UNKNOWN](#)

**Note:**

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

**See also:**

[cuD3D9CtxCreate](#), [cuD3D9GetDevice](#), [cuD3D9GetDirect3DDevice](#), [cuD3D9MapResources](#),  
[cuD3D9RegisterResource](#), [cuD3D9ResourceGetMappedArray](#), [cuD3D9ResourceGetMappedPitch](#),  
[cuD3D9ResourceGetMappedPointer](#), [cuD3D9ResourceGetMappedSize](#), [cuD3D9ResourceGetSurfaceDimensions](#),  
[cuD3D9ResourceSetMapFlags](#), [cuD3D9UnmapResources](#)

## 3.29 Direct3D 10 Interoperability

### Functions

- **CUresult cuD3D10CtxCreate** (CUcontext \*pCtx, CUdevice \*pCuDevice, unsigned int Flags, ID3D10Device \*pDxDevice)  
*Create a CUDA context for interoperability with Direct3D.*
- **CUresult cuD3D10GetDevice** (CUdevice \*pDevice, IDXGIAdapter \*pAdapter)  
*Gets the device number for an adapter.*
- **CUresult cuD3D10MapResources** (unsigned int count, ID3D10Resource \*\*ppResources)  
*Map Direct3D resources for access by CUDA.*
- **CUresult cuD3D10RegisterResource** (ID3D10Resource \*pResource, unsigned int flags)  
*Register a Direct3D resource for access by CUDA.*
- **CUresult cuD3D10ResourceGetMappedArray** (CUarray \*pArray, ID3D10Resource \*pResource, U32 SubResource)  
*Get an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- **CUresult cuD3D10ResourceGetMappedPitch** (unsigned int \*pPitch, unsigned int \*pPitchSlice, ID3D10Resource \*pResource, U32 SubResource)  
*Get the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- **CUresult cuD3D10ResourceGetMappedPointer** (CUdeviceptr \*pDevPtr, ID3D10Resource \*pResource, U32 SubResource)  
*Get a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- **CUresult cuD3D10ResourceGetMappedSize** (unsigned int \*pSize, ID3D10Resource \*pResource, U32 SubResource)  
*Get the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- **CUresult cuD3D10ResourceGetSurfaceDimensions** (unsigned int \*pWidth, unsigned int \*pHeight, unsigned int \*pDepth, ID3D10Resource \*pResource, U32 SubResource)  
*Get the dimensions of a registered surface.*
- **CUresult cuD3D10ResourceSetMapFlags** (ID3D10Resource \*pResource, unsigned int Flags)  
*Set usage flags for mapping a Direct3D resource.*
- **CUresult cuD3D10UnmapResources** (unsigned int count, ID3D10Resource \*\*ppResources)  
*Unmap Direct3D resources.*
- **CUresult cuD3D10UnregisterResource** (ID3D10Resource \*pResource)  
*Unregister a Direct3D resource.*

### 3.29.1 Detailed Description

This section describes the Direct3D 10 interoperability functions of the low-level CUDA driver application programming interface.

### 3.29.2 Function Documentation

#### 3.29.2.1 CUresult cuD3D10CtxCreate (CUcontext \* pCtx, CUdevice \* pCuDevice, unsigned int Flags, IDXGIAdapter \* pDxDevice)

Creates a new CUDA context, enables interoperability for that context with the Direct3D device `pDxDevice`, and associates the created CUDA context with the calling thread. The `CUcontext` will be returned in `*pCtx`. If `pCuDevice` is non-NULL, then the `CUdevice` on which this CUDA context was created will be returned in `*pCuDevice`. For usage of the `Flags` parameter, see `cuCtxCreate()`. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context.

This context will function only until its Direct3D device is destroyed. On success, this call will increase the internal reference count on `pDxDevice`. This reference count will be decremented upon destruction of this context through `cuCtxDestroy()`.

#### Parameters:

- `pCtx` - Returned newly created CUDA context
- `pCuDevice` - Returned pointer to device on which context was created
- `Flags` - Context creation flags (see `cuCtxCreate()` for details)
- `pDxDevice` - Direct3D device to create interoperability context with

#### Returns:

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_OUT_OF_MEMORY`, `CUDA_ERROR_UNKNOWN`

#### Note:

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

#### See also:

`cuD3D10GetDevice`, `cuD3D10MapResources`, `cuD3D10RegisterResource`, `cuD3D10ResourceGetMappedArray`, `cuD3D10ResourceGetMappedPitch`, `cuD3D10ResourceGetMappedPointer`, `cuD3D10ResourceGetMappedSize`, `cuD3D10ResourceGetSurfaceDimensions`, `cuD3D10ResourceSetMapFlags`, `cuD3D10UnmapResources`, `cuD3D10UnregisterResource`

#### 3.29.2.2 CUresult cuD3D10GetDevice (CUdevice \* pDevice, IDXGIAdapter \* pAdapter)

Returns in `*pDevice` the Cuda-compatible device corresponding to the adapter `pAdapter` obtained from `IDXGIAdapter::EnumAdapters`. This call will succeed only if a device on adapter `pAdapter` is Cuda-compatible.

#### Parameters:

- `pDevice` - Returns the device corresponding to `pAdapter`
- `pAdapter` - D3D10 adapter to get device for

#### Returns:

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_UNKNOWN`

#### Note:

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



**See also:**

[cuD3D10CtxCreate](#), [cuD3D10MapResources](#), [cuD3D10RegisterResource](#), [cuD3D10ResourceGetMappedArray](#), [cuD3D10ResourceGetMappedPitch](#), [cuD3D10ResourceGetMappedPointer](#), [cuD3D10ResourceGetMappedSize](#), [cuD3D10ResourceGetSurfaceDimensions](#), [cuD3D10ResourceSetMapFlags](#), [cuD3D10UnmapResources](#), [cuD3D10UnregisterResource](#)

**3.29.2.3 CUresult cuD3D10MapResources (unsigned int *count*, ID3D10Resource \*\* *ppResources*)**

Maps the `count` Direct3D resources in `ppResources` for access by CUDA.

The resources in `ppResources` may be accessed in CUDA kernels until they are unmapped. Direct3D should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any Direct3D calls issued before [cuD3D10MapResources\(\)](#) will complete before any CUDA kernels issued after [cuD3D10MapResources\(\)](#) begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If any of `ppResources` are presently mapped for access by CUDA, then [CUDA\\_ERROR\\_ALREADY\\_MAPPED](#) is returned.

**Parameters:**

*count* - Number of resources to map for CUDA

*ppResources* - Resources to map for CUDA

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#), [CUDA\\_ERROR\\_ALREADY\\_MAPPED](#), [CUDA\\_ERROR\\_UNKNOWN](#)

**Note:**

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

**See also:**

[cuD3D10CtxCreate](#), [cuD3D10GetDevice](#), [cuD3D10RegisterResource](#), [cuD3D10ResourceGetMappedArray](#), [cuD3D10ResourceGetMappedPitch](#), [cuD3D10ResourceGetMappedPointer](#), [cuD3D10ResourceGetMappedSize](#), [cuD3D10ResourceGetSurfaceDimensions](#), [cuD3D10ResourceSetMapFlags](#), [cuD3D10UnmapResources](#), [cuD3D10UnregisterResource](#)

**3.29.2.4 CUresult cuD3D10RegisterResource (ID3D10Resource \* *pResource*, unsigned int *flags*)**

Registers the Direct3D resource `pResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through [cuD3D10UnregisterResource\(\)](#). Also on success, this call will increase the internal reference count on `pResource`. This reference count will be decremented when this resource is unregistered through [cuD3D10UnregisterResource\(\)](#).

This call is potentially high-overhead and should not be called every frame in interactive applications.

The type of `pResource` must be one of the following.

- `ID3D10Buffer`: Cannot be used with `flags` set to `CU_D3D10_REGISTER_FLAGS_ARRAY`.

- ID3D10Texture1D: No restrictions.
- ID3D10Texture2D: No restrictions.
- ID3D10Texture3D: No restrictions.

The `flags` argument specifies the mechanism through which CUDA will access the Direct3D resource. The following values are allowed.

- `CU_D3D10_REGISTER_FLAGS_NONE`: Specifies that CUDA will access this resource through a [CUdeviceptr](#). The pointer, size, and (for textures), pitch for each subresource of this allocation may be queried through [cuD3D10ResourceGetMappedPointer\(\)](#), [cuD3D10ResourceGetMappedSize\(\)](#), and [cuD3D10ResourceGetMappedPitch\(\)](#) respectively. This option is valid for all resource types.
- `CU_D3D10_REGISTER_FLAGS_ARRAY`: Specifies that CUDA will access this resource through a [CUarray](#) queried on a sub-resource basis through [cuD3D10ResourceGetMappedArray\(\)](#). This option is only valid for resources of type ID3D10Texture1D, ID3D10Texture2D, and ID3D10Texture3D.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized on this context then [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#) is returned. If `pResource` is of incorrect type or is already registered, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If `pResource` cannot be registered, then [CUDA\\_ERROR\\_UNKNOWN](#) is returned.

**Parameters:**

*pResource* - Resource to register

*flags* - Parameters for resource registration

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#), [CUDA\\_ERROR\\_OUT\\_OF\\_MEMORY](#), [CUDA\\_ERROR\\_UNKNOWN](#)

**Note:**

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

**See also:**

[cuD3D10CtxCreate](#), [cuD3D10GetDevice](#), [cuD3D10MapResources](#), [cuD3D10ResourceGetMappedArray](#), [cuD3D10ResourceGetMappedPitch](#), [cuD3D10ResourceGetMappedPointer](#), [cuD3D10ResourceGetMappedSize](#), [cuD3D10ResourceGetSurfaceDimensions](#), [cuD3D10ResourceSetMapFlags](#), [cuD3D10UnmapResources](#), [cuD3D10UnregisterResource](#)

### 3.29.2.5 CUresult cuD3D10ResourceGetMappedArray (CUarray \* pArray, ID3D10Resource \* pResource, U32 SubResource)

Returns in \*pArray an array through which the subresource of the mapped Direct3D resource pResource, which corresponds to SubResource may be accessed. The value set in pArray may change every time that pResource is mapped.

If pResource is not registered, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If pResource was not registered with usage flags CU\_D3D10\_REGISTER\_FLAGS\_ARRAY, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If pResource is not mapped, then [CUDA\\_ERROR\\_NOT\\_MAPPED](#) is returned.

For usage requirements of the SubResource parameter, see [cuD3D10ResourceGetMappedPointer\(\)](#).

#### Parameters:

*pArray* - Returned array corresponding to subresource

*pResource* - Mapped resource to access

*SubResource* - Subresource of pResource to access

#### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#), [CUDA\\_ERROR\\_NOT\\_MAPPED](#)

#### Note:

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

#### See also:

[cuD3D10CtxCreate](#), [cuD3D10GetDevice](#), [cuD3D10MapResources](#), [cuD3D10RegisterResource](#), [cuD3D10ResourceGetMappedPitch](#), [cuD3D10ResourceGetMappedPointer](#), [cuD3D10ResourceGetMappedSize](#), [cuD3D10ResourceGetSurfaceDimensions](#), [cuD3D10ResourceSetMapFlags](#), [cuD3D10UnmapResources](#), [cuD3D10UnregisterResource](#)

### 3.29.2.6 CUresult cuD3D10ResourceGetMappedPitch (unsigned int \* pPitch, unsigned int \* pPitchSlice, ID3D10Resource \* pResource, U32 SubResource)

Returns in \*pPitch and \*pPitchSlice the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource pResource, which corresponds to SubResource. The values set in pPitch and pPitchSlice may change every time that pResource is mapped.

The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position **x**, **y** from the base pointer of the surface is:

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

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

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

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

If pResource is not of type IDirect3DBaseTexture10 or one of its sub-types or if pResource has not been registered for use with CUDA, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If pResource was not registered with usage flags CU\_D3D10\_REGISTER\_FLAGS\_NONE, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If pResource is not mapped for access by CUDA, then [CUDA\\_ERROR\\_NOT\\_MAPPED](#) is returned.

For usage requirements of the `SubResource` parameter, see [cuD3D10ResourceGetMappedPointer\(\)](#).

**Parameters:**

- pPitch* - Returned pitch of subresource
- pPitchSlice* - Returned Z-slice pitch of subresource
- pResource* - Mapped resource to access
- SubResource* - Subresource of pResource to access

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#), [CUDA\\_ERROR\\_NOT\\_MAPPED](#)

**Note:**

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

**See also:**

[cuD3D10CtxCreate](#), [cuD3D10GetDevice](#), [cuD3D10MapResources](#), [cuD3D10RegisterResource](#), [cuD3D10ResourceGetMappedArray](#), [cuD3D10ResourceGetMappedPointer](#), [cuD3D10ResourceGetMappedSize](#), [cuD3D10ResourceGetSurfaceDimensions](#), [cuD3D10ResourceSetMapFlags](#), [cuD3D10UnmapResources](#), [cuD3D10UnregisterResource](#)

**3.29.2.7 CUresult cuD3D10ResourceGetMappedPointer (CUdeviceptr \*pDevPtr, ID3D10Resource \*pResource, U32 SubResource)**

Returns in \*pDevPtr the base pointer of the subresource of the mapped Direct3D resource pResource, which corresponds to SubResource. The value set in pDevPtr may change every time that pResource is mapped.

If pResource is not registered, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If pResource was not registered with usage flags `CU_D3D10_REGISTER_FLAGS_NONE`, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If pResource is not mapped, then [CUDA\\_ERROR\\_NOT\\_MAPPED](#) is returned.

If pResource is of type `ID3D10Buffer`, then SubResource must be 0. If pResource is of any other type, then the value of SubResource must come from the subresource calculation in [D3D10CalcSubResource\(\)](#).

**Parameters:**

- pDevPtr* - Returned pointer corresponding to subresource
- pResource* - Mapped resource to access
- SubResource* - Subresource of pResource to access

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#), [CUDA\\_ERROR\\_NOT\\_MAPPED](#)

**Note:**

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

**See also:**

[cuD3D10CtxCreate](#), [cuD3D10GetDevice](#), [cuD3D10MapResources](#), [cuD3D10RegisterResource](#),  
[cuD3D10ResourceGetMappedArray](#), [cuD3D10ResourceGetMappedPitch](#), [cuD3D10ResourceGetMappedSize](#),  
[cuD3D10ResourceGetSurfaceDimensions](#), [cuD3D10ResourceSetMapFlags](#), [cuD3D10UnmapResources](#),  
[cuD3D10UnregisterResource](#)

### 3.29.2.8 CUresult cuD3D10ResourceGetMappedSize (unsigned int \* pSize, ID3D10Resource \* pResource, U32 SubResource)

Returns in \*pSize the size of the subresource of the mapped Direct3D resource pResource, which corresponds to SubResource. The value set in pSize may change every time that pResource is mapped.

If pResource has not been registered for use with CUDA, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If pResource was not registered with usage flags CU\_D3D10\_REGISTER\_FLAGS\_NONE, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If pResource is not mapped for access by CUDA, then [CUDA\\_ERROR\\_NOT\\_MAPPED](#) is returned.

For usage requirements of the SubResource parameter, see [cuD3D10ResourceGetMappedPointer\(\)](#).

**Parameters:**

*pSize* - Returned size of subresource  
*pResource* - Mapped resource to access  
*SubResource* - Subresource of pResource to access

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#),  
[CUDA\\_ERROR\\_NOT\\_MAPPED](#)

**Note:**

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

**See also:**

[cuD3D10CtxCreate](#), [cuD3D10GetDevice](#), [cuD3D10MapResources](#), [cuD3D10RegisterResource](#),  
[cuD3D10ResourceGetMappedArray](#), [cuD3D10ResourceGetMappedPitch](#), [cuD3D10ResourceGetMappedPointer](#),  
[cuD3D10ResourceGetSurfaceDimensions](#), [cuD3D10ResourceSetMapFlags](#), [cuD3D10UnmapResources](#),  
[cuD3D10UnregisterResource](#)

### 3.29.2.9 CUresult cuD3D10ResourceGetSurfaceDimensions (unsigned int \* pWidth, unsigned int \* pHeight, unsigned int \* pDepth, ID3D10Resource \* pResource, U32 SubResource)

Returns in \*pWidth, \*pHeight, and \*pDepth the dimensions of the subresource of the mapped Direct3D resource pResource, which corresponds to SubResource.

Because anti-aliased surfaces may have multiple samples per pixel, it is possible that the dimensions of a resource will be an integer factor larger than the dimensions reported by the Direct3D runtime.

The parameters pWidth, pHeight, and pDepth are optional. For 2D surfaces, the value returned in \*pDepth will be 0.

If `pResource` is not of type `IDirect3DBaseTexture10` or `IDirect3DSurface10` or if `pResource` has not been registered for use with CUDA, then `CUDA_ERROR_INVALID_HANDLE` is returned.

For usage requirements of the `SubResource` parameter, see `cuD3D10ResourceGetMappedPointer()`.

**Parameters:**

- pWidth* - Returned width of surface
- pHeight* - Returned height of surface
- pDepth* - Returned depth of surface
- pResource* - Registered resource to access
- SubResource* - Subresource of `pResource` to access

**Returns:**

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_INVALID_HANDLE`

**Note:**

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

**See also:**

`cuD3D10CtxCreate`, `cuD3D10GetDevice`, `cuD3D10MapResources`, `cuD3D10RegisterResource`, `cuD3D10ResourceGetMappedArray`, `cuD3D10ResourceGetMappedPitch`, `cuD3D10ResourceGetMappedPointer`, `cuD3D10ResourceGetMappedSize`, `cuD3D10ResourceSetMapFlags`, `cuD3D10UnmapResources`, `cuD3D10UnregisterResource`

### 3.29.2.10 CUresult cuD3D10ResourceSetMapFlags (ID3D10Resource \*pResource, unsigned int Flags)

Set flags for mapping the Direct3D resource `pResource`.

Changes to flags will take effect the next time `pResource` is mapped. The `Flags` argument may be any of the following.

- `CU_D3D10_MAPRESOURCE_FLAGS_NONE`: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- `CU_D3D10_MAPRESOURCE_FLAGS_READONLY`: Specifies that CUDA kernels which access this resource will not write to this resource.
- `CU_D3D10_MAPRESOURCE_FLAGS_WRITEDISCARD`: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If `pResource` has not been registered for use with CUDA, then `CUDA_ERROR_INVALID_HANDLE` is returned. If `pResource` is presently mapped for access by CUDA then `CUDA_ERROR_ALREADY_MAPPED` is returned.

**Parameters:**

- pResource* - Registered resource to set flags for
- Flags* - Parameters for resource mapping

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_INVALID\_HANDLE, CUDA\_ERROR\_ALREADY\_MAPPED

**Note:**

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

**See also:**

cuD3D10CtxCreate, cuD3D10GetDevice, cuD3D10MapResources, cuD3D10RegisterResource, cuD3D10ResourceGetMappedArray, cuD3D10ResourceGetMappedPitch, cuD3D10ResourceGetMappedPointer, cuD3D10ResourceGetMappedSize, cuD3D10ResourceGetSurfaceDimensions, cuD3D10UnmapResources, cuD3D10UnregisterResource

**3.29.2.11 CUresult cuD3D10UnmapResources (unsigned int *count*, ID3D10Resource \*\* *ppResources*)**

Unmaps the *count* Direct3D resources in *ppResources*.

This function provides the synchronization guarantee that any CUDA kernels issued before `cuD3D10UnmapResources()` will complete before any Direct3D calls issued after `cuD3D10UnmapResources()` begin.

If any of *ppResources* have not been registered for use with CUDA or if *ppResources* contains any duplicate entries, then `CUDA_ERROR_INVALID_HANDLE` is returned. If any of *ppResources* are not presently mapped for access by CUDA, then `CUDA_ERROR_NOT_MAPPED` is returned.

**Parameters:**

*count* - Number of resources to unmap for CUDA

*ppResources* - Resources to unmap for CUDA

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_INVALID\_HANDLE, CUDA\_ERROR\_NOT\_MAPPED, CUDA\_ERROR\_UNKNOWN

**Note:**

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

**See also:**

cuD3D10CtxCreate, cuD3D10GetDevice, cuD3D10MapResources, cuD3D10RegisterResource, cuD3D10ResourceGetMappedArray, cuD3D10ResourceGetMappedPitch, cuD3D10ResourceGetMappedPointer, cuD3D10ResourceGetMappedSize, cuD3D10ResourceGetSurfaceDimensions, cuD3D10ResourceSetMapFlags, cuD3D10UnregisterResource

**3.29.2.12 CUresult cuD3D10UnregisterResource (ID3D10Resource \* *pResource*)**

Unregisters the Direct3D resource *pResource* so it is not accessible by CUDA unless registered again.

If *pResource* is not registered, then `CUDA_ERROR_INVALID_HANDLE` is returned.

**Parameters:**

*pResource* - Resources to unregister

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_HANDLE, CUDA\_ERROR\_UNKNOWN

**Note:**

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

**See also:**

cuD3D10CtxCreate, cuD3D10GetDevice, cuD3D10MapResources, cuD3D10RegisterResource, cuD3D10ResourceGetMappedArray, cuD3D10ResourceGetMappedPitch, cuD3D10ResourceGetMappedPointer, cuD3D10ResourceGetMappedSize, cuD3D10ResourceGetSurfaceDimensions, cuD3D10ResourceSetMapFlags, cuD3D10UnmapResources



## 3.30 Data types used by CUDA driver

### Data Structures

- struct [CUDA\\_ARRAY3D\\_DESCRIPTOR](#)
- struct [CUDA\\_ARRAY\\_DESCRIPTOR](#)
- struct [CUDA\\_MEMCPY2D\\_st](#)
- struct [CUDA\\_MEMCPY3D\\_st](#)
- struct [CUdevprop\\_st](#)

### Data types used by CUDA driver

Data types used by CUDA driver

#### Author:

NVIDIA Corporation

- enum [CUaddress\\_mode\\_enum](#) {  
[CU\\_TR\\_ADDRESS\\_MODE\\_WRAP](#),  
[CU\\_TR\\_ADDRESS\\_MODE\\_CLAMP](#),  
[CU\\_TR\\_ADDRESS\\_MODE\\_MIRROR](#) }
- enum [CUarray\\_format\\_enum](#) {  
[CU\\_AD\\_FORMAT\\_UNSIGNED\\_INT8](#),  
[CU\\_AD\\_FORMAT\\_UNSIGNED\\_INT16](#),  
[CU\\_AD\\_FORMAT\\_UNSIGNED\\_INT32](#),  
[CU\\_AD\\_FORMAT\\_SIGNED\\_INT8](#),  
[CU\\_AD\\_FORMAT\\_SIGNED\\_INT16](#),  
[CU\\_AD\\_FORMAT\\_SIGNED\\_INT32](#),  
[CU\\_AD\\_FORMAT\\_HALF](#),  
[CU\\_AD\\_FORMAT\\_FLOAT](#) }
- enum [CUcomputemode\\_enum](#) {  
[CU\\_COMPUTEMODE\\_DEFAULT](#),  
[CU\\_COMPUTEMODE\\_EXCLUSIVE](#),  
[CU\\_COMPUTEMODE\\_PROHIBITED](#) }
- enum [CUctx\\_flags\\_enum](#) {  
[CU\\_CTX\\_SCHED\\_AUTO](#),  
[CU\\_CTX\\_SCHED\\_SPIN](#),  
[CU\\_CTX\\_SCHED\\_YIELD](#) ,  
[CU\\_CTX\\_BLOCKING\\_SYNC](#),  
[CU\\_CTX\\_MAP\\_HOST](#) }
- enum [cudaError\\_enum](#) {  
[CUDA\\_SUCCESS](#),  
[CUDA\\_ERROR\\_INVALID\\_VALUE](#),  
[CUDA\\_ERROR\\_OUT\\_OF\\_MEMORY](#),

```
CUDA_ERROR_NOT_INITIALIZED,  
CUDA_ERROR_DEINITIALIZED,  
CUDA_ERROR_NO_DEVICE,  
CUDA_ERROR_INVALID_DEVICE,  
CUDA_ERROR_INVALID_IMAGE,  
CUDA_ERROR_INVALID_CONTEXT,  
CUDA_ERROR_CONTEXT_ALREADY_CURRENT,  
CUDA_ERROR_MAP_FAILED,  
CUDA_ERROR_UNMAP_FAILED,  
CUDA_ERROR_ARRAY_IS_MAPPED,  
CUDA_ERROR_ALREADY_MAPPED,  
CUDA_ERROR_NO_BINARY_FOR_GPU,  
CUDA_ERROR_ALREADY_ACQUIRED,  
CUDA_ERROR_NOT_MAPPED,  
CUDA_ERROR_INVALID_SOURCE,  
CUDA_ERROR_FILE_NOT_FOUND,  
CUDA_ERROR_INVALID_HANDLE,  
CUDA_ERROR_NOT_FOUND,  
CUDA_ERROR_NOT_READY,  
CUDA_ERROR_LAUNCH_FAILED,  
CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES,  
CUDA_ERROR_LAUNCH_TIMEOUT,  
CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING,  
CUDA_ERROR_UNKNOWN }  
• enum CUdevice_attribute_enum {  
  CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK,  
  CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X,  
  CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y,  
  CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z,  
  CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X,  
  CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y,  
  CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z,  
  CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK,  
  CU_DEVICE_ATTRIBUTE_SHARED_MEMORY_PER_BLOCK,  
  CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY,  
  CU_DEVICE_ATTRIBUTE_WARP_SIZE,  
  CU_DEVICE_ATTRIBUTE_MAX_PITCH,  
  CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK,  
  CU_DEVICE_ATTRIBUTE_REGISTERS_PER_BLOCK,  
  CU_DEVICE_ATTRIBUTE_CLOCK_RATE,  
  CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT,
```

```
CU_DEVICE_ATTRIBUTE_GPU_OVERLAP,  
CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT,  
CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT,  
CU_DEVICE_ATTRIBUTE_INTEGRATED,  
CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY,  
CU_DEVICE_ATTRIBUTE_COMPUTE_MODE }  
• enum CUevent_flags_enum {  
    CU_EVENT_DEFAULT,  
    CU_EVENT_BLOCKING_SYNC }  
• enum CUfilter_mode_enum {  
    CU_TR_FILTER_MODE_POINT,  
    CU_TR_FILTER_MODE_LINEAR }  
• enum CUfunction_attribute_enum {  
    CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK,  
    CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES,  
    CU_FUNC_ATTRIBUTE_CONST_SIZE_BYTES,  
    CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES,  
    CU_FUNC_ATTRIBUTE_NUM_REGS }  
• enum CUjit_fallback_enum {  
    CU_PREFER_PTX,  
    CU_PREFER_BINARY }  
• enum CUjit_option_enum {  
    CU_JIT_MAX_REGISTERS,  
    CU_JIT_THREADS_PER_BLOCK,  
    CU_JIT_WALL_TIME,  
    CU_JIT_INFO_LOG_BUFFER,  
    CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES,  
    CU_JIT_ERROR_LOG_BUFFER,  
    CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES,  
    CU_JIT_OPTIMIZATION_LEVEL,  
    CU_JIT_TARGET_FROM_CUCONTEXT,  
    CU_JIT_TARGET,  
    CU_JIT_FALLBACK_STRATEGY }  
• enum CUjit_target_enum {  
    CU_TARGET_COMPUTE_10,  
    CU_TARGET_COMPUTE_11,  
    CU_TARGET_COMPUTE_12,  
    CU_TARGET_COMPUTE_13 }  
• enum CUmemorytype_enum {  
    CU_MEMORYTYPE_HOST,  
    CU_MEMORYTYPE_DEVICE,  
    CU_MEMORYTYPE_ARRAY }
```

- typedef enum [CUaddress\\_mode\\_enum](#) [CUaddress\\_mode](#)
- typedef struct [CUarray\\_st](#) \* [CUarray](#)  
*CUDA array.*
- typedef enum [CUarray\\_format\\_enum](#) [CUarray\\_format](#)
- typedef enum [CUcomputemode\\_enum](#) [CUcomputemode](#)
- typedef struct [CUctx\\_st](#) \* [CUcontext](#)  
*CUDA context.*
- typedef enum [CUctx\\_flags\\_enum](#) [CUctx\\_flags](#)
- typedef struct [CUDA\\_MEMCPY2D\\_st](#) [CUDA\\_MEMCPY2D](#)
- typedef struct [CUDA\\_MEMCPY3D\\_st](#) [CUDA\\_MEMCPY3D](#)
- typedef int [CUdevice](#)  
*CUDA device.*
- typedef enum [CUdevice\\_attribute\\_enum](#) [CUdevice\\_attribute](#)
- typedef unsigned int [CUdeviceptr](#)  
*CUDA device pointer.*
- typedef struct [CUdevprop\\_st](#) [CUdevprop](#)
- typedef struct [CUevent\\_st](#) \* [CUevent](#)  
*CUDA event.*
- typedef enum [CUevent\\_flags\\_enum](#) [CUevent\\_flags](#)
- typedef enum [CUfilter\\_mode\\_enum](#) [CUfilter\\_mode](#)
- typedef struct [CUfunc\\_st](#) \* [CUfunction](#)  
*CUDA function.*
- typedef enum [CUfunction\\_attribute\\_enum](#) [CUfunction\\_attribute](#)
- typedef enum [CUjit\\_fallback\\_enum](#) [CUjit\\_fallback](#)
- typedef enum [CUjit\\_option\\_enum](#) [CUjit\\_option](#)
- typedef enum [CUjit\\_target\\_enum](#) [CUjit\\_target](#)
- typedef enum [CUmemorytype\\_enum](#) [CUmemorytype](#)
- typedef struct [CUmod\\_st](#) \* [CUmodule](#)  
*CUDA module.*
- typedef enum [cudaError\\_enum](#) [CUresult](#)
- typedef struct [CUstream\\_st](#) \* [CUstream](#)  
*CUDA stream.*
- typedef struct [CUtexref\\_st](#) \* [CUtexref](#)  
*CUDA texture reference.*
- #define [CU\\_MEMHOSTALLOC\\_DEVICEMAP](#)
- #define [CU\\_MEMHOSTALLOC\\_PORTABLE](#)
- #define [CU\\_MEMHOSTALLOC\\_WRITECOMBINED](#)
- #define [CU\\_PARAM\\_TR\\_DEFAULT](#)
- #define [CU\\_TRSA\\_OVERRIDE\\_FORMAT](#)
- #define [CU\\_TRSF\\_NORMALIZED\\_COORDINATES](#)
- #define [CU\\_TRSF\\_READ\\_AS\\_INTEGER](#)
- #define [CUDA\\_VERSION](#)

### 3.30.1 Define Documentation

#### 3.30.1.1 `#define CU_MEMHOSTALLOC_DEVICEMAP`

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

#### 3.30.1.2 `#define CU_MEMHOSTALLOC_PORTABLE`

If set, host memory is portable between CUDA contexts. Flag for `cuMemHostAlloc()`

#### 3.30.1.3 `#define CU_MEMHOSTALLOC_WRITECOMBINED`

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

#### 3.30.1.4 `#define CU_PARAM_TR_DEFAULT`

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

#### 3.30.1.5 `#define CU_TRSA_OVERRIDE_FORMAT`

Override the texref format with a format inferred from the array. Flag for `cuTexRefSetArray()`

#### 3.30.1.6 `#define CU_TRSF_NORMALIZED_COORDINATES`

Use normalized texture coordinates in the range [0,1) instead of [0,dim). Flag for `cuTexRefSetFlags()`

#### 3.30.1.7 `#define CU_TRSF_READ_AS_INTEGER`

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

#### 3.30.1.8 `#define CUDA_VERSION`

CUDA API version number

### 3.30.2 Typedef Documentation

#### 3.30.2.1 `typedef enum CUaddress_mode_enum CUaddress_mode`

Texture reference addressing modes

#### 3.30.2.2 `typedef enum CUarray_format_enum CUarray_format`

Array formats

**3.30.2.3 typedef enum CUcomputemode\_enum CUcomputemode**

Compute Modes

**3.30.2.4 typedef enum CUctx\_flags\_enum CUctx\_flags**

Context creation flags

**3.30.2.5 typedef struct CUDA\_MEMCPY2D\_st CUDA\_MEMCPY2D**

2D memory copy parameters

**3.30.2.6 typedef struct CUDA\_MEMCPY3D\_st CUDA\_MEMCPY3D**

3D memory copy parameters

**3.30.2.7 typedef enum CUdevice\_attribute\_enum CUdevice\_attribute**

Device properties

**3.30.2.8 typedef struct CUdevprop\_st CUdevprop**

Legacy device properties

**3.30.2.9 typedef enum CUevent\_flags\_enum CUevent\_flags**

Event creation flags

**3.30.2.10 typedef enum CUfilter\_mode\_enum CUfilter\_mode**

Texture reference filtering modes

**3.30.2.11 typedef enum CUfunction\_attribute\_enum CUfunction\_attribute**

Function properties

**3.30.2.12 typedef enum CUjit\_fallback\_enum CUjit\_fallback**

Cubin matching fallback strategies

**3.30.2.13 typedef enum CUjit\_option\_enum CUjit\_option**

Online compiler options

**3.30.2.14 typedef enum CUjit\_target\_enum CUjit\_target**

Online compilation targets

### 3.30.2.15 typedef enum CUmemorytype\_enum CUmemorytype

Memory types

### 3.30.2.16 typedef enum cudaError\_enum CUresult

Error codes

## 3.30.3 Enumeration Type Documentation

### 3.30.3.1 enum CUaddress\_mode\_enum

Texture reference addressing modes

**Enumerator:**

- CU\_TR\_ADDRESS\_MODE\_WRAP* Wrapping address mode.
- CU\_TR\_ADDRESS\_MODE\_CLAMP* Clamp to edge address mode.
- CU\_TR\_ADDRESS\_MODE\_MIRROR* Mirror address mode.

### 3.30.3.2 enum CUarray\_format\_enum

Array formats

**Enumerator:**

- CU\_AD\_FORMAT\_UNSIGNED\_INT8* Unsigned 8-bit integers.
- CU\_AD\_FORMAT\_UNSIGNED\_INT16* Unsigned 16-bit integers.
- CU\_AD\_FORMAT\_UNSIGNED\_INT32* Unsigned 32-bit integers.
- CU\_AD\_FORMAT\_SIGNED\_INT8* Signed 8-bit integers.
- CU\_AD\_FORMAT\_SIGNED\_INT16* Signed 16-bit integers.
- CU\_AD\_FORMAT\_SIGNED\_INT32* Signed 32-bit integers.
- CU\_AD\_FORMAT\_HALF* 16-bit floating point
- CU\_AD\_FORMAT\_FLOAT* 32-bit floating point

### 3.30.3.3 enum CUcomputemode\_enum

Compute Modes

**Enumerator:**

- CU\_COMPUTEMODE\_DEFAULT* Default compute mode (Multiple contexts allowed per device).
- CU\_COMPUTEMODE\_EXCLUSIVE* Compute-exclusive mode (Only one context can be present on this device at a time).
- CU\_COMPUTEMODE\_PROHIBITED* Compute-prohibited mode (No contexts can be created on this device at this time).

### 3.30.3.4 enum CUctx\_flags\_enum

Context creation flags

#### Enumerator:

*CUDA\_CTX\_SCHED\_AUTO* Automatic scheduling.  
*CUDA\_CTX\_SCHED\_SPIN* Set spin as default scheduling.  
*CUDA\_CTX\_SCHED\_YIELD* Set yield as default scheduling.  
*CUDA\_CTX\_BLOCKING\_SYNC* Use blocking synchronization.  
*CUDA\_CTX\_MAP\_HOST* Support mapped pinned allocations.

### 3.30.3.5 enum cudaError\_enum

Error codes

#### Enumerator:

*CUDA\_SUCCESS* No errors.  
*CUDA\_ERROR\_INVALID\_VALUE* Invalid value.  
*CUDA\_ERROR\_OUT\_OF\_MEMORY* Out of memory.  
*CUDA\_ERROR\_NOT\_INITIALIZED* Driver not initialized.  
*CUDA\_ERROR\_DEINITIALIZED* Driver deinitialized.  
*CUDA\_ERROR\_NO\_DEVICE* No CUDA-capable device available.  
*CUDA\_ERROR\_INVALID\_DEVICE* Invalid device.  
*CUDA\_ERROR\_INVALID\_IMAGE* Invalid kernel image.  
*CUDA\_ERROR\_INVALID\_CONTEXT* Invalid context.  
*CUDA\_ERROR\_CONTEXT\_ALREADY\_CURRENT* Context already current.  
*CUDA\_ERROR\_MAP\_FAILED* Map failed.  
*CUDA\_ERROR\_UNMAP\_FAILED* Unmap failed.  
*CUDA\_ERROR\_ARRAY\_IS\_MAPPED* Array is mapped.  
*CUDA\_ERROR\_ALREADY\_MAPPED* Already mapped.  
*CUDA\_ERROR\_NO\_BINARY\_FOR\_GPU* No binary for GPU.  
*CUDA\_ERROR\_ALREADY\_ACQUIRED* Already acquired.  
*CUDA\_ERROR\_NOT\_MAPPED* Not mapped.  
*CUDA\_ERROR\_INVALID\_SOURCE* Invalid source.  
*CUDA\_ERROR\_FILE\_NOT\_FOUND* File not found.  
*CUDA\_ERROR\_INVALID\_HANDLE* Invalid handle.  
*CUDA\_ERROR\_NOT\_FOUND* Not found.  
*CUDA\_ERROR\_NOT\_READY* CUDA not ready.  
*CUDA\_ERROR\_LAUNCH\_FAILED* Launch failed.  
*CUDA\_ERROR\_LAUNCH\_OUT\_OF\_RESOURCES* Launch exceeded resources.  
*CUDA\_ERROR\_LAUNCH\_TIMEOUT* Launch exceeded timeout.  
*CUDA\_ERROR\_LAUNCH\_INCOMPATIBLE\_TEXTURING* Launch with incompatible texturing.  
*CUDA\_ERROR\_UNKNOWN* Unknown error.



**3.30.3.6 enum CUdevice\_attribute\_enum**

Device properties

**Enumerator:**

- CU\_DEVICE\_ATTRIBUTE\_MAX\_THREADS\_PER\_BLOCK* Maximum number of threads per block.
- CU\_DEVICE\_ATTRIBUTE\_MAX\_BLOCK\_DIM\_X* Maximum block dimension X.
- CU\_DEVICE\_ATTRIBUTE\_MAX\_BLOCK\_DIM\_Y* Maximum block dimension Y.
- CU\_DEVICE\_ATTRIBUTE\_MAX\_BLOCK\_DIM\_Z* Maximum block dimension Z.
- CU\_DEVICE\_ATTRIBUTE\_MAX\_GRID\_DIM\_X* Maximum grid dimension X.
- CU\_DEVICE\_ATTRIBUTE\_MAX\_GRID\_DIM\_Y* Maximum grid dimension Y.
- CU\_DEVICE\_ATTRIBUTE\_MAX\_GRID\_DIM\_Z* Maximum grid dimension Z.
- CU\_DEVICE\_ATTRIBUTE\_MAX\_SHARED\_MEMORY\_PER\_BLOCK* Maximum shared memory available per block in bytes.
- CU\_DEVICE\_ATTRIBUTE\_SHARED\_MEMORY\_PER\_BLOCK* Deprecated, use *CU\_DEVICE\_ATTRIBUTE\_MAX\_SHARED\_MEMORY\_PER\_BLOCK*.
- CU\_DEVICE\_ATTRIBUTE\_TOTAL\_CONSTANT\_MEMORY* Memory available on device for `__constant_` variables in a CUDA C kernel in bytes.
- CU\_DEVICE\_ATTRIBUTE\_WARP\_SIZE* Warp size in threads.
- CU\_DEVICE\_ATTRIBUTE\_MAX\_PITCH* Maximum pitch in bytes allowed by memory copies.
- CU\_DEVICE\_ATTRIBUTE\_MAX\_REGISTERS\_PER\_BLOCK* Maximum number of 32-bit registers available per block.
- CU\_DEVICE\_ATTRIBUTE\_REGISTERS\_PER\_BLOCK* Deprecated, use *CU\_DEVICE\_ATTRIBUTE\_MAX\_REGISTERS\_PER\_BLOCK*.
- CU\_DEVICE\_ATTRIBUTE\_CLOCK\_RATE* Peak clock frequency in kilohertz.
- CU\_DEVICE\_ATTRIBUTE\_TEXTURE\_ALIGNMENT* Alignment requirement for textures.
- CU\_DEVICE\_ATTRIBUTE\_GPU\_OVERLAP* Device can possibly copy memory and execute a kernel concurrently.
- CU\_DEVICE\_ATTRIBUTE\_MULTIPROCESSOR\_COUNT* Number of multiprocessors on device.
- CU\_DEVICE\_ATTRIBUTE\_KERNEL\_EXEC\_TIMEOUT* Specifies whether there is a run time limit on kernels.
- CU\_DEVICE\_ATTRIBUTE\_INTEGRATED* Device is integrated with host memory.
- CU\_DEVICE\_ATTRIBUTE\_CAN\_MAP\_HOST\_MEMORY* Device can map host memory into CUDA address space.
- CU\_DEVICE\_ATTRIBUTE\_COMPUTE\_MODE* Compute mode (See [CUcomputemode](#) for details).

**3.30.3.7 enum CUevent\_flags\_enum**

Event creation flags

**Enumerator:**

- CU\_EVENT\_DEFAULT* Default event flag.
- CU\_EVENT\_BLOCKING\_SYNC* Event uses blocking synchronization.

**3.30.3.8 enum CUfilter\_mode\_enum**

Texture reference filtering modes

**Enumerator:**

*CU\_TR\_FILTER\_MODE\_POINT* Point filter mode.

*CU\_TR\_FILTER\_MODE\_LINEAR* Linear filter mode.

**3.30.3.9 enum CUfunction\_attribute\_enum**

Function properties

**Enumerator:**

*CU\_FUNC\_ATTRIBUTE\_MAX\_THREADS\_PER\_BLOCK* The number of threads beyond which a launch of the function would fail. This number depends on both the function and the device on which the function is currently loaded.

*CU\_FUNC\_ATTRIBUTE\_SHARED\_SIZE\_BYTES* The size in bytes of statically-allocated shared memory required by this function. This does not include dynamically-allocated shared memory requested by the user at runtime.

*CU\_FUNC\_ATTRIBUTE\_CONST\_SIZE\_BYTES* The size in bytes of user-allocated constant memory required by this function.

*CU\_FUNC\_ATTRIBUTE\_LOCAL\_SIZE\_BYTES* The size in bytes of thread local memory used by this function.

*CU\_FUNC\_ATTRIBUTE\_NUM\_REGS* The number of registers used by each thread of this function.

**3.30.3.10 enum CUjit\_fallback\_enum**

Cubin matching fallback strategies

**Enumerator:**

*CU\_PREFER\_PTX* Prefer to compile ptx

*CU\_PREFER\_BINARY* Prefer to fall back to compatible binary code

**3.30.3.11 enum CUjit\_option\_enum**

Online compiler options

**Enumerator:**

*CU\_JIT\_MAX\_REGISTERS* Max number of registers that a thread may use.

*CU\_JIT\_THREADS\_PER\_BLOCK* IN: Specifies minimum number of threads per block to target compilation for

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

- CU\_JIT\_WALL\_TIME*** Returns a float value in the option of the wall clock time, in milliseconds, spent creating the cubin
- CU\_JIT\_INFO\_LOG\_BUFFER*** Pointer to a buffer in which to print any log messages from PTXAS that are informational in nature
- CU\_JIT\_INFO\_LOG\_BUFFER\_SIZE\_BYTES*** IN: Log buffer size in bytes. Log messages will be capped at this size (including null terminator)  
OUT: Amount of log buffer filled with messages
- CU\_JIT\_ERROR\_LOG\_BUFFER*** Pointer to a buffer in which to print any log messages from PTXAS that reflect errors
- CU\_JIT\_ERROR\_LOG\_BUFFER\_SIZE\_BYTES*** IN: Log buffer size in bytes. Log messages will be capped at this size (including null terminator)  
OUT: Amount of log buffer filled with messages
- CU\_JIT\_OPTIMIZATION\_LEVEL*** Level of optimizations to apply to generated code (0 - 4), with 4 being the default and highest level of optimizations.
- CU\_JIT\_TARGET\_FROM\_CUCONTEXT*** No option value required. Determines the target based on the current attached context (default)
- CU\_JIT\_TARGET*** Target is chosen based on supplied CUjit\_target\_enum.
- CU\_JIT\_FALLBACK\_STRATEGY*** Specifies choice of fallback strategy if matching cubin is not found. Choice is based on supplied CUjit\_fallback\_enum.

#### 3.30.3.12 enum CUjit\_target\_enum

Online compilation targets

##### Enumerator:

- CU\_TARGET\_COMPUTE\_10*** Compute device class 1.0.
- CU\_TARGET\_COMPUTE\_11*** Compute device class 1.1.
- CU\_TARGET\_COMPUTE\_12*** Compute device class 1.2.
- CU\_TARGET\_COMPUTE\_13*** Compute device class 1.3.

#### 3.30.3.13 enum CUmemorytype\_enum

Memory types

##### Enumerator:

- CU\_MEMORYTYPE\_HOST*** Host memory.
- CU\_MEMORYTYPE\_DEVICE*** Device memory.
- CU\_MEMORYTYPE\_ARRAY*** Array memory.



# Chapter 4

## Data Structure Documentation

### 4.1 CUDA\_ARRAY3D\_DESCRIPTOR Struct Reference

#### Data Fields

- unsigned int [Depth](#)  
*Depth of 3D array.*
- unsigned int [Flags](#)  
*Flags.*
- [CUarray\\_format](#) Format  
*Array format.*
- unsigned int [Height](#)  
*Height of 3D array.*
- unsigned int [NumChannels](#)  
*Channels per array element.*
- unsigned int [Width](#)  
*Width of 3D array.*

#### 4.1.1 Detailed Description

3D array descriptor

## 4.2 CUDA\_ARRAY\_DESCRIPTOR Struct Reference

### Data Fields

- [CUarray\\_format](#) Format  
*Array format.*
- unsigned int [Height](#)  
*Height of array.*
- unsigned int [NumChannels](#)  
*Channels per array element.*
- unsigned int [Width](#)  
*Width of array.*

### 4.2.1 Detailed Description

Array descriptor

## 4.3 CUDA\_MEMCPY2D\_st Struct Reference

### Data Fields

- [CUarray dstArray](#)  
*Destination array reference.*
- [CUdeviceptr dstDevice](#)  
*Destination device pointer.*
- `void * dstHost`  
*Destination host pointer.*
- [CUMemorytype dstMemoryType](#)  
*Destination memory type (host, device, array).*
- `unsigned int dstPitch`  
*Destination pitch (ignored when dst is array).*
- `unsigned int dstXInBytes`  
*Destination X in bytes.*
- `unsigned int dstY`  
*Destination Y.*
- `unsigned int Height`  
*Height of 2D memory copy.*
- [CUarray srcArray](#)  
*Source array reference.*
- [CUdeviceptr srcDevice](#)  
*Source device pointer.*
- `const void * srcHost`  
*Source host pointer.*
- [CUMemorytype srcMemoryType](#)  
*Source memory type (host, device, array).*
- `unsigned int srcPitch`  
*Source pitch (ignored when src is array).*
- `unsigned int srcXInBytes`  
*Source X in bytes.*
- `unsigned int srcY`  
*Source Y.*
- `unsigned int WidthInBytes`  
*Width of 2D memory copy in bytes.*

### 4.3.1 Detailed Description

2D memory copy parameters



## 4.4 CUDA\_MEMCPY3D\_st Struct Reference

### Data Fields

- unsigned int [Depth](#)  
*Depth of 3D memory copy.*
- [CUarray dstArray](#)  
*Destination array reference.*
- [CUdeviceptr dstDevice](#)  
*Destination device pointer.*
- unsigned int [dstHeight](#)  
*Destination height (ignored when dst is array; may be 0 if Depth==1).*
- void \* [dstHost](#)  
*Destination host pointer.*
- unsigned int [dstLOD](#)  
*Destination LOD.*
- [CUmemorytype dstMemoryType](#)  
*Destination memory type (host, device, array).*
- unsigned int [dstPitch](#)  
*Destination pitch (ignored when dst is array).*
- unsigned int [dstXInBytes](#)  
*Destination X in bytes.*
- unsigned int [dstY](#)  
*Destination Y.*
- unsigned int [dstZ](#)  
*Destination Z.*
- unsigned int [Height](#)  
*Height of 3D memory copy.*
- void \* [reserved0](#)  
*Must be NULL.*
- void \* [reserved1](#)  
*Must be NULL.*
- [CUarray srcArray](#)  
*Source array reference.*
- [CUdeviceptr srcDevice](#)

*Source device pointer.*

- unsigned int `srcHeight`  
*Source height (ignored when src is array; may be 0 if Depth==1).*
- const void \* `srcHost`  
*Source host pointer.*
- unsigned int `srcLOD`  
*Source LOD.*
- `CUmemorytype srcMemoryType`  
*Source memory type (host, device, array).*
- unsigned int `srcPitch`  
*Source pitch (ignored when src is array).*
- unsigned int `srcXInBytes`  
*Source X in bytes.*
- unsigned int `srcY`  
*Source Y.*
- unsigned int `srcZ`  
*Source Z.*
- unsigned int `WidthInBytes`  
*Width of 3D memory copy in bytes.*

#### 4.4.1 Detailed Description

3D memory copy parameters

## 4.5 cudaChannelFormatDesc Struct Reference

### Data Fields

- enum [cudaChannelFormatKind](#) `f`  
*Channel format kind.*
- int `w`  
`w`
- int `x`  
`x`
- int `y`  
`y`
- int `z`  
`z`

### 4.5.1 Detailed Description

CUDA Channel format descriptor

## 4.6 cudaDeviceProp Struct Reference

### Data Fields

- int [canMapHostMemory](#)  
*Device can map host memory with `cudaHostAlloc/cudaHostGetDevicePointer`.*
- int [clockRate](#)  
*Clock frequency in kilohertz.*
- int [computeMode](#)  
*Compute mode (See `cudaComputeMode`).*
- int [deviceOverlap](#)  
*Device can concurrently copy memory and execute a kernel.*
- int [integrated](#)  
*Device is integrated as opposed to discrete.*
- int [kernelExecTimeoutEnabled](#)  
*Specified whether there is a run time limit on kernels.*
- int [major](#)  
*Major compute capability.*
- int [maxGridSize](#) [3]  
*Maximum size of each dimension of a grid.*
- int [maxThreadsDim](#) [3]  
*Maximum size of each dimension of a block.*
- int [maxThreadsPerBlock](#)  
*Maximum number of threads per block.*
- size\_t [memPitch](#)  
*Maximum pitch in bytes allowed by memory copies.*
- int [minor](#)  
*Minor compute capability.*
- int [multiProcessorCount](#)  
*Number of multiprocessors on device.*
- char [name](#) [256]  
*ASCII string identifying device.*
- int [regsPerBlock](#)  
*32-bit registers available per block*
- size\_t [sharedMemPerBlock](#)

*Shared memory available per block in bytes.*

- `size_t textureAlignment`  
*Alignment requirement for textures.*
- `size_t totalConstMem`  
*Constant memory available on device in bytes.*
- `size_t totalGlobalMem`  
*Global memory available on device in bytes.*
- `int warpSize`  
*Warp size in threads.*

### 4.6.1 Detailed Description

CUDA device properties

## 4.7 cudaExtent Struct Reference

### Data Fields

- `size_t depth`  
*Depth in bytes.*
- `size_t height`  
*Height in bytes.*
- `size_t width`  
*Width in bytes.*

### 4.7.1 Detailed Description

CUDA extent

## 4.8 cudaFuncAttributes Struct Reference

### Data Fields

- `size_t constSizeBytes`  
*Size of constant memory in bytes.*
- `size_t localSizeBytes`  
*Size of local memory in bytes.*
- `int maxThreadsPerBlock`  
*Maximum number of threads per block.*
- `int numRegs`  
*Number of registers used.*
- `size_t sharedSizeBytes`  
*Size of shared memory in bytes.*

### 4.8.1 Detailed Description

CUDA function attributes

## 4.9 cudaMemcpy3DParms Struct Reference

### Data Fields

- struct `cudaArray * dstArray`  
*Destination memory address.*
- struct `cudaPos dstPos`  
*Destination position offset.*
- struct `cudaPitchedPtr dstPtr`  
*Pitched destination memory address.*
- struct `cudaExtent extent`  
*Requested memory copy size.*
- enum `cudaMemcpyKind kind`  
*Type of transfer.*
- struct `cudaArray * srcArray`  
*Source memory address.*
- struct `cudaPos srcPos`  
*Source position offset.*
- struct `cudaPitchedPtr srcPtr`  
*Pitched source memory address.*

### 4.9.1 Detailed Description

CUDA 3D memory copying parameters



## 4.10 cudaPitchedPtr Struct Reference

### Data Fields

- `size_t pitch`  
*Pitch of allocated memory in bytes.*
- `void * ptr`  
*Pointer to allocated memory.*
- `size_t xsize`  
*Logical width of allocation in bytes.*
- `size_t ysize`  
*Logical height of allocation in bytes.*

### 4.10.1 Detailed Description

CUDA Pitched memory pointer

## 4.11 cudaPos Struct Reference

### Data Fields

- [size\\_t x](#)  
x
- [size\\_t y](#)  
y
- [size\\_t z](#)  
z

### 4.11.1 Detailed Description

CUDA 3D position

## 4.12 CUdevprop\_st Struct Reference

### Data Fields

- int [clockRate](#)  
*Clock frequency in kilohertz.*
- int [maxGridSize](#) [3]  
*Maximum size of each dimension of a grid.*
- int [maxThreadsDim](#) [3]  
*Maximum size of each dimension of a block.*
- int [maxThreadsPerBlock](#)  
*Maximum number of threads per block.*
- int [memPitch](#)  
*Maximum pitch in bytes allowed by memory copies.*
- int [regsPerBlock](#)  
*32-bit registers available per block*
- int [sharedMemPerBlock](#)  
*Shared memory available per block in bytes.*
- int [SIMDWidth](#)  
*Warp size in threads.*
- int [textureAlign](#)  
*Alignment requirement for textures.*
- int [totalConstantMemory](#)  
*Constant memory available on device in bytes.*

### 4.12.1 Detailed Description

Legacy device properties

# Index

C++ API Routines, [81](#)  
Context Management, [103](#)  
CU\_AD\_FORMAT\_FLOAT  
    CUDA\_TYPES, [201](#)  
CU\_AD\_FORMAT\_HALF  
    CUDA\_TYPES, [201](#)  
CU\_AD\_FORMAT\_SIGNED\_INT16  
    CUDA\_TYPES, [201](#)  
CU\_AD\_FORMAT\_SIGNED\_INT32  
    CUDA\_TYPES, [201](#)  
CU\_AD\_FORMAT\_SIGNED\_INT8  
    CUDA\_TYPES, [201](#)  
CU\_AD\_FORMAT\_UNSIGNED\_INT16  
    CUDA\_TYPES, [201](#)  
CU\_AD\_FORMAT\_UNSIGNED\_INT32  
    CUDA\_TYPES, [201](#)  
CU\_AD\_FORMAT\_UNSIGNED\_INT8  
    CUDA\_TYPES, [201](#)  
CU\_COMPUTEMODE\_DEFAULT  
    CUDA\_TYPES, [201](#)  
CU\_COMPUTEMODE\_EXCLUSIVE  
    CUDA\_TYPES, [201](#)  
CU\_COMPUTEMODE\_PROHIBITED  
    CUDA\_TYPES, [201](#)  
CU\_CTX\_BLOCKING\_SYNC  
    CUDA\_TYPES, [202](#)  
CU\_CTX\_MAP\_HOST  
    CUDA\_TYPES, [202](#)  
CU\_CTX\_SCHED\_AUTO  
    CUDA\_TYPES, [202](#)  
CU\_CTX\_SCHED\_SPIN  
    CUDA\_TYPES, [202](#)  
CU\_CTX\_SCHED\_YIELD  
    CUDA\_TYPES, [202](#)  
CU\_DEVICE\_ATTRIBUTE\_CAN\_MAP\_HOST\_-  
    MEMORY  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_CLOCK\_RATE  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_COMPUTE\_MODE  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_GPU\_OVERLAP  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_INTEGRATED  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_KERNEL\_EXEC\_-  
    TIMEOUT  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_MAX\_BLOCK\_DIM\_X  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_MAX\_BLOCK\_DIM\_Y  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_MAX\_BLOCK\_DIM\_Z  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_MAX\_GRID\_DIM\_X  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_MAX\_GRID\_DIM\_Y  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_MAX\_GRID\_DIM\_Z  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_MAX\_PITCH  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_MAX\_REGISTERS\_-  
    PER\_BLOCK  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_MAX\_SHARED\_-  
    MEMORY\_PER\_BLOCK  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_MAX\_THREADS\_PER\_-  
    BLOCK  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_MULTIPROCESSOR\_-  
    COUNT  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_REGISTERS\_PER\_-  
    BLOCK  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_SHARED\_MEMORY\_-  
    PER\_BLOCK  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_TEXTURE\_ALIGNMENT  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_TOTAL\_CONSTANT\_-  
    MEMORY  
    CUDA\_TYPES, [203](#)  
CU\_DEVICE\_ATTRIBUTE\_WARP\_SIZE  
    CUDA\_TYPES, [203](#)  
CU\_EVENT\_BLOCKING\_SYNC  
    CUDA\_TYPES, [203](#)  
CU\_EVENT\_DEFAULT

- CUDA\_TYPES, 203
- CU\_FUNC\_ATTRIBUTE\_CONST\_SIZE\_BYTES
  - CUDA\_TYPES, 204
- CU\_FUNC\_ATTRIBUTE\_LOCAL\_SIZE\_BYTES
  - CUDA\_TYPES, 204
- CU\_FUNC\_ATTRIBUTE\_MAX\_THREADS\_PER\_BLOCK
  - CUDA\_TYPES, 204
- CU\_FUNC\_ATTRIBUTE\_NUM\_REGS
  - CUDA\_TYPES, 204
- CU\_FUNC\_ATTRIBUTE\_SHARED\_SIZE\_BYTES
  - CUDA\_TYPES, 204
- CU\_JIT\_ERROR\_LOG\_BUFFER
  - CUDA\_TYPES, 205
- CU\_JIT\_ERROR\_LOG\_BUFFER\_SIZE\_BYTES
  - CUDA\_TYPES, 205
- CU\_JIT\_FALLBACK\_STRATEGY
  - CUDA\_TYPES, 205
- CU\_JIT\_INFO\_LOG\_BUFFER
  - CUDA\_TYPES, 205
- CU\_JIT\_INFO\_LOG\_BUFFER\_SIZE\_BYTES
  - CUDA\_TYPES, 205
- CU\_JIT\_MAX\_REGISTERS
  - CUDA\_TYPES, 204
- CU\_JIT\_OPTIMIZATION\_LEVEL
  - CUDA\_TYPES, 205
- CU\_JIT\_TARGET
  - CUDA\_TYPES, 205
- CU\_JIT\_TARGET\_FROM\_CUCONTEXT
  - CUDA\_TYPES, 205
- CU\_JIT\_THREADS\_PER\_BLOCK
  - CUDA\_TYPES, 204
- CU\_JIT\_WALL\_TIME
  - CUDA\_TYPES, 204
- CU\_MEMORYTYPE\_ARRAY
  - CUDA\_TYPES, 205
- CU\_MEMORYTYPE\_DEVICE
  - CUDA\_TYPES, 205
- CU\_MEMORYTYPE\_HOST
  - CUDA\_TYPES, 205
- CU\_PREFER\_BINARY
  - CUDA\_TYPES, 204
- CU\_PREFER\_PTX
  - CUDA\_TYPES, 204
- CU\_TARGET\_COMPUTE\_10
  - CUDA\_TYPES, 205
- CU\_TARGET\_COMPUTE\_11
  - CUDA\_TYPES, 205
- CU\_TARGET\_COMPUTE\_12
  - CUDA\_TYPES, 205
- CU\_TARGET\_COMPUTE\_13
  - CUDA\_TYPES, 205
- CU\_TR\_ADDRESS\_MODE\_CLAMP
  - CUDA\_TYPES, 201
- CU\_TR\_ADDRESS\_MODE\_MIRROR
  - CUDA\_TYPES, 201
- CU\_TR\_ADDRESS\_MODE\_WRAP
  - CUDA\_TYPES, 201
- CU\_TR\_FILTER\_MODE\_LINEAR
  - CUDA\_TYPES, 204
- CU\_TR\_FILTER\_MODE\_POINT
  - CUDA\_TYPES, 204
- CU\_MEMHOSTALLOC\_DEVICEMAP
  - CUDA\_TYPES, 199
- CU\_MEMHOSTALLOC\_PORTABLE
  - CUDA\_TYPES, 199
- CU\_MEMHOSTALLOC\_WRITECOMBINED
  - CUDA\_TYPES, 199
- CU\_PARAM\_TR\_DEFAULT
  - CUDA\_TYPES, 199
- CU\_TRSA\_OVERRIDE\_FORMAT
  - CUDA\_TYPES, 199
- CU\_TRSF\_NORMALIZED\_COORDINATES
  - CUDA\_TYPES, 199
- CU\_TRSF\_READ\_AS\_INTEGER
  - CUDA\_TYPES, 199
- CUaddress\_mode
  - CUDA\_TYPES, 199
- CUaddress\_mode\_enum
  - CUDA\_TYPES, 201
- cuArray3DCreate
  - CUMEM, 129
- cuArray3DGetDescriptor
  - CUMEM, 131
- CUarray\_format
  - CUDA\_TYPES, 199
- CUarray\_format\_enum
  - CUDA\_TYPES, 201
- cuArrayCreate
  - CUMEM, 131
- cuArrayDestroy
  - CUMEM, 133
- cuArrayGetDescriptor
  - CUMEM, 133
- CUcomputemode
  - CUDA\_TYPES, 199
- CUcomputemode\_enum
  - CUDA\_TYPES, 201
- CUCTX
  - cuCtxAttach, 103
  - cuCtxCreate, 104
  - cuCtxDestroy, 105
  - cuCtxDetach, 105
  - cuCtxGetDevice, 106
  - cuCtxPopCurrent, 106
  - cuCtxPushCurrent, 106
  - cuCtxSynchronize, 107
- CUctx\_flags

- CUDA\_TYPES, 200
- CUctx\_flags\_enum
  - CUDA\_TYPES, 201
- cuCtxAttach
  - CUCTX, 103
- cuCtxCreate
  - CUCTX, 104
- cuCtxDestroy
  - CUCTX, 105
- cuCtxDetach
  - CUCTX, 105
- cuCtxGetDevice
  - CUCTX, 106
- cuCtxPopCurrent
  - CUCTX, 106
- cuCtxPushCurrent
  - CUCTX, 106
- cuCtxSynchronize
  - CUCTX, 107
- CUD3D10
  - cuD3D10CtxCreate, 186
  - cuD3D10GetDevice, 186
  - cuD3D10MapResources, 187
  - cuD3D10RegisterResource, 187
  - cuD3D10ResourceGetMappedArray, 188
  - cuD3D10ResourceGetMappedPitch, 189
  - cuD3D10ResourceGetMappedPointer, 190
  - cuD3D10ResourceGetMappedSize, 191
  - cuD3D10ResourceGetSurfaceDimensions, 191
  - cuD3D10ResourceSetMapFlags, 192
  - cuD3D10UnmapResources, 193
  - cuD3D10UnregisterResource, 193
- cuD3D10CtxCreate
  - CUD3D10, 186
- cuD3D10GetDevice
  - CUD3D10, 186
- cuD3D10MapResources
  - CUD3D10, 187
- cuD3D10RegisterResource
  - CUD3D10, 187
- cuD3D10ResourceGetMappedArray
  - CUD3D10, 188
- cuD3D10ResourceGetMappedPitch
  - CUD3D10, 189
- cuD3D10ResourceGetMappedPointer
  - CUD3D10, 190
- cuD3D10ResourceGetMappedSize
  - CUD3D10, 191
- cuD3D10ResourceGetSurfaceDimensions
  - CUD3D10, 191
- cuD3D10ResourceSetMapFlags
  - CUD3D10, 192
- cuD3D10UnmapResources
  - CUD3D10, 193
- cuD3D10UnregisterResource
  - CUD3D10, 193
- CUD3D9
  - cuD3D9CtxCreate, 175
  - cuD3D9GetDevice, 175
  - cuD3D9GetDirect3DDevice, 176
  - cuD3D9MapResources, 176
  - cuD3D9RegisterResource, 177
  - cuD3D9ResourceGetMappedArray, 178
  - cuD3D9ResourceGetMappedPitch, 179
  - cuD3D9ResourceGetMappedPointer, 180
  - cuD3D9ResourceGetMappedSize, 180
  - cuD3D9ResourceGetSurfaceDimensions, 181
  - cuD3D9ResourceSetMapFlags, 182
  - cuD3D9UnmapResources, 183
  - cuD3D9UnregisterResource, 183
- cuD3D9CtxCreate
  - CUD3D9, 175
- cuD3D9GetDevice
  - CUD3D9, 175
- cuD3D9GetDirect3DDevice
  - CUD3D9, 176
- cuD3D9MapResources
  - CUD3D9, 176
- cuD3D9RegisterResource
  - CUD3D9, 177
- cuD3D9ResourceGetMappedArray
  - CUD3D9, 178
- cuD3D9ResourceGetMappedPitch
  - CUD3D9, 179
- cuD3D9ResourceGetMappedPointer
  - CUD3D9, 180
- cuD3D9ResourceGetMappedSize
  - CUD3D9, 180
- cuD3D9ResourceGetSurfaceDimensions
  - CUD3D9, 181
- cuD3D9ResourceSetMapFlags
  - CUD3D9, 182
- cuD3D9UnmapResources
  - CUD3D9, 183
- cuD3D9UnregisterResource
  - CUD3D9, 183
- CUDA Driver API, 94
- CUDA Runtime API, 5
- CUDA\_ERROR\_ALREADY\_ACQUIRED
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_ALREADY\_MAPPED
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_ARRAY\_IS\_MAPPED
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_CONTEXT\_ALREADY\_CURRENT
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_DEINITIALIZED
  - CUDA\_TYPES, 202

- CUDA\_ERROR\_FILE\_NOT\_FOUND
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_INVALID\_CONTEXT
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_INVALID\_DEVICE
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_INVALID\_HANDLE
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_INVALID\_IMAGE
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_INVALID\_SOURCE
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_INVALID\_VALUE
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_LAUNCH\_FAILED
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_LAUNCH\_INCOMPATIBLE\_-  
TEXTURING
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_LAUNCH\_OUT\_OF\_RESOURCES
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_LAUNCH\_TIMEOUT
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_MAP\_FAILED
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_NO\_BINARY\_FOR\_GPU
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_NO\_DEVICE
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_NOT\_FOUND
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_NOT\_INITIALIZED
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_NOT\_MAPPED
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_NOT\_READY
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_OUT\_OF\_MEMORY
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_UNKNOWN
  - CUDA\_TYPES, 202
- CUDA\_ERROR\_UNMAP\_FAILED
  - CUDA\_TYPES, 202
- CUDA\_SUCCESS
  - CUDA\_TYPES, 202
- CUDA\_TYPES
  - CU\_AD\_FORMAT\_FLOAT, 201
  - CU\_AD\_FORMAT\_HALF, 201
  - CU\_AD\_FORMAT\_SIGNED\_INT16, 201
  - CU\_AD\_FORMAT\_SIGNED\_INT32, 201
  - CU\_AD\_FORMAT\_SIGNED\_INT8, 201
  - CU\_AD\_FORMAT\_UNSIGNED\_INT16, 201
  - CU\_AD\_FORMAT\_UNSIGNED\_INT32, 201
  - CU\_AD\_FORMAT\_UNSIGNED\_INT8, 201
- CU\_COMPUTEMODE\_DEFAULT, 201
- CU\_COMPUTEMODE\_EXCLUSIVE, 201
- CU\_COMPUTEMODE\_PROHIBITED, 201
- CU\_CTX\_BLOCKING\_SYNC, 202
- CU\_CTX\_MAP\_HOST, 202
- CU\_CTX\_SCHED\_AUTO, 202
- CU\_CTX\_SCHED\_SPIN, 202
- CU\_CTX\_SCHED\_YIELD, 202
- CU\_DEVICE\_ATTRIBUTE\_CAN\_MAP\_HOST\_-  
MEMORY, 203
- CU\_DEVICE\_ATTRIBUTE\_CLOCK\_RATE, 203
- CU\_DEVICE\_ATTRIBUTE\_COMPUTE\_MODE,  
203
- CU\_DEVICE\_ATTRIBUTE\_GPU\_OVERLAP,  
203
- CU\_DEVICE\_ATTRIBUTE\_INTEGRATED, 203
- CU\_DEVICE\_ATTRIBUTE\_KERNEL\_EXEC\_-  
TIMEOUT, 203
- CU\_DEVICE\_ATTRIBUTE\_MAX\_BLOCK\_-  
DIM\_X, 203
- CU\_DEVICE\_ATTRIBUTE\_MAX\_BLOCK\_-  
DIM\_Y, 203
- CU\_DEVICE\_ATTRIBUTE\_MAX\_BLOCK\_-  
DIM\_Z, 203
- CU\_DEVICE\_ATTRIBUTE\_MAX\_GRID\_DIM\_-  
X, 203
- CU\_DEVICE\_ATTRIBUTE\_MAX\_GRID\_DIM\_-  
Y, 203
- CU\_DEVICE\_ATTRIBUTE\_MAX\_GRID\_DIM\_-  
Z, 203
- CU\_DEVICE\_ATTRIBUTE\_MAX\_PITCH, 203
- CU\_DEVICE\_ATTRIBUTE\_MAX\_-  
REGISTERS\_PER\_BLOCK, 203
- CU\_DEVICE\_ATTRIBUTE\_MAX\_SHARED\_-  
MEMORY\_PER\_BLOCK, 203
- CU\_DEVICE\_ATTRIBUTE\_MAX\_THREADS\_-  
PER\_BLOCK, 203
- CU\_DEVICE\_ATTRIBUTE\_-  
MULTIPROCESSOR\_COUNT, 203
- CU\_DEVICE\_ATTRIBUTE\_REGISTERS\_PER\_-  
BLOCK, 203
- CU\_DEVICE\_ATTRIBUTE\_SHARED\_-  
MEMORY\_PER\_BLOCK, 203
- CU\_DEVICE\_ATTRIBUTE\_TEXTURE\_-  
ALIGNMENT, 203
- CU\_DEVICE\_ATTRIBUTE\_TOTAL\_-  
CONSTANT\_MEMORY, 203
- CU\_DEVICE\_ATTRIBUTE\_WARP\_SIZE, 203
- CU\_EVENT\_BLOCKING\_SYNC, 203
- CU\_EVENT\_DEFAULT, 203
- CU\_FUNC\_ATTRIBUTE\_CONST\_SIZE\_BYTES,  
204
- CU\_FUNC\_ATTRIBUTE\_LOCAL\_SIZE\_BYTES,  
204

- CU\_FUNC\_ATTRIBUTE\_MAX\_THREADS\_-  
PER\_BLOCK, 204
- CU\_FUNC\_ATTRIBUTE\_NUM\_REGS, 204
- CU\_FUNC\_ATTRIBUTE\_SHARED\_SIZE\_-  
BYTES, 204
- CU\_JIT\_ERROR\_LOG\_BUFFER, 205
- CU\_JIT\_ERROR\_LOG\_BUFFER\_SIZE\_BYTES,  
205
- CU\_JIT\_FALLBACK\_STRATEGY, 205
- CU\_JIT\_INFO\_LOG\_BUFFER, 205
- CU\_JIT\_INFO\_LOG\_BUFFER\_SIZE\_BYTES,  
205
- CU\_JIT\_MAX\_REGISTERS, 204
- CU\_JIT\_OPTIMIZATION\_LEVEL, 205
- CU\_JIT\_TARGET, 205
- CU\_JIT\_TARGET\_FROM\_CUCONTEXT, 205
- CU\_JIT\_THREADS\_PER\_BLOCK, 204
- CU\_JIT\_WALL\_TIME, 204
- CU\_MEMORYTYPE\_ARRAY, 205
- CU\_MEMORYTYPE\_DEVICE, 205
- CU\_MEMORYTYPE\_HOST, 205
- CU\_PREFER\_BINARY, 204
- CU\_PREFER\_PTX, 204
- CU\_TARGET\_COMPUTE\_10, 205
- CU\_TARGET\_COMPUTE\_11, 205
- CU\_TARGET\_COMPUTE\_12, 205
- CU\_TARGET\_COMPUTE\_13, 205
- CU\_TR\_ADDRESS\_MODE\_CLAMP, 201
- CU\_TR\_ADDRESS\_MODE\_MIRROR, 201
- CU\_TR\_ADDRESS\_MODE\_WRAP, 201
- CU\_TR\_FILTER\_MODE\_LINEAR, 204
- CU\_TR\_FILTER\_MODE\_POINT, 204
- CUDA\_ERROR\_ALREADY\_ACQUIRED, 202
- CUDA\_ERROR\_ALREADY\_MAPPED, 202
- CUDA\_ERROR\_ARRAY\_IS\_MAPPED, 202
- CUDA\_ERROR\_CONTEXT\_ALREADY\_-  
CURRENT, 202
- CUDA\_ERROR\_DEINITIALIZED, 202
- CUDA\_ERROR\_FILE\_NOT\_FOUND, 202
- CUDA\_ERROR\_INVALID\_CONTEXT, 202
- CUDA\_ERROR\_INVALID\_DEVICE, 202
- CUDA\_ERROR\_INVALID\_HANDLE, 202
- CUDA\_ERROR\_INVALID\_IMAGE, 202
- CUDA\_ERROR\_INVALID\_SOURCE, 202
- CUDA\_ERROR\_INVALID\_VALUE, 202
- CUDA\_ERROR\_LAUNCH\_FAILED, 202
- CUDA\_ERROR\_LAUNCH\_INCOMPATIBLE\_-  
TEXTURING, 202
- CUDA\_ERROR\_LAUNCH\_OUT\_OF\_-  
RESOURCES, 202
- CUDA\_ERROR\_LAUNCH\_TIMEOUT, 202
- CUDA\_ERROR\_MAP\_FAILED, 202
- CUDA\_ERROR\_NO\_BINARY\_FOR\_GPU, 202
- CUDA\_ERROR\_NO\_DEVICE, 202
- CUDA\_ERROR\_NOT\_FOUND, 202
- CUDA\_ERROR\_NOT\_INITIALIZED, 202
- CUDA\_ERROR\_NOT\_MAPPED, 202
- CUDA\_ERROR\_NOT\_READY, 202
- CUDA\_ERROR\_OUT\_OF\_MEMORY, 202
- CUDA\_ERROR\_UNKNOWN, 202
- CUDA\_ERROR\_UNMAP\_FAILED, 202
- CUDA\_SUCCESS, 202
- CUDA\_ARRAY3D\_DESCRIPTOR, 207
- CUDA\_ARRAY\_DESCRIPTOR, 208
- CUDA\_MEMCPY2D  
CUDA\_TYPES, 200
- CUDA\_MEMCPY2D\_st, 209
- CUDA\_MEMCPY3D  
CUDA\_TYPES, 200
- CUDA\_MEMCPY3D\_st, 211
- CUDA\_TYPES  
CU\_MEMHOSTALLOC\_DEVICEMAP, 199
- CU\_MEMHOSTALLOC\_PORTABLE, 199
- CU\_MEMHOSTALLOC\_WRITECOMBINED,  
199
- CU\_PARAM\_TR\_DEFAULT, 199
- CU\_TRSA\_OVERRIDE\_FORMAT, 199
- CU\_TRSF\_NORMALIZED\_COORDINATES, 199
- CU\_TRSF\_READ\_AS\_INTEGER, 199
- CUaddress\_mode, 199
- CUaddress\_mode\_enum, 201
- CUarray\_format, 199
- CUarray\_format\_enum, 201
- CUcomputemode, 199
- CUcomputemode\_enum, 201
- CUctx\_flags, 200
- CUctx\_flags\_enum, 201
- CUDA\_MEMCPY2D, 200
- CUDA\_MEMCPY3D, 200
- CUDA\_VERSION, 199
- cudaError\_enum, 202
- CUdevice\_attribute, 200
- CUdevice\_attribute\_enum, 202
- CUdevprop, 200
- CUevent\_flags, 200
- CUevent\_flags\_enum, 203
- CUfilter\_mode, 200
- CUfilter\_mode\_enum, 203
- CUfunction\_attribute, 200
- CUfunction\_attribute\_enum, 204
- CUjit\_fallback, 200
- CUjit\_fallback\_enum, 204
- CUjit\_option, 200
- CUjit\_option\_enum, 204
- CUjit\_target, 200
- CUjit\_target\_enum, 205
- CUmemorytype, 200
- CUmemorytype\_enum, 205



- CUresult, 201
- CUDA\_VERSION
  - CUDA\_TYPES, 199
- cudaBindTexture
  - CUDART\_HIGHLEVEL, 82
  - CUDART\_TEXTURE, 75
- cudaBindTexture2D
  - CUDART\_HIGHLEVEL, 83
  - CUDART\_TEXTURE, 76
- cudaBindTextureToArray
  - CUDART\_HIGHLEVEL, 84
  - CUDART\_TEXTURE, 77
- cudaChannelFormatDesc, 213
- cudaChannelFormatKind
  - CUDART\_TYPES, 91
- cudaChannelFormatKindFloat
  - CUDART\_TYPES, 92
- cudaChannelFormatKindNone
  - CUDART\_TYPES, 92
- cudaChannelFormatKindSigned
  - CUDART\_TYPES, 92
- cudaChannelFormatKindUnsigned
  - CUDART\_TYPES, 92
- cudaChooseDevice
  - CUDART\_DEVICE, 9
- cudaComputeMode
  - CUDART\_TYPES, 92
- cudaComputeModeDefault
  - CUDART\_TYPES, 92
- cudaComputeModeExclusive
  - CUDART\_TYPES, 92
- cudaComputeModeProhibited
  - CUDART\_TYPES, 92
- cudaConfigureCall
  - CUDART\_EXECUTION, 20
- cudaCreateChannelDesc
  - CUDART\_HIGHLEVEL, 85
  - CUDART\_TEXTURE, 77
- cudaD3D10GetDevice
  - CUDART\_D3D10, 66
- cudaD3D10MapFlags
  - CUDART\_D3D10, 66
- cudaD3D10MapFlagsNone
  - CUDART\_D3D10, 66
- cudaD3D10MapFlagsReadOnly
  - CUDART\_D3D10, 66
- cudaD3D10MapFlagsWriteDiscard
  - CUDART\_D3D10, 66
- cudaD3D10MapResources
  - CUDART\_D3D10, 67
- cudaD3D10RegisterFlags
  - CUDART\_D3D10, 66
- cudaD3D10RegisterFlagsArray
  - CUDART\_D3D10, 66
- cudaD3D10RegisterFlagsNone
  - CUDART\_D3D10, 66
- cudaD3D10RegisterResource
  - CUDART\_D3D10, 67
- cudaD3D10ResourceGetMappedArray
  - CUDART\_D3D10, 68
- cudaD3D10ResourceGetMappedPitch
  - CUDART\_D3D10, 69
- cudaD3D10ResourceGetMappedPointer
  - CUDART\_D3D10, 70
- cudaD3D10ResourceGetMappedSize
  - CUDART\_D3D10, 70
- cudaD3D10ResourceGetSurfaceDimensions
  - CUDART\_D3D10, 71
- cudaD3D10ResourceSetMapFlags
  - CUDART\_D3D10, 72
- cudaD3D10SetDirect3DDevice
  - CUDART\_D3D10, 72
- cudaD3D10UnmapResources
  - CUDART\_D3D10, 73
- cudaD3D10UnregisterResource
  - CUDART\_D3D10, 74
- cudaD3D9GetDevice
  - CUDART\_D3D9, 55
- cudaD3D9GetDirect3DDevice
  - CUDART\_D3D9, 56
- cudaD3D9MapFlags
  - CUDART\_D3D9, 55
- cudaD3D9MapFlagsNone
  - CUDART\_D3D9, 55
- cudaD3D9MapFlagsReadOnly
  - CUDART\_D3D9, 55
- cudaD3D9MapFlagsWriteDiscard
  - CUDART\_D3D9, 55
- cudaD3D9MapResources
  - CUDART\_D3D9, 56
- cudaD3D9RegisterFlags
  - CUDART\_D3D9, 55
- cudaD3D9RegisterFlagsArray
  - CUDART\_D3D9, 55
- cudaD3D9RegisterFlagsNone
  - CUDART\_D3D9, 55
- cudaD3D9RegisterResource
  - CUDART\_D3D9, 57
- cudaD3D9ResourceGetMappedArray
  - CUDART\_D3D9, 58
- cudaD3D9ResourceGetMappedPitch
  - CUDART\_D3D9, 59
- cudaD3D9ResourceGetMappedPointer
  - CUDART\_D3D9, 60
- cudaD3D9ResourceGetMappedSize
  - CUDART\_D3D9, 60
- cudaD3D9ResourceGetSurfaceDimensions
  - CUDART\_D3D9, 61

- cudaD3D9ResourceSetMapFlags
  - CUDART\_D3D9, 62
- cudaD3D9SetDirect3DDevice
  - CUDART\_D3D9, 62
- cudaD3D9UnmapResources
  - CUDART\_D3D9, 63
- cudaD3D9UnregisterResource
  - CUDART\_D3D9, 64
- cudaDeviceProp, 214
- cudaDriverGetVersion
  - CUDART\_VERSION, 80
- cudaError
  - CUDART\_TYPES, 92
- cudaError\_enum
  - CUDA\_TYPES, 202
- cudaError\_t
  - CUDART\_TYPES, 91
- cudaErrorAddressOfConstant
  - CUDART\_TYPES, 93
- cudaErrorApiFailureBase
  - CUDART\_TYPES, 93
- cudaErrorCudartUnloading
  - CUDART\_TYPES, 93
- cudaErrorInitializationError
  - CUDART\_TYPES, 92
- cudaErrorInsufficientDriver
  - CUDART\_TYPES, 93
- cudaErrorInvalidChannelDescriptor
  - CUDART\_TYPES, 92
- cudaErrorInvalidConfiguration
  - CUDART\_TYPES, 92
- cudaErrorInvalidDevice
  - CUDART\_TYPES, 92
- cudaErrorInvalidDeviceFunction
  - CUDART\_TYPES, 92
- cudaErrorInvalidDevicePointer
  - CUDART\_TYPES, 92
- cudaErrorInvalidFilterSetting
  - CUDART\_TYPES, 93
- cudaErrorInvalidHostPointer
  - CUDART\_TYPES, 92
- cudaErrorInvalidMemcpyDirection
  - CUDART\_TYPES, 93
- cudaErrorInvalidNormSetting
  - CUDART\_TYPES, 93
- cudaErrorInvalidPitchValue
  - CUDART\_TYPES, 92
- cudaErrorInvalidResourceHandle
  - CUDART\_TYPES, 93
- cudaErrorInvalidSymbol
  - CUDART\_TYPES, 92
- cudaErrorInvalidTexture
  - CUDART\_TYPES, 92
- cudaErrorInvalidTextureBinding
  - CUDART\_TYPES, 92
- cudaErrorInvalidValue
  - CUDART\_TYPES, 92
- cudaErrorLaunchFailure
  - CUDART\_TYPES, 92
- cudaErrorLaunchOutOfResources
  - CUDART\_TYPES, 92
- cudaErrorLaunchTimeout
  - CUDART\_TYPES, 92
- cudaErrorMapBufferObjectFailed
  - CUDART\_TYPES, 92
- cudaErrorMemoryAllocation
  - CUDART\_TYPES, 92
- cudaErrorMemoryValueTooLarge
  - CUDART\_TYPES, 93
- cudaErrorMissingConfiguration
  - CUDART\_TYPES, 92
- cudaErrorMixedDeviceExecution
  - CUDART\_TYPES, 93
- cudaErrorNoDevice
  - CUDART\_TYPES, 93
- cudaErrorNotReady
  - CUDART\_TYPES, 93
- cudaErrorNotYetImplemented
  - CUDART\_TYPES, 93
- cudaErrorPriorLaunchFailure
  - CUDART\_TYPES, 92
- cudaErrorSetOnActiveProcess
  - CUDART\_TYPES, 93
- cudaErrorStartupFailure
  - CUDART\_TYPES, 93
- cudaErrorSynchronizationError
  - CUDART\_TYPES, 93
- cudaErrorTextureFetchFailed
  - CUDART\_TYPES, 93
- cudaErrorTextureNotBound
  - CUDART\_TYPES, 93
- cudaErrorUnknown
  - CUDART\_TYPES, 93
- cudaErrorUnmapBufferObjectFailed
  - CUDART\_TYPES, 92
- cudaEvent\_t
  - CUDART\_TYPES, 91
- cudaEventCreate
  - CUDART\_EVENT, 16
- cudaEventCreateWithFlags
  - CUDART\_EVENT, 16
- cudaEventDestroy
  - CUDART\_EVENT, 17
- cudaEventElapsedTime
  - CUDART\_EVENT, 17
- cudaEventQuery
  - CUDART\_EVENT, 18
- cudaEventRecord

- CUDART\_EVENT, 18
- cudaEventSynchronize
  - CUDART\_EVENT, 19
- cudaExtent, 216
- cudaFree
  - CUDART\_MEMORY, 25
- cudaFreeArray
  - CUDART\_MEMORY, 26
- cudaFreeHost
  - CUDART\_MEMORY, 26
- cudaFuncAttributes, 217
- cudaFuncGetAttributes
  - CUDART\_EXECUTION, 21
- cudaGetChannelDesc
  - CUDART\_TEXTURE, 78
- cudaGetDevice
  - CUDART\_DEVICE, 9
- cudaGetDeviceCount
  - CUDART\_DEVICE, 10
- cudaGetDeviceProperties
  - CUDART\_DEVICE, 10
- cudaGetErrorString
  - CUDART\_ERROR, 7
- cudaGetLastError
  - CUDART\_ERROR, 7
- cudaGetSymbolAddress
  - CUDART\_HIGHLEVEL, 85
  - CUDART\_MEMORY, 26
- cudaGetSymbolSize
  - CUDART\_HIGHLEVEL, 85
  - CUDART\_MEMORY, 27
- cudaGetTextureAlignmentOffset
  - CUDART\_HIGHLEVEL, 86
  - CUDART\_TEXTURE, 78
- cudaGetTextureReference
  - CUDART\_TEXTURE, 79
- cudaGLMapBufferObject
  - CUDART\_OPENGL, 51
- cudaGLRegisterBufferObject
  - CUDART\_OPENGL, 52
- cudaGLSetGLDevice
  - CUDART\_OPENGL, 52
- cudaGLUnmapBufferObject
  - CUDART\_OPENGL, 52
- cudaGLUnregisterBufferObject
  - CUDART\_OPENGL, 53
- cudaHostAlloc
  - CUDART\_MEMORY, 27
- cudaHostGetDevicePointer
  - CUDART\_MEMORY, 28
- cudaLaunch
  - CUDART\_EXECUTION, 21
  - CUDART\_HIGHLEVEL, 86
- cudaMalloc
  - CUDART\_MEMORY, 29
- cudaMalloc3D
  - CUDART\_MEMORY, 29
- cudaMalloc3DArray
  - CUDART\_MEMORY, 30
- cudaMallocArray
  - CUDART\_MEMORY, 31
- cudaMallocHost
  - CUDART\_MEMORY, 31
- cudaMallocPitch
  - CUDART\_MEMORY, 32
- cudaMemcpy
  - CUDART\_MEMORY, 32
- cudaMemcpy2D
  - CUDART\_MEMORY, 33
- cudaMemcpy2DArrayToArray
  - CUDART\_MEMORY, 34
- cudaMemcpy2DAsync
  - CUDART\_MEMORY, 34
- cudaMemcpy2DFromArray
  - CUDART\_MEMORY, 35
- cudaMemcpy2DFromArrayAsync
  - CUDART\_MEMORY, 36
- cudaMemcpy2DToArray
  - CUDART\_MEMORY, 37
- cudaMemcpy2DToArrayAsync
  - CUDART\_MEMORY, 38
- cudaMemcpy3D
  - CUDART\_MEMORY, 39
- cudaMemcpy3DAsync
  - CUDART\_MEMORY, 40
- cudaMemcpy3DParms, 218
- cudaMemcpyArrayToArray
  - CUDART\_MEMORY, 41
- cudaMemcpyAsync
  - CUDART\_MEMORY, 42
- cudaMemcpyDeviceToDevice
  - CUDART\_TYPES, 93
- cudaMemcpyDeviceToHost
  - CUDART\_TYPES, 93
- cudaMemcpyFromArray
  - CUDART\_MEMORY, 43
- cudaMemcpyFromArrayAsync
  - CUDART\_MEMORY, 43
- cudaMemcpyFromSymbol
  - CUDART\_MEMORY, 44
- cudaMemcpyFromSymbolAsync
  - CUDART\_MEMORY, 45
- cudaMemcpyHostToDevice
  - CUDART\_TYPES, 93
- cudaMemcpyHostToHost
  - CUDART\_TYPES, 93
- cudaMemcpyKind
  - CUDART\_TYPES, 93

- cudaMemcpyToArray
  - CUDART\_MEMORY, 46
- cudaMemcpyToArrayAsync
  - CUDART\_MEMORY, 46
- cudaMemcpyToSymbol
  - CUDART\_MEMORY, 47
- cudaMemcpyToSymbolAsync
  - CUDART\_MEMORY, 48
- cudaMemset
  - CUDART\_MEMORY, 48
- cudaMemset2D
  - CUDART\_MEMORY, 49
- cudaMemset3D
  - CUDART\_MEMORY, 49
- cudaPitchedPtr, 219
- cudaPos, 220
- CUDART\_D3D10
  - cudaD3D10MapFlagsNone, 66
  - cudaD3D10MapFlagsReadOnly, 66
  - cudaD3D10MapFlagsWriteDiscard, 66
  - cudaD3D10RegisterFlagsArray, 66
  - cudaD3D10RegisterFlagsNone, 66
- CUDART\_D3D9
  - cudaD3D9MapFlagsNone, 55
  - cudaD3D9MapFlagsReadOnly, 55
  - cudaD3D9MapFlagsWriteDiscard, 55
  - cudaD3D9RegisterFlagsArray, 55
  - cudaD3D9RegisterFlagsNone, 55
- CUDART\_TYPES
  - cudaChannelFormatKindFloat, 92
  - cudaChannelFormatKindNone, 92
  - cudaChannelFormatKindSigned, 92
  - cudaChannelFormatKindUnsigned, 92
  - cudaComputeModeDefault, 92
  - cudaComputeModeExclusive, 92
  - cudaComputeModeProhibited, 92
  - cudaErrorAddressOfConstant, 93
  - cudaErrorApiFailureBase, 93
  - cudaErrorCudartUnloading, 93
  - cudaErrorInitializationError, 92
  - cudaErrorInsufficientDriver, 93
  - cudaErrorInvalidChannelDescriptor, 92
  - cudaErrorInvalidConfiguration, 92
  - cudaErrorInvalidDevice, 92
  - cudaErrorInvalidDeviceFunction, 92
  - cudaErrorInvalidDevicePointer, 92
  - cudaErrorInvalidFilterSetting, 93
  - cudaErrorInvalidHostPointer, 92
  - cudaErrorInvalidMemcpyDirection, 93
  - cudaErrorInvalidNormSetting, 93
  - cudaErrorInvalidPitchValue, 92
  - cudaErrorInvalidResourceHandle, 93
  - cudaErrorInvalidSymbol, 92
  - cudaErrorInvalidTexture, 92
  - cudaErrorInvalidTextureBinding, 92
  - cudaErrorInvalidValue, 92
  - cudaErrorLaunchFailure, 92
  - cudaErrorLaunchOutOfResources, 92
  - cudaErrorLaunchTimeout, 92
  - cudaErrorMapBufferObjectFailed, 92
  - cudaErrorMemoryAllocation, 92
  - cudaErrorMemoryValueTooLarge, 93
  - cudaErrorMissingConfiguration, 92
  - cudaErrorMixedDeviceExecution, 93
  - cudaErrorNoDevice, 93
  - cudaErrorNotReady, 93
  - cudaErrorNotYetImplemented, 93
  - cudaErrorPriorLaunchFailure, 92
  - cudaErrorSetOnActiveProcess, 93
  - cudaErrorStartupFailure, 93
  - cudaErrorSynchronizationError, 93
  - cudaErrorTextureFetchFailed, 93
  - cudaErrorTextureNotBound, 93
  - cudaErrorUnknown, 93
  - cudaErrorUnmapBufferObjectFailed, 92
  - cudaMemcpyDeviceToDevice, 93
  - cudaMemcpyDeviceToHost, 93
  - cudaMemcpyHostToDevice, 93
  - cudaMemcpyHostToHost, 93
  - cudaSuccess, 92
- CUDART\_D3D10
  - cudaD3D10GetDevice, 66
  - cudaD3D10MapFlags, 66
  - cudaD3D10MapResources, 67
  - cudaD3D10RegisterFlags, 66
  - cudaD3D10RegisterResource, 67
  - cudaD3D10ResourceGetMappedArray, 68
  - cudaD3D10ResourceGetMappedPitch, 69
  - cudaD3D10ResourceGetMappedPointer, 70
  - cudaD3D10ResourceGetMappedSize, 70
  - cudaD3D10ResourceGetSurfaceDimensions, 71
  - cudaD3D10ResourceSetMapFlags, 72
  - cudaD3D10SetDirect3DDevice, 72
  - cudaD3D10UnmapResources, 73
  - cudaD3D10UnregisterResource, 74
- CUDART\_D3D9
  - cudaD3D9GetDevice, 55
  - cudaD3D9GetDirect3DDevice, 56
  - cudaD3D9MapFlags, 55
  - cudaD3D9MapResources, 56
  - cudaD3D9RegisterFlags, 55
  - cudaD3D9RegisterResource, 57
  - cudaD3D9ResourceGetMappedArray, 58
  - cudaD3D9ResourceGetMappedPitch, 59
  - cudaD3D9ResourceGetMappedPointer, 60
  - cudaD3D9ResourceGetMappedSize, 60
  - cudaD3D9ResourceGetSurfaceDimensions, 61
  - cudaD3D9ResourceSetMapFlags, 62

- cudaD3D9SetDirect3DDevice, 62
- cudaD3D9UnmapResources, 63
- cudaD3D9UnregisterResource, 64
- CUDART\_DEVICE
  - cudaChooseDevice, 9
  - cudaGetDevice, 9
  - cudaGetDeviceCount, 10
  - cudaGetDeviceProperties, 10
  - cudaSetDevice, 12
  - cudaSetDeviceFlags, 12
  - cudaSetValidDevices, 13
- CUDART\_ERROR
  - cudaGetErrorString, 7
  - cudaGetLastError, 7
- CUDART\_EVENT
  - cudaEventCreate, 16
  - cudaEventCreateWithFlags, 16
  - cudaEventDestroy, 17
  - cudaEventElapsedTime, 17
  - cudaEventQuery, 18
  - cudaEventRecord, 18
  - cudaEventSynchronize, 19
- CUDART\_EXECUTION
  - cudaConfigureCall, 20
  - cudaFuncGetAttributes, 21
  - cudaLaunch, 21
  - cudaSetDoubleForDevice, 21
  - cudaSetDoubleForHost, 22
  - cudaSetupArgument, 22
- CUDART\_HIGHLEVEL
  - cudaBindTexture, 82
  - cudaBindTexture2D, 83
  - cudaBindTextureToArray, 84
  - cudaCreateChannelDesc, 85
  - cudaGetSymbolAddress, 85
  - cudaGetSymbolSize, 85
  - cudaGetTextureAlignmentOffset, 86
  - cudaLaunch, 86
  - cudaSetupArgument, 87
  - cudaUnbindTexture, 87
- CUDART\_MEMORY
  - cudaFree, 25
  - cudaFreeArray, 26
  - cudaFreeHost, 26
  - cudaGetSymbolAddress, 26
  - cudaGetSymbolSize, 27
  - cudaHostAlloc, 27
  - cudaHostGetDevicePointer, 28
  - cudaMalloc, 29
  - cudaMalloc3D, 29
  - cudaMalloc3DArray, 30
  - cudaMallocArray, 31
  - cudaMallocHost, 31
  - cudaMallocPitch, 32
  - cudaMemcpy, 32
  - cudaMemcpy2D, 33
  - cudaMemcpy2DArrayToArray, 34
  - cudaMemcpy2DAsync, 34
  - cudaMemcpy2DFromArray, 35
  - cudaMemcpy2DFromArrayAsync, 36
  - cudaMemcpy2DToArray, 37
  - cudaMemcpy2DToArrayAsync, 38
  - cudaMemcpy3D, 39
  - cudaMemcpy3DAsync, 40
  - cudaMemcpyArrayToArray, 41
  - cudaMemcpyAsync, 42
  - cudaMemcpyFromArray, 43
  - cudaMemcpyFromArrayAsync, 43
  - cudaMemcpyFromSymbol, 44
  - cudaMemcpyFromSymbolAsync, 45
  - cudaMemcpyToArray, 46
  - cudaMemcpyToArrayAsync, 46
  - cudaMemcpyToSymbol, 47
  - cudaMemcpyToSymbolAsync, 48
  - cudaMemset, 48
  - cudaMemset2D, 49
  - cudaMemset3D, 49
- CUDART\_OPENGL
  - cudaGLMapBufferObject, 51
  - cudaGLRegisterBufferObject, 52
  - cudaGLSetGLDevice, 52
  - cudaGLUnmapBufferObject, 52
  - cudaGLUnregisterBufferObject, 53
  - cudaWGLGetDevice, 53
- CUDART\_STREAM
  - cudaStreamCreate, 14
  - cudaStreamDestroy, 14
  - cudaStreamQuery, 15
  - cudaStreamSynchronize, 15
- CUDART\_TEXTURE
  - cudaBindTexture, 75
  - cudaBindTexture2D, 76
  - cudaBindTextureToArray, 77
  - cudaCreateChannelDesc, 77
  - cudaGetChannelDesc, 78
  - cudaGetTextureAlignmentOffset, 78
  - cudaGetTextureReference, 79
  - cudaUnbindTexture, 79
- CUDART\_THREAD
  - cudaThreadExit, 6
  - cudaThreadSynchronize, 6
- CUDART\_TYPES
  - cudaChannelFormatKind, 91
  - cudaComputeMode, 92
  - cudaError, 92
  - cudaError\_t, 91
  - cudaEvent\_t, 91
  - cudaMemcpyKind, 93

- cudaStream\_t, 91
- CUDART\_VERSION
  - cudaDriverGetVersion, 80
  - cudaRuntimeGetVersion, 80
- cudaRuntimeGetVersion
  - CUDART\_VERSION, 80
- cudaSetDevice
  - CUDART\_DEVICE, 12
- cudaSetDeviceFlags
  - CUDART\_DEVICE, 12
- cudaSetDoubleForDevice
  - CUDART\_EXECUTION, 21
- cudaSetDoubleForHost
  - CUDART\_EXECUTION, 22
- cudaSetupArgument
  - CUDART\_EXECUTION, 22
  - CUDART\_HIGHLEVEL, 87
- cudaSetValidDevices
  - CUDART\_DEVICE, 13
- cudaStream\_t
  - CUDA\_TYPERES, 91
- cudaStreamCreate
  - CUDART\_STREAM, 14
- cudaStreamDestroy
  - CUDART\_STREAM, 14
- cudaStreamQuery
  - CUDART\_STREAM, 15
- cudaStreamSynchronize
  - CUDART\_STREAM, 15
- cudaSuccess
  - CUDART\_TYPERES, 92
- cudaThreadExit
  - CUDART\_THREAD, 6
- cudaThreadSynchronize
  - CUDART\_THREAD, 6
- cudaUnbindTexture
  - CUDART\_HIGHLEVEL, 87
  - CUDART\_TEXTURE, 79
- cudaWGLGetDevice
  - CUDART\_OPENGL, 53
- CUDEVICE
  - cuDeviceComputeCapability, 96
  - cuDeviceGet, 97
  - cuDeviceGetAttribute, 97
  - cuDeviceGetCount, 98
  - cuDeviceGetName, 99
  - cuDeviceGetProperties, 99
  - cuDeviceTotalMem, 100
- CUdevice\_attribute
  - CUDA\_TYPERES, 200
- CUdevice\_attribute\_enum
  - CUDA\_TYPERES, 202
- cuDeviceComputeCapability
  - CUDEVICE, 96
- cuDeviceGet
  - CUDEVICE, 97
- cuDeviceGetAttribute
  - CUDEVICE, 97
- cuDeviceGetCount
  - CUDEVICE, 98
- cuDeviceGetName
  - CUDEVICE, 99
- cuDeviceGetProperties
  - CUDEVICE, 99
- cuDeviceTotalMem
  - CUDEVICE, 100
- CUdevprop
  - CUDA\_TYPERES, 200
- CUdevprop\_st, 221
- cuDriverGetVersion
  - CUVERSION, 102
- CUEVENT
  - cuEventCreate, 116
  - cuEventDestroy, 117
  - cuEventElapsedTime, 117
  - cuEventQuery, 117
  - cuEventRecord, 118
  - cuEventSynchronize, 118
- CUevent\_flags
  - CUDA\_TYPERES, 200
- CUevent\_flags\_enum
  - CUDA\_TYPERES, 203
- cuEventCreate
  - CUEVENT, 116
- cuEventDestroy
  - CUEVENT, 117
- cuEventElapsedTime
  - CUEVENT, 117
- cuEventQuery
  - CUEVENT, 117
- cuEventRecord
  - CUEVENT, 118
- cuEventSynchronize
  - CUEVENT, 118
- CUEXEC
  - cuFuncGetAttribute, 120
  - cuFuncSetBlockShape, 121
  - cuFuncSetSharedSize, 121
  - cuLaunch, 122
  - cuLaunchGrid, 122
  - cuLaunchGridAsync, 123
  - cuParamSetf, 123
  - cuParamSeti, 124
  - cuParamSetSize, 124
  - cuParamSetTexRef, 125
  - cuParamSetv, 125
- CUfilter\_mode
  - CUDA\_TYPERES, 200

- CUfilter\_mode\_enum
  - CUDA\_TYPES, 203
- cuFuncGetAttribute
  - CUEXEC, 120
- cuFuncSetBlockShape
  - CUEXEC, 121
- cuFuncSetSharedSize
  - CUEXEC, 121
- CUfunction\_attribute
  - CUDA\_TYPES, 200
- CUfunction\_attribute\_enum
  - CUDA\_TYPES, 204
- CUGL
  - cuGLCtxCreate, 170
  - cuGLInit, 171
  - cuGLMapBufferObject, 171
  - cuGLRegisterBufferObject, 171
  - cuGLUnmapBufferObject, 172
  - cuGLUnregisterBufferObject, 172
  - cuWGLGetDevice, 173
- cuGLCtxCreate
  - CUGL, 170
- cuGLInit
  - CUGL, 171
- cuGLMapBufferObject
  - CUGL, 171
- cuGLRegisterBufferObject
  - CUGL, 171
- cuGLUnmapBufferObject
  - CUGL, 172
- cuGLUnregisterBufferObject
  - CUGL, 172
- CUINIT
  - cuInit, 95
- cuInit
  - CUINIT, 95
- CUjit\_fallback
  - CUDA\_TYPES, 200
- CUjit\_fallback\_enum
  - CUDA\_TYPES, 204
- CUjit\_option
  - CUDA\_TYPES, 200
- CUjit\_option\_enum
  - CUDA\_TYPES, 204
- CUjit\_target
  - CUDA\_TYPES, 200
- CUjit\_target\_enum
  - CUDA\_TYPES, 205
- cuLaunch
  - CUEXEC, 122
- cuLaunchGrid
  - CUEXEC, 122
- cuLaunchGridAsync
  - CUEXEC, 123
- CUMEM
  - cuArray3DCreate, 129
  - cuArray3DGetDescriptor, 131
  - cuArrayCreate, 131
  - cuArrayDestroy, 133
  - cuArrayGetDescriptor, 133
  - cuMemAlloc, 134
  - cuMemAllocHost, 134
  - cuMemAllocPitch, 135
  - cuMemcpy2D, 136
  - cuMemcpy2DAsync, 138
  - cuMemcpy2DUnaligned, 140
  - cuMemcpy3D, 142
  - cuMemcpy3DAsync, 145
  - cuMemcpyAtoA, 147
  - cuMemcpyAtoD, 148
  - cuMemcpyAtoH, 148
  - cuMemcpyAtoHAsync, 149
  - cuMemcpyDtoA, 149
  - cuMemcpyDtoD, 150
  - cuMemcpyDtoH, 151
  - cuMemcpyDtoHAsync, 151
  - cuMemcpyHtoA, 152
  - cuMemcpyHtoAAsync, 152
  - cuMemcpyHtoD, 153
  - cuMemcpyHtoDAsync, 154
  - cuMemFree, 154
  - cuMemFreeHost, 155
  - cuMemGetAddressRange, 155
  - cuMemGetInfo, 156
  - cuMemHostAlloc, 156
  - cuMemHostGetDevicePointer, 157
  - cuMemsetD16, 158
  - cuMemsetD2D16, 159
  - cuMemsetD2D32, 159
  - cuMemsetD2D8, 160
  - cuMemsetD32, 160
  - cuMemsetD8, 161
- cuMemAlloc
  - CUMEM, 134
- cuMemAllocHost
  - CUMEM, 134
- cuMemAllocPitch
  - CUMEM, 135
- cuMemcpy2D
  - CUMEM, 136
- cuMemcpy2DAsync
  - CUMEM, 138
- cuMemcpy2DUnaligned
  - CUMEM, 140
- cuMemcpy3D
  - CUMEM, 142
- cuMemcpy3DAsync
  - CUMEM, 145

- cuMemcpyAtoA
  - CUMEM, 147
- cuMemcpyAtoD
  - CUMEM, 148
- cuMemcpyAtoH
  - CUMEM, 148
- cuMemcpyAtoHAsync
  - CUMEM, 149
- cuMemcpyDtoA
  - CUMEM, 149
- cuMemcpyDtoD
  - CUMEM, 150
- cuMemcpyDtoH
  - CUMEM, 151
- cuMemcpyDtoHAsync
  - CUMEM, 151
- cuMemcpyHtoA
  - CUMEM, 152
- cuMemcpyHtoAAsync
  - CUMEM, 152
- cuMemcpyHtoD
  - CUMEM, 153
- cuMemcpyHtoDAsync
  - CUMEM, 154
- cuMemFree
  - CUMEM, 154
- cuMemFreeHost
  - CUMEM, 155
- cuMemGetAddressRange
  - CUMEM, 155
- cuMemGetInfo
  - CUMEM, 156
- cuMemHostAlloc
  - CUMEM, 156
- cuMemHostGetDevicePointer
  - CUMEM, 157
- CUmemorytype
  - CUDA\_TYPES, 200
- CUmemorytype\_enum
  - CUDA\_TYPES, 205
- cuMemsetD16
  - CUMEM, 158
- cuMemsetD2D16
  - CUMEM, 159
- cuMemsetD2D32
  - CUMEM, 159
- cuMemsetD2D8
  - CUMEM, 160
- cuMemsetD32
  - CUMEM, 160
- cuMemsetD8
  - CUMEM, 161
- CUMODULE
  - cuModuleGetFunction, 108
  - cuModuleGetGlobal, 109
  - cuModuleGetTexRef, 109
  - cuModuleLoad, 110
  - cuModuleLoadData, 110
  - cuModuleLoadDataEx, 111
  - cuModuleLoadFatBinary, 112
  - cuModuleUnload, 112
- cuModuleGetFunction
  - CUMODULE, 108
- cuModuleGetGlobal
  - CUMODULE, 109
- cuModuleGetTexRef
  - CUMODULE, 109
- cuModuleLoad
  - CUMODULE, 110
- cuModuleLoadData
  - CUMODULE, 110
- cuModuleLoadDataEx
  - CUMODULE, 111
- cuModuleLoadFatBinary
  - CUMODULE, 112
- cuModuleUnload
  - CUMODULE, 112
- cuParamSetf
  - CUEXEC, 123
- cuParamSeti
  - CUEXEC, 124
- cuParamSetSize
  - CUEXEC, 124
- cuParamSetTexRef
  - CUEXEC, 125
- cuParamSetv
  - CUEXEC, 125
- CUresult
  - CUDA\_TYPES, 201
- CUSTREAM
  - cuStreamCreate, 114
  - cuStreamDestroy, 114
  - cuStreamQuery, 115
  - cuStreamSynchronize, 115
- cuStreamCreate
  - CUSTREAM, 114
- cuStreamDestroy
  - CUSTREAM, 114
- cuStreamQuery
  - CUSTREAM, 115
- cuStreamSynchronize
  - CUSTREAM, 115
- CUTEXREF
  - cuTexRefCreate, 163
  - cuTexRefDestroy, 163
  - cuTexRefGetAddress, 163
  - cuTexRefGetAddressMode, 164
  - cuTexRefGetArray, 164



- cuTexRefGetFilterMode, [165](#)
- cuTexRefGetFlags, [165](#)
- cuTexRefGetFormat, [165](#)
- cuTexRefSetAddress, [166](#)
- cuTexRefSetAddress2D, [166](#)
- cuTexRefSetAddressMode, [167](#)
- cuTexRefSetArray, [167](#)
- cuTexRefSetFilterMode, [168](#)
- cuTexRefSetFlags, [168](#)
- cuTexRefSetFormat, [169](#)
- cuTexRefCreate
  - CUTEXREF, [163](#)
- cuTexRefDestroy
  - CUTEXREF, [163](#)
- cuTexRefGetAddress
  - CUTEXREF, [163](#)
- cuTexRefGetAddressMode
  - CUTEXREF, [164](#)
- cuTexRefGetArray
  - CUTEXREF, [164](#)
- cuTexRefGetFilterMode
  - CUTEXREF, [165](#)
- cuTexRefGetFlags
  - CUTEXREF, [165](#)
- cuTexRefGetFormat
  - CUTEXREF, [165](#)
- cuTexRefSetAddress
  - CUTEXREF, [166](#)
- cuTexRefSetAddress2D
  - CUTEXREF, [166](#)
- cuTexRefSetAddressMode
  - CUTEXREF, [167](#)
- cuTexRefSetArray
  - CUTEXREF, [167](#)
- cuTexRefSetFilterMode
  - CUTEXREF, [168](#)
- cuTexRefSetFlags
  - CUTEXREF, [168](#)
- cuTexRefSetFormat
  - CUTEXREF, [169](#)
- CUVERSION
  - cuDriverGetVersion, [102](#)
- cuWGLGetDevice
  - CUGL, [173](#)
  
- Data types used by CUDA driver, [195](#)
- Data types used by CUDA Runtime, [89](#)
- Device Management, [9](#), [96](#)
- Direct3D 10 Interoperability, [65](#), [185](#)
- Direct3D 9 Interoperability, [54](#), [174](#)
  
- Error Handling, [7](#)
- Event Management, [16](#), [116](#)
- Execution Control, [20](#), [120](#)
  
- Initialization, [95](#)
- Memory Management, [23](#), [127](#)
- Module Management, [108](#)
  
- OpenGL Interoperability, [51](#), [170](#)
  
- Stream Management, [14](#), [114](#)
  
- Texture Reference Management, [75](#), [162](#)
- Thread Management, [6](#)
  
- Version Management, [80](#), [102](#)



## **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 or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

## **Trademarks**

NVIDIA, the NVIDIA logo, GeForce, Tesla, and Quadro are trademarks or registered trademarks of NVIDIA Corporation. Other company and product names may be trademarks of the respective companies with which they are associated.

## **Copyright**

© 2007-2009 NVIDIA Corporation. All rights reserved.



**NVIDIA**

NVIDIA Corporation  
2701 San Tomas Expressway  
Santa Clara, CA 95050  
[www.nvidia.com](http://www.nvidia.com)