



# NVIDIA CUDA

## Reference Manual

Version 3.2 Beta

---

August 2010



# Contents

<b>1</b>	<b>Deprecated List</b>	<b>1</b>
<b>2</b>	<b>Module Index</b>	<b>5</b>
2.1	Modules . . . . .	5
<b>3</b>	<b>Data Structure Index</b>	<b>7</b>
3.1	Data Structures . . . . .	7
<b>4</b>	<b>Module Documentation</b>	<b>9</b>
4.1	CUDA Runtime API . . . . .	9
4.1.1	Detailed Description . . . . .	10
4.1.2	Define Documentation . . . . .	10
4.1.2.1	CUDART_VERSION . . . . .	10
4.2	Thread Management . . . . .	11
4.2.1	Detailed Description . . . . .	11
4.2.2	Function Documentation . . . . .	11
4.2.2.1	cudaThreadExit . . . . .	11
4.2.2.2	cudaThreadGetCacheConfig . . . . .	12
4.2.2.3	cudaThreadGetLimit . . . . .	12
4.2.2.4	cudaThreadSetCacheConfig . . . . .	13
4.2.2.5	cudaThreadSetLimit . . . . .	13
4.2.2.6	cudaThreadSynchronize . . . . .	14
4.3	Error Handling . . . . .	15
4.3.1	Detailed Description . . . . .	15
4.3.2	Function Documentation . . . . .	15
4.3.2.1	cudaGetErrorString . . . . .	15
4.3.2.2	cudaGetLastError . . . . .	15
4.3.2.3	cudaPeekAtLastError . . . . .	16
4.4	Device Management . . . . .	17

---

4.4.1	Detailed Description	17
4.4.2	Function Documentation	17
4.4.2.1	cudaChooseDevice	17
4.4.2.2	cudaGetDevice	18
4.4.2.3	cudaGetDeviceCount	18
4.4.2.4	cudaGetDeviceProperties	18
4.4.2.5	cudaSetDevice	20
4.4.2.6	cudaSetDeviceFlags	21
4.4.2.7	cudaSetValidDevices	21
4.5	Stream Management	23
4.5.1	Detailed Description	23
4.5.2	Function Documentation	23
4.5.2.1	cudaStreamCreate	23
4.5.2.2	cudaStreamDestroy	23
4.5.2.3	cudaStreamQuery	24
4.5.2.4	cudaStreamSynchronize	24
4.5.2.5	cudaStreamWaitEvent	25
4.6	Event Management	26
4.6.1	Detailed Description	26
4.6.2	Function Documentation	26
4.6.2.1	cudaEventCreate	26
4.6.2.2	cudaEventCreateWithFlags	27
4.6.2.3	cudaEventDestroy	27
4.6.2.4	cudaEventElapsedTime	28
4.6.2.5	cudaEventQuery	28
4.6.2.6	cudaEventRecord	29
4.6.2.7	cudaEventSynchronize	29
4.7	Execution Control	30
4.7.1	Detailed Description	30
4.7.2	Function Documentation	30
4.7.2.1	cudaConfigureCall	30
4.7.2.2	cudaFuncGetAttributes	31
4.7.2.3	cudaFuncSetCacheConfig	31
4.7.2.4	cudaLaunch	32
4.7.2.5	cudaSetDoubleForDevice	32
4.7.2.6	cudaSetDoubleForHost	33
4.7.2.7	cudaSetupArgument	33

---

4.8	Memory Management	34
4.8.1	Detailed Description	37
4.8.2	Function Documentation	37
4.8.2.1	cudaFree	37
4.8.2.2	cudaFreeArray	37
4.8.2.3	cudaFreeHost	38
4.8.2.4	cudaGetSymbolAddress	38
4.8.2.5	cudaGetSymbolSize	39
4.8.2.6	cudaHostAlloc	39
4.8.2.7	cudaHostGetDevicePointer	40
4.8.2.8	cudaHostGetFlags	41
4.8.2.9	cudaMalloc	41
4.8.2.10	cudaMalloc3D	41
4.8.2.11	cudaMalloc3DArray	42
4.8.2.12	cudaMallocArray	43
4.8.2.13	cudaMallocHost	44
4.8.2.14	cudaMallocPitch	44
4.8.2.15	cudaMemcpy	45
4.8.2.16	cudaMemcpy2D	45
4.8.2.17	cudaMemcpy2DArrayToArray	46
4.8.2.18	cudaMemcpy2DAsync	47
4.8.2.19	cudaMemcpy2DFromArray	48
4.8.2.20	cudaMemcpy2DFromArrayAsync	48
4.8.2.21	cudaMemcpy2DToArray	49
4.8.2.22	cudaMemcpy2DToArrayAsync	50
4.8.2.23	cudaMemcpy3D	51
4.8.2.24	cudaMemcpy3DAsync	52
4.8.2.25	cudaMemcpyArrayToArray	54
4.8.2.26	cudaMemcpyAsync	54
4.8.2.27	cudaMemcpyFromArray	55
4.8.2.28	cudaMemcpyFromArrayAsync	56
4.8.2.29	cudaMemcpyFromSymbol	56
4.8.2.30	cudaMemcpyFromSymbolAsync	57
4.8.2.31	cudaMemcpyToArray	58
4.8.2.32	cudaMemcpyToArrayAsync	58
4.8.2.33	cudaMemcpyToSymbol	59
4.8.2.34	cudaMemcpyToSymbolAsync	60

4.8.2.35	<code>cudaMemGetInfo</code>	60
4.8.2.36	<code>cudaMemset</code>	61
4.8.2.37	<code>cudaMemset2D</code>	61
4.8.2.38	<code>cudaMemset2DAsync</code>	62
4.8.2.39	<code>cudaMemset3D</code>	62
4.8.2.40	<code>cudaMemset3DAsync</code>	63
4.8.2.41	<code>cudaMemsetAsync</code>	64
4.8.2.42	<code>make_cudaExtent</code>	64
4.8.2.43	<code>make_cudaPitchedPtr</code>	64
4.8.2.44	<code>make_cudaPos</code>	65
4.9	OpenGL Interoperability	66
4.9.1	Detailed Description	66
4.9.2	Enumeration Type Documentation	66
4.9.2.1	<code>cudaGLMapFlags</code>	66
4.9.3	Function Documentation	67
4.9.3.1	<code>cudaGLSetGLDevice</code>	67
4.9.3.2	<code>cudaGraphicsGLRegisterBuffer</code>	67
4.9.3.3	<code>cudaGraphicsGLRegisterImage</code>	68
4.9.3.4	<code>cudaWGLGetDevice</code>	68
4.10	Direct3D 9 Interoperability	70
4.10.1	Detailed Description	70
4.10.2	Enumeration Type Documentation	71
4.10.2.1	<code>cudaD3D9DeviceList</code>	71
4.10.2.2	<code>cudaD3D9MapFlags</code>	71
4.10.2.3	<code>cudaD3D9RegisterFlags</code>	71
4.10.3	Function Documentation	71
4.10.3.1	<code>cudaD3D9GetDevice</code>	71
4.10.3.2	<code>cudaD3D9GetDevices</code>	72
4.10.3.3	<code>cudaD3D9GetDirect3DDevice</code>	72
4.10.3.4	<code>cudaD3D9SetDirect3DDevice</code>	73
4.10.3.5	<code>cudaGraphicsD3D9RegisterResource</code>	73
4.11	Direct3D 10 Interoperability	75
4.11.1	Detailed Description	75
4.11.2	Enumeration Type Documentation	76
4.11.2.1	<code>cudaD3D10DeviceList</code>	76
4.11.2.2	<code>cudaD3D10MapFlags</code>	76
4.11.2.3	<code>cudaD3D10RegisterFlags</code>	76

4.11.3	Function Documentation	76
4.11.3.1	cudaD3D10GetDevice	76
4.11.3.2	cudaD3D10GetDevices	77
4.11.3.3	cudaD3D10GetDirect3DDevice	77
4.11.3.4	cudaD3D10SetDirect3DDevice	78
4.11.3.5	cudaGraphicsD3D10RegisterResource	78
4.12	Direct3D 11 Interoperability	80
4.12.1	Detailed Description	80
4.12.2	Enumeration Type Documentation	80
4.12.2.1	cudaD3D11DeviceList	80
4.12.3	Function Documentation	81
4.12.3.1	cudaD3D11GetDevice	81
4.12.3.2	cudaD3D11GetDevices	81
4.12.3.3	cudaD3D11GetDirect3DDevice	82
4.12.3.4	cudaD3D11SetDirect3DDevice	82
4.12.3.5	cudaGraphicsD3D11RegisterResource	82
4.13	VDPAU Interoperability	84
4.13.1	Detailed Description	84
4.13.2	Function Documentation	84
4.13.2.1	cudaGraphicsVDPAURegisterOutputSurface	84
4.13.2.2	cudaGraphicsVDPAURegisterVideoSurface	85
4.13.2.3	cudaVDPAUGetDevice	85
4.13.2.4	cudaVDPAUSetVDPAUDevice	86
4.14	Graphics Interoperability	87
4.14.1	Detailed Description	87
4.14.2	Function Documentation	87
4.14.2.1	cudaGraphicsMapResources	87
4.14.2.2	cudaGraphicsResourceGetMappedPointer	88
4.14.2.3	cudaGraphicsResourceSetMapFlags	88
4.14.2.4	cudaGraphicsSubResourceGetMappedArray	89
4.14.2.5	cudaGraphicsUnmapResources	90
4.14.2.6	cudaGraphicsUnregisterResource	90
4.15	Texture Reference Management	91
4.15.1	Detailed Description	91
4.15.2	Function Documentation	91
4.15.2.1	cudaBindTexture	91
4.15.2.2	cudaBindTexture2D	92

4.15.2.3	<code>cudaBindTextureToArray</code>	93
4.15.2.4	<code>cudaCreateChannelDesc</code>	93
4.15.2.5	<code>cudaGetChannelDesc</code>	94
4.15.2.6	<code>cudaGetTextureAlignmentOffset</code>	94
4.15.2.7	<code>cudaGetTextureReference</code>	95
4.15.2.8	<code>cudaUnbindTexture</code>	95
4.16	Surface Reference Management	96
4.16.1	Detailed Description	96
4.16.2	Function Documentation	96
4.16.2.1	<code>cudaBindSurfaceToArray</code>	96
4.16.2.2	<code>cudaGetSurfaceReference</code>	96
4.17	Version Management	98
4.17.1	Function Documentation	98
4.17.1.1	<code>cudaDriverGetVersion</code>	98
4.17.1.2	<code>cudaRuntimeGetVersion</code>	98
4.18	C++ API Routines	99
4.18.1	Detailed Description	100
4.18.2	Function Documentation	100
4.18.2.1	<code>cudaBindSurfaceToArray</code>	100
4.18.2.2	<code>cudaBindSurfaceToArray</code>	101
4.18.2.3	<code>cudaBindTexture</code>	101
4.18.2.4	<code>cudaBindTexture</code>	102
4.18.2.5	<code>cudaBindTexture2D</code>	103
4.18.2.6	<code>cudaBindTexture2D</code>	103
4.18.2.7	<code>cudaBindTextureToArray</code>	104
4.18.2.8	<code>cudaBindTextureToArray</code>	105
4.18.2.9	<code>cudaCreateChannelDesc</code>	105
4.18.2.10	<code>cudaEventCreate</code>	106
4.18.2.11	<code>cudaFuncGetAttributes</code>	106
4.18.2.12	<code>cudaFuncSetCacheConfig</code>	107
4.18.2.13	<code>cudaGetSymbolAddress</code>	107
4.18.2.14	<code>cudaGetSymbolSize</code>	108
4.18.2.15	<code>cudaGetTextureAlignmentOffset</code>	108
4.18.2.16	<code>cudaLaunch</code>	109
4.18.2.17	<code>cudaMallocHost</code>	109
4.18.2.18	<code>cudaSetupArgument</code>	110
4.18.2.19	<code>cudaUnbindTexture</code>	111

4.19 Interactions with the CUDA Driver API . . . . .	112
4.19.1 Context Management . . . . .	112
4.19.2 Interactions between CUstream and cudaStream_t . . . . .	112
4.19.3 Interactions between CUevent and cudaEvent_t . . . . .	112
4.19.4 Interactions between CUarray and struct cudaArray * . . . . .	112
4.19.5 Interactions between CUgraphicsResource and cudaGraphicsResource_t . . . . .	113
4.20 Direct3D 9 Interoperability [DEPRECATED] . . . . .	114
4.20.1 Detailed Description . . . . .	114
4.20.2 Function Documentation . . . . .	114
4.20.2.1 cudaD3D9MapResources . . . . .	114
4.20.2.2 cudaD3D9RegisterResource . . . . .	115
4.20.2.3 cudaD3D9ResourceGetMappedArray . . . . .	116
4.20.2.4 cudaD3D9ResourceGetMappedPitch . . . . .	117
4.20.2.5 cudaD3D9ResourceGetMappedPointer . . . . .	118
4.20.2.6 cudaD3D9ResourceGetMappedSize . . . . .	119
4.20.2.7 cudaD3D9ResourceGetSurfaceDimensions . . . . .	119
4.20.2.8 cudaD3D9ResourceSetMapFlags . . . . .	120
4.20.2.9 cudaD3D9UnmapResources . . . . .	121
4.20.2.10 cudaD3D9UnregisterResource . . . . .	121
4.21 Direct3D 10 Interoperability [DEPRECATED] . . . . .	122
4.21.1 Detailed Description . . . . .	122
4.21.2 Function Documentation . . . . .	122
4.21.2.1 cudaD3D10MapResources . . . . .	122
4.21.2.2 cudaD3D10RegisterResource . . . . .	123
4.21.2.3 cudaD3D10ResourceGetMappedArray . . . . .	124
4.21.2.4 cudaD3D10ResourceGetMappedPitch . . . . .	125
4.21.2.5 cudaD3D10ResourceGetMappedPointer . . . . .	126
4.21.2.6 cudaD3D10ResourceGetMappedSize . . . . .	126
4.21.2.7 cudaD3D10ResourceGetSurfaceDimensions . . . . .	127
4.21.2.8 cudaD3D10ResourceSetMapFlags . . . . .	128
4.21.2.9 cudaD3D10UnmapResources . . . . .	128
4.21.2.10 cudaD3D10UnregisterResource . . . . .	129
4.22 OpenGL Interoperability [DEPRECATED] . . . . .	130
4.22.1 Detailed Description . . . . .	130
4.22.2 Function Documentation . . . . .	130
4.22.2.1 cudaGLMapBufferObject . . . . .	130
4.22.2.2 cudaGLMapBufferObjectAsync . . . . .	131

---

4.22.2.3	<code>cudaGLRegisterBufferObject</code>	131
4.22.2.4	<code>cudaGLSetBufferObjectMapFlags</code>	132
4.22.2.5	<code>cudaGLUnmapBufferObject</code>	133
4.22.2.6	<code>cudaGLUnmapBufferObjectAsync</code>	133
4.22.2.7	<code>cudaGLUnregisterBufferObject</code>	134
4.23	Data types used by CUDA Runtime	135
4.23.1	Define Documentation	138
4.23.1.1	<code>cudaArrayDefault</code>	138
4.23.1.2	<code>cudaArraySurfaceLoadStore</code>	138
4.23.1.3	<code>cudaDeviceBlockingSync</code>	138
4.23.1.4	<code>cudaDeviceLmemResizeToMax</code>	138
4.23.1.5	<code>cudaDeviceMapHost</code>	138
4.23.1.6	<code>cudaDeviceMask</code>	138
4.23.1.7	<code>cudaDevicePropDontCare</code>	139
4.23.1.8	<code>cudaDeviceScheduleAuto</code>	139
4.23.1.9	<code>cudaDeviceScheduleSpin</code>	139
4.23.1.10	<code>cudaDeviceScheduleYield</code>	139
4.23.1.11	<code>cudaEventBlockingSync</code>	139
4.23.1.12	<code>cudaEventDefault</code>	139
4.23.1.13	<code>cudaEventDisableTiming</code>	139
4.23.1.14	<code>cudaHostAllocDefault</code>	139
4.23.1.15	<code>cudaHostAllocMapped</code>	139
4.23.1.16	<code>cudaHostAllocPortable</code>	139
4.23.1.17	<code>cudaHostAllocWriteCombined</code>	139
4.23.2	Typedef Documentation	140
4.23.2.1	<code>cudaError_t</code>	140
4.23.2.2	<code>cudaEvent_t</code>	140
4.23.2.3	<code>cudaGraphicsResource_t</code>	140
4.23.2.4	<code>cudaStream_t</code>	140
4.23.2.5	<code>cudaUUID_t</code>	140
4.23.3	Enumeration Type Documentation	140
4.23.3.1	<code>cudaChannelFormatKind</code>	140
4.23.3.2	<code>cudaComputeMode</code>	140
4.23.3.3	<code>cudaError</code>	141
4.23.3.4	<code>cudaFuncCache</code>	144
4.23.3.5	<code>cudaGraphicsCubeFace</code>	144
4.23.3.6	<code>cudaGraphicsMapFlags</code>	144

4.23.3.7	<code>cudaGraphicsRegisterFlags</code>	144
4.23.3.8	<code>cudaLimit</code>	145
4.23.3.9	<code>cudaMemcpyKind</code>	145
4.23.3.10	<code>cudaSurfaceBoundaryMode</code>	145
4.23.3.11	<code>cudaSurfaceFormatMode</code>	145
4.23.3.12	<code>cudaTextureAddressMode</code>	145
4.23.3.13	<code>cudaTextureFilterMode</code>	146
4.23.3.14	<code>cudaTextureReadMode</code>	146
4.24	CUDA Driver API	147
4.24.1	Detailed Description	147
4.25	Data types used by CUDA driver	148
4.25.1	Define Documentation	153
4.25.1.1	<code>CU_MEMHOSTALLOC_DEVICEMAP</code>	153
4.25.1.2	<code>CU_MEMHOSTALLOC_PORTABLE</code>	153
4.25.1.3	<code>CU_MEMHOSTALLOC_WRITECOMBINED</code>	153
4.25.1.4	<code>CU_PARAM_TR_DEFAULT</code>	153
4.25.1.5	<code>CU_TRSA_OVERRIDE_FORMAT</code>	153
4.25.1.6	<code>CU_TRSF_NORMALIZED_COORDINATES</code>	153
4.25.1.7	<code>CU_TRSF_READ_AS_INTEGER</code>	153
4.25.1.8	<code>CU_TRSF_SRGB</code>	153
4.25.1.9	<code>CUDA_ARRAY3D_2DARRAY</code>	153
4.25.1.10	<code>CUDA_ARRAY3D_SURFACE_LDST</code>	153
4.25.1.11	<code>CUDA_VERSION</code>	154
4.25.2	Typedef Documentation	154
4.25.2.1	<code>CUaddress_mode</code>	154
4.25.2.2	<code>CUarray</code>	154
4.25.2.3	<code>CUarray_cubemap_face</code>	154
4.25.2.4	<code>CUarray_format</code>	154
4.25.2.5	<code>CUcomputemode</code>	154
4.25.2.6	<code>CUcontext</code>	154
4.25.2.7	<code>CUctx_flags</code>	154
4.25.2.8	<code>CUDA_ARRAY3D_DESCRIPTOR</code>	154
4.25.2.9	<code>CUDA_ARRAY_DESCRIPTOR</code>	154
4.25.2.10	<code>CUDA_MEMCPY2D</code>	154
4.25.2.11	<code>CUDA_MEMCPY3D</code>	155
4.25.2.12	<code>CUdevice</code>	155
4.25.2.13	<code>CUdevice_attribute</code>	155

---

4.25.2.14	CUdeviceptr	155
4.25.2.15	CUdevprop	155
4.25.2.16	CUevent	155
4.25.2.17	CUevent_flags	155
4.25.2.18	CUfilter_mode	155
4.25.2.19	CUfunc_cache	155
4.25.2.20	CUfunction	155
4.25.2.21	CUfunction_attribute	155
4.25.2.22	CUgraphicsMapResourceFlags	155
4.25.2.23	CUgraphicsRegisterFlags	156
4.25.2.24	CUgraphicsResource	156
4.25.2.25	CUjit_fallback	156
4.25.2.26	CUjit_option	156
4.25.2.27	CUjit_target	156
4.25.2.28	CULimit	156
4.25.2.29	CUmemorytype	156
4.25.2.30	CUmodule	156
4.25.2.31	CUresult	156
4.25.2.32	CUstream	156
4.25.2.33	CUsurfref	156
4.25.2.34	CUtexref	156
4.25.3	Enumeration Type Documentation	157
4.25.3.1	CUaddress_mode_enum	157
4.25.3.2	CUarray_cubemap_face_enum	157
4.25.3.3	CUarray_format_enum	157
4.25.3.4	CUcomputemode_enum	157
4.25.3.5	CUctx_flags_enum	158
4.25.3.6	cudaError_enum	158
4.25.3.7	CUdevice_attribute_enum	160
4.25.3.8	CUevent_flags_enum	161
4.25.3.9	CUfilter_mode_enum	161
4.25.3.10	CUfunc_cache_enum	161
4.25.3.11	CUfunction_attribute_enum	162
4.25.3.12	CUgraphicsMapResourceFlags_enum	162
4.25.3.13	CUgraphicsRegisterFlags_enum	162
4.25.3.14	CUjit_fallback_enum	162
4.25.3.15	CUjit_option_enum	162

---

4.25.3.16	CUjit_target_enum	163
4.25.3.17	CULimit_enum	164
4.25.3.18	CUMemoryType_enum	164
4.26	Initialization	165
4.26.1	Detailed Description	165
4.26.2	Function Documentation	165
4.26.2.1	cuInit	165
4.27	Version Management	166
4.27.1	Detailed Description	166
4.27.2	Function Documentation	166
4.27.2.1	cuDriverGetVersion	166
4.28	Device Management	167
4.28.1	Detailed Description	167
4.28.2	Function Documentation	167
4.28.2.1	cuDeviceComputeCapability	167
4.28.2.2	cuDeviceGet	168
4.28.2.3	cuDeviceGetAttribute	168
4.28.2.4	cuDeviceGetCount	170
4.28.2.5	cuDeviceGetName	170
4.28.2.6	cuDeviceGetProperties	171
4.28.2.7	cuDeviceTotalMem	172
4.29	Context Management	173
4.29.1	Detailed Description	173
4.29.2	Function Documentation	174
4.29.2.1	cuCtxAttach	174
4.29.2.2	cuCtxCreate	174
4.29.2.3	cuCtxDestroy	175
4.29.2.4	cuCtxDetach	176
4.29.2.5	cuCtxGetApiVersion	176
4.29.2.6	cuCtxGetCacheConfig	177
4.29.2.7	cuCtxGetDevice	177
4.29.2.8	cuCtxGetLimit	178
4.29.2.9	cuCtxPopCurrent	178
4.29.2.10	cuCtxPushCurrent	179
4.29.2.11	cuCtxSetCacheConfig	179
4.29.2.12	cuCtxSetLimit	180
4.29.2.13	cuCtxSynchronize	181

---

4.30	Module Management	182
4.30.1	Detailed Description	182
4.30.2	Function Documentation	182
4.30.2.1	cuModuleGetFunction	182
4.30.2.2	cuModuleGetGlobal	183
4.30.2.3	cuModuleGetSurfRef	183
4.30.2.4	cuModuleGetTexRef	184
4.30.2.5	cuModuleLoad	184
4.30.2.6	cuModuleLoadData	185
4.30.2.7	cuModuleLoadDataEx	185
4.30.2.8	cuModuleLoadFatBinary	187
4.30.2.9	cuModuleUnload	187
4.31	Memory Management	188
4.31.1	Detailed Description	191
4.31.2	Function Documentation	191
4.31.2.1	cuArray3DCreate	191
4.31.2.2	cuArray3DGetDescriptor	192
4.31.2.3	cuArrayCreate	193
4.31.2.4	cuArrayDestroy	195
4.31.2.5	cuArrayGetDescriptor	195
4.31.2.6	cuMemAlloc	196
4.31.2.7	cuMemAllocHost	196
4.31.2.8	cuMemAllocPitch	197
4.31.2.9	cuMemcpy2D	198
4.31.2.10	cuMemcpy2DAsync	200
4.31.2.11	cuMemcpy2DUnaligned	202
4.31.2.12	cuMemcpy3D	204
4.31.2.13	cuMemcpy3DAsync	207
4.31.2.14	cuMemcpyAtoA	209
4.31.2.15	cuMemcpyAtoD	210
4.31.2.16	cuMemcpyAtoH	210
4.31.2.17	cuMemcpyAtoHAsync	211
4.31.2.18	cuMemcpyDtoA	211
4.31.2.19	cuMemcpyDtoD	212
4.31.2.20	cuMemcpyDtoDAsync	213
4.31.2.21	cuMemcpyDtoH	213
4.31.2.22	cuMemcpyDtoHAsync	214

4.31.2.23	cuMemcpyHtoA	214
4.31.2.24	cuMemcpyHtoAAsync	215
4.31.2.25	cuMemcpyHtoD	216
4.31.2.26	cuMemcpyHtoDAsync	216
4.31.2.27	cuMemFree	217
4.31.2.28	cuMemFreeHost	218
4.31.2.29	cuMemGetAddressRange	218
4.31.2.30	cuMemGetInfo	219
4.31.2.31	cuMemHostAlloc	219
4.31.2.32	cuMemHostGetDevicePointer	220
4.31.2.33	cuMemHostGetFlags	221
4.31.2.34	cuMemsetD16	221
4.31.2.35	cuMemsetD16Async	222
4.31.2.36	cuMemsetD2D16	222
4.31.2.37	cuMemsetD2D16Async	223
4.31.2.38	cuMemsetD2D32	224
4.31.2.39	cuMemsetD2D32Async	224
4.31.2.40	cuMemsetD2D8	225
4.31.2.41	cuMemsetD2D8Async	226
4.31.2.42	cuMemsetD32	226
4.31.2.43	cuMemsetD32Async	227
4.31.2.44	cuMemsetD8	228
4.31.2.45	cuMemsetD8Async	228
4.32	Stream Management	230
4.32.1	Detailed Description	230
4.32.2	Function Documentation	230
4.32.2.1	cuStreamCreate	230
4.32.2.2	cuStreamDestroy	231
4.32.2.3	cuStreamQuery	231
4.32.2.4	cuStreamSynchronize	231
4.32.2.5	cuStreamWaitEvent	232
4.33	Event Management	233
4.33.1	Detailed Description	233
4.33.2	Function Documentation	233
4.33.2.1	cuEventCreate	233
4.33.2.2	cuEventDestroy	234
4.33.2.3	cuEventElapsedTime	234

4.33.2.4	cuEventQuery	235
4.33.2.5	cuEventRecord	235
4.33.2.6	cuEventSynchronize	236
4.34	Execution Control	237
4.34.1	Detailed Description	237
4.34.2	Function Documentation	238
4.34.2.1	cuFuncGetAttribute	238
4.34.2.2	cuFuncSetBlockShape	239
4.34.2.3	cuFuncSetCacheConfig	239
4.34.2.4	cuFuncSetSharedSize	240
4.34.2.5	cuLaunch	240
4.34.2.6	cuLaunchGrid	241
4.34.2.7	cuLaunchGridAsync	241
4.34.2.8	cuParamSetf	242
4.34.2.9	cuParamSeti	242
4.34.2.10	cuParamSetSize	243
4.34.2.11	cuParamSetv	243
4.35	Execution Control [DEPRECATED]	244
4.35.1	Detailed Description	244
4.35.2	Function Documentation	244
4.35.2.1	cuParamSetTexRef	244
4.36	Texture Reference Management	245
4.36.1	Detailed Description	246
4.36.2	Function Documentation	246
4.36.2.1	cuTexRefGetAddress	246
4.36.2.2	cuTexRefGetAddressMode	246
4.36.2.3	cuTexRefGetArray	247
4.36.2.4	cuTexRefGetFilterMode	247
4.36.2.5	cuTexRefGetFlags	247
4.36.2.6	cuTexRefGetFormat	248
4.36.2.7	cuTexRefSetAddress	248
4.36.2.8	cuTexRefSetAddress2D	249
4.36.2.9	cuTexRefSetAddressMode	249
4.36.2.10	cuTexRefSetArray	250
4.36.2.11	cuTexRefSetFilterMode	250
4.36.2.12	cuTexRefSetFlags	251
4.36.2.13	cuTexRefSetFormat	251

4.37 Texture Reference Management [DEPRECATED]	253
4.37.1 Detailed Description	253
4.37.2 Function Documentation	253
4.37.2.1 cuTexRefCreate	253
4.37.2.2 cuTexRefDestroy	253
4.38 Surface Reference Management	255
4.38.1 Detailed Description	255
4.38.2 Function Documentation	255
4.38.2.1 cuSurfRefGetArray	255
4.38.2.2 cuSurfRefSetArray	255
4.39 Graphics Interoperability	257
4.39.1 Detailed Description	257
4.39.2 Function Documentation	257
4.39.2.1 cuGraphicsMapResources	257
4.39.2.2 cuGraphicsResourceGetMappedPointer	258
4.39.2.3 cuGraphicsResourceSetMapFlags	258
4.39.2.4 cuGraphicsSubResourceGetMappedArray	259
4.39.2.5 cuGraphicsUnmapResources	260
4.39.2.6 cuGraphicsUnregisterResource	260
4.40 OpenGL Interoperability	262
4.40.1 Detailed Description	262
4.40.2 Function Documentation	262
4.40.2.1 cuGLCtxCreate	262
4.40.2.2 cuGraphicsGLRegisterBuffer	263
4.40.2.3 cuGraphicsGLRegisterImage	263
4.40.2.4 cuWGLGetDevice	264
4.41 OpenGL Interoperability [DEPRECATED]	265
4.41.1 Detailed Description	265
4.41.2 Typedef Documentation	265
4.41.2.1 CUGLmap_flags	265
4.41.3 Enumeration Type Documentation	266
4.41.3.1 CUGLmap_flags_enum	266
4.41.4 Function Documentation	266
4.41.4.1 cuGLInit	266
4.41.4.2 cuGLMapBufferObject	266
4.41.4.3 cuGLMapBufferObjectAsync	267
4.41.4.4 cuGLRegisterBufferObject	267

4.41.4.5	cuGLSetBufferObjectMapFlags . . . . .	268
4.41.4.6	cuGLUnmapBufferObject . . . . .	269
4.41.4.7	cuGLUnmapBufferObjectAsync . . . . .	269
4.41.4.8	cuGLUnregisterBufferObject . . . . .	270
4.42	Direct3D 9 Interoperability . . . . .	271
4.42.1	Detailed Description . . . . .	271
4.42.2	Typedef Documentation . . . . .	272
4.42.2.1	CUd3d9DeviceList . . . . .	272
4.42.3	Enumeration Type Documentation . . . . .	272
4.42.3.1	CUd3d9DeviceList_enum . . . . .	272
4.42.4	Function Documentation . . . . .	272
4.42.4.1	cuD3D9CtxCreate . . . . .	272
4.42.4.2	cuD3D9CtxCreateOnDevice . . . . .	273
4.42.4.3	cuD3D9GetDevice . . . . .	273
4.42.4.4	cuD3D9GetDevices . . . . .	274
4.42.4.5	cuD3D9GetDirect3DDevice . . . . .	274
4.42.4.6	cuGraphicsD3D9RegisterResource . . . . .	275
4.43	Direct3D 9 Interoperability [DEPRECATED] . . . . .	277
4.43.1	Detailed Description . . . . .	278
4.43.2	Typedef Documentation . . . . .	278
4.43.2.1	CUd3d9map_flags . . . . .	278
4.43.2.2	CUd3d9register_flags . . . . .	278
4.43.3	Enumeration Type Documentation . . . . .	278
4.43.3.1	CUd3d9map_flags_enum . . . . .	278
4.43.3.2	CUd3d9register_flags_enum . . . . .	278
4.43.4	Function Documentation . . . . .	278
4.43.4.1	cuD3D9MapResources . . . . .	278
4.43.4.2	cuD3D9RegisterResource . . . . .	279
4.43.4.3	cuD3D9ResourceGetMappedArray . . . . .	280
4.43.4.4	cuD3D9ResourceGetMappedPitch . . . . .	281
4.43.4.5	cuD3D9ResourceGetMappedPointer . . . . .	282
4.43.4.6	cuD3D9ResourceGetMappedSize . . . . .	282
4.43.4.7	cuD3D9ResourceGetSurfaceDimensions . . . . .	283
4.43.4.8	cuD3D9ResourceSetMapFlags . . . . .	284
4.43.4.9	cuD3D9UnmapResources . . . . .	285
4.43.4.10	cuD3D9UnregisterResource . . . . .	285
4.44	Direct3D 10 Interoperability . . . . .	286

4.44.1	Detailed Description	286
4.44.2	Typedef Documentation	287
4.44.2.1	CUD3d10DeviceList	287
4.44.3	Enumeration Type Documentation	287
4.44.3.1	CUD3d10DeviceList_enum	287
4.44.4	Function Documentation	287
4.44.4.1	cuD3D10CtxCreate	287
4.44.4.2	cuD3D10CtxCreateOnDevice	288
4.44.4.3	cuD3D10GetDevice	288
4.44.4.4	cuD3D10GetDevices	289
4.44.4.5	cuD3D10GetDirect3DDevice	289
4.44.4.6	cuGraphicsD3D10RegisterResource	290
4.45	Direct3D 10 Interoperability [DEPRECATED]	292
4.45.1	Detailed Description	293
4.45.2	Typedef Documentation	293
4.45.2.1	CUD3D10map_flags	293
4.45.2.2	CUD3D10register_flags	293
4.45.3	Enumeration Type Documentation	293
4.45.3.1	CUD3D10map_flags_enum	293
4.45.3.2	CUD3D10register_flags_enum	293
4.45.4	Function Documentation	293
4.45.4.1	cuD3D10MapResources	293
4.45.4.2	cuD3D10RegisterResource	294
4.45.4.3	cuD3D10ResourceGetMappedArray	295
4.45.4.4	cuD3D10ResourceGetMappedPitch	296
4.45.4.5	cuD3D10ResourceGetMappedPointer	297
4.45.4.6	cuD3D10ResourceGetMappedSize	297
4.45.4.7	cuD3D10ResourceGetSurfaceDimensions	298
4.45.4.8	cuD3D10ResourceSetMapFlags	299
4.45.4.9	cuD3D10UnmapResources	299
4.45.4.10	cuD3D10UnregisterResource	300
4.46	Direct3D 11 Interoperability	301
4.46.1	Detailed Description	301
4.46.2	Typedef Documentation	301
4.46.2.1	CUD3d11DeviceList	301
4.46.3	Enumeration Type Documentation	302
4.46.3.1	CUD3d11DeviceList_enum	302

4.46.4	Function Documentation	302
4.46.4.1	cuD3D11CtxCreate	302
4.46.4.2	cuD3D11CtxCreateOnDevice	302
4.46.4.3	cuD3D11GetDevice	303
4.46.4.4	cuD3D11GetDevices	304
4.46.4.5	cuD3D11GetDirect3DDevice	304
4.46.4.6	cuGraphicsD3D11RegisterResource	305
4.47	VDPAU Interoperability	307
4.47.1	Detailed Description	307
4.47.2	Function Documentation	307
4.47.2.1	cuGraphicsVDPAURegisterOutputSurface	307
4.47.2.2	cuGraphicsVDPAURegisterVideoSurface	308
4.47.2.3	cuVDPAUCtxCreate	309
4.47.2.4	cuVDPAUGetDevice	309
<b>5</b>	<b>Data Structure Documentation</b>	<b>311</b>
5.1	CUDA_ARRAY3D_DESCRIPTOR_st Struct Reference	311
5.1.1	Detailed Description	311
5.1.2	Field Documentation	311
5.1.2.1	Depth	311
5.1.2.2	Flags	311
5.1.2.3	Format	311
5.1.2.4	Height	311
5.1.2.5	NumChannels	312
5.1.2.6	Width	312
5.2	CUDA_ARRAY_DESCRIPTOR_st Struct Reference	313
5.2.1	Detailed Description	313
5.2.2	Field Documentation	313
5.2.2.1	Format	313
5.2.2.2	Height	313
5.2.2.3	NumChannels	313
5.2.2.4	Width	313
5.3	CUDA_MEMCPY2D_st Struct Reference	314
5.3.1	Detailed Description	314
5.3.2	Field Documentation	314
5.3.2.1	dstArray	314
5.3.2.2	dstDevice	314

---

5.3.2.3	dstHost	314
5.3.2.4	dstMemoryType	314
5.3.2.5	dstPitch	314
5.3.2.6	dstXInBytes	315
5.3.2.7	dstY	315
5.3.2.8	Height	315
5.3.2.9	srcArray	315
5.3.2.10	srcDevice	315
5.3.2.11	srcHost	315
5.3.2.12	srcMemoryType	315
5.3.2.13	srcPitch	315
5.3.2.14	srcXInBytes	315
5.3.2.15	srcY	315
5.3.2.16	WidthInBytes	315
5.4	CUDA_MEMCPY3D_st Struct Reference	316
5.4.1	Detailed Description	316
5.4.2	Field Documentation	316
5.4.2.1	Depth	316
5.4.2.2	dstArray	316
5.4.2.3	dstDevice	316
5.4.2.4	dstHeight	317
5.4.2.5	dstHost	317
5.4.2.6	dstLOD	317
5.4.2.7	dstMemoryType	317
5.4.2.8	dstPitch	317
5.4.2.9	dstXInBytes	317
5.4.2.10	dstY	317
5.4.2.11	dstZ	317
5.4.2.12	Height	317
5.4.2.13	reserved0	317
5.4.2.14	reserved1	317
5.4.2.15	srcArray	317
5.4.2.16	srcDevice	318
5.4.2.17	srcHeight	318
5.4.2.18	srcHost	318
5.4.2.19	srcLOD	318
5.4.2.20	srcMemoryType	318

---

5.4.2.21	srcPitch	318
5.4.2.22	srcXInBytes	318
5.4.2.23	srcY	318
5.4.2.24	srcZ	318
5.4.2.25	WidthInBytes	318
5.5	cudaChannelFormatDesc Struct Reference	319
5.5.1	Detailed Description	319
5.5.2	Field Documentation	319
5.5.2.1	f	319
5.5.2.2	w	319
5.5.2.3	x	319
5.5.2.4	y	319
5.5.2.5	z	319
5.6	cudaDeviceProp Struct Reference	320
5.6.1	Detailed Description	320
5.6.2	Field Documentation	320
5.6.2.1	canMapHostMemory	320
5.6.2.2	clockRate	320
5.6.2.3	computeMode	321
5.6.2.4	concurrentKernels	321
5.6.2.5	deviceOverlap	321
5.6.2.6	ECCEEnabled	321
5.6.2.7	integrated	321
5.6.2.8	kernelExecTimeoutEnabled	321
5.6.2.9	major	321
5.6.2.10	maxGridSize	321
5.6.2.11	maxTexture1D	321
5.6.2.12	maxTexture2D	321
5.6.2.13	maxTexture2DArray	321
5.6.2.14	maxTexture3D	321
5.6.2.15	maxThreadsDim	322
5.6.2.16	maxThreadsPerBlock	322
5.6.2.17	memPitch	322
5.6.2.18	minor	322
5.6.2.19	multiProcessorCount	322
5.6.2.20	name	322
5.6.2.21	pciBusID	322

5.6.2.22	pciDeviceID	322
5.6.2.23	regsPerBlock	322
5.6.2.24	sharedMemPerBlock	322
5.6.2.25	surfaceAlignment	322
5.6.2.26	tccDriver	322
5.6.2.27	textureAlignment	323
5.6.2.28	totalConstMem	323
5.6.2.29	totalGlobalMem	323
5.6.2.30	warpSize	323
5.7	cudaExtent Struct Reference	324
5.7.1	Detailed Description	324
5.7.2	Field Documentation	324
5.7.2.1	depth	324
5.7.2.2	height	324
5.7.2.3	width	324
5.8	cudaFuncAttributes Struct Reference	325
5.8.1	Detailed Description	325
5.8.2	Field Documentation	325
5.8.2.1	binaryVersion	325
5.8.2.2	constSizeBytes	325
5.8.2.3	localSizeBytes	325
5.8.2.4	maxThreadsPerBlock	325
5.8.2.5	numRegs	325
5.8.2.6	ptxVersion	325
5.8.2.7	sharedSizeBytes	326
5.9	cudaMemcpy3DParms Struct Reference	327
5.9.1	Detailed Description	327
5.9.2	Field Documentation	327
5.9.2.1	dstArray	327
5.9.2.2	dstPos	327
5.9.2.3	dstPtr	327
5.9.2.4	extent	327
5.9.2.5	kind	327
5.9.2.6	srcArray	327
5.9.2.7	srcPos	327
5.9.2.8	srcPtr	328
5.10	cudaPitchedPtr Struct Reference	329

---

5.10.1	Detailed Description	329
5.10.2	Field Documentation	329
5.10.2.1	pitch	329
5.10.2.2	ptr	329
5.10.2.3	xsize	329
5.10.2.4	ysize	329
5.11	cudaPos Struct Reference	330
5.11.1	Detailed Description	330
5.11.2	Field Documentation	330
5.11.2.1	x	330
5.11.2.2	y	330
5.11.2.3	z	330
5.12	CUdevprop_st Struct Reference	331
5.12.1	Detailed Description	331
5.12.2	Field Documentation	331
5.12.2.1	clockRate	331
5.12.2.2	maxGridSize	331
5.12.2.3	maxThreadsDim	331
5.12.2.4	maxThreadsPerBlock	331
5.12.2.5	memPitch	331
5.12.2.6	regsPerBlock	331
5.12.2.7	sharedMemPerBlock	331
5.12.2.8	SIMDWidth	332
5.12.2.9	textureAlign	332
5.12.2.10	totalConstantMemory	332
5.13	surfaceReference Struct Reference	333
5.13.1	Detailed Description	333
5.13.2	Field Documentation	333
5.13.2.1	channelDesc	333
5.14	textureReference Struct Reference	334
5.14.1	Detailed Description	334
5.14.2	Field Documentation	334
5.14.2.1	addressMode	334
5.14.2.2	channelDesc	334
5.14.2.3	filterMode	334
5.14.2.4	normalized	334

# Chapter 1

## Deprecated List

Global [cudaD3D9MapResources](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D9RegisterResource](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D9ResourceGetMappedArray](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D9ResourceGetMappedPitch](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D9ResourceGetMappedPointer](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D9ResourceGetMappedSize](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D9ResourceGetSurfaceDimensions](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D9ResourceSetMapFlags](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D9UnmapResources](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D9UnregisterResource](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D10MapResources](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D10RegisterResource](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D10ResourceGetMappedArray](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D10ResourceGetMappedPitch](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D10ResourceGetMappedPointer](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D10ResourceGetMappedSize](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D10ResourceGetSurfaceDimensions](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D10ResourceSetMapFlags](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D10UnmapResources](#) This function is deprecated as of Cuda 3.0.

Global [cudaD3D10UnregisterResource](#) This function is deprecated as of Cuda 3.0.

Global [cudaGLMapBufferObject](#) This function is deprecated as of Cuda 3.0.

Global [cudaGLMapBufferObjectAsync](#) This function is deprecated as of Cuda 3.0.

Global [cudaGLRegisterBufferObject](#) This function is deprecated as of Cuda 3.0.

Global [cudaGLSetBufferObjectMapFlags](#) This function is deprecated as of Cuda 3.0.

Global [cudaGLUnmapBufferObject](#) This function is deprecated as of Cuda 3.0.

Global [cudaGLUnmapBufferObjectAsync](#) This function is deprecated as of Cuda 3.0.

Global [cudaGLUnregisterBufferObject](#) This function is deprecated as of Cuda 3.0.

Global [cudaErrorPriorLaunchFailure](#) This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Global [cudaErrorAddressOfConstant](#) This error return is deprecated as of CUDA 3.1. Variables in constant memory may now have their address taken by the runtime via [cudaGetSymbolAddress\(\)](#).

Global [cudaErrorTextureFetchFailed](#) This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

---

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

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

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

**Global `cudaErrorCudartUnloading`** This error return is deprecated as of CUDA 3.2.

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

**Global `CUDA_ERROR_CONTEXT_ALREADY_CURRENT`** This error return is deprecated as of CUDA 3.2. It is no longer an error to attempt to push the active context via `cuCtxPushCurrent()`.

**Global `cuParamSetTexRef`**

**Global `cuTexRefCreate`**

**Global `cuTexRefDestroy`**

**Global `cuGLInit`** This function is deprecated as of Cuda 3.0.

**Global `cuGLMapBufferObject`** This function is deprecated as of Cuda 3.0.

**Global `cuGLMapBufferObjectAsync`** This function is deprecated as of Cuda 3.0.

**Global `cuGLRegisterBufferObject`** This function is deprecated as of Cuda 3.0.

**Global `cuGLSetBufferObjectMapFlags`** This function is deprecated as of Cuda 3.0.

**Global `cuGLUnmapBufferObject`** This function is deprecated as of Cuda 3.0.

**Global `cuGLUnmapBufferObjectAsync`** This function is deprecated as of Cuda 3.0.

**Global `cuGLUnregisterBufferObject`** This function is deprecated as of Cuda 3.0.

Global [cuD3D9MapResources](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D9RegisterResource](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D9ResourceGetMappedArray](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D9ResourceGetMappedPitch](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D9ResourceGetMappedPointer](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D9ResourceGetMappedSize](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D9ResourceGetSurfaceDimensions](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D9ResourceSetMapFlags](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D9UnmapResources](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D9UnregisterResource](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D10MapResources](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D10RegisterResource](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D10ResourceGetMappedArray](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D10ResourceGetMappedPitch](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D10ResourceGetMappedPointer](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D10ResourceGetMappedSize](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D10ResourceGetSurfaceDimensions](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D10ResourceSetMapFlags](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D10UnmapResources](#) This function is deprecated as of Cuda 3.0.

Global [cuD3D10UnregisterResource](#) This function is deprecated as of Cuda 3.0.

# Chapter 2

## Module Index

### 2.1 Modules

Here is a list of all modules:

CUDA Runtime API . . . . .	9
Thread Management . . . . .	11
Error Handling . . . . .	15
Device Management . . . . .	17
Stream Management . . . . .	23
Event Management . . . . .	26
Execution Control . . . . .	30
Memory Management . . . . .	34
OpenGL Interoperability . . . . .	66
OpenGL Interoperability [DEPRECATED] . . . . .	130
Direct3D 9 Interoperability . . . . .	70
Direct3D 9 Interoperability [DEPRECATED] . . . . .	114
Direct3D 10 Interoperability . . . . .	75
Direct3D 10 Interoperability [DEPRECATED] . . . . .	122
Direct3D 11 Interoperability . . . . .	80
VDPAU Interoperability . . . . .	84
Graphics Interoperability . . . . .	87
Texture Reference Management . . . . .	91
Surface Reference Management . . . . .	96
Version Management . . . . .	98
C++ API Routines . . . . .	99
Interactions with the CUDA Driver API . . . . .	112
Data types used by CUDA Runtime . . . . .	135
CUDA Driver API . . . . .	147
Data types used by CUDA driver . . . . .	148
Initialization . . . . .	165
Version Management . . . . .	166
Device Management . . . . .	167
Context Management . . . . .	173
Module Management . . . . .	182
Memory Management . . . . .	188
Stream Management . . . . .	230
Event Management . . . . .	233

---

Execution Control . . . . .	237
Execution Control [DEPRECATED] . . . . .	244
Texture Reference Management . . . . .	245
Texture Reference Management [DEPRECATED] . . . . .	253
Surface Reference Management . . . . .	255
Graphics Interoperability . . . . .	257
OpenGL Interoperability . . . . .	262
OpenGL Interoperability [DEPRECATED] . . . . .	265
Direct3D 9 Interoperability . . . . .	271
Direct3D 9 Interoperability [DEPRECATED] . . . . .	277
Direct3D 10 Interoperability . . . . .	286
Direct3D 10 Interoperability [DEPRECATED] . . . . .	292
Direct3D 11 Interoperability . . . . .	301
VDPAU Interoperability . . . . .	307

# Chapter 3

## Data Structure Index

### 3.1 Data Structures

Here are the data structures with brief descriptions:

<a href="#">CUDA_ARRAY3D_DESCRIPTOR_st</a>	311
<a href="#">CUDA_ARRAY_DESCRIPTOR_st</a>	313
<a href="#">CUDA_MEMCPY2D_st</a>	314
<a href="#">CUDA_MEMCPY3D_st</a>	316
<a href="#">cudaChannelFormatDesc</a>	319
<a href="#">cudaDeviceProp</a>	320
<a href="#">cudaExtent</a>	324
<a href="#">cudaFuncAttributes</a>	325
<a href="#">cudaMemcpy3DParms</a>	327
<a href="#">cudaPitchedPtr</a>	329
<a href="#">cudaPos</a>	330
<a href="#">CUdevprop_st</a>	331
<a href="#">surfaceReference</a>	333
<a href="#">textureReference</a>	334



# Chapter 4

## Module Documentation

### 4.1 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](#)
- [Direct3D 11 Interoperability](#)
- [VDPAU Interoperability](#)
- [Graphics Interoperability](#)
- [Texture Reference Management](#)
- [Surface Reference Management](#)
- [Version Management](#)
- [C++ API Routines](#)

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

- [Interactions with the CUDA Driver API](#)

*Interactions between the CUDA Driver API and the CUDA Runtime API.*

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

#### Defines

- `#define CUDART_VERSION 3020`

### 4.1.1 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`.

### 4.1.2 Define Documentation

#### 4.1.2.1 `#define CUDART_VERSION 3020`

CUDA Runtime API Version 3.2

## 4.2 Thread Management

### Functions

- [cudaError\\_t cudaThreadExit](#) (void)  
*Exit and clean up from CUDA launches.*
- [cudaError\\_t cudaThreadGetCacheConfig](#) (enum [cudaFuncCache](#) \*pCacheConfig)  
*Returns the preferred cache configuration for the current host thread.*
- [cudaError\\_t cudaThreadGetLimit](#) (size\_t \*pValue, enum [cudaLimit](#) limit)  
*Returns resource limits.*
- [cudaError\\_t cudaThreadSetCacheConfig](#) (enum [cudaFuncCache](#) cacheConfig)  
*Sets the preferred cache configuration for the current host thread.*
- [cudaError\\_t cudaThreadSetLimit](#) (enum [cudaLimit](#) limit, size\_t value)  
*Set resource limits.*
- [cudaError\\_t cudaThreadSynchronize](#) (void)  
*Wait for compute device to finish.*

### 4.2.1 Detailed Description

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

### 4.2.2 Function Documentation

#### 4.2.2.1 [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](#)

#### 4.2.2.2 `cudaError_t cudaThreadGetCacheConfig (enum cudaFuncCache * pCacheConfig)`

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

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

The supported cache configurations are:

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

##### Parameters:

*pCacheConfig* - Returned cache configuration

##### Returns:

`cudaSuccess`, `cudaErrorInitializationError`

##### Note:

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

##### See also:

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

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

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

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

##### Parameters:

*limit* - Limit to query

*pValue* - Returned size in bytes of limit

##### Returns:

`cudaSuccess`, `cudaErrorUnsupportedLimit`, `cudaErrorInvalidValue`

##### Note:

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

##### See also:

`cudaThreadSetLimit`

#### 4.2.2.4 `cudaError_t cudaThreadSetCacheConfig` (enum `cudaFuncCache cacheConfig`)

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

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

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

The supported cache configurations are:

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

#### Parameters:

`cacheConfig` - Requested cache configuration

#### Returns:

`cudaSuccess`, `cudaErrorInitializationError`

#### Note:

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

#### See also:

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

#### 4.2.2.5 `cudaError_t cudaThreadSetLimit` (enum `cudaLimit limit`, `size_t value`)

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

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

- `cudaLimitStackSize` controls the stack size of each GPU thread. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error `cudaErrorUnsupportedLimit` being returned.
- `cudaLimitPrintfFifoSize` controls the size of the FIFO used by the `printf()` device system call. Setting `cudaLimitPrintfFifoSize` must be performed before launching any kernel that uses the `printf()` device system call, otherwise `cudaErrorInvalidValue` will be returned. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error `cudaErrorUnsupportedLimit` being returned.

- [cudaLimitMallocHeapSize](#) controls the size of the heap used by the `malloc()` and `free()` device system calls. Setting [cudaLimitMallocHeapSize](#) must be performed before launching any kernel that uses the `malloc()` or `free()` device system calls, otherwise [cudaErrorInvalidValue](#) will be returned. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error [cudaErrorUnsupportedLimit](#) being returned.

**Parameters:**

*limit* - Limit to set

*value* - Size in bytes of limit

**Returns:**

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

**Note:**

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

**See also:**

[cudaThreadGetLimit](#)

#### 4.2.2.6 [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. If the [cudaDeviceBlockingSync](#) flag was set for this device, the host thread will block until the device has finished its work.

**Returns:**

[cudaSuccess](#)

**Note:**

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

**See also:**

[cudaThreadExit](#)

## 4.3 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.*
- `cudaError_t cudaPeekAtLastError (void)`  
*Returns the last error from a runtime call.*

### 4.3.1 Detailed Description

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

### 4.3.2 Function Documentation

#### 4.3.2.1 `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](#), [cudaPeekAtLastError](#), [cudaError](#)

#### 4.3.2.2 `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](#), [cudaErrorLaunchTimeout](#), [cudaErrorLaunchOutOfResources](#), [cudaErrorInvalidDeviceFunction](#), [cudaErrorInvalidConfiguration](#), [cudaErrorInvalidDevice](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidPitchValue](#), [cudaErrorInvalidSymbol](#), [cudaErrorUnmapBufferObjectFailed](#), [cudaErrorInvalidHostPointer](#), [cudaErrorInvalidDevicePointer](#), [cudaErrorInvalidTexture](#), [cudaErrorInvalidTextureBinding](#), [cudaErrorInvalidChannelDescriptor](#), [cudaErrorInvalidMemcpyDirection](#), [cudaErrorInvalidFilterSetting](#), [cudaErrorInvalidNormSetting](#), [cudaErrorUnknown](#), [cudaErrorNotYetImplemented](#), [cudaErrorInvalidResourceHandle](#), [cudaErrorInsufficientDriver](#), [cudaErrorSetOnActiveProcess](#), [cudaErrorStartupFailure](#), [cudaErrorApiFailureBase](#)

**Note:**

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

**See also:**

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

**4.3.2.3 cudaError\_t cudaPeekAtLastError (void)**

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

**Returns:**

[cudaSuccess](#), [cudaErrorMissingConfiguration](#), [cudaErrorMemoryAllocation](#), [cudaErrorInitializationError](#), [cudaErrorLaunchFailure](#), [cudaErrorLaunchTimeout](#), [cudaErrorLaunchOutOfResources](#), [cudaErrorInvalidDeviceFunction](#), [cudaErrorInvalidConfiguration](#), [cudaErrorInvalidDevice](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidPitchValue](#), [cudaErrorInvalidSymbol](#), [cudaErrorUnmapBufferObjectFailed](#), [cudaErrorInvalidHostPointer](#), [cudaErrorInvalidDevicePointer](#), [cudaErrorInvalidTexture](#), [cudaErrorInvalidTextureBinding](#), [cudaErrorInvalidChannelDescriptor](#), [cudaErrorInvalidMemcpyDirection](#), [cudaErrorInvalidFilterSetting](#), [cudaErrorInvalidNormSetting](#), [cudaErrorUnknown](#), [cudaErrorNotYetImplemented](#), [cudaErrorInvalidResourceHandle](#), [cudaErrorInsufficientDriver](#), [cudaErrorSetOnActiveProcess](#), [cudaErrorStartupFailure](#), [cudaErrorApiFailureBase](#)

**Note:**

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

**See also:**

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

## 4.4 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](#) (unsigned 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.*

### 4.4.1 Detailed Description

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

### 4.4.2 Function Documentation

#### 4.4.2.1 [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](#)

#### 4.4.2.2 `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](#)

#### 4.4.2.3 `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](#)

#### 4.4.2.4 `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;
    int concurrentKernels;
    int ECCEnabled;
    int pciBusID;
    int pciDeviceID;
    int tccDriver;
}

```

where:

- [name\[256\]](#) is an ASCII string identifying the device;
- [totalGlobalMem](#) is the total amount of global memory available on the device in bytes;
- [sharedMemPerBlock](#) is the maximum amount of shared memory available to a thread block in bytes; 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.
- `concurrentKernels` is 1 if the device supports executing multiple kernels within the same context simultaneously, or 0 if not. It is not guaranteed that multiple kernels will be resident on the device concurrently so this feature should not be relied upon for correctness;
- `ECCEnabled` is 1 if the device has ECC support turned on, or 0 if not.
- `pciBusID` is the PCI bus identifier of the device.
- `pciDeviceID` is the PCI device (sometimes called slot) identifier of the device.
- `tccDriver` is 1 if the device is using a TCC driver or 0 if not.

**Parameters:**

*prop* - Properties for the specified device

*device* - Device number to get properties for

**Returns:**

`cudaSuccess`, `cudaErrorInvalidDevice`

**See also:**

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

#### 4.4.2.5 `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 or if there exists a CUDA driver context active on the host thread, then 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`

#### 4.4.2.6 `cudaError_t cudaSetDeviceFlags (unsigned 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 or if there exists a CUDA driver context active on the host thread, then this call returns `cudaErrorSetOnActiveProcess`.

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.
- `cudaDeviceLmemResizeToMax`: Instruct CUDA to not reduce local memory after resizing local memory for a kernel. This can prevent thrashing by local memory allocations when launching many kernels with high local memory usage at the cost of potentially increased memory usage.

#### Parameters:

*flags* - Parameters for device operation

#### Returns:

`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorSetOnActiveProcess`

#### See also:

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

#### 4.4.2.7 `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` exceeds 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](#)

## 4.5 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.*
- [cudaError\\_t cudaStreamWaitEvent \(cudaStream\\_t stream, cudaEvent\\_t event, unsigned int flags\)](#)  
*Make a compute stream wait on an event.*

### 4.5.1 Detailed Description

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

### 4.5.2 Function Documentation

#### 4.5.2.1 [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](#), [cudaStreamWaitEvent](#), [cudaStreamDestroy](#)

#### 4.5.2.2 [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](#), [cudaStreamWaitEvent](#), [cudaStreamSynchronize](#)

#### 4.5.2.3 `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](#), [cudaStreamWaitEvent](#), [cudaStreamSynchronize](#), [cudaStreamDestroy](#)

#### 4.5.2.4 `cudaError_t cudaStreamSynchronize (cudaStream_t stream)`

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

**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](#), [cudaStreamWaitEvent](#), [cudaStreamDestroy](#)

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

Makes all future work submitted to `stream` wait until `event` reports completion before beginning execution. This synchronization will be performed efficiently on the device.

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

If `stream` is NULL, any future work submitted in any stream will wait for `event` to complete before beginning execution. This effectively creates a barrier for all future work submitted to the device on this thread.

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

##### Parameters:

*stream* - Stream to wait

*event* - Event to wait on

*flags* - Parameters for the operation (must be 0)

##### Returns:

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

##### Note:

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

##### See also:

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

## 4.6 Event Management

### Functions

- [cudaError\\_t cudaEventCreate \(cudaEvent\\_t \\*event\)](#)  
*Creates an event object.*
- [cudaError\\_t cudaEventCreateWithFlags \(cudaEvent\\_t \\*event, unsigned int flags\)](#)  
*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 \(cudaEvent\\_t event\)](#)  
*Queries an event's status.*
- [cudaError\\_t cudaEventRecord \(cudaEvent\\_t event, cudaStream\\_t stream=0\)](#)  
*Records an event.*
- [cudaError\\_t cudaEventSynchronize \(cudaEvent\\_t event\)](#)  
*Waits for an event to complete.*

### 4.6.1 Detailed Description

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

### 4.6.2 Function Documentation

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

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

#### Parameters:

*event* - Newly created event

#### Returns:

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

#### Note:

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

#### See also:

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

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

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

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

##### Parameters:

*event* - Newly created event

*flags* - Flags for new event

##### Returns:

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

##### Note:

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

##### See also:

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

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

Destroys the event specified by `event`.

##### Parameters:

*event* - Event to destroy

##### Returns:

`cudaSuccess`, `cudaErrorInitializationError`, `cudaErrorInvalidValue`, `cudaErrorLaunchFailure`

##### Note:

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

##### See also:

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

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

If `cudaEventRecord()` has not been called on either event, then `cudaErrorInvalidResourceHandle` is returned. If `cudaEventRecord()` has been called on both events but one or both of them has not yet been completed (that is, `cudaEventQuery()` would return `cudaErrorNotReady` on at least one of the events), `cudaErrorNotReady` is returned. If either event was created with the `cudaEventDisableTiming` flag, then this function will return `cudaErrorInvalidResourceHandle`.

##### Parameters:

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

*start* - Starting event

*end* - Ending event

##### Returns:

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

##### Note:

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

##### See also:

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

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

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

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

##### Parameters:

*event* - Event to query

##### Returns:

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

##### Note:

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

##### See also:

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

#### 4.6.2.6 `cudaError_t cudaEventRecord(cudaEvent_t event, cudaStream_t stream = 0)`

Records an event. If `stream` is non-zero, the event is recorded after all preceding operations in `stream` have been completed; otherwise, it is recorded after all preceding operations in the CUDA context have been completed. Since 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 on `event`, then this call will overwrite any existing state in `event`. Any subsequent calls which examine the status of `event` will only examine the completion of this most recent call to [cudaEventRecord\(\)](#).

**Parameters:**

- event* - Event to record
- stream* - Stream in which to record event

**Returns:**

[cudaSuccess](#), [cudaErrorInvalidValue](#), [cudaErrorInitializationError](#), [cudaErrorInvalidResourceHandle](#), [cudaError-LaunchFailure](#)

**Note:**

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

**See also:**

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

#### 4.6.2.7 `cudaError_t cudaEventSynchronize(cudaEvent_t event)`

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

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

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

**Parameters:**

- event* - Event to wait for

**Returns:**

[cudaSuccess](#), [cudaErrorInitializationError](#), [cudaErrorInvalidValue](#), [cudaErrorInvalidResourceHandle](#), [cudaError-LaunchFailure](#)

**Note:**

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

**See also:**

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

## 4.7 Execution Control

### Functions

- [cudaError\\_t cudaConfigureCall](#) (dim3 gridDim, dim3 blockDim, size\_t sharedMem=0, [cudaStream\\_t](#) stream=0)  
*Configure a device-launch.*
- [cudaError\\_t cudaFuncGetAttributes](#) (struct [cudaFuncAttributes](#) \*attr, const char \*func)  
*Find out attributes for a given function.*
- [cudaError\\_t cudaFuncSetCacheConfig](#) (const char \*func, enum [cudaFuncCache](#) cacheConfig)  
*Sets the preferred cache configuration for a device 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.*

### 4.7.1 Detailed Description

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

### 4.7.2 Function Documentation

#### 4.7.2.1 [cudaError\\_t cudaConfigureCall](#) (dim3 gridDim, dim3 blockDim, size\_t sharedMem = 0, [cudaStream\\_t](#) stream = 0)

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

#### 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:**

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

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

This function obtains the attributes of a function specified via `func`, which is a character string that specifies the fully-decorated (C++) name for a function that executes on the device. The parameter specified by `func` must be declared as a `__global__` function. The fetched attributes are placed in `attr`. If the specified function does not exist, then [cudaErrorInvalidDeviceFunction](#) is returned.

Note that some function attributes such as [maxThreadsPerBlock](#) may vary based on the device that is currently being used.

**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:**

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

**4.7.2.3 `cudaError_t cudaFuncSetCacheConfig (const char * func, enum cudaFuncCache cacheConfig)`**

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

`func` is a character string that specifies the fully-decorated (C++) name for a function that executes on the device. The parameter specified by `func` must be declared as a `__global__` function. If the specified function does not exist, then [cudaErrorInvalidDeviceFunction](#) is returned.

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

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

The supported cache configurations are:

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

- [cudaFuncCachePreferL1](#): prefer larger L1 cache and smaller shared memory

**Parameters:**

*func* - Char string naming device function

*cacheConfig* - Requested cache configuration

**Returns:**

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

**Note:**

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

**See also:**

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

**4.7.2.4 `cudaError_t cudaLaunch (const char * entry)`**

Launches the function `entry` on the device. The parameter `entry` must be a character string naming a function that executes on the device. The parameter specified by `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 char string naming device function to execute

**Returns:**

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

**Note:**

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

**See also:**

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

**4.7.2.5 `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:**

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

**4.7.2.6 `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:**

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

**4.7.2.7 `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:**

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

## 4.8 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 \*\*pHost, 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 cudaHostGetFlags](#) (unsigned int \*pFlags, void \*pHost)  
*Passes back flags used to allocate pinned 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 \*\*array, const struct [cudaChannelFormatDesc](#) \*desc, struct [cudaExtent](#) extent, unsigned int flags=0)  
*Allocate an array on the device.*
- [cudaError\\_t cudaMallocArray](#) (struct cudaArray \*\*array, const struct [cudaChannelFormatDesc](#) \*desc, size\_t width, size\_t height=0, unsigned int flags=0)  
*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 cudaMemcpy2DFromArray` (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=`cudaMemcpyDeviceToDevice`)  
*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=0)  
*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=0)  
*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=0)  
*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=0)  
*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=`cudaMemcpyDeviceToDevice`)  
*Copies data between host and device.*
- `cudaError_t cudaMemcpyAsync` (void \*dst, const void \*src, size\_t count, enum `cudaMemcpyKind` kind, `cudaStream_t` stream=0)  
*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=0)

*Copies data between host and device.*

- [cudaError\\_t cudaMemcpyFromSymbol](#) (void \*dst, const char \*symbol, size\_t count, size\_t offset=0, enum [cudaMemcpyKind](#) kind=cudaMemcpyDeviceToHost)

*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=0)

*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=0)

*Copies data between host and device.*

- [cudaError\\_t cudaMemcpyToSymbol](#) (const char \*symbol, const void \*src, size\_t count, size\_t offset=0, enum [cudaMemcpyKind](#) kind=cudaMemcpyHostToDevice)

*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=0)

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

- [cudaError\\_t cudaMemGetInfo](#) (size\_t \*free, size\_t \*total)

*Gets free and total device memory.*

- [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 cudaMemset2DAsync](#) (void \*devPtr, size\_t pitch, int value, size\_t width, size\_t height, [cudaStream\\_t](#) stream=0)

*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.*

- [cudaError\\_t cudaMemset3DAsync](#) (struct [cudaPitchedPtr](#) pitchedDevPtr, int value, struct [cudaExtent](#) extent, [cudaStream\\_t](#) stream=0)

*Initializes or sets device memory to a value.*

- [cudaError\\_t cudaMemsetAsync](#) (void \*devPtr, int value, size\_t count, [cudaStream\\_t](#) stream=0)

*Initializes or sets device memory to a value.*

- struct [cudaExtent](#) [make\\_cudaExtent](#) (size\_t w, size\_t h, size\_t d)

*Returns a [cudaExtent](#) based on input parameters.*

- struct `cudaPitchedPtr` `make_cudaPitchedPtr` (void \*d, size\_t p, size\_t xsz, size\_t ysz)  
*Returns a `cudaPitchedPtr` based on input parameters.*
- struct `cudaPos` `make_cudaPos` (size\_t x, size\_t y, size\_t z)  
*Returns a `cudaPos` based on input parameters.*

### 4.8.1 Detailed Description

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

### 4.8.2 Function Documentation

#### 4.8.2.1 `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` (C API), `cudaFreeHost`, `cudaMalloc3D`, `cudaMalloc3DArray`, `cudaHostAlloc`

#### 4.8.2.2 `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 \(C API\)](#), [cudaFreeHost](#), [cudaHostAlloc](#)

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

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

**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 \(C API\)](#), [cudaMalloc3D](#), [cudaMalloc3DArray](#), [cudaHostAlloc](#)

**4.8.2.4 `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 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 the global or constant memory space, `*devPtr` is unchanged and the error [cudaErrorInvalidSymbol](#) is returned. If there are multiple global or constant variables with the same string name (from separate files) and the lookup is done via character string, [cudaErrorDuplicateVariableName](#) is returned.

**Parameters:**

*devPtr* - Return device pointer associated with symbol

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

**Returns:**

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

**Note:**

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

**See also:**

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

#### 4.8.2.5 `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 \(C++ API\)](#)

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

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

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

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

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

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

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

Memory allocated by this function must be freed with [cudaFreeHost\(\)](#).

**Parameters:**

- pHost* - 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 \(C API\)](#), [cudaFreeHost](#)

#### 4.8.2.7 `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](#)

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

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

##### Parameters:

*pFlags* - Returned flags word

*pHost* - Host pointer

##### Returns:

`cudaSuccess`, `cudaErrorInvalidValue`

##### Note:

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

##### See also:

`cudaHostAlloc`

#### 4.8.2.9 `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` (C API), `cudaFreeHost`, `cudaHostAlloc`

#### 4.8.2.10 `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 (*width* field in bytes)

**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](#) (C API), [cudaFreeHost](#), [cudaHostAlloc](#), [make\\_cudaPitchedPtr](#), [make\\_cudaExtent](#)

#### 4.8.2.11 `cudaError_t cudaMalloc3DArray(struct cudaArray **array, const struct cudaChannelFormatDesc *desc, struct cudaExtent extent, unsigned int flags = 0)`

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

The [cudaChannelFormatDesc](#) is defined as:

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

where [cudaChannelFormatKind](#) is one of [cudaChannelFormatKindSigned](#), [cudaChannelFormatKindUnsigned](#), or [cudaChannelFormatKindFloat](#).

[cudaMalloc3DArray\(\)](#) 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.

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

**Parameters:**

*array* - Pointer to allocated array in device memory

*desc* - Requested channel format  
*extent* - Requested allocation size (width field in elements)  
*flags* - Flags for extensions (must be 0 for now)

**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 \(C API\)](#), [cudaFreeHost](#), [cudaHostAlloc](#), [make\\_cudaExtent](#)

#### 4.8.2.12 `cudaError_t cudaMallocArray (struct cudaArray ** array, const struct cudaChannelFormatDesc * desc, size_t width, size_t height = 0, unsigned int flags = 0)`

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

The [cudaChannelFormatDesc](#) is defined as:

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

where [cudaChannelFormatKind](#) is one of [cudaChannelFormatKindSigned](#), [cudaChannelFormatKindUnsigned](#), or [cudaChannelFormatKindFloat](#).

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

- [cudaArrayDefault](#): This flag's value is defined to be 0 and provides default array allocation
- [cudaArraySurfaceLoadStore](#): Allocates an array that can be read from or written to using a surface reference

**Parameters:**

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

**Returns:**

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

**Note:**

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

**See also:**

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

#### 4.8.2.13 `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`, `cudaMallocHost (C++ API)`, `cudaFreeHost`, `cudaHostAlloc`

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

Allocates at least `width` (in bytes) \* `height` bytes of linear memory on the device and returns in `*devPtr` a pointer to the allocated memory. The function may pad the allocation to ensure that corresponding pointers in any given row will continue to meet the alignment requirements for coalescing as the address is updated from row to row. The pitch returned in `*pitch` by `cudaMallocPitch()` is the width in bytes of the allocation. The intended usage of `pitch` is as a separate parameter of the allocation, used to compute addresses within the 2D array. Given the row and column of an array element of type `T`, the address is computed as:

```
T* pElement = (T*)((char*)BaseAddress + Row * pitch) + Column;
```

For allocations of 2D arrays, it is recommended that programmers consider performing pitch allocations using `cudaMallocPitch()`. Due to pitch alignment restrictions in the hardware, this is especially true if the application will be performing 2D memory copies between different regions of device memory (whether linear memory or CUDA arrays).

##### Parameters:

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

##### Returns:

`cudaSuccess`, `cudaErrorMemoryAllocation`

**Note:**

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

**See also:**

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

**4.8.2.15 `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](#)

**4.8.2.16 `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. `width` must not exceed either `dpitch` or `spitch`. Calling [cudaMemcpy2D\(\)](#) with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior. [cudaMemcpy2D\(\)](#) returns an error if `dpitch` or `spitch` exceeds the maximum allowed.

**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](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpyFromArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

#### 4.8.2.17 `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 = cudaMemcpyDeviceToDevice)`

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. `wOffsetDst + width` must not exceed the width of the CUDA array `dst`. `wOffsetSrc + width` must not exceed the width of the CUDA array `src`.

**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](#)

**4.8.2.18 `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 = 0)`**

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. `width` must not exceed either `dpitch` or `spitch`. Calling [cudaMemcpy2DAsync\(\)](#) with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior. [cudaMemcpy2DAsync\(\)](#) returns an error if `dpitch` or `spitch` is greater than the maximum allowed.

[cudaMemcpy2DAsync\(\)](#) is asynchronous with respect to the host, so the call may return before the copy is complete. 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.

**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](#), [cudaMemcpyArrayToArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpyFromArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

#### 4.8.2.19 `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. `wOffset + width` must not exceed the width of the CUDA array `src`. `width` must not exceed `dpitch`. `cudaMemcpy2DFromArray()` returns an error if `dpitch` exceeds the maximum allowed.

##### 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`](#)

#### 4.8.2.20 `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 = 0`)

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. `wOffset + width` must not exceed the width of the CUDA array `src`. `width` must not exceed `dpitch`. `cudaMemcpy2DFromArrayAsync()` returns an error if `dpitch` exceeds the maximum allowed.

`cudaMemcpy2DFromArrayAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. 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.

**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](#)

**4.8.2.21 `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. `wOffset + width` must not exceed the width of the CUDA array `dst`. `width` must not exceed `spitch`. [cudaMemcpy2DToArray\(\)](#) returns an error if `spitch` exceeds the maximum allowed.

**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](#)

#### 4.8.2.22 `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 = 0)`

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. `wOffset + width` must not exceed the width of the CUDA array `dst`. `width` must not exceed `spitch`. [cudaMemcpy2DToArrayAsync\(\)](#) returns an error if `spitch` exceeds the maximum allowed.

[cudaMemcpy2DToArrayAsync\(\)](#) is asynchronous with respect to the host, so the call may return before the copy is complete. 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.

**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](#), [cudaMemcpyArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpyFromArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

**4.8.2.23 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.

The source object must lie entirely within the region defined by `srcPos` and `extent`. The destination object must lie entirely within the region defined by `dstPos` and `extent`.

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

#### 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`

#### 4.8.2.24 `cudaError_t cudaMemcpy3DAsync (const struct cudaMemcpy3DParms *p, cudaStream_t stream = 0)`

```
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.

The source object must lie entirely within the region defined by `srcPos` and `extent`. The destination object must lie entirely within the region defined by `dstPos` and `extent`.

`cudaMemcpy3DAsync()` returns an error if the pitch of `srcPtr` or `dstPtr` exceeds the maximum allowed. The pitch of a `cudaPitchedPtr` allocated with `cudaMalloc3D()` will always be valid.

`cudaMemcpy3DAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. 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.

#### 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`, `cudaMemcpyFromArrayToArray`, `cudaMemcpy2DFromArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`, `make_cudaExtent`, `make_cudaPos`

#### 4.8.2.25 `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 = cudaMemcpyDeviceToDevice)`

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](#)

#### 4.8.2.26 `cudaError_t cudaMemcpyAsync (void * dst, const void * src, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream = 0)`

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.

##### 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](#)

#### 4.8.2.27 `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](#)

#### 4.8.2.28 `cudaError_t cudaMemcpyFromArrayAsync` (`void * dst`, `const struct cudaArray * src`, `size_t wOffset`, `size_t hOffset`, `size_t count`, `enum cudaMemcpyKind kind`, `cudaStream_t stream = 0`)

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.

##### 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](#)

#### 4.8.2.29 `cudaError_t cudaMemcpyFromSymbol` (`void * dst`, `const char * symbol`, `size_t count`, `size_t offset = 0`, `enum cudaMemcpyKind kind = cudaMemcpyDeviceToHost`)

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](#)

**4.8.2.30 `cudaError_t cudaMemcpyFromSymbolAsync(void *dst, const char *symbol, size_t count, size_t offset, enum cudaMemcpyKind kind, cudaStream_t stream = 0)`**

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.

**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](#)

#### 4.8.2.31 `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`](#)

#### 4.8.2.32 `cudaError_t cudaMemcpyToArrayAsync` (`struct cudaArray * dst`, `size_t wOffset`, `size_t hOffset`, `const void * src`, `size_t count`, `enum cudaMemcpyKind kind`, `cudaStream_t stream = 0`)

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.

##### 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](#)

#### 4.8.2.33 `cudaError_t cudaMemcpyToSymbol(const char * symbol, const void * src, size_t count, size_t offset = 0, enum cudaMemcpyKind kind = cudaMemcpyHostToDevice)`

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](#), [cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpyFromArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

#### 4.8.2.34 `cudaError_t cudaMemcpyToSymbolAsync (const char * symbol, const void * src, size_t count, size_t offset, enum cudaMemcpyKind kind, cudaStream_t stream = 0)`

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.

##### 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`

#### 4.8.2.35 `cudaError_t cudaMemGetInfo (size_t * free, size_t * total)`

Returns in `*free` and `*total` respectively, the free and total amount of memory available for allocation by the device in bytes.

##### Parameters:

*free* - Returned free memory in bytes  
*total* - Returned total memory in bytes

##### Returns:

`cudaSuccess`, `cudaErrorInitializationError`, `cudaErrorInvalidValue`, `cudaErrorLaunchFailure`

##### Note:

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

**4.8.2.36** `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](#), [cudaMemsetAsync](#), [cudaMemset2DAsync](#), [cudaMemset3DAsync](#)

**4.8.2.37** `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](#), [cudaMemsetAsync](#), [cudaMemset2DAsync](#), [cudaMemset3DAsync](#)

#### 4.8.2.38 `cudaError_t cudaMemset2DAsync (void * devPtr, size_t pitch, int value, size_t width, size_t height, cudaStream_t stream = 0)`

Sets to the specified value *value* a matrix (*height* rows of *width* bytes each) pointed to by *dstPtr*. *pitch* is the width in bytes of the 2D array pointed to by *dstPtr*, including any padding added to the end of each row. This function performs fastest when the pitch is one that has been passed back by [cudaMallocPitch\(\)](#).

[cudaMemset2DAsync\(\)](#) is asynchronous with respect to the host, so the call may return before the memset is complete. The operation can optionally be associated to a stream by passing a non-zero *stream* argument. If *stream* is non-zero, the operation may overlap with operations in other streams.

##### Parameters:

- devPtr* - Pointer to 2D device memory
- pitch* - Pitch in bytes of 2D device memory
- value* - Value to set for each byte of specified memory
- width* - Width of matrix set (columns in bytes)
- height* - Height of matrix set (rows)
- stream* - Stream identifier

##### Returns:

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

##### Note:

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

##### See also:

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

#### 4.8.2.39 `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 (*width* field in bytes)

**Returns:**

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

**Note:**

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

**See also:**

[cudaMemset](#), [cudaMemset2D](#), [cudaMemsetAsync](#), [cudaMemset2DAsync](#), [cudaMemset3DAsync](#), [cudaMalloc3D](#), [make\\_cudaPitchedPtr](#), [make\\_cudaExtent](#)

**4.8.2.40 `cudaError_t cudaMemset3DAsync` (`struct cudaPitchedPtr pitchedDevPtr`, `int value`, `struct cudaExtent extent`, `cudaStream_t stream = 0`)**

Initializes each element of a 3D array to the specified value `value`. The object to initialize is defined by `pitchedDevPtr`. The `pitch` field of `pitchedDevPtr` is the width in memory in bytes of the 3D array pointed to by `pitchedDevPtr`, including any padding added to the end of each row. The `xsize` field specifies the logical width of each row in bytes, while the `ysize` field specifies the height of each 2D slice in rows.

The extents of the initialized region are specified as a `width` in bytes, a `height` in rows, and a `depth` in slices.

Extents with `width` greater than or equal to the `xsize` of `pitchedDevPtr` may perform significantly faster than extents narrower than the `xsize`. Secondly, extents with `height` equal to the `ysize` of `pitchedDevPtr` will perform faster than when the `height` is shorter than the `ysize`.

This function performs fastest when the `pitchedDevPtr` has been allocated by [cudaMalloc3D\(\)](#).

[cudaMemset3DAsync\(\)](#) is asynchronous with respect to the host, so the call may return before the memset is complete. The operation can optionally be associated to a stream by passing a non-zero `stream` argument. If `stream` is non-zero, the operation may overlap with operations in other streams.

**Parameters:**

*pitchedDevPtr* - Pointer to pitched device memory

*value* - Value to set for each byte of specified memory

*extent* - Size parameters for where to set device memory (`width` field in bytes)

*stream* - Stream identifier

**Returns:**

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

**Note:**

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

**See also:**

[cudaMemset](#), [cudaMemset2D](#), [cudaMemset3D](#), [cudaMemsetAsync](#), [cudaMemset2DAsync](#), [cudaMalloc3D](#), [make\\_cudaPitchedPtr](#), [make\\_cudaExtent](#)

#### 4.8.2.41 `cudaError_t cudaMemsetAsync (void * devPtr, int value, size_t count, cudaStream_t stream = 0)`

Fills the first `count` bytes of the memory area pointed to by `devPtr` with the constant byte value `value`.

`cudaMemsetAsync()` is asynchronous with respect to the host, so the call may return before the memset is complete. The operation can optionally be associated to a stream by passing a non-zero `stream` argument. If `stream` is non-zero, the operation may overlap with operations in other streams.

##### Parameters:

- devPtr* - Pointer to device memory
- value* - Value to set for each byte of specified memory
- count* - Size in bytes to set
- stream* - Stream identifier

##### Returns:

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

##### Note:

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

##### See also:

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

#### 4.8.2.42 `struct cudaExtent make_cudaExtent (size_t w, size_t h, size_t d) [read]`

Returns a [cudaExtent](#) based on the specified input parameters `w`, `h`, and `d`.

##### Parameters:

- w* - Width in bytes
- h* - Height in elements
- d* - Depth in elements

##### Returns:

[cudaExtent](#) specified by `w`, `h`, and `d`

##### See also:

[make\\_cudaPitchedPtr](#), [make\\_cudaPos](#)

#### 4.8.2.43 `struct cudaPitchedPtr make_cudaPitchedPtr (void * d, size_t p, size_t xsz, size_t ysz) [read]`

Returns a [cudaPitchedPtr](#) based on the specified input parameters `d`, `p`, `xsz`, and `ysz`.

##### Parameters:

- d* - Pointer to allocated memory
- p* - Pitch of allocated memory in bytes

*xsz* - Logical width of allocation in elements

*ysz* - Logical height of allocation in elements

**Returns:**

[cudaPitchedPtr](#) specified by *d*, *p*, *xsz*, and *ysz*

**See also:**

[make\\_cudaExtent](#), [make\\_cudaPos](#)

**4.8.2.44 struct cudaPos make\_cudaPos (size\_t *x*, size\_t *y*, size\_t *z*)** [read]

Returns a [cudaPos](#) based on the specified input parameters *x*, *y*, and *z*.

**Parameters:**

*x* - X position

*y* - Y position

*z* - Z position

**Returns:**

[cudaPos](#) specified by *x*, *y*, and *z*

**See also:**

[make\\_cudaExtent](#), [make\\_cudaPitchedPtr](#)

## 4.9 OpenGL Interoperability

### Modules

- [OpenGL Interoperability \[DEPRECATED\]](#)

### Enumerations

- enum [cudaGLMapFlags](#) {  
[cudaGLMapFlagsNone](#) = 0,  
[cudaGLMapFlagsReadOnly](#) = 1,  
[cudaGLMapFlagsWriteDiscard](#) = 2 }

### Functions

- [cudaError\\_t cudaGLSetGLDevice](#) (int device)  
*Sets the CUDA device for use with OpenGL interoperability.*
- [cudaError\\_t cudaGraphicsGLRegisterBuffer](#) (struct [cudaGraphicsResource](#) \*\*resource, GLuint buffer, unsigned int flags)  
*Registers an OpenGL buffer object.*
- [cudaError\\_t cudaGraphicsGLRegisterImage](#) (struct [cudaGraphicsResource](#) \*\*resource, GLuint image, GLenum target, unsigned int flags)  
*Register an OpenGL texture or renderbuffer object.*
- [cudaError\\_t cudaWGLGetDevice](#) (int \*device, HGPUNV hGpu)  
*Gets the CUDA device associated with hGpu.*

#### 4.9.1 Detailed Description

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

#### 4.9.2 Enumeration Type Documentation

##### 4.9.2.1 enum [cudaGLMapFlags](#)

CUDA GL Map Flags

##### Enumerator:

- [cudaGLMapFlagsNone](#)* Default; Assume resource can be read/written
- [cudaGLMapFlagsReadOnly](#)* CUDA kernels will not write to this resource
- [cudaGLMapFlagsWriteDiscard](#)* CUDA kernels will only write to and will not read from this resource

### 4.9.3 Function Documentation

#### 4.9.3.1 `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 or if there exists a CUDA driver context active on the host thread, then this call returns [cudaErrorSetActiveProcess](#).

**Parameters:**

*device* - Device to use for OpenGL interoperability

**Returns:**

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

**Note:**

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

**See also:**

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

#### 4.9.3.2 `cudaError_t cudaGraphicsGLRegisterBuffer (struct cudaGraphicsResource ** resource, GLuint buffer, unsigned int flags)`

Registers the buffer object specified by `buffer` for access by CUDA. A handle to the registered object is returned as `resource`. The map flags `flags` specify the intended usage, as follows:

- [cudaGraphicsMapFlagsNone](#): Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- [cudaGraphicsMapFlagsReadOnly](#): Specifies that CUDA will not write to this resource.
- [cudaGraphicsMapFlagsWriteDiscard](#): Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

**Parameters:**

*resource* - Pointer to the returned object handle

*buffer* - name of buffer object to be registered

*flags* - Map flags

**Returns:**

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

**Note:**

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

**See also:**

[cudaGLCtxCreate](#), [cudaGraphicsUnregisterResource](#), [cudaGraphicsMapResources](#), [cudaGraphicsResourceGetMappedPointer](#)

#### 4.9.3.3 `cudaError_t cudaGraphicsGLRegisterImage (struct cudaGraphicsResource ** resource, GLuint image, GLenum target, unsigned int flags)`

Registers the texture or renderbuffer object specified by `image` for access by CUDA. `target` must match the type of the object. A handle to the registered object is returned as `resource`. The map flags `flags` specify the intended usage, as follows:

- `cudaGraphicsMapFlagsNone`: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- `cudaGraphicsMapFlagsReadOnly`: Specifies that CUDA will not write to this resource.
- `cudaGraphicsMapFlagsWriteDiscard`: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

The following image classes are currently disallowed:

- Textures with borders
- Multisampled renderbuffers

#### Parameters:

*resource* - Pointer to the returned object handle

*image* - name of texture or renderbuffer object to be registered

*target* - Identifies the type of object specified by `image`, and must be one of `GL_TEXTURE_2D`, `GL_TEXTURE_RECTANGLE`, `GL_TEXTURE_CUBE_MAP`, `GL_TEXTURE_3D`, `GL_TEXTURE_2D_ARRAY`, or `GL_RENDERBUFFER`.

*flags* - Map flags

#### Returns:

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

#### Note:

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

#### See also:

`cudaGLSetGLDevice`, `cudaGraphicsUnregisterResource`, `cudaGraphicsMapResources`, `cudaGraphicsSubResourceGetMappedArray`

#### 4.9.3.4 `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](#)

## 4.10 Direct3D 9 Interoperability

### Modules

- [Direct3D 9 Interoperability \[DEPRECATED\]](#)

### Enumerations

- enum `cudaD3D9DeviceList` {  
`cudaD3D9DeviceListAll` = 1,  
`cudaD3D9DeviceListCurrentFrame` = 2,  
`cudaD3D9DeviceListNextFrame` = 3 }
- enum `cudaD3D9MapFlags` {  
`cudaD3D9MapFlagsNone` = 0,  
`cudaD3D9MapFlagsReadOnly` = 1,  
`cudaD3D9MapFlagsWriteDiscard` = 2 }
- enum `cudaD3D9RegisterFlags` {  
`cudaD3D9RegisterFlagsNone` = 0,  
`cudaD3D9RegisterFlagsArray` = 1 }

### Functions

- `cudaError_t cudaD3D9GetDevice` (int \*device, const char \*pszAdapterName)  
*Gets the device number for an adapter.*
- `cudaError_t cudaD3D9GetDevices` (unsigned int \*pCudaDeviceCount, int \*pCudaDevices, unsigned int cudaDeviceCount, IDirect3DDevice9 \*pD3D9Device, enum `cudaD3D9DeviceList` deviceList)  
*Gets the CUDA devices corresponding to a Direct3D 9 device.*
- `cudaError_t cudaD3D9GetDirect3DDevice` (IDirect3DDevice9 \*\*ppD3D9Device)  
*Gets the Direct3D device against which the current CUDA context was created.*
- `cudaError_t cudaD3D9SetDirect3DDevice` (IDirect3DDevice9 \*pD3D9Device, int device=-1)  
*Sets the Direct3D device to use for interoperability in this thread.*
- `cudaError_t cudaGraphicsD3D9RegisterResource` (struct `cudaGraphicsResource` \*\*resource, IDirect3DResource9 \*pD3DResource, unsigned int flags)  
*Register a Direct3D 9 resource for access by CUDA.*

#### 4.10.1 Detailed Description

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

## 4.10.2 Enumeration Type Documentation

### 4.10.2.1 enum cudaD3D9DeviceList

CUDA devices corresponding to a D3D9 device

**Enumerator:**

*cudaD3D9DeviceListAll* The CUDA devices for all GPUs used by a D3D9 device

*cudaD3D9DeviceListCurrentFrame* The CUDA devices for the GPUs used by a D3D9 device in its currently rendering frame

*cudaD3D9DeviceListNextFrame* The CUDA devices for the GPUs to be used by a D3D9 device in the next frame

### 4.10.2.2 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

### 4.10.2.3 enum cudaD3D9RegisterFlags

CUDA D3D9 Register Flags

**Enumerator:**

*cudaD3D9RegisterFlagsNone* Default; Resource can be accessed through a void\*

*cudaD3D9RegisterFlagsArray* Resource can be accessed through a CUarray\*

## 4.10.3 Function Documentation

### 4.10.3.1 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](#), [cudaGraphicsD3D9RegisterResource](#),

#### 4.10.3.2 `cudaError_t cudaD3D9GetDevices (unsigned int * pCudaDeviceCount, int * pCudaDevices, unsigned int cudaDeviceCount, IDirect3DDevice9 * pD3D9Device, enum cudaD3D9DeviceList deviceList)`

Returns in `*pCudaDeviceCount` the number of CUDA-compatible devices corresponding to the Direct3D 9 device `pD3D9Device`. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of the the CUDA-compatible devices corresponding to the Direct3D 9 device `pD3D9Device`.

If any of the GPUs being used to render `pDevice` are not CUDA capable then the call will return [cudaErrorNoDevice](#).

**Parameters:**

*pCudaDeviceCount* - Returned number of CUDA devices corresponding to `pD3D9Device`

*pCudaDevices* - Returned CUDA devices corresponding to `pD3D9Device`

*cudaDeviceCount* - The size of the output device array `pCudaDevices`

*pD3D9Device* - Direct3D 9 device to query for CUDA devices

*deviceList* - The set of devices to return. This set may be [cudaD3D9DeviceListAll](#) for all devices, [cudaD3D9DeviceListCurrentFrame](#) for the devices used to render the current frame (in SLI), or [cudaD3D9DeviceListNextFrame](#) for the devices used to render the next frame (in SLI).

**Returns:**

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

**Note:**

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

See also:

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

#### 4.10.3.3 `cudaError_t cudaD3D9GetDirect3DDevice (IDirect3DDevice9 ** ppD3D9Device)`

Returns in `*ppD3D9Device` the Direct3D device against which this CUDA context was created in [cudaD3D9SetDirect3DDevice\(\)](#).

**Parameters:**

*ppD3D9Device* - 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:

[cudaD3D9SetDirect3DDevice](#)

#### 4.10.3.4 `cudaError_t cudaD3D9SetDirect3DDevice (IDirect3DDevice9 * pD3D9Device, int device = -1)`

Records `pD3D9Device` as the Direct3D device to use for Direct3D interoperability on this host thread. If the host thread has already initialized the CUDA runtime by calling non-device management runtime functions or if there exists a CUDA driver context active on the host thread, then this call returns `cudaErrorSetActiveProcess`.

Successful context creation on `pD3D9Device` will increase the internal reference count on `pD3D9Device`. This reference count will be decremented upon destruction of this context through `cudaThreadExit()`.

##### Parameters:

*pD3D9Device* - Direct3D device to use for this thread

*device* - The CUDA device to use. This device must be among the devices returned when querying `cudaD3D9DeviceListAll` from `cudaD3D9GetDevices`, may be set to -1 to automatically select an appropriate CUDA device.

##### Returns:

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

##### Note:

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

##### See also:

`cudaD3D9GetDevice`, `cudaGraphicsD3D9RegisterResource`,

#### 4.10.3.5 `cudaError_t cudaGraphicsD3D9RegisterResource (struct cudaGraphicsResource ** resource, IDirect3DResource9 * pD3DResource, unsigned int flags)`

Registers the Direct3D 9 resource `pD3DResource` for access by CUDA.

If this call is successful then the application will be able to map and unmap this resource until it is unregistered through `cudaGraphicsUnregisterResource()`. Also on success, this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through `cudaGraphicsUnregisterResource()`.

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

The type of `pD3DResource` must be one of the following.

- `IDirect3DVertexBuffer9`: may be accessed through a device pointer
- `IDirect3DIndexBuffer9`: may be accessed through a device pointer
- `IDirect3DSurface9`: may be accessed through an array. Only stand-alone objects of type `IDirect3DSurface9` may be explicitly shared. In particular, individual mipmap levels and faces of cube maps may not be registered directly. To access individual surfaces associated with a texture, one must register the base texture object.
- `IDirect3DBaseTexture9`: individual surfaces on this texture may be accessed through an array.

The `flags` argument may be used to specify additional parameters at register time. The only valid value for this parameter is

- `cudaGraphicsRegisterFlagsNone`

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 using [cudaD3D9SetDirect3DDevice](#) then [cudaErrorInvalidDevice](#) is returned. If `pD3DResource` is of incorrect type or is already registered, then [cudaErrorInvalidResourceHandle](#) is returned. If `pD3DResource` cannot be registered, then [cudaErrorUnknown](#) is returned.

**Parameters:**

*resource* - Pointer to returned resource handle

*pD3DResource* - Direct3D 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:**

[cudaD3D9SetDirect3DDevice](#), [cudaGraphicsUnregisterResource](#), [cudaGraphicsMapResources](#), [cudaGraphicsSubResourceGetMappedArray](#), [cudaGraphicsResourceGetMappedPointer](#)

## 4.11 Direct3D 10 Interoperability

### Modules

- [Direct3D 10 Interoperability \[DEPRECATED\]](#)

### Enumerations

- enum `cudaD3D10DeviceList` {  
`cudaD3D10DeviceListAll` = 1,  
`cudaD3D10DeviceListCurrentFrame` = 2,  
`cudaD3D10DeviceListNextFrame` = 3 }
- enum `cudaD3D10MapFlags` {  
`cudaD3D10MapFlagsNone` = 0,  
`cudaD3D10MapFlagsReadOnly` = 1,  
`cudaD3D10MapFlagsWriteDiscard` = 2 }
- enum `cudaD3D10RegisterFlags` {  
`cudaD3D10RegisterFlagsNone` = 0,  
`cudaD3D10RegisterFlagsArray` = 1 }

### Functions

- `cudaError_t cudaD3D10GetDevice` (int \*device, IDXGIAdapter \*pAdapter)  
*Gets the device number for an adapter.*
- `cudaError_t cudaD3D10GetDevices` (unsigned int \*pCudaDeviceCount, int \*pCudaDevices, unsigned int cudaDeviceCount, ID3D10Device \*pD3D10Device, enum `cudaD3D10DeviceList` deviceList)  
*Gets the CUDA devices corresponding to a Direct3D 10 device.*
- `cudaError_t cudaD3D10GetDirect3DDevice` (ID3D10Device \*\*ppD3D10Device)  
*Gets the Direct3D device against which the current CUDA context was created.*
- `cudaError_t cudaD3D10SetDirect3DDevice` (ID3D10Device \*pD3D10Device, int device=-1)  
*Sets the Direct3D 10 device to use for interoperability in this thread.*
- `cudaError_t cudaGraphicsD3D10RegisterResource` (struct `cudaGraphicsResource` \*\*resource, ID3D10Resource \*pD3DResource, unsigned int flags)  
*Register a Direct3D 10 resource for access by CUDA.*

#### 4.11.1 Detailed Description

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

## 4.11.2 Enumeration Type Documentation

### 4.11.2.1 enum cudaD3D10DeviceList

CUDA devices corresponding to a D3D10 device

**Enumerator:**

*cudaD3D10DeviceListAll* The CUDA devices for all GPUs used by a D3D10 device

*cudaD3D10DeviceListCurrentFrame* The CUDA devices for the GPUs used by a D3D10 device in its currently rendering frame

*cudaD3D10DeviceListNextFrame* The CUDA devices for the GPUs to be used by a D3D10 device in the next frame

### 4.11.2.2 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

### 4.11.2.3 enum cudaD3D10RegisterFlags

CUDA D3D10 Register Flags

**Enumerator:**

*cudaD3D10RegisterFlagsNone* Default; Resource can be accessed through a void\*

*cudaD3D10RegisterFlagsArray* Resource can be accessed through a CUarray\*

## 4.11.3 Function Documentation

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

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

**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](#), [cudaGraphicsD3D10RegisterResource](#),

#### 4.11.3.2 `cudaError_t cudaD3D10GetDevices (unsigned int * pCudaDeviceCount, int * pCudaDevices, unsigned int cudaDeviceCount, ID3D10Device * pD3D10Device, enum cudaD3D10DeviceList deviceList)`

Returns in `*pCudaDeviceCount` the number of CUDA-compatible devices corresponding to the Direct3D 10 device `pD3D10Device`. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of the the CUDA-compatible devices corresponding to the Direct3D 10 device `pD3D10Device`.

If any of the GPUs being used to render `pDevice` are not CUDA capable then the call will return [cudaErrorNoDevice](#).

**Parameters:**

*pCudaDeviceCount* - Returned number of CUDA devices corresponding to `pD3D10Device`

*pCudaDevices* - Returned CUDA devices corresponding to `pD3D10Device`

*cudaDeviceCount* - The size of the output device array `pCudaDevices`

*pD3D10Device* - Direct3D 10 device to query for CUDA devices

*deviceList* - The set of devices to return. This set may be [cudaD3D10DeviceListAll](#) for all devices, [cudaD3D10DeviceListCurrentFrame](#) for the devices used to render the current frame (in SLI), or [cudaD3D10DeviceListNextFrame](#) for the devices used to render the next frame (in SLI).

**Returns:**

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

**Note:**

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

See also:

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

#### 4.11.3.3 `cudaError_t cudaD3D10GetDirect3DDevice (ID3D10Device ** ppD3D10Device)`

Returns in `*ppD3D10Device` the Direct3D device against which this CUDA context was created in [cudaD3D10SetDirect3DDevice\(\)](#).

**Parameters:**

*ppD3D10Device* - 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:

[cudaD3D10SetDirect3DDevice](#)

#### 4.11.3.4 `cudaError_t cudaD3D10SetDirect3DDevice (ID3D10Device * pD3D10Device, int device = -1)`

Records `pD3D10Device` as the Direct3D 10 device to use for Direct3D 10 interoperability on this host thread. If the host thread has already initialized the CUDA runtime by calling non-device management runtime functions or if there exists a CUDA driver context active on the host thread, then this call returns `cudaErrorSetActiveProcess`.

Successful context creation on `pD3D10Device` will increase the internal reference count on `pD3D10Device`. This reference count will be decremented upon destruction of this context through `cudaThreadExit()`.

##### Parameters:

*pD3D10Device* - Direct3D device to use for interoperability

*device* - The CUDA device to use. This device must be among the devices returned when querying `cudaD3D10DeviceListAll` from `cudaD3D10GetDevices`, may be set to -1 to automatically select an appropriate CUDA device.

##### Returns:

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

##### Note:

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

##### See also:

`cudaD3D10GetDevice`, `cudaGraphicsD3D10RegisterResource`

#### 4.11.3.5 `cudaError_t cudaGraphicsD3D10RegisterResource (struct cudaGraphicsResource ** resource, ID3D10Resource * pD3DResource, unsigned int flags)`

Registers the Direct3D 10 resource `pD3DResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through `cudaGraphicsUnregisterResource()`. Also on success, this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through `cudaGraphicsUnregisterResource()`.

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

The type of `pD3DResource` must be one of the following.

- `ID3D10Buffer`: may be accessed via a device pointer
- `ID3D10Texture1D`: individual subresources of the texture may be accessed via arrays
- `ID3D10Texture2D`: individual subresources of the texture may be accessed via arrays
- `ID3D10Texture3D`: individual subresources of the texture may be accessed via arrays

The `flags` argument may be used to specify additional parameters at register time. The only valid value for this parameter is

- `cudaGraphicsRegisterFlagsNone`

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 using [cudaD3D10SetDirect3DDevice](#) then [cudaErrorInvalidDevice](#) is returned. If `pD3DResource` is of incorrect type or is already registered, then [cudaErrorInvalidResourceHandle](#) is returned. If `pD3DResource` cannot be registered, then [cudaErrorUnknown](#) is returned.

**Parameters:**

*resource* - Pointer to returned resource handle  
*pD3DResource* - Direct3D 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:**

[cudaD3D10SetDirect3DDevice](#), [cudaGraphicsUnregisterResource](#), [cudaGraphicsMapResources](#), [cudaGraphicsSubResourceGetMappedArray](#), [cudaGraphicsResourceGetMappedPointer](#)

## 4.12 Direct3D 11 Interoperability

### Enumerations

- enum `cudaD3D11DeviceList` {  
`cudaD3D11DeviceListAll` = 1,  
`cudaD3D11DeviceListCurrentFrame` = 2,  
`cudaD3D11DeviceListNextFrame` = 3 }

### Functions

- `cudaError_t cudaD3D11GetDevice` (int \*device, IDXGIAdapter \*pAdapter)  
*Gets the device number for an adapter.*
- `cudaError_t cudaD3D11GetDevices` (unsigned int \*pCudaDeviceCount, int \*pCudaDevices, unsigned int cudaDeviceCount, ID3D11Device \*pD3D11Device, enum `cudaD3D11DeviceList` deviceList)  
*Gets the CUDA devices corresponding to a Direct3D 11 device.*
- `cudaError_t cudaD3D11GetDirect3DDevice` (ID3D11Device \*\*ppD3D11Device)  
*Gets the Direct3D device against which the current CUDA context was created.*
- `cudaError_t cudaD3D11SetDirect3DDevice` (ID3D11Device \*pD3D11Device, int device=-1)  
*Sets the Direct3D 11 device to use for interoperability in this thread.*
- `cudaError_t cudaGraphicsD3D11RegisterResource` (struct `cudaGraphicsResource` \*\*resource, ID3D11Resource \*pD3DResource, unsigned int flags)  
*Register a Direct3D 11 resource for access by CUDA.*

#### 4.12.1 Detailed Description

This section describes the Direct3D 11 interoperability functions of the CUDA runtime application programming interface.

#### 4.12.2 Enumeration Type Documentation

##### 4.12.2.1 enum `cudaD3D11DeviceList`

CUDA devices corresponding to a D3D11 device

##### Enumerator:

**`cudaD3D11DeviceListAll`** The CUDA devices for all GPUs used by a D3D11 device

**`cudaD3D11DeviceListCurrentFrame`** The CUDA devices for the GPUs used by a D3D11 device in its currently rendering frame

**`cudaD3D11DeviceListNextFrame`** The CUDA devices for the GPUs to be used by a D3D11 device in the next frame

### 4.12.3 Function Documentation

#### 4.12.3.1 `cudaError_t cudaD3D11GetDevice (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* - D3D11 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:**

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

#### 4.12.3.2 `cudaError_t cudaD3D11GetDevices (unsigned int * pCudaDeviceCount, int * pCudaDevices, unsigned int cudaDeviceCount, ID3D11Device * pD3D11Device, enum cudaD3D11DeviceList deviceList)`

Returns in `*pCudaDeviceCount` the number of CUDA-compatible devices corresponding to the Direct3D 11 device `pD3D11Device`. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of the the CUDA-compatible devices corresponding to the Direct3D 11 device `pD3D11Device`.

If any of the GPUs being used to render `pDevice` are not CUDA capable then the call will return [cudaErrorNoDevice](#).

**Parameters:**

*pCudaDeviceCount* - Returned number of CUDA devices corresponding to `pD3D11Device`  
*pCudaDevices* - Returned CUDA devices corresponding to `pD3D11Device`  
*cudaDeviceCount* - The size of the output device array `pCudaDevices`  
*pD3D11Device* - Direct3D 11 device to query for CUDA devices  
*deviceList* - The set of devices to return. This set may be [cudaD3D11DeviceListAll](#) for all devices, [cudaD3D11DeviceListCurrentFrame](#) for the devices used to render the current frame (in SLI), or [cudaD3D11DeviceListNextFrame](#) for the devices used to render the next frame (in SLI).

**Returns:**

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

**Note:**

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

**See also:**

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

#### 4.12.3.3 `cudaError_t cudaD3D11GetDirect3DDevice (ID3D11Device ** ppD3D11Device)`

Returns in `*ppD3D11Device` the Direct3D device against which this CUDA context was created in `cudaD3D11SetDirect3DDevice()`.

##### Parameters:

`ppD3D11Device` - 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:

`cudaD3D11SetDirect3DDevice`

#### 4.12.3.4 `cudaError_t cudaD3D11SetDirect3DDevice (ID3D11Device * pD3D11Device, int device = -1)`

Records `pD3D11Device` as the Direct3D 11 device to use for Direct3D 11 interoperability on this host thread. If the host thread has already initialized the CUDA runtime by calling non-device management runtime functions or if there exists a CUDA driver context active on the host thread, then this call returns `cudaErrorSetActiveProcess`.

Successful context creation on `pD3D11Device` will increase the internal reference count on `pD3D11Device`. This reference count will be decremented upon destruction of this context through `cudaThreadExit()`.

##### Parameters:

`pD3D11Device` - Direct3D device to use for interoperability

`device` - The CUDA device to use. This device must be among the devices returned when querying `cudaD3D11DeviceListAll` from `cudaD3D11GetDevices`, may be set to -1 to automatically select an appropriate CUDA device.

##### Returns:

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

##### Note:

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

##### See also:

`cudaD3D11GetDevice`, `cudaGraphicsD3D11RegisterResource`

#### 4.12.3.5 `cudaError_t cudaGraphicsD3D11RegisterResource (struct cudaGraphicsResource ** resource, ID3D11Resource * pD3DResource, unsigned int flags)`

Registers the Direct3D 11 resource `pD3DResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through `cudaGraphicsUnregisterResource()`. Also on success, this call will increase the internal reference count on

`pD3DResource`. This reference count will be decremented when this resource is unregistered through [cudaGraphicsUnregisterResource\(\)](#).

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

The type of `pD3DResource` must be one of the following.

- `ID3D11Buffer`: may be accessed via a device pointer
- `ID3D11Texture1D`: individual subresources of the texture may be accessed via arrays
- `ID3D11Texture2D`: individual subresources of the texture may be accessed via arrays
- `ID3D11Texture3D`: individual subresources of the texture may be accessed via arrays

The `flags` argument may be used to specify additional parameters at register time. The only valid value for this parameter is

- [cudaGraphicsRegisterFlagsNone](#)

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 using [cudaD3D11SetDirect3DDevice](#) then [cudaErrorInvalidDevice](#) is returned. If `pD3DResource` is of incorrect type or is already registered, then [cudaErrorInvalidResourceHandle](#) is returned. If `pD3DResource` cannot be registered, then [cudaErrorUnknown](#) is returned.

#### Parameters:

*resource* - Pointer to returned resource handle

*pD3DResource* - Direct3D 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:

[cudaD3D11SetDirect3DDevice](#), [cudaGraphicsUnregisterResource](#), [cudaGraphicsMapResources](#), [cudaGraphicsSubResourceGetMappedArray](#), [cudaGraphicsResourceGetMappedPointer](#)

## 4.13 VDPAU Interoperability

### Functions

- [cudaError\\_t cudaGraphicsVDPAURegisterOutputSurface](#) (struct cudaGraphicsResource \*\*resource, VdpOutputSurface vdpSurface, unsigned int flags)  
*Register a VdpOutputSurface object.*
- [cudaError\\_t cudaGraphicsVDPAURegisterVideoSurface](#) (struct cudaGraphicsResource \*\*resource, VdpVideoSurface vdpSurface, unsigned int flags)  
*Register a VdpVideoSurface object.*
- [cudaError\\_t cudaVDPAUGetDevice](#) (int \*device, VdpDevice vdpDevice, VdpGetProcAddress \*vdpGetProcAddress)  
*Gets the CUDA device associated with a VdpDevice.*
- [cudaError\\_t cudaVDPAUSetVDPAUDevice](#) (int device, VdpDevice vdpDevice, VdpGetProcAddress \*vdpGetProcAddress)  
*Sets the CUDA device for use with VDPAU interoperability.*

### 4.13.1 Detailed Description

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

### 4.13.2 Function Documentation

#### 4.13.2.1 [cudaError\\_t cudaGraphicsVDPAURegisterOutputSurface](#) (struct cudaGraphicsResource \*\*resource, VdpOutputSurface vdpSurface, unsigned int flags)

Registers the VdpOutputSurface specified by `vdpSurface` for access by CUDA. A handle to the registered object is returned as `resource`. The surface's intended usage is specified using `flags`, as follows:

- [cudaGraphicsMapFlagsNone](#): Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- [cudaGraphicsMapFlagsReadOnly](#): Specifies that CUDA will not write to this resource.
- [cudaGraphicsMapFlagsWriteDiscard](#): Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

#### Parameters:

*resource* - Pointer to the returned object handle  
*vdpSurface* - VDPAU object to be registered  
*flags* - Map flags

#### Returns:

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

**Note:**

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

**See also:**

[cudaVDPAUSetVDPAUDevice](#), [cudaGraphicsUnregisterResource](#), [cudaGraphicsSubResourceGetMappedArray](#)

#### 4.13.2.2 `cudaError_t cudaGraphicsVDPAURegisterVideoSurface (struct cudaGraphicsResource ** resource, VdpVideoSurface vdpSurface, unsigned int flags)`

Registers the `VdpVideoSurface` specified by `vdpSurface` for access by CUDA. A handle to the registered object is returned as `resource`. The surface's intended usage is specified using `flags`, as follows:

- `cudaGraphicsMapFlagsNone`: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- `cudaGraphicsMapFlagsReadOnly`: Specifies that CUDA will not write to this resource.
- `cudaGraphicsMapFlagsWriteDiscard`: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

**Parameters:**

*resource* - Pointer to the returned object handle

*vdpSurface* - VDPAU object to be registered

*flags* - Map flags

**Returns:**

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

**Note:**

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

**See also:**

[cudaVDPAUSetVDPAUDevice](#), [cudaGraphicsUnregisterResource](#), [cudaGraphicsSubResourceGetMappedArray](#)

#### 4.13.2.3 `cudaError_t cudaVDPAUGetDevice (int * device, VdpDevice vdpDevice, VdpGetProcAddress * vdpGetProcAddress)`

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

**Parameters:**

*device* - Returns the device associated with `vdpDevice`, or -1 if the device associated with `vdpDevice` is not a compute device.

*vdpDevice* - A `VdpDevice` handle

*vdpGetProcAddress* - VDPAU's `VdpGetProcAddress` function pointer

**Returns:**

[cudaSuccess](#)

**Note:**

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

**See also:**

[cudaVDPAUSetVDPAUDevice](#)

**4.13.2.4 `cudaError_t cudaVDPAUSetVDPAUDevice (int device, VdpDevice vdpDevice, VdpGetProcAddress * vdpGetProcAddress)`**

Records `device` as the device on which the active host thread executes the device code. Records the thread as using VDPAU interoperability. If the host thread has already initialized the CUDA runtime by calling non-device management runtime functions or if there exists a CUDA driver context active on the host thread, then this call returns [cudaErrorSetOnActiveProcess](#).

**Parameters:**

*device* - Device to use for VDPAU interoperability

*vdpDevice* - The `VdpDevice` to interoperate with

*vdpGetProcAddress* - VDPAU's `VdpGetProcAddress` function pointer

**Returns:**

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

**Note:**

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

**See also:**

[cudaGraphicsVDPAURegisterVideoSurface](#), [cudaGraphicsVDPAURegisterOutputSurface](#)

## 4.14 Graphics Interoperability

### Functions

- `cudaError_t cudaGraphicsMapResources` (int *count*, `cudaGraphicsResource_t` \**resources*, `cudaStream_t` *stream*=0)  
*Map graphics resources for access by CUDA.*
- `cudaError_t cudaGraphicsResourceGetMappedPointer` (void \*\**devPtr*, `size_t` \**size*, `cudaGraphicsResource_t` *resource*)  
*Get an device pointer through which to access a mapped graphics resource.*
- `cudaError_t cudaGraphicsResourceSetMapFlags` (`cudaGraphicsResource_t` *resource*, unsigned int *flags*)  
*Set usage flags for mapping a graphics resource.*
- `cudaError_t cudaGraphicsSubResourceGetMappedArray` (struct `cudaArray` \*\**array*, `cudaGraphicsResource_t` *resource*, unsigned int *arrayIndex*, unsigned int *mipLevel*)  
*Get an array through which to access a subresource of a mapped graphics resource.*
- `cudaError_t cudaGraphicsUnmapResources` (int *count*, `cudaGraphicsResource_t` \**resources*, `cudaStream_t` *stream*=0)  
*Unmap graphics resources.*
- `cudaError_t cudaGraphicsUnregisterResource` (`cudaGraphicsResource_t` *resource*)  
*Unregisters a graphics resource for access by CUDA.*

### 4.14.1 Detailed Description

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

### 4.14.2 Function Documentation

#### 4.14.2.1 `cudaError_t cudaGraphicsMapResources` (int *count*, `cudaGraphicsResource_t` \* *resources*, `cudaStream_t` *stream* = 0)

Maps the `count` graphics resources in `resources` for access by CUDA.

The resources in `resources` may be accessed by CUDA until they are unmapped. The graphics API from which `resources` were registered should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any graphics calls issued before `cudaGraphicsMapResources()` will complete before any subsequent CUDA work issued in `stream` begins.

If `resources` contains any duplicate entries then `cudaErrorInvalidResourceHandle` is returned. If any of `resources` are presently mapped for access by CUDA then `cudaErrorUnknown` is returned.

#### Parameters:

*count* - Number of resources to map

*resources* - Resources to map for CUDA

*stream* - Stream for synchronization

**Returns:**

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

**Note:**

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

**See also:**

[cudaGraphicsResourceGetMappedPointer](#) [cudaGraphicsSubResourceGetMappedArray](#) [cudaGraphicsUnmapResources](#)

**4.14.2.2 `cudaError_t cudaGraphicsResourceGetMappedPointer (void ** devPtr, size_t * size, cudaGraphicsResource_t resource)`**

Returns in `*devPtr` a pointer through which the mapped graphics resource `resource` may be accessed. Returns in `*size` the size of the memory in bytes which may be accessed from that pointer. The value set in `devPtr` may change every time that `resource` is mapped.

If `resource` is not a buffer then it cannot be accessed via a pointer and [cudaErrorUnknown](#) is returned. If `resource` is not mapped then [cudaErrorUnknown](#) is returned. \*

**Parameters:**

*devPtr* - Returned pointer through which `resource` may be accessed

*size* - Returned size of the buffer accessible starting at `*devPtr`

*resource* - Mapped resource to access

**Returns:**

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

**Note:**

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

**See also:**

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

**4.14.2.3 `cudaError_t cudaGraphicsResourceSetMapFlags (cudaGraphicsResource_t resource, unsigned int flags)`**

Set `flags` for mapping the graphics resource `resource`.

Changes to `flags` will take effect the next time `resource` is mapped. The `flags` argument may be any of the following:

- [cudaGraphicsMapFlagsNone](#): Specifies no hints about how `resource` will be used. It is therefore assumed that CUDA may read from or write to `resource`.
- [cudaGraphicsMapFlagsReadOnly](#): Specifies that CUDA will not write to `resource`.

- [cudaGraphicsMapFlagsWriteDiscard](#): Specifies CUDA will not read from `resource` and will write over the entire contents of `resource`, so none of the data previously stored in `resource` will be preserved.

If `resource` is presently mapped for access by CUDA then [cudaErrorUnknown](#) is returned. If `flags` is not one of the above values then [cudaErrorInvalidValue](#) is returned.

**Parameters:**

*resource* - 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:**

[cudaGraphicsMapResources](#)

#### 4.14.2.4 [cudaError\\_t cudaGraphicsSubResourceGetMappedArray \(struct cudaArray \\*\\* array, cudaGraphicsResource\\_t resource, unsigned int arrayIndex, unsigned int mipLevel\)](#)

Returns in `*array` an array through which the subresource of the mapped graphics resource `resource` which corresponds to array index `arrayIndex` and mipmap level `mipLevel` may be accessed. The value set in `array` may change every time that `resource` is mapped.

If `resource` is not a texture then it cannot be accessed via an array and [cudaErrorUnknown](#) is returned. If `arrayIndex` is not a valid array index for `resource` then [cudaErrorInvalidValue](#) is returned. If `mipLevel` is not a valid mipmap level for `resource` then [cudaErrorInvalidValue](#) is returned. If `resource` is not mapped then [cudaErrorUnknown](#) is returned.

**Parameters:**

*array* - Returned array through which a subresource of `resource` may be accessed

*resource* - Mapped resource to access

*arrayIndex* - Array index for array textures or cubemap face index as defined by [cudaGraphicsCubeFace](#) for cubemap textures for the subresource to access

*mipLevel* - Mipmap level for the subresource to access

**Returns:**

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

**Note:**

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

**See also:**

[cudaGraphicsResourceGetMappedPointer](#)

#### 4.14.2.5 `cudaError_t cudaGraphicsUnmapResources (int count, cudaGraphicsResource_t * resources, cudaStream_t stream = 0)`

Unmaps the `count` graphics resources in `resources`.

Once unmapped, the resources in `resources` may not be accessed by CUDA until they are mapped again.

This function provides the synchronization guarantee that any CUDA work issued in `stream` before `cudaGraphicsUnmapResources()` will complete before any subsequently issued graphics work begins.

If `resources` contains any duplicate entries then `cudaErrorInvalidResourceHandle` is returned. If any of `resources` are not presently mapped for access by Cuda then `cudaErrorUnknown` is returned.

##### Parameters:

*count* - Number of resources to unmap

*resources* - Resources to unmap

*stream* - Stream for synchronization

##### Returns:

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

##### Note:

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

##### See also:

[cudaGraphicsMapResources](#)

#### 4.14.2.6 `cudaError_t cudaGraphicsUnregisterResource (cudaGraphicsResource_t resource)`

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

If `resource` is invalid then `cudaErrorInvalidResourceHandle` is returned.

##### Parameters:

*resource* - Resource to unregister

##### Returns:

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

##### Note:

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

##### See also:

[cudaGraphicsD3D9RegisterResource](#), [cudaGraphicsD3D10RegisterResource](#), [cudaGraphicsD3D11RegisterResource](#), [cudaGraphicsGLRegisterBuffer](#), [cudaGraphicsGLRegisterImage](#)

## 4.15 Texture Reference Management

### Functions

- `cudaError_t cudaBindTexture` (`size_t *offset`, `const struct textureReference *texref`, `const void *devPtr`, `const struct cudaChannelFormatDesc *desc`, `size_t size=UINT_MAX`)  
*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.*

### 4.15.1 Detailed Description

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

### 4.15.2 Function Documentation

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

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)

**4.15.2.2 `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), [cudaBindTexture2D](#) (C++ API, inherited channel descriptor), [cudaBindTextureToArray](#) (C API), [cudaBindTextureToArray](#) (C API), [cudaGetTextureAlignmentOffset](#) (C API)

#### 4.15.2.3 `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)

#### 4.15.2.4 `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 \(C API\)](#)

**4.15.2.5 `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 \(C API\)](#)

**4.15.2.6 `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 \(C++ API\)](#)

**4.15.2.7 `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 \(C API\)](#)

**4.15.2.8 `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 \(C API\)](#)

## 4.16 Surface Reference Management

### Functions

- [cudaError\\_t cudaBindSurfaceToArray](#) (const struct [surfaceReference](#) \*surfref, const struct cudaArray \*array, const struct [cudaChannelFormatDesc](#) \*desc)  
*Binds an array to a surface.*
- [cudaError\\_t cudaGetSurfaceReference](#) (const struct [surfaceReference](#) \*\*surfref, const char \*symbol)  
*Get the surface reference associated with a symbol.*

### 4.16.1 Detailed Description

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

### 4.16.2 Function Documentation

#### 4.16.2.1 [cudaError\\_t cudaBindSurfaceToArray](#) (const struct [surfaceReference](#) \* *surfref*, const struct [cudaArray](#) \* *array*, const struct [cudaChannelFormatDesc](#) \* *desc*)

Binds the CUDA array *array* to the surface reference *surfref*. *desc* describes how the memory is interpreted when fetching values from the surface. Any CUDA array previously bound to *surfref* is unbound.

#### Parameters:

- surfref* - Surface to bind
- array* - Memory array on device
- desc* - Channel format

#### Returns:

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

#### Note:

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

#### See also:

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

#### 4.16.2.2 [cudaError\\_t cudaGetSurfaceReference](#) (const struct [surfaceReference](#) \*\* *surfref*, const char \* *symbol*)

Returns in \**surfref* the structure associated to the surface reference defined by symbol *symbol*.

#### Parameters:

- surfref* - Surface associated with symbol

*symbol* - Symbol to find surface reference for

**Returns:**

[cudaSuccess](#), [cudaErrorInvalidSurface](#)

**Note:**

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

**See also:**

[cudaBindSurfaceToArray \(C API\)](#)

## 4.17 Version Management

### Functions

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

### 4.17.1 Function Documentation

#### 4.17.1.1 [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](#)

#### 4.17.1.2 [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](#)

## 4.18 C++ API Routines

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

### Functions

- `template<class T, int dim>`  
`cudaError_t cudaBindSurfaceToArray` (const struct surface< T, dim > &surf, const struct cudaArray \*array)  
*[C++ API] Binds an array to a surface*
- `template<class T, int dim>`  
`cudaError_t cudaBindSurfaceToArray` (const struct surface< T, dim > &surf, const struct cudaArray \*array, const struct cudaChannelFormatDesc &desc)  
*[C++ API] Binds an array to a surface*
- `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, 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 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*
- `cudaError_t cudaEventCreate` (cudaEvent\_t \*event, unsigned int flags)  
*[C++ API] Creates an event object with the specified flags*

- `template<class T >`  
`cudaError_t cudaFuncGetAttributes` (struct `cudaFuncAttributes` \*attr, T \*entry)  
*[C++ API] Find out attributes for a given function*
- `template<class T >`  
`cudaError_t cudaFuncSetCacheConfig` (T \*func, enum `cudaFuncCache` cacheConfig)  
*Sets the preferred cache configuration for a device function.*
- `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*
- `cudaError_t cudaMallocHost` (void \*\*ptr, size\_t size, unsigned int flags)  
*[C++ API] Allocates page-locked memory on the host*
- `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*

### 4.18.1 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.

### 4.18.2 Function Documentation

#### 4.18.2.1 `template<class T, int dim> cudaError_t cudaBindSurfaceToArray` (const struct surface< T, dim > &surf, const struct cudaArray \*array)

Binds the CUDA array `array` to the surface reference `surf`. The channel descriptor is inherited from the CUDA array. Any CUDA array previously bound to `surf` is unbound.

**Parameters:**

*surf* - Surface to bind  
*array* - Memory array on device

**Returns:**

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

**Note:**

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

**See also:**

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

#### 4.18.2.2 `template<class T , int dim> cudaError_t cudaBindSurfaceToArray (const struct surface< T, dim > & surf, const struct cudaArray * array, const struct cudaChannelFormatDesc & desc)`

Binds the CUDA array `array` to the surface reference `surf`. `desc` describes how the memory is interpreted when dealing with the surface. Any CUDA array previously bound to `surf` is unbound.

**Parameters:**

*surf* - Surface to bind  
*array* - Memory array on device  
*desc* - Channel format

**Returns:**

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

**Note:**

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

**See also:**

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

#### 4.18.2.3 `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), [cudaBindTexture2D](#) (C++ API), [cudaBindTexture2D](#) (C++ API, inherited channel descriptor), [cudaBindTextureToArray](#) (C++ API), [cudaBindTextureToArray](#) (C++ API, inherited channel descriptor), [cudaUnbindTexture](#) (C++ API), [cudaGetTextureAlignmentOffset](#) (C++ API)

**4.18.2.4** `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), [cudaBindTexture2D](#) (C++ API, inherited channel descriptor), [cudaBindTextureToArray](#) (C++ API), [cudaBindTextureToArray](#) (C++ API, inherited channel descriptor), [cudaUnbindTexture](#) (C++ API), [cudaGetTextureAlignmentOffset](#) (C++ API)

**4.18.2.5** `template<class T , int dim, enum cudaTextureReadMode readMode> cudaError_t  
 cudaBindTexture2D (size_t * offset, const struct texture< T, dim, readMode > & tex, const void *  
devPtr, 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. The channel descriptor is inherited from the texture reference type. Any memory previously bound to `tex` is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, `cudaBindTexture2D()` returns in `*offset` a byte offset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the `tex2D()` function. If the device memory pointer was returned from `cudaMalloc()`, the offset is guaranteed to be 0 and NULL may be passed as the `offset` parameter.

**Parameters:**

*offset* - Offset in bytes  
*tex* - Texture reference to bind  
*devPtr* - 2D memory area on device  
*width* - Width in texel units  
*height* - Height in texel units  
*pitch* - Pitch in bytes

**Returns:**

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

**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), [cudaBindTexture2D](#) (C++ API), [cudaBindTextureToArray](#) (C++ API), [cudaBindTextureToArray](#) (C++ API, inherited channel descriptor), [cudaUnbindTexture](#) (C++ API), [cudaGetTextureAlignmentOffset](#) (C++ API)

**4.18.2.6** `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), [cudaBindTexture2D](#) (C++ API, inherited channel descriptor), [cudaBindTextureToArray](#) (C++ API), [cudaBindTextureToArray](#) (C++ API, inherited channel descriptor), [cudaUnbindTexture](#) (C++ API), [cudaGetTextureAlignmentOffset](#) (C++ API)

**4.18.2.7** `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), [cudaBindTexture2D](#) (C++ API, inherited channel descriptor), [cudaBindTextureToArray](#) (C API), [cudaBindTextureToArray](#) (C++ API), [cudaUnbindTexture](#) (C++ API), [cudaGetTextureAlignmentOffset](#) (C++ API)

**4.18.2.8** `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), [cudaBindTexture2D](#) (C++ API, inherited channel descriptor), [cudaBindTextureToArray](#) (C API), [cudaBindTextureToArray](#) (C++ API, inherited channel descriptor), [cudaUnbindTexture](#) (C++ API), [cudaGetTextureAlignmentOffset](#) (C++ API)

**4.18.2.9** `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)

#### 4.18.2.10 `cudaError_t cudaEventCreate (cudaEvent_t * event, unsigned int flags)`

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

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

##### Parameters:

*event* - Newly created event

*flags* - Flags for new event

##### Returns:

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

##### Note:

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

##### See also:

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

#### 4.18.2.11 `template<class T> cudaError_t cudaFuncGetAttributes (struct cudaFuncAttributes * attr, T * entry)`

This function obtains the attributes of a function specified via `entry`. The parameter `entry` can either be a pointer to a function that executes on the device, or it can be a character string specifying the fully-decorated (C++) name of a function that executes on the device. The parameter specified by `entry` must be declared as a `__global__` function. The fetched attributes are placed in `attr`. If the specified function does not exist, then `cudaErrorInvalidDeviceFunction` is returned.

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

##### Parameters:

*attr* - Return pointer to function's attributes

*entry* - 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:**

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

#### 4.18.2.12 `template<class T > cudaError_t cudaFuncSetCacheConfig (T *func, enum cudaFuncCache cacheConfig)`

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

`func` can either be a pointer to a function that executes on the device, or it can be a character string specifying the fully-decorated (C++) name for a function that executes on the device. The parameter specified by `func` must be declared as a `__global__` function. If the specified function does not exist, then [cudaErrorInvalidDeviceFunction](#) is returned.

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

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

The supported cache configurations are:

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

**Parameters:**

*func* - Char string naming device function

*cacheConfig* - Requested cache configuration

**Returns:**

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

**Note:**

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

**See also:**

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

#### 4.18.2.13 `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 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 the global or constant memory space, `*devPtr` is unchanged and the error [cudaErrorInvalidSymbol](#) is returned. If there are multiple global or constant variables with the same string name (from separate files) and the lookup is done via character string, [cudaErrorDuplicateVariableName](#) is returned.

**Parameters:**

- devPtr* - Return device pointer associated with symbol
- symbol* - Global/constant variable or string symbol to search for

**Returns:**

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

**Note:**

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

**See also:**

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

**4.18.2.14** `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. If there are multiple global variables with the same string name (from separate files) and the lookup is done via character string, [cudaErrorDuplicateVariableName](#) is returned.

**Parameters:**

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

**Returns:**

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

**Note:**

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

**See also:**

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

**4.18.2.15** `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), [cudaBindTexture2D](#) (C++ API, inherited channel descriptor), [cudaBindTextureToArray](#) (C++ API), [cudaBindTextureToArray](#) (C++ API, inherited channel descriptor), [cudaUnbindTexture](#) (C++ API), [cudaGetTextureAlignmentOffset](#) (C API)

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

Launches the function `entry` on the device. The parameter `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. The parameter specified by `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](#), [cudaErrorLaunchFailure](#), [cudaErrorLaunchTimeout](#), [cudaErrorLaunchOutOfResources](#), [cudaErrorSharedObjectSymbolNotFound](#), [cudaErrorSharedObjectInitFailed](#)

**Note:**

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

**See also:**

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

**4.18.2.17 `cudaError_t cudaMallocHost (void ** ptr, size_t size, unsigned int flags)`**

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

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

- [cudaHostAllocDefault](#): This flag's value is defined to be 0.
- [cudaHostAllocPortable](#): The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.

- [cudaHostAllocMapped](#): Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling [cudaHostGetDevicePointer\(\)](#).
- [cudaHostAllocWriteCombined](#): Allocates the memory as write-combined (WC). WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs. WC memory is a good option for buffers that will be written by the CPU and read by the device via mapped pinned memory or host->device transfers.

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

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

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

Memory allocated by this function must be freed with [cudaFreeHost\(\)](#).

**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 \(C API\)](#), [cudaFreeHost](#), [cudaHostAlloc](#)

#### 4.18.2.18 `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:**

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

**4.18.2.19** `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), [cudaBindTexture2D](#) (C++ API, inherited channel descriptor), [cudaBindTextureToArray](#) (C++ API), [cudaBindTextureToArray](#) (C++ API, inherited channel descriptor), [cudaUnbindTexture](#) (C API), [cudaGetTextureAlignmentOffset](#) (C++ API)

## 4.19 Interactions with the CUDA Driver API

Interactions between the CUDA Driver API and the CUDA Runtime API.

This section describes the interactions between the CUDA Driver API and the CUDA Runtime API

### 4.19.1 Context Management

CUDA Runtime API calls operate on the CUDA Driver API [CUcontext](#) which is bound to the current host thread.

If there exists no CUDA Driver API [CUcontext](#) bound to the current thread at the time of a CUDA Runtime API call which requires a [CUcontext](#) then the CUDA Runtime will implicitly create a new [CUcontext](#) before executing the call.

If the CUDA Runtime creates a [CUcontext](#) then the [CUcontext](#) will be created using the parameters specified by the CUDA Runtime API functions [cudaSetDevice](#), [cudaSetValidDevices](#), [cudaSetDeviceFlags](#), [cudaGLSetGLDevice](#), [cudaD3D9SetDirect3DDevice](#), [cudaD3D10SetDirect3DDevice](#), and [cudaD3D11SetDirect3DDevice](#). Note that these functions will fail with [cudaErrorSetOnActiveProcess](#) if they are called when a [CUcontext](#) is bound to the current host thread.

The lifetime of a [CUcontext](#) is managed by a reference counting mechanism. The reference count of a [CUcontext](#) is initially set to 0, and is incremented by [cuCtxAttach](#) and decremented by [cuCtxDetach](#).

If a [CUcontext](#) is created by the CUDA Runtime, then the CUDA runtime will decrement the reference count of that [CUcontext](#) in the function [cudaThreadExit](#). If a [CUcontext](#) is created by the CUDA Driver API (or is created by a separate instance of the CUDA Runtime API library), then the CUDA Runtime will not increment or decrement the reference count of that [CUcontext](#).

All CUDA Runtime API state (e.g, global variables' addresses and values) travels with its underlying [CUcontext](#). In particular, if a [CUcontext](#) is moved from one thread to another (using [cuCtxPopCurrent](#) and [cuCtxPushCurrent](#)) then all CUDA Runtime API state will move to that thread as well.

Please note that attaching to legacy contexts (those with a version of 3010 as returned by [cuCtxGetApiVersion\(\)](#)) is not possible. The CUDA Runtime will return [cudaErrorIncompatibleDriverContext](#) in such cases.

### 4.19.2 Interactions between CUstream and cudaStream\_t

The types [CUstream](#) and [cudaStream\\_t](#) are identical and may be used interchangeably.

### 4.19.3 Interactions between CUEvent and cudaEvent\_t

The types [CUEvent](#) and [cudaEvent\\_t](#) are identical and may be used interchangeably.

### 4.19.4 Interactions between CUarray and struct cudaArray \*

The types [CUarray](#) and `struct cudaArray *` represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a [CUarray](#) in a CUDA Runtime API function which takes a `struct cudaArray *`, it is necessary to explicitly cast the [CUarray](#) to a `struct cudaArray *`.

In order to use a `struct cudaArray *` in a CUDA Driver API function which takes a [CUarray](#), it is necessary to explicitly cast the `struct cudaArray *` to a [CUarray](#).

### 4.19.5 Interactions between CUgraphicsResource and cudaGraphicsResource\_t

The types [CUgraphicsResource](#) and struct `cudaGraphicsResource *` represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a [CUgraphicsResource](#) in a CUDA Runtime API function which takes a struct `cudaGraphicsResource *`, it is necessary to explicitly cast the [CUgraphicsResource](#) to a struct `cudaGraphicsResource *`.

In order to use a struct `cudaGraphicsResource *` in a CUDA Driver API function which takes a [CUgraphicsResource](#), it is necessary to explicitly cast the struct `cudaGraphicsResource *` to a [CUgraphicsResource](#) .

## 4.20 Direct3D 9 Interoperability [DEPRECATED]

### Functions

- [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 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.*

### 4.20.1 Detailed Description

This section describes deprecated Direct3D 9 interoperability functions.

### 4.20.2 Function Documentation

#### 4.20.2.1 [cudaError\\_t cudaD3D9MapResources](#) (int count, IDirect3DResource9 \*\* ppResources)

#### Deprecated

This function is deprecated as of Cuda 3.0.

Maps the `count` Direct3D resources in `ppResources` for access by CUDA.

The resources in `ppResources` may be accessed in CUDA kernels until they are unmapped. Direct3D should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any Direct3D calls issued before `cudaD3D9MapResources()` will complete before any CUDA kernels issued after `cudaD3D9MapResources()` begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries then `cudaErrorInvalidResourceHandle` is returned. If any of `ppResources` are presently mapped for access by CUDA then `cudaErrorUnknown` is returned.

**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:**

`cudaGraphicsMapResources`

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

**Deprecated**

This function is deprecated as of Cuda 3.0.

Registers the Direct3D resource `pResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through `cudaD3D9UnregisterResource()`. Also on success, this call will increase the internal reference count on `pResource`. This reference count will be decremented when this resource is unregistered through `cudaD3D9UnregisterResource()`.

This call 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:

[cudaGraphicsD3D9RegisterResource](#)

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

#### Deprecated

This function is deprecated as of Cuda 3.0.

Returns in `*ppArray` an array through which the subresource of the mapped Direct3D resource `pResource`, which corresponds to `face` and `level` may be accessed. The value set in `ppArray` may change every time that `pResource` is mapped.

If `pResource` is not registered then [cudaErrorInvalidResourceHandle](#) is returned. If `pResource` was not registered with usage flags [cudaD3D9RegisterFlagsArray](#), then [cudaErrorInvalidResourceHandle](#) is returned. If `pResource` is not mapped, then [cudaErrorUnknown](#) is returned.

For usage requirements of `face` and `level` parameters, see [cudaD3D9ResourceGetMappedPointer\(\)](#).

**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:**

[cudaGraphicsSubResourceGetMappedArray](#)

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

**Deprecated**

This function is deprecated as of Cuda 3.0.

Returns in *pPitch* and *pPitchSlice* the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource *pResource*, which corresponds to *face* and *level*. The values set in *pPitch* and *pPitchSlice* may change every time that *pResource* is mapped.

The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position *x*, *y* from the base pointer of the surface is:

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

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

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

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

If *pResource* is not of type `IDirect3DBaseTexture9` or one of its sub-types or if *pResource* has not been registered for use with CUDA, then [cudaErrorInvalidResourceHandle](#) is returned. If *pResource* was not registered with usage flags [cudaD3D9RegisterFlagsNone](#), then [cudaErrorInvalidResourceHandle](#) is returned. If *pResource* is not mapped for access by CUDA then [cudaErrorUnknown](#) is returned.

For usage requirements of *face* and *level* parameters, see [cudaD3D9ResourceGetMappedPointer\(\)](#).

**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:**

[cudaGraphicsResourceGetMappedPointer](#)

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

**Deprecated**

This function is deprecated as of Cuda 3.0.

Returns in `*pPointer` the base pointer of the subresource of the mapped Direct3D resource `pResource`, which corresponds to `face` and `level`. The value set in `pPointer` may change every time that `pResource` is mapped.

If `pResource` is not registered, then [cudaErrorInvalidResourceHandle](#) is returned. If `pResource` was not registered with usage flags [cudaD3D9RegisterFlagsNone](#), then [cudaErrorInvalidResourceHandle](#) is returned. If `pResource` is not mapped, then [cudaErrorUnknown](#) is returned.

If `pResource` is of type `IDirect3DCubeTexture9`, then `face` must one of the values enumerated by type `D3DCUBEMAP_FACES`. For all other types, `face` must be 0. If `face` is invalid, then [cudaErrorInvalidValue](#) is returned.

If `pResource` is of type `IDirect3DBaseTexture9`, then `level` must correspond to a valid mipmap level. Only mipmap level 0 is supported for now. For all other types `level` must be 0. If `level` is invalid, then [cudaErrorInvalidValue](#) is returned.

**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:**

[cudaGraphicsResourceGetMappedPointer](#)

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

##### Deprecated

This function is deprecated as of Cuda 3.0.

Returns in `*pSize` the size of the subresource of the mapped Direct3D resource `pResource`, which corresponds to `face` and `level`. The value set in `pSize` may change every time that `pResource` is mapped.

If `pResource` has not been registered for use with CUDA then `cudaErrorInvalidResourceHandle` is returned. If `pResource` was not registered with usage flags `cudaD3D9RegisterFlagsNone`, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` is not mapped for access by CUDA then `cudaErrorUnknown` is returned.

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

##### 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:

`cudaGraphicsResourceGetMappedPointer`

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

##### Deprecated

This function is deprecated as of Cuda 3.0.

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

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

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

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

[cudaGraphicsSubResourceGetMappedArray](#)

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

**Deprecated**

This function is deprecated as of Cuda 3.0.

Set flags for mapping the Direct3D resource `pResource`.

Changes to flags will take effect the next time `pResource` is mapped. The `flags` argument may be any of the following:

- [cudaD3D9MapFlagsNone](#): Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- [cudaD3D9MapFlagsReadOnly](#): Specifies that CUDA kernels which access this resource will not write to this resource.
- [cudaD3D9MapFlagsWriteDiscard](#): Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If `pResource` has not been registered for use with CUDA, then [cudaErrorInvalidResourceHandle](#) is returned. If `pResource` is presently mapped for access by CUDA, then [cudaErrorUnknown](#) is returned.

**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:**

[cudaInteropResourceSetMapFlags](#)

#### 4.20.2.9 `cudaError_t cudaD3D9UnmapResources` (`int count`, `IDirect3DResource9 ** ppResources`)

##### Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the `count` Direct3D resources in `ppResources`.

This function provides the synchronization guarantee that any CUDA kernels issued before `cudaD3D9UnmapResources()` will complete before any Direct3D calls issued after `cudaD3D9UnmapResources()` begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries, then `cudaErrorInvalidResourceHandle` is returned. If any of `ppResources` are not presently mapped for access by CUDA then `cudaErrorUnknown` is returned.

##### 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:

[cudaGraphicsUnmapResources](#)

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

##### Deprecated

This function is deprecated as of Cuda 3.0.

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

If `pResource` is not registered, then `cudaErrorInvalidResourceHandle` is returned.

##### 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:

[cudaGraphicsUnregisterResource](#)

## 4.21 Direct3D 10 Interoperability [DEPRECATED]

### Functions

- [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 10 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 \*\*pPointer, 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 cudaD3D10UnmapResources](#) (int count, ID3D10Resource \*\*ppResources)  
*Unmaps Direct3D resources.*
- [cudaError\\_t cudaD3D10UnregisterResource](#) (ID3D10Resource \*pResource)  
*Unregisters a Direct3D resource.*

### 4.21.1 Detailed Description

This section describes deprecated Direct3D 10 interoperability functions.

### 4.21.2 Function Documentation

#### 4.21.2.1 [cudaError\\_t cudaD3D10MapResources](#) (int count, ID3D10Resource \*\* ppResources)

#### Deprecated

This function is deprecated as of Cuda 3.0.

Maps the `count` Direct3D resources in `ppResources` for access by CUDA.

The resources in `ppResources` may be accessed in CUDA kernels until they are unmapped. Direct3D should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any Direct3D calls issued before `cudaD3D10MapResources()` will complete before any CUDA kernels issued after `cudaD3D10MapResources()` begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries then `cudaErrorInvalidResourceHandle` is returned. If any of `ppResources` are presently mapped for access by CUDA then `cudaErrorUnknown` is returned.

**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:**

[cudaGraphicsMapResources](#)

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

**Deprecated**

This function is deprecated as of Cuda 3.0.

Registers the Direct3D resource `pResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through `cudaD3D10UnregisterResource()`. Also on success, this call will increase the internal reference count on `pResource`. This reference count will be decremented when this resource is unregistered through `cudaD3D10UnregisterResource()`.

This call 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:

[cudaGraphicsD3D10RegisterResource](#)

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

#### Deprecated

This function is deprecated as of Cuda 3.0.

Returns in `*ppArray` an array through which the subresource of the mapped Direct3D resource `pResource` which corresponds to `subResource` may be accessed. The value set in `ppArray` may change every time that `pResource` is mapped.

If `pResource` is not registered, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` was not registered with usage flags `cudaD3D10RegisterFlagsArray`, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` is not mapped then `cudaErrorUnknown` is returned.

For usage requirements of the `subResource` parameter, see [cudaD3D10ResourceGetMappedPointer\(\)](#).

**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:**

[cudaGraphicsSubResourceGetMappedArray](#)

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

**Deprecated**

This function is deprecated as of Cuda 3.0.

Returns in *pPitch* and *pPitchSlice* the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource *pResource*, which corresponds to *subResource*. The values set in *pPitch* and *pPitchSlice* may change every time that *pResource* is mapped.

The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position *x*, *y* from the base pointer of the surface is:

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

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

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

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

If *pResource* is not of type `ID3D10Texture1D`, `ID3D10Texture2D`, or `ID3D10Texture3D`, or if *pResource* has not been registered for use with CUDA, then [cudaErrorInvalidResourceHandle](#) is returned. If *pResource* was not registered with usage flags `cudaD3D10RegisterFlagsNone`, then [cudaErrorInvalidResourceHandle](#) is returned. If *pResource* is not mapped for access by CUDA then [cudaErrorUnknown](#) is returned.

For usage requirements of the *subResource* parameter see [cudaD3D10ResourceGetMappedPointer\(\)](#).

**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:**

[cudaGraphicsSubResourceGetMappedArray](#)

**4.21.2.5 `cudaError_t cudaD3D10ResourceGetMappedPointer (void ** pPointer, ID3D10Resource * pResource, unsigned int subResource)`****Deprecated**

This function is deprecated as of Cuda 3.0.

Returns in `*pPointer` the base pointer of the subresource of the mapped Direct3D resource `pResource` which corresponds to `subResource`. The value set in `pPointer` may change every time that `pResource` is mapped.

If `pResource` is not registered, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` was not registered with usage flags `cudaD3D9RegisterFlagsNone`, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` is not mapped then `cudaErrorUnknown` is returned.

If `pResource` is of type `ID3D10Buffer` then `subResource` must be 0. If `pResource` is of any other type, then the value of `subResource` must come from the subresource calculation in `D3D10CalcSubResource()`.

**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:**

[cudaGraphicsResourceGetMappedPointer](#)

**4.21.2.6 `cudaError_t cudaD3D10ResourceGetMappedSize (size_t * pSize, ID3D10Resource * pResource, unsigned int subResource)`****Deprecated**

This function is deprecated as of Cuda 3.0.

Returns in `*pSize` the size of the subresource of the mapped Direct3D resource `pResource` which corresponds to `subResource`. The value set in `pSize` may change every time that `pResource` is mapped.

If `pResource` has not been registered for use with CUDA then `cudaErrorInvalidHandle` is returned. If `pResource` was not registered with usage flags `cudaD3D10RegisterFlagsNone`, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` is not mapped for access by CUDA then `cudaErrorUnknown` is returned.

For usage requirements of the `subResource` parameter see [cudaD3D10ResourceGetMappedPointer\(\)](#).

**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:**

[cudaGraphicsResourceGetMappedPointer](#)

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

**Deprecated**

This function is deprecated as of Cuda 3.0.

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

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

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

If *pResource* is not of type `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:**

[cudaGraphicsSubResourceGetMappedArray](#)

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

##### Deprecated

This function is deprecated as of Cuda 3.0.

Set usage flags for mapping the Direct3D resource `pResource`.

Changes to flags will take effect the next time `pResource` is mapped. The `flags` argument may be any of the following:

- `cudaD3D10MapFlagsNone`: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- `cudaD3D10MapFlagsReadOnly`: Specifies that CUDA kernels which access this resource will not write to this resource.
- `cudaD3D10MapFlagsWriteDiscard`: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If `pResource` has not been registered for use with CUDA then `cudaErrorInvalidHandle` is returned. If `pResource` is presently mapped for access by CUDA then `cudaErrorUnknown` is returned.

##### 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:

`cudaGraphicsResourceSetMapFlags`

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

##### Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the `count` Direct3D resource in `ppResources`.

This function provides the synchronization guarantee that any CUDA kernels issued before `cudaD3D10UnmapResources()` will complete before any Direct3D calls issued after `cudaD3D10UnmapResources()` begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries, then `cudaErrorInvalidResourceHandle` is returned. If any of `ppResources` are not presently mapped for access by CUDA then `cudaErrorUnknown` is returned.

**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:**

[cudaGraphicsUnmapResources](#)

**4.21.2.10 `cudaError_t cudaD3D10UnregisterResource (ID3D10Resource * pResource)`****Deprecated**

This function is deprecated as of Cuda 3.0.

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

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

**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:**

[cudaGraphicsUnregisterResource](#)

## 4.22 OpenGL Interoperability [DEPRECATED]

### Functions

- [cudaError\\_t cudaGLMapBufferObject](#) (void \*\*devPtr, GLuint bufObj)  
*Maps a buffer object for access by CUDA.*
- [cudaError\\_t cudaGLMapBufferObjectAsync](#) (void \*\*devPtr, GLuint bufObj, [cudaStream\\_t](#) stream)  
*Maps a buffer object for access by CUDA.*
- [cudaError\\_t cudaGLRegisterBufferObject](#) (GLuint bufObj)  
*Registers a buffer object for access by CUDA.*
- [cudaError\\_t cudaGLSetBufferObjectMapFlags](#) (GLuint bufObj, unsigned int flags)  
*Set usage flags for mapping an OpenGL buffer.*
- [cudaError\\_t cudaGLUnmapBufferObject](#) (GLuint bufObj)  
*Unmaps a buffer object for access by CUDA.*
- [cudaError\\_t cudaGLUnmapBufferObjectAsync](#) (GLuint bufObj, [cudaStream\\_t](#) stream)  
*Unmaps a buffer object for access by CUDA.*
- [cudaError\\_t cudaGLUnregisterBufferObject](#) (GLuint bufObj)  
*Unregisters a buffer object for access by CUDA.*

### 4.22.1 Detailed Description

This section describes deprecated OpenGL interoperability functionality.

### 4.22.2 Function Documentation

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

##### Deprecated

This function is deprecated as of Cuda 3.0.

Maps the buffer object of ID `bufObj` into the address space of CUDA and returns in `*devPtr` the base pointer of the resulting mapping. The buffer must have previously been registered by calling [cudaGLRegisterBufferObject\(\)](#). While a buffer is mapped by CUDA, any OpenGL operation which references the buffer will result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

All streams in the current thread are synchronized with the current GL context.

##### 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:**

[cudaGraphicsMapResources](#)

#### 4.22.2.2 `cudaError_t cudaGLMapBufferObjectAsync (void ** devPtr, GLuint bufObj, cudaStream_t stream)`

**Deprecated**

This function is deprecated as of Cuda 3.0.

Maps the buffer object of ID `bufObj` into the address space of CUDA and returns in `*devPtr` the base pointer of the resulting mapping. The buffer must have previously been registered by calling [cudaGLRegisterBufferObject\(\)](#). While a buffer is mapped by CUDA, any OpenGL operation which references the buffer will result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

Stream `/p stream` is synchronized with the current GL context.

**Parameters:**

*devPtr* - Returned device pointer to CUDA object

*bufObj* - Buffer object ID to map

*stream* - Stream to synchronize

**Returns:**

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

**Note:**

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

**See also:**

[cudaGraphicsMapResources](#)

#### 4.22.2.3 `cudaError_t cudaGLRegisterBufferObject (GLuint bufObj)`

**Deprecated**

This function is deprecated as of Cuda 3.0.

Registers the buffer object of ID `bufObj` for access by CUDA. This function must be called before CUDA can map the buffer object. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

**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:**

[cudaGraphicsGLRegisterBuffer](#)

#### 4.22.2.4 `cudaError_t cudaGLSetBufferObjectMapFlags (GLuint bufObj, unsigned int flags)`

**Deprecated**

This function is deprecated as of Cuda 3.0.

Set flags for mapping the OpenGL buffer `bufObj`

Changes to flags will take effect the next time `bufObj` is mapped. The `flags` argument may be any of the following:

- [cudaGLMapFlagsNone](#): Specifies no hints about how this buffer will be used. It is therefore assumed that this buffer will be read from and written to by CUDA kernels. This is the default value.
- [cudaGLMapFlagsReadOnly](#): Specifies that CUDA kernels which access this buffer will not write to the buffer.
- [cudaGLMapFlagsWriteDiscard](#): Specifies that CUDA kernels which access this buffer will not read from the buffer and will write over the entire contents of the buffer, so none of the data previously stored in the buffer will be preserved.

If `bufObj` has not been registered for use with CUDA, then [cudaErrorInvalidResourceHandle](#) is returned. If `bufObj` is presently mapped for access by CUDA, then [cudaErrorUnknown](#) is returned.

**Parameters:**

*bufObj* - Registered buffer object to set flags for

*flags* - Parameters for buffer mapping

**Returns:**

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

**Note:**

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

**See also:**

[cudaGraphicsResourceSetMapFlags](#)

#### 4.22.2.5 `cudaError_t cudaGLUnmapBufferObject (GLuint bufObj)`

##### Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the buffer object of ID `bufObj` for access by CUDA. When a buffer is unmapped, the base address returned by `cudaGLMapBufferObject()` is invalid and subsequent references to the address result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

All streams in the current thread are synchronized with the current GL context.

##### 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:

[cudaGraphicsUnmapResources](#)

#### 4.22.2.6 `cudaError_t cudaGLUnmapBufferObjectAsync (GLuint bufObj, cudaStream_t stream)`

##### Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the buffer object of ID `bufObj` for access by CUDA. When a buffer is unmapped, the base address returned by `cudaGLMapBufferObject()` is invalid and subsequent references to the address result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

Stream `/p stream` is synchronized with the current GL context.

##### Parameters:

*bufObj* - Buffer object to unmap

*stream* - Stream to synchronize

##### Returns:

`cudaSuccess`, `cudaErrorInvalidDevicePointer`, `cudaErrorUnmapBufferObjectFailed`

##### Note:

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

##### See also:

[cudaGraphicsUnmapResources](#)

#### 4.22.2.7 `cudaError_t cudaGLUnregisterBufferObject (GLuint bufObj)`

##### Deprecated

This function is deprecated as of Cuda 3.0.

Unregisters the buffer object of ID `bufObj` for access by CUDA and releases any CUDA resources associated with the buffer. Once a buffer is unregistered, it may no longer be mapped by CUDA. The GL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

##### Parameters:

*bufObj* - Buffer object to unregister

##### Returns:

`cudaSuccess`

##### Note:

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

##### See also:

[cudaGraphicsUnregisterResource](#)

## 4.23 Data types used by CUDA Runtime

### Data Structures

- struct `cudaChannelFormatDesc`
- struct `cudaDeviceProp`
- struct `cudaExtent`
- struct `cudaFuncAttributes`
- struct `cudaMemcpy3DParms`
- struct `cudaPitchedPtr`
- struct `cudaPos`
- struct `surfaceReference`
- struct `textureReference`

### Enumerations

- enum `cudaSurfaceBoundaryMode` {  
    `cudaBoundaryModeZero` = 0,  
    `cudaBoundaryModeClamp` = 1,  
    `cudaBoundaryModeTrap` = 2 }
- enum `cudaSurfaceFormatMode` {  
    `cudaFormatModeForced` = 0,  
    `cudaFormatModeAuto` = 1 }
- enum `cudaTextureAddressMode` {  
    `cudaAddressModeWrap` = 0,  
    `cudaAddressModeClamp` = 1,  
    `cudaAddressModeMirror` = 2,  
    `cudaAddressModeBorder` = 3 }
- enum `cudaTextureFilterMode` {  
    `cudaFilterModePoint` = 0,  
    `cudaFilterModeLinear` = 1 }
- enum `cudaTextureReadMode` {  
    `cudaReadModeElementType` = 0,  
    `cudaReadModeNormalizedFloat` = 1 }

### Data types used by CUDA Runtime

Data types used by CUDA Runtime

#### Author:

NVIDIA Corporation

- enum `cudaChannelFormatKind` {  
    `cudaChannelFormatKindSigned` = 0,  
    `cudaChannelFormatKindUnsigned` = 1,  
    `cudaChannelFormatKindFloat` = 2,  
    `cudaChannelFormatKindNone` = 3 }

- enum `cudaComputeMode` {  
    `cudaComputeModeDefault` = 0,  
    `cudaComputeModeExclusive` = 1,  
    `cudaComputeModeProhibited` = 2 }
- enum `cudaError` {  
    `cudaSuccess` = 0,  
    `cudaErrorMissingConfiguration` = 1,  
    `cudaErrorMemoryAllocation` = 2,  
    `cudaErrorInitializationError` = 3,  
    `cudaErrorLaunchFailure` = 4,  
    `cudaErrorPriorLaunchFailure` = 5,  
    `cudaErrorLaunchTimeout` = 6,  
    `cudaErrorLaunchOutOfResources` = 7,  
    `cudaErrorInvalidDeviceFunction` = 8,  
    `cudaErrorInvalidConfiguration` = 9,  
    `cudaErrorInvalidDevice` = 10,  
    `cudaErrorInvalidValue` = 11,  
    `cudaErrorInvalidPitchValue` = 12,  
    `cudaErrorInvalidSymbol` = 13,  
    `cudaErrorMapBufferObjectFailed` = 14,  
    `cudaErrorUnmapBufferObjectFailed` = 15,  
    `cudaErrorInvalidHostPointer` = 16,  
    `cudaErrorInvalidDevicePointer` = 17,  
    `cudaErrorInvalidTexture` = 18,  
    `cudaErrorInvalidTextureBinding` = 19,  
    `cudaErrorInvalidChannelDescriptor` = 20,  
    `cudaErrorInvalidMemcpyDirection` = 21,  
    `cudaErrorAddressOfConstant` = 22,  
    `cudaErrorTextureFetchFailed` = 23,  
    `cudaErrorTextureNotBound` = 24,  
    `cudaErrorSynchronizationError` = 25,  
    `cudaErrorInvalidFilterSetting` = 26,  
    `cudaErrorInvalidNormSetting` = 27,  
    `cudaErrorMixedDeviceExecution` = 28,  
    `cudaErrorCudartUnloading` = 29,  
    `cudaErrorUnknown` = 30,  
    `cudaErrorNotYetImplemented` = 31,  
    `cudaErrorMemoryValueTooLarge` = 32,  
    `cudaErrorInvalidResourceHandle` = 33,  
    `cudaErrorNotReady` = 34,  
    `cudaErrorInsufficientDriver` = 35,

```
cudaErrorSetOnActiveProcess = 36,  
cudaErrorInvalidSurface = 37,  
cudaErrorNoDevice = 38,  
cudaErrorECCUncorrectable = 39,  
cudaErrorSharedObjectSymbolNotFound = 40,  
cudaErrorSharedObjectInitFailed = 41,  
cudaErrorUnsupportedLimit = 42,  
cudaErrorDuplicateVariableName = 43,  
cudaErrorDuplicateTextureName = 44,  
cudaErrorDuplicateSurfaceName = 45,  
cudaErrorDevicesUnavailable = 46,  
cudaErrorInvalidKernelImage = 47,  
cudaErrorNoKernelImageForDevice = 48,  
cudaErrorIncompatibleDriverContext = 49,  
cudaErrorStartupFailure = 0x7f,  
cudaErrorApiFailureBase = 10000 }  
• enum cudaFuncCache {  
    cudaFuncCachePreferNone = 0,  
    cudaFuncCachePreferShared = 1,  
    cudaFuncCachePreferL1 = 2 }  
• enum cudaGraphicsCubeFace {  
    cudaGraphicsCubeFacePositiveX = 0x00,  
    cudaGraphicsCubeFaceNegativeX = 0x01,  
    cudaGraphicsCubeFacePositiveY = 0x02,  
    cudaGraphicsCubeFaceNegativeY = 0x03,  
    cudaGraphicsCubeFacePositiveZ = 0x04,  
    cudaGraphicsCubeFaceNegativeZ = 0x05 }  
• enum cudaGraphicsMapFlags {  
    cudaGraphicsMapFlagsNone = 0,  
    cudaGraphicsMapFlagsReadOnly = 1,  
    cudaGraphicsMapFlagsWriteDiscard = 2 }  
• enum cudaGraphicsRegisterFlags { cudaGraphicsRegisterFlagsNone = 0 }  
• enum cudaLimit {  
    cudaLimitStackSize = 0x00,  
    cudaLimitPrintfFifoSize = 0x01,  
    cudaLimitMallocHeapSize = 0x02 }  
• enum cudaMemcpyKind {  
    cudaMemcpyHostToHost = 0,  
    cudaMemcpyHostToDevice = 1,  
    cudaMemcpyDeviceToHost = 2,  
    cudaMemcpyDeviceToDevice = 3 }  
• typedef enum cudaError cudaError_t
```

- typedef struct CUevent\_st \* [cudaEvent\\_t](#)
- typedef struct cudaGraphicsResource \* [cudaGraphicsResource\\_t](#)
- typedef struct CUstream\_st \* [cudaStream\\_t](#)
- typedef struct CUuuid\_st [cudaUUID\\_t](#)
- #define [cudaArrayDefault](#) 0x00
- #define [cudaArraySurfaceLoadStore](#) 0x02
- #define [cudaDeviceBlockingSync](#) 4
- #define [cudaDeviceLmemResizeToMax](#) 16
- #define [cudaDeviceMapHost](#) 8
- #define [cudaDeviceMask](#) 0x1f
- #define [cudaDevicePropDontCare](#)
- #define [cudaDeviceScheduleAuto](#) 0
- #define [cudaDeviceScheduleSpin](#) 1
- #define [cudaDeviceScheduleYield](#) 2
- #define [cudaEventBlockingSync](#) 1
- #define [cudaEventDefault](#) 0
- #define [cudaEventDisableTiming](#) 2
- #define [cudaHostAllocDefault](#) 0
- #define [cudaHostAllocMapped](#) 2
- #define [cudaHostAllocPortable](#) 1
- #define [cudaHostAllocWriteCombined](#) 4

### 4.23.1 Define Documentation

#### 4.23.1.1 #define [cudaArrayDefault](#) 0x00

Default CUDA array allocation flag

#### 4.23.1.2 #define [cudaArraySurfaceLoadStore](#) 0x02

Must be set in `cudaMallocArray` in order to bind surfaces to the CUDA array

#### 4.23.1.3 #define [cudaDeviceBlockingSync](#) 4

Device flag - Use blocking synchronization

#### 4.23.1.4 #define [cudaDeviceLmemResizeToMax](#) 16

Device flag - Keep local memory allocation after launch

#### 4.23.1.5 #define [cudaDeviceMapHost](#) 8

Device flag - Support mapped pinned allocations

#### 4.23.1.6 #define [cudaDeviceMask](#) 0x1f

Device flags mask

**4.23.1.7 #define cudaDevicePropDontCare**

Empty device properties

**4.23.1.8 #define cudaDeviceScheduleAuto 0**

Device flag - Automatic scheduling

**4.23.1.9 #define cudaDeviceScheduleSpin 1**

Device flag - Spin default scheduling

**4.23.1.10 #define cudaDeviceScheduleYield 2**

Device flag - Yield default scheduling

**4.23.1.11 #define cudaEventBlockingSync 1**

Event uses blocking synchronization

**4.23.1.12 #define cudaEventDefault 0**

Default event flag

**4.23.1.13 #define cudaEventDisableTiming 2**

Event will not record timing data

**4.23.1.14 #define cudaHostAllocDefault 0**

Default page-locked allocation flag

**4.23.1.15 #define cudaHostAllocMapped 2**

Map allocation into device space

**4.23.1.16 #define cudaHostAllocPortable 1**

Pinned memory accessible by all CUDA contexts

**4.23.1.17 #define cudaHostAllocWriteCombined 4**

Write-combined memory

## 4.23.2 Typedef Documentation

### 4.23.2.1 typedef enum cudaError\_t cudaError\_t

CUDA Error types

### 4.23.2.2 typedef struct CUevent\_st\* cudaEvent\_t

CUDA event types

### 4.23.2.3 typedef struct cudaGraphicsResource\* cudaGraphicsResource\_t

CUDA graphics resource types

### 4.23.2.4 typedef struct CUstream\_st\* cudaStream\_t

CUDA stream

### 4.23.2.5 typedef struct CUuuid\_st cudaUUID\_t

CUDA UUID types

## 4.23.3 Enumeration Type Documentation

### 4.23.3.1 enum cudaChannelFormatKind

Channel format kind

**Enumerator:**

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

### 4.23.3.2 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)

### 4.23.3.3 enum cudaError

CUDA error types

#### Enumerator:

**cudaSuccess** The API call returned with no errors. In the case of query calls, this can also mean that the operation being queried is complete (see [cudaEventQuery\(\)](#) and [cudaStreamQuery\(\)](#)).

**cudaErrorMissingConfiguration** The device function being invoked (usually via [cudaLaunch\(\)](#)) was not previously configured via the [cudaConfigureCall\(\)](#) function.

**cudaErrorMemoryAllocation** The API call failed because it was unable to allocate enough memory to perform the requested operation.

**cudaErrorInitializationError** The API call failed because the CUDA driver and runtime could not be initialized.

**cudaErrorLaunchFailure** An exception occurred on the device while executing a kernel. Common causes include dereferencing an invalid device pointer and accessing out of bounds shared memory. The device cannot be used until [cudaThreadExit\(\)](#) is called. All existing device memory allocations are invalid and must be reconstructed if the program is to continue using CUDA.

**cudaErrorPriorLaunchFailure** This indicated that a previous kernel launch failed. This was previously used for device emulation of kernel launches.

#### Deprecated

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

**cudaErrorLaunchTimeout** This indicates that the device kernel took too long to execute. This can only occur if timeouts are enabled - see the device property [kernelExecTimeoutEnabled](#) for more information. The device cannot be used until [cudaThreadExit\(\)](#) is called. All existing device memory allocations are invalid and must be reconstructed if the program is to continue using CUDA.

**cudaErrorLaunchOutOfResources** This indicates that a launch did not occur because it did not have appropriate resources. Although this error is similar to [cudaErrorInvalidConfiguration](#), this error usually indicates that the user has attempted to pass too many arguments to the device kernel, or the kernel launch specifies too many threads for the kernel's register count.

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

**cudaErrorInvalidConfiguration** This indicates that a kernel launch is requesting resources that can never be satisfied by the current device. Requesting more shared memory per block than the device supports will trigger this error, as will requesting too many threads or blocks. See [cudaDeviceProp](#) for more device limitations.

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

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

**cudaErrorInvalidPitchValue** This indicates that one or more of the pitch-related parameters passed to the API call is not within the acceptable range for pitch.

**cudaErrorInvalidSymbol** This indicates that the symbol name/identifier passed to the API call is not a valid name or identifier.

**cudaErrorMapBufferObjectFailed** This indicates that the buffer object could not be mapped.

**cudaErrorUnmapBufferObjectFailed** This indicates that the buffer object could not be unmapped.

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

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

***cudaErrorInvalidTexture*** This indicates that the texture passed to the API call is not a valid texture.

***cudaErrorInvalidTextureBinding*** This indicates that the texture binding is not valid. This occurs if you call [cudaGetTextureAlignmentOffset\(\)](#) with an unbound texture.

***cudaErrorInvalidChannelDescriptor*** This indicates that the channel descriptor passed to the API call is not valid. This occurs if the format is not one of the formats specified by [cudaChannelFormatKind](#), or if one of the dimensions is invalid.

***cudaErrorInvalidMemcpyDirection*** This indicates that the direction of the memcpy passed to the API call is not one of the types specified by [cudaMemcpyKind](#).

***cudaErrorAddressOfConstant*** This indicated that the user has taken the address of a constant variable, which was forbidden up until the CUDA 3.1 release.

#### Deprecated

This error return is deprecated as of CUDA 3.1. Variables in constant memory may now have their address taken by the runtime via [cudaGetSymbolAddress\(\)](#).

***cudaErrorTextureFetchFailed*** This indicated that a texture fetch was not able to be performed. This was previously used for device emulation of texture operations.

#### Deprecated

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

***cudaErrorTextureNotBound*** This indicated that a texture was not bound for access. This was previously used for device emulation of texture operations.

#### Deprecated

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

***cudaErrorSynchronizationError*** This indicated that a synchronization operation had failed. This was previously used for some device emulation functions.

#### Deprecated

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

***cudaErrorInvalidFilterSetting*** This indicates that a non-float texture was being accessed with linear filtering. This is not supported by CUDA.

***cudaErrorInvalidNormSetting*** This indicates that an attempt was made to read a non-float texture as a normalized float. This is not supported by CUDA.

***cudaErrorMixedDeviceExecution*** Mixing of device and device emulation code was not allowed.

#### Deprecated

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

***cudaErrorCudartUnloading*** This indicated an issue with calling API functions during the unload process of the CUDA runtime in prior releases.

#### Deprecated

This error return is deprecated as of CUDA 3.2.

***cudaErrorUnknown*** This indicates that an unknown internal error has occurred.

***cudaErrorNotYetImplemented*** This indicates that the API call is not yet implemented. Production releases of CUDA will never return this error.

***cudaErrorMemoryValueTooLarge*** This indicated that an emulated device pointer exceeded the 32-bit address range.

**Deprecated**

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

***cudaErrorInvalidResourceHandle*** This indicates that a resource handle passed to the API call was not valid. Resource handles are opaque types like [cudaStream\\_t](#) and [cudaEvent\\_t](#).

***cudaErrorNotReady*** This indicates that asynchronous operations issued previously have not completed yet. This result is not actually an error, but must be indicated differently than [cudaSuccess](#) (which indicates completion). Calls that may return this value include [cudaEventQuery\(\)](#) and [cudaStreamQuery\(\)](#).

***cudaErrorInsufficientDriver*** This indicates that the installed NVIDIA CUDA driver is older than the CUDA runtime library. This is not a supported configuration. Users should install an updated NVIDIA display driver to allow the application to run.

***cudaErrorSetOnActiveProcess*** This indicates that the user has called [cudaSetDevice\(\)](#), [cudaSetValidDevices\(\)](#), [cudaSetDeviceFlags\(\)](#), [cudaD3D9SetDirect3DDevice\(\)](#), [cudaD3D10SetDirect3DDevice](#), [cudaD3D11SetDirect3DDevice\(\)](#), \* or [cudaVDPAUSetVDPAUDevice\(\)](#) after initializing the CUDA runtime by calling non-device management operations (allocating memory and launching kernels are examples of non-device management operations). This error can also be returned if using runtime/driver interoperability and there is an existing [CUcontext](#) active on the host thread.

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

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

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

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

***cudaErrorSharedObjectInitFailed*** This indicates that initialization of a shared object failed.

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

***cudaErrorDuplicateVariableName*** This indicates that multiple global or constant variables (across separate CUDA source files in the application) share the same string name.

***cudaErrorDuplicateTextureName*** This indicates that multiple textures (across separate CUDA source files in the application) share the same string name.

***cudaErrorDuplicateSurfaceName*** This indicates that multiple surfaces (across separate CUDA source files in the application) share the same string name.

***cudaErrorDevicesUnavailable*** This indicates that all CUDA devices are busy or unavailable at the current time. Devices are often busy/unavailable due to use of [cudaComputeModeExclusive](#) or [cudaComputeModeProhibited](#). They can also be unavailable due to memory constraints on a device that already has active CUDA work being performed.

***cudaErrorInvalidKernelImage*** This indicates that the device kernel image is invalid.

***cudaErrorNoKernelImageForDevice*** This indicates that there is no kernel image available that is suitable for the device. This can occur when a user specifies code generation options for a particular CUDA source file that do not include the corresponding device configuration.

***cudaErrorIncompatibleDriverContext*** This indicates that the current context is not compatible with this version of the CUDA Runtime. This can only occur if you are using CUDA Runtime/Driver interoperability and have created an existing Driver context using an older API. Please see [Interactions with the CUDA Driver API](#) for more information.

*cudaErrorStartupFailure* This indicates an internal startup failure in the CUDA runtime.

*cudaErrorApiFailureBase* Any unhandled CUDA driver error is added to this value and returned via the runtime. Production releases of CUDA should not return such errors.

#### 4.23.3.4 enum cudaFuncCache

CUDA function cache configurations

**Enumerator:**

*cudaFuncCachePreferNone* Default function cache configuration, no preference

*cudaFuncCachePreferShared* Prefer larger shared memory and smaller L1 cache

*cudaFuncCachePreferL1* Prefer larger L1 cache and smaller shared memory

#### 4.23.3.5 enum cudaGraphicsCubeFace

CUDA graphics interop array indices for cube maps

**Enumerator:**

*cudaGraphicsCubeFacePositiveX* Positive X face of cubemap

*cudaGraphicsCubeFaceNegativeX* Negative X face of cubemap

*cudaGraphicsCubeFacePositiveY* Positive Y face of cubemap

*cudaGraphicsCubeFaceNegativeY* Negative Y face of cubemap

*cudaGraphicsCubeFacePositiveZ* Positive Z face of cubemap

*cudaGraphicsCubeFaceNegativeZ* Negative Z face of cubemap

#### 4.23.3.6 enum cudaGraphicsMapFlags

CUDA graphics interop map flags

**Enumerator:**

*cudaGraphicsMapFlagsNone* Default; Assume resource can be read/written

*cudaGraphicsMapFlagsReadOnly* CUDA will not write to this resource

*cudaGraphicsMapFlagsWriteDiscard* CUDA will only write to and will not read from this resource

#### 4.23.3.7 enum cudaGraphicsRegisterFlags

CUDA graphics interop register flags

**Enumerator:**

*cudaGraphicsRegisterFlagsNone* Default

#### 4.23.3.8 enum cudaLimit

CUDA Limits

**Enumerator:**

*cudaLimitStackSize* GPU thread stack size  
*cudaLimitPrintfFifoSize* GPU printf FIFO size  
*cudaLimitMallocHeapSize* GPU malloc heap size

#### 4.23.3.9 enum cudaMemcpyKind

CUDA memory copy types

**Enumerator:**

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

#### 4.23.3.10 enum cudaSurfaceBoundaryMode

CUDA Surface boundary modes

**Enumerator:**

*cudaBoundaryModeZero* Zero boundary mode  
*cudaBoundaryModeClamp* Clamp boundary mode  
*cudaBoundaryModeTrap* Trap boundary mode

#### 4.23.3.11 enum cudaSurfaceFormatMode

CUDA Surface format modes

**Enumerator:**

*cudaFormatModeForced* Forced format mode  
*cudaFormatModeAuto* Auto format mode

#### 4.23.3.12 enum cudaTextureAddressMode

CUDA texture address modes

**Enumerator:**

*cudaAddressModeWrap* Wrapping address mode  
*cudaAddressModeClamp* Clamp to edge address mode  
*cudaAddressModeMirror* Mirror address mode  
*cudaAddressModeBorder* Border address mode

#### 4.23.3.13 enum `cudaTextureFilterMode`

CUDA texture filter modes

**Enumerator:**

*`cudaFilterModePoint`* Point filter mode

*`cudaFilterModeLinear`* Linear filter mode

#### 4.23.3.14 enum `cudaTextureReadMode`

CUDA texture read modes

**Enumerator:**

*`cudaReadModeElementType`* Read texture as specified element type

*`cudaReadModeNormalizedFloat`* Read texture as normalized float

## 4.24 CUDA Driver API

### Modules

- [Data types used by CUDA driver](#)
- [Initialization](#)
- [Version Management](#)
- [Device Management](#)
- [Context Management](#)
- [Module Management](#)
- [Memory Management](#)
- [Stream Management](#)
- [Event Management](#)
- [Execution Control](#)
- [Texture Reference Management](#)
- [Surface Reference Management](#)
- [Graphics Interoperability](#)
- [OpenGL Interoperability](#)
- [Direct3D 9 Interoperability](#)
- [Direct3D 10 Interoperability](#)
- [Direct3D 11 Interoperability](#)
- [VDPAU Interoperability](#)

### 4.24.1 Detailed Description

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

## 4.25 Data types used by CUDA driver

### Data Structures

- struct [CUDA\\_ARRAY3D\\_DESCRIPTOR\\_st](#)
- struct [CUDA\\_ARRAY\\_DESCRIPTOR\\_st](#)
- struct [CUDA\\_MEMCPY2D\\_st](#)
- struct [CUDA\\_MEMCPY3D\\_st](#)
- struct [CUdevprop\\_st](#)

### Defines

- #define [CU\\_MEMHOSTALLOC\\_DEVICEMAP](#) 0x02
- #define [CU\\_MEMHOSTALLOC\\_PORTABLE](#) 0x01
- #define [CU\\_MEMHOSTALLOC\\_WRITECOMBINED](#) 0x04
- #define [CU\\_PARAM\\_TR\\_DEFAULT](#) -1
- #define [CU\\_TRSA\\_OVERRIDE\\_FORMAT](#) 0x01
- #define [CU\\_TRSF\\_NORMALIZED\\_COORDINATES](#) 0x02
- #define [CU\\_TRSF\\_READ\\_AS\\_INTEGER](#) 0x01
- #define [CU\\_TRSF\\_SRGB](#) 0x10
- #define [CUDA\\_ARRAY3D\\_2DARRAY](#) 0x01
- #define [CUDA\\_ARRAY3D\\_SURFACE\\_LDST](#) 0x02
- #define [CUDA\\_VERSION](#) 3020

### Typedefs

- typedef enum [CUaddress\\_mode\\_enum](#) [CUaddress\\_mode](#)
- typedef struct [CUarray\\_st](#) \* [CUarray](#)
- typedef enum [CUarray\\_cubemap\\_face\\_enum](#) [CUarray\\_cubemap\\_face](#)
- typedef enum [CUarray\\_format\\_enum](#) [CUarray\\_format](#)
- typedef enum [CUcomputemode\\_enum](#) [CUcomputemode](#)
- typedef struct [CUctx\\_st](#) \* [CUcontext](#)
- typedef enum [CUctx\\_flags\\_enum](#) [CUctx\\_flags](#)
- typedef struct [CUDA\\_ARRAY3D\\_DESCRIPTOR\\_st](#) [CUDA\\_ARRAY3D\\_DESCRIPTOR](#)
- typedef struct [CUDA\\_ARRAY\\_DESCRIPTOR\\_st](#) [CUDA\\_ARRAY\\_DESCRIPTOR](#)
- typedef struct [CUDA\\_MEMCPY2D\\_st](#) [CUDA\\_MEMCPY2D](#)
- typedef struct [CUDA\\_MEMCPY3D\\_st](#) [CUDA\\_MEMCPY3D](#)
- typedef int [CUdevice](#)
- typedef enum [CUdevice\\_attribute\\_enum](#) [CUdevice\\_attribute](#)
- typedef unsigned int [CUdeviceptr](#)
- typedef struct [CUdevprop\\_st](#) [CUdevprop](#)
- typedef struct [CUevent\\_st](#) \* [CUevent](#)
- typedef enum [CUevent\\_flags\\_enum](#) [CUevent\\_flags](#)
- typedef enum [CUfilter\\_mode\\_enum](#) [CUfilter\\_mode](#)
- typedef enum [CUfunc\\_cache\\_enum](#) [CUfunc\\_cache](#)
- typedef struct [CUfunc\\_st](#) \* [CUfunction](#)
- typedef enum [CUfunction\\_attribute\\_enum](#) [CUfunction\\_attribute](#)
- typedef enum [CUgraphicsMapResourceFlags\\_enum](#) [CUgraphicsMapResourceFlags](#)
- typedef enum [CUgraphicsRegisterFlags\\_enum](#) [CUgraphicsRegisterFlags](#)

- typedef struct CUgraphicsResource\_st \* CUgraphicsResource
- typedef enum CUjit\_fallback\_enum CUjit\_fallback
- typedef enum CUjit\_option\_enum CUjit\_option
- typedef enum CUjit\_target\_enum CUjit\_target
- typedef enum CULimit\_enum CULimit
- typedef enum CUmemorytype\_enum CUmemorytype
- typedef struct CUmod\_st \* CUmodule
- typedef enum cudaError\_enum CUresult
- typedef struct CUstream\_st \* CUstream
- typedef struct CUsurfref\_st \* CUsurfref
- typedef struct CUtexref\_st \* CUtexref

## Enumerations

- enum CUaddress\_mode\_enum {  
     CU\_TR\_ADDRESS\_MODE\_WRAP = 0,  
     CU\_TR\_ADDRESS\_MODE\_CLAMP = 1,  
     CU\_TR\_ADDRESS\_MODE\_MIRROR = 2,  
     CU\_TR\_ADDRESS\_MODE\_BORDER = 3 }
- enum CUarray\_cubemap\_face\_enum {  
     CU\_CUBEMAP\_FACE\_POSITIVE\_X = 0x00,  
     CU\_CUBEMAP\_FACE\_NEGATIVE\_X = 0x01,  
     CU\_CUBEMAP\_FACE\_POSITIVE\_Y = 0x02,  
     CU\_CUBEMAP\_FACE\_NEGATIVE\_Y = 0x03,  
     CU\_CUBEMAP\_FACE\_POSITIVE\_Z = 0x04,  
     CU\_CUBEMAP\_FACE\_NEGATIVE\_Z = 0x05 }
- 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 }
- enum CUcomputemode\_enum {  
     CU\_COMPUTEMODE\_DEFAULT = 0,  
     CU\_COMPUTEMODE\_EXCLUSIVE = 1,  
     CU\_COMPUTEMODE\_PROHIBITED = 2 }
- enum CUctx\_flags\_enum {  
     CU\_CTX\_SCHED\_AUTO = 0,  
     CU\_CTX\_SCHED\_SPIN = 1,  
     CU\_CTX\_SCHED\_YIELD = 2 ,  
     CU\_CTX\_BLOCKING\_SYNC = 4,  
     CU\_CTX\_MAP\_HOST = 8,  
     CU\_CTX\_LMEM\_RESIZE\_TO\_MAX = 16 }

- enum `cudaError_enum` {
  - `CUDA_SUCCESS` = 0,
  - `CUDA_ERROR_INVALID_VALUE` = 1,
  - `CUDA_ERROR_OUT_OF_MEMORY` = 2,
  - `CUDA_ERROR_NOT_INITIALIZED` = 3,
  - `CUDA_ERROR_DEINITIALIZED` = 4,
  - `CUDA_ERROR_NO_DEVICE` = 100,
  - `CUDA_ERROR_INVALID_DEVICE` = 101,
  - `CUDA_ERROR_INVALID_IMAGE` = 200,
  - `CUDA_ERROR_INVALID_CONTEXT` = 201,
  - `CUDA_ERROR_CONTEXT_ALREADY_CURRENT` = 202,
  - `CUDA_ERROR_MAP_FAILED` = 205,
  - `CUDA_ERROR_UNMAP_FAILED` = 206,
  - `CUDA_ERROR_ARRAY_IS_MAPPED` = 207,
  - `CUDA_ERROR_ALREADY_MAPPED` = 208,
  - `CUDA_ERROR_NO_BINARY_FOR_GPU` = 209,
  - `CUDA_ERROR_ALREADY_ACQUIRED` = 210,
  - `CUDA_ERROR_NOT_MAPPED` = 211,
  - `CUDA_ERROR_NOT_MAPPED_AS_ARRAY` = 212,
  - `CUDA_ERROR_NOT_MAPPED_AS_POINTER` = 213,
  - `CUDA_ERROR_ECC_UNCORRECTABLE` = 214,
  - `CUDA_ERROR_UNSUPPORTED_LIMIT` = 215,
  - `CUDA_ERROR_INVALID_SOURCE` = 300,
  - `CUDA_ERROR_FILE_NOT_FOUND` = 301,
  - `CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND` = 302,
  - `CUDA_ERROR_SHARED_OBJECT_INIT_FAILED` = 303,
  - `CUDA_ERROR_OPERATING_SYSTEM` = 304,
  - `CUDA_ERROR_INVALID_HANDLE` = 400,
  - `CUDA_ERROR_NOT_FOUND` = 500,
  - `CUDA_ERROR_NOT_READY` = 600,
  - `CUDA_ERROR_LAUNCH_FAILED` = 700,
  - `CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES` = 701,
  - `CUDA_ERROR_LAUNCH_TIMEOUT` = 702,
  - `CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING` = 703,
  - `CUDA_ERROR_UNKNOWN` = 999 }
- enum `CUdevice_attribute_enum` {
  - `CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK` = 1,
  - `CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X` = 2,
  - `CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y` = 3,
  - `CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z` = 4,
  - `CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X` = 5,

```
CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y = 6,  
CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z = 7,  
CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK = 8,  
CU_DEVICE_ATTRIBUTE_SHARED_MEMORY_PER_BLOCK = 8,  
CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY = 9,  
CU_DEVICE_ATTRIBUTE_WARP_SIZE = 10,  
CU_DEVICE_ATTRIBUTE_MAX_PITCH = 11,  
CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK = 12,  
CU_DEVICE_ATTRIBUTE_REGISTERS_PER_BLOCK = 12,  
CU_DEVICE_ATTRIBUTE_CLOCK_RATE = 13,  
CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT = 14,  
CU_DEVICE_ATTRIBUTE_GPU_OVERLAP = 15,  
CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT = 16,  
CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT = 17,  
CU_DEVICE_ATTRIBUTE_INTEGRATED = 18,  
CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY = 19,  
CU_DEVICE_ATTRIBUTE_COMPUTE_MODE = 20,  
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH = 21,  
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_WIDTH = 22,  
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_HEIGHT = 23,  
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH = 24,  
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT = 25,  
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH = 26,  
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_WIDTH = 27,  
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_HEIGHT = 28,  
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_NUMSLICES = 29,  
CU_DEVICE_ATTRIBUTE_SURFACE_ALIGNMENT = 30,  
CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS = 31,  
CU_DEVICE_ATTRIBUTE_ECC_ENABLED = 32,  
CU_DEVICE_ATTRIBUTE_PCI_BUS_ID = 33,  
CU_DEVICE_ATTRIBUTE_PCI_DEVICE_ID = 34,  
CU_DEVICE_ATTRIBUTE_TCC_DRIVER = 35 }  
• enum CUevent_flags_enum {  
    CU_EVENT_DEFAULT = 0,  
    CU_EVENT_BLOCKING_SYNC = 1,  
    CU_EVENT_DISABLE_TIMING = 2 }  
• enum CUfilter_mode_enum {  
    CU_TR_FILTER_MODE_POINT = 0,  
    CU_TR_FILTER_MODE_LINEAR = 1 }
```

- enum CUfunc\_cache\_enum {  
CU\_FUNC\_CACHE\_PREFER\_NONE = 0x00,  
CU\_FUNC\_CACHE\_PREFER\_SHARED = 0x01,  
CU\_FUNC\_CACHE\_PREFER\_L1 = 0x02 }
- enum CUfunction\_attribute\_enum {  
CU\_FUNC\_ATTRIBUTE\_MAX\_THREADS\_PER\_BLOCK = 0,  
CU\_FUNC\_ATTRIBUTE\_SHARED\_SIZE\_BYTES = 1,  
CU\_FUNC\_ATTRIBUTE\_CONST\_SIZE\_BYTES = 2,  
CU\_FUNC\_ATTRIBUTE\_LOCAL\_SIZE\_BYTES = 3,  
CU\_FUNC\_ATTRIBUTE\_NUM\_REGS = 4,  
CU\_FUNC\_ATTRIBUTE\_PTX\_VERSION = 5,  
CU\_FUNC\_ATTRIBUTE\_BINARY\_VERSION = 6 }
- enum CUgraphicsMapResourceFlags\_enum
- enum CUgraphicsRegisterFlags\_enum
- enum CUjit\_fallback\_enum {  
CU\_PREFER\_PTX = 0,  
CU\_PREFER\_BINARY }
- enum CUjit\_option\_enum {  
CU\_JIT\_MAX\_REGISTERS = 0,  
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 = 0,  
CU\_TARGET\_COMPUTE\_11,  
CU\_TARGET\_COMPUTE\_12,  
CU\_TARGET\_COMPUTE\_13,  
CU\_TARGET\_COMPUTE\_20,  
CU\_TARGET\_COMPUTE\_21 }
- enum CUlimit\_enum {  
CU\_LIMIT\_STACK\_SIZE = 0x00,  
CU\_LIMIT\_PRINTF\_FIFO\_SIZE = 0x01,  
CU\_LIMIT\_MALLOC\_HEAP\_SIZE = 0x02 }
- enum CUmemorytype\_enum {  
CU\_MEMORYTYPE\_HOST = 0x01,  
CU\_MEMORYTYPE\_DEVICE = 0x02,  
CU\_MEMORYTYPE\_ARRAY = 0x03 }

## 4.25.1 Define Documentation

### 4.25.1.1 #define CU\_MEMHOSTALLOC\_DEVICEMAP 0x02

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

### 4.25.1.2 #define CU\_MEMHOSTALLOC\_PORTABLE 0x01

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

### 4.25.1.3 #define CU\_MEMHOSTALLOC\_WRITECOMBINED 0x04

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

### 4.25.1.4 #define CU\_PARAM\_TR\_DEFAULT -1

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

### 4.25.1.5 #define CU\_TRSA\_OVERRIDE\_FORMAT 0x01

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

### 4.25.1.6 #define CU\_TRSF\_NORMALIZED\_COORDINATES 0x02

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

### 4.25.1.7 #define CU\_TRSF\_READ\_AS\_INTEGER 0x01

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

### 4.25.1.8 #define CU\_TRSF\_SRGB 0x10

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

### 4.25.1.9 #define CUDA\_ARRAY3D\_2DARRAY 0x01

If set, the CUDA array contains an array of 2D slices and the Depth member of [CUDA\\_ARRAY3D\\_DESCRIPTOR](#) specifies the number of slices, not the depth of a 3D array.

### 4.25.1.10 #define CUDA\_ARRAY3D\_SURFACE\_LDST 0x02

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

#### 4.25.1.11 `#define CUDA_VERSION 3020`

CUDA API version number

### 4.25.2 Typedef Documentation

#### 4.25.2.1 `typedef enum CUaddress_mode_enum CUaddress_mode`

Texture reference addressing modes

#### 4.25.2.2 `typedef struct CUarray_st* CUarray`

CUDA array

#### 4.25.2.3 `typedef enum CUarray_cubemap_face_enum CUarray_cubemap_face`

Array indices for cube faces

#### 4.25.2.4 `typedef enum CUarray_format_enum CUarray_format`

Array formats

#### 4.25.2.5 `typedef enum CUcomputemode_enum CUcomputemode`

Compute Modes

#### 4.25.2.6 `typedef struct CUctx_st* CUcontext`

CUDA context

#### 4.25.2.7 `typedef enum CUctx_flags_enum CUctx_flags`

Context creation flags

#### 4.25.2.8 `typedef struct CUDA_ARRAY3D_DESCRIPTOR_st CUDA_ARRAY3D_DESCRIPTOR`

3D array descriptor

#### 4.25.2.9 `typedef struct CUDA_ARRAY_DESCRIPTOR_st CUDA_ARRAY_DESCRIPTOR`

Array descriptor

#### 4.25.2.10 `typedef struct CUDA_MEMCPY2D_st CUDA_MEMCPY2D`

2D memory copy parameters

**4.25.2.11 typedef struct CUDA\_MEMCPY3D\_st CUDA\_MEMCPY3D**

3D memory copy parameters

**4.25.2.12 typedef int CUdevice**

CUDA device

**4.25.2.13 typedef enum CUdevice\_attribute\_enum CUdevice\_attribute**

Device properties

**4.25.2.14 typedef unsigned int CUdeviceptr**

CUDA device pointer

**4.25.2.15 typedef struct CUdevprop\_st CUdevprop**

Legacy device properties

**4.25.2.16 typedef struct CUevent\_st\* CUevent**

CUDA event

**4.25.2.17 typedef enum CUevent\_flags\_enum CUevent\_flags**

Event creation flags

**4.25.2.18 typedef enum CUfilter\_mode\_enum CUfilter\_mode**

Texture reference filtering modes

**4.25.2.19 typedef enum CUfunc\_cache\_enum CUfunc\_cache**

Function cache configurations

**4.25.2.20 typedef struct CUfunc\_st\* CUfunction**

CUDA function

**4.25.2.21 typedef enum CUfunction\_attribute\_enum CUfunction\_attribute**

Function properties

**4.25.2.22 typedef enum CUgraphicsMapResourceFlags\_enum CUgraphicsMapResourceFlags**

Flags for mapping and unmapping interop resources

**4.25.2.23 typedef enum CUgraphicsRegisterFlags\_enum CUgraphicsRegisterFlags**

Flags to register a graphics resource

**4.25.2.24 typedef struct CUgraphicsResource\_st\* CUgraphicsResource**

CUDA graphics interop resource

**4.25.2.25 typedef enum CUjit\_fallback\_enum CUjit\_fallback**

Cubin matching fallback strategies

**4.25.2.26 typedef enum CUjit\_option\_enum CUjit\_option**

Online compiler options

**4.25.2.27 typedef enum CUjit\_target\_enum CUjit\_target**

Online compilation targets

**4.25.2.28 typedef enum CUlimit\_enum CUlimit**

Limits

**4.25.2.29 typedef enum CUmemorytype\_enum CUmemorytype**

Memory types

**4.25.2.30 typedef struct CUmod\_st\* CUmodule**

CUDA module

**4.25.2.31 typedef enum cudaError\_enum CUresult**

Error codes

**4.25.2.32 typedef struct CUstream\_st\* CUstream**

CUDA stream

**4.25.2.33 typedef struct CUsurfref\_st\* CUsurfref**

CUDA surface reference

**4.25.2.34 typedef struct CUTexref\_st\* CUTexref**

CUDA texture reference

### 4.25.3 Enumeration Type Documentation

#### 4.25.3.1 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  
*CU\_TR\_ADDRESS\_MODE\_BORDER* Border address mode

#### 4.25.3.2 enum CUarray\_cubemap\_face\_enum

Array indices for cube faces

**Enumerator:**

*CU\_CUBEMAP\_FACE\_POSITIVE\_X* Positive X face of cubemap  
*CU\_CUBEMAP\_FACE\_NEGATIVE\_X* Negative X face of cubemap  
*CU\_CUBEMAP\_FACE\_POSITIVE\_Y* Positive Y face of cubemap  
*CU\_CUBEMAP\_FACE\_NEGATIVE\_Y* Negative Y face of cubemap  
*CU\_CUBEMAP\_FACE\_POSITIVE\_Z* Positive Z face of cubemap  
*CU\_CUBEMAP\_FACE\_NEGATIVE\_Z* Negative Z face of cubemap

#### 4.25.3.3 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

#### 4.25.3.4 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)

#### 4.25.3.5 enum CUctx\_flags\_enum

Context creation flags

##### Enumerator:

***CU\_CTX\_SCHED\_AUTO*** Automatic scheduling

***CU\_CTX\_SCHED\_SPIN*** Set spin as default scheduling

***CU\_CTX\_SCHED\_YIELD*** Set yield as default scheduling

***CU\_CTX\_BLOCKING\_SYNC*** Use blocking synchronization

***CU\_CTX\_MAP\_HOST*** Support mapped pinned allocations

***CU\_CTX\_LMEM\_RESIZE\_TO\_MAX*** Keep local memory allocation after launch

#### 4.25.3.6 enum cudaError\_enum

Error codes

##### Enumerator:

***CUDA\_SUCCESS*** The API call returned with no errors. In the case of query calls, this can also mean that the operation being queried is complete (see [cuEventQuery\(\)](#) and [cuStreamQuery\(\)](#)).

***CUDA\_ERROR\_INVALID\_VALUE*** This indicates that one or more of the parameters passed to the API call is not within an acceptable range of values.

***CUDA\_ERROR\_OUT\_OF\_MEMORY*** The API call failed because it was unable to allocate enough memory to perform the requested operation.

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

***CUDA\_ERROR\_DEINITIALIZED*** This indicates that the CUDA driver is in the process of shutting down.

***CUDA\_ERROR\_NO\_DEVICE*** This indicates that no CUDA-capable devices were detected by the installed CUDA driver.

***CUDA\_ERROR\_INVALID\_DEVICE*** This indicates that the device ordinal supplied by the user does not correspond to a valid CUDA device.

***CUDA\_ERROR\_INVALID\_IMAGE*** This indicates that the device kernel image is invalid. This can also indicate an invalid CUDA module.

***CUDA\_ERROR\_INVALID\_CONTEXT*** This most frequently indicates that there is no context bound to the current thread. This can also be returned if the context passed to an API call is not a valid handle (such as a context that has had [cuCtxDestroy\(\)](#) invoked on it). This can also be returned if a user mixes different API versions (i.e. 3010 context with 3020 API calls). See [cuCtxGetApiVersion\(\)](#) for more details.

***CUDA\_ERROR\_CONTEXT\_ALREADY\_CURRENT*** This indicated that the context being supplied as a parameter to the API call was already the active context.

##### Deprecated

This error return is deprecated as of CUDA 3.2. It is no longer an error to attempt to push the active context via [cuCtxPushCurrent\(\)](#).

- CUDA\_ERROR\_MAP\_FAILED*** This indicates that a map or register operation has failed.
- CUDA\_ERROR\_UNMAP\_FAILED*** This indicates that an unmap or unregister operation has failed.
- CUDA\_ERROR\_ARRAY\_IS\_MAPPED*** This indicates that the specified array is currently mapped and thus cannot be destroyed.
- CUDA\_ERROR\_ALREADY\_MAPPED*** This indicates that the resource is already mapped.
- CUDA\_ERROR\_NO\_BINARY\_FOR\_GPU*** This indicates that there is no kernel image available that is suitable for the device. This can occur when a user specifies code generation options for a particular CUDA source file that do not include the corresponding device configuration.
- CUDA\_ERROR\_ALREADY\_ACQUIRED*** This indicates that a resource has already been acquired.
- CUDA\_ERROR\_NOT\_MAPPED*** This indicates that a resource is not mapped.
- CUDA\_ERROR\_NOT\_MAPPED\_AS\_ARRAY*** This indicates that a mapped resource is not available for access as an array.
- CUDA\_ERROR\_NOT\_MAPPED\_AS\_POINTER*** This indicates that a mapped resource is not available for access as a pointer.
- CUDA\_ERROR\_ECC\_UNCORRECTABLE*** This indicates that an uncorrectable ECC error was detected during execution.
- CUDA\_ERROR\_UNSUPPORTED\_LIMIT*** This indicates that the [CUlimit](#) passed to the API call is not supported by the active device.
- CUDA\_ERROR\_INVALID\_SOURCE*** This indicates that the device kernel source is invalid.
- CUDA\_ERROR\_FILE\_NOT\_FOUND*** This indicates that the file specified was not found.
- CUDA\_ERROR\_SHARED\_OBJECT\_SYMBOL\_NOT\_FOUND*** This indicates that a link to a shared object failed to resolve.
- CUDA\_ERROR\_SHARED\_OBJECT\_INIT\_FAILED*** This indicates that initialization of a shared object failed.
- CUDA\_ERROR\_OPERATING\_SYSTEM*** This indicates that an OS call failed.
- CUDA\_ERROR\_INVALID\_HANDLE*** This indicates that a resource handle passed to the API call was not valid. Resource handles are opaque types like [CUstream](#) and [CUevent](#).
- CUDA\_ERROR\_NOT\_FOUND*** This indicates that a named symbol was not found. Examples of symbols are global/constant variable names, texture names, and surface names.
- CUDA\_ERROR\_NOT\_READY*** This indicates that asynchronous operations issued previously have not completed yet. This result is not actually an error, but must be indicated differently than [CUDA\\_SUCCESS](#) (which indicates completion). Calls that may return this value include [cuEventQuery\(\)](#) and [cuStreamQuery\(\)](#).
- CUDA\_ERROR\_LAUNCH\_FAILED*** An exception occurred on the device while executing a kernel. Common causes include dereferencing an invalid device pointer and accessing out of bounds shared memory. The context cannot be used, so it must be destroyed (and a new one should be created). All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.
- CUDA\_ERROR\_LAUNCH\_OUT\_OF\_RESOURCES*** This indicates that a launch did not occur because it did not have appropriate resources. This error usually indicates that the user has attempted to pass too many arguments to the device kernel, or the kernel launch specifies too many threads for the kernel's register count. Passing arguments of the wrong size (i.e. a 64-bit pointer when a 32-bit int is expected) is equivalent to passing too many arguments and can also result in this error.
- CUDA\_ERROR\_LAUNCH\_TIMEOUT*** This indicates that the device kernel took too long to execute. This can only occur if timeouts are enabled - see the device attribute [CU\\_DEVICE\\_ATTRIBUTE\\_KERNEL\\_EXEC\\_TIMEOUT](#) for more information. The context cannot be used (and must be destroyed similar to [CUDA\\_ERROR\\_LAUNCH\\_FAILED](#)). All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.

***CUDA\_ERROR\_LAUNCH\_INCOMPATIBLE\_TEXTURING*** This error indicates a kernel launch that uses an incompatible texturing mode.

***CUDA\_ERROR\_UNKNOWN*** This indicates that an unknown internal error has occurred.

#### 4.25.3.7 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)

***CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE1D\_WIDTH*** Maximum 1D texture width

***CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE2D\_WIDTH*** Maximum 2D texture width

***CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE2D\_HEIGHT*** Maximum 2D texture height

***CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE3D\_WIDTH*** Maximum 3D texture width

***CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE3D\_HEIGHT*** Maximum 3D texture height

*CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE3D\_DEPTH* Maximum 3D texture depth  
*CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE2D\_ARRAY\_WIDTH* Maximum texture array width  
*CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE2D\_ARRAY\_HEIGHT* Maximum texture array height  
*CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE2D\_ARRAY\_NUMSLICES* Maximum slices in a texture array  
*CU\_DEVICE\_ATTRIBUTE\_SURFACE\_ALIGNMENT* Alignment requirement for surfaces  
*CU\_DEVICE\_ATTRIBUTE\_CONCURRENT\_KERNELS* Device can possibly execute multiple kernels concurrently  
*CU\_DEVICE\_ATTRIBUTE\_ECC\_ENABLED* Device has ECC support enabled  
*CU\_DEVICE\_ATTRIBUTE\_PCI\_BUS\_ID* PCI bus ID of the device  
*CU\_DEVICE\_ATTRIBUTE\_PCI\_DEVICE\_ID* PCI device ID of the device  
*CU\_DEVICE\_ATTRIBUTE\_TCC\_DRIVER* Device is using TCC driver model

#### 4.25.3.8 enum CUevent\_flags\_enum

Event creation flags

##### Enumerator:

*CU\_EVENT\_DEFAULT* Default event flag  
*CU\_EVENT\_BLOCKING\_SYNC* Event uses blocking synchronization  
*CU\_EVENT\_DISABLE\_TIMING* Event will not record timing data

#### 4.25.3.9 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

#### 4.25.3.10 enum CUfunc\_cache\_enum

Function cache configurations

##### Enumerator:

*CU\_FUNC\_CACHE\_PREFER\_NONE* no preference for shared memory or L1 (default)  
*CU\_FUNC\_CACHE\_PREFER\_SHARED* prefer larger shared memory and smaller L1 cache  
*CU\_FUNC\_CACHE\_PREFER\_L1* prefer larger L1 cache and smaller shared memory

#### 4.25.3.11 enum CUfunction\_attribute\_enum

Function properties

##### Enumerator:

**CU\_FUNC\_ATTRIBUTE\_MAX\_THREADS\_PER\_BLOCK** The maximum number of threads per block, beyond which a launch of the function would fail. This number depends on both the function and the device on which the function is currently loaded.

**CU\_FUNC\_ATTRIBUTE\_SHARED\_SIZE\_BYTES** The size in bytes of statically-allocated shared memory required by this function. This does not include dynamically-allocated shared memory requested by the user at runtime.

**CU\_FUNC\_ATTRIBUTE\_CONST\_SIZE\_BYTES** The size in bytes of user-allocated constant memory required by this function.

**CU\_FUNC\_ATTRIBUTE\_LOCAL\_SIZE\_BYTES** The size in bytes of local memory used by each thread of this function.

**CU\_FUNC\_ATTRIBUTE\_NUM\_REGS** The number of registers used by each thread of this function.

**CU\_FUNC\_ATTRIBUTE\_PTX\_VERSION** The PTX virtual architecture version for which the function was compiled. This value is the major PTX version \* 10 + the minor PTX version, so a PTX version 1.3 function would return the value 13. Note that this may return the undefined value of 0 for cubins compiled prior to CUDA 3.0.

**CU\_FUNC\_ATTRIBUTE\_BINARY\_VERSION** The binary architecture version for which the function was compiled. This value is the major binary version \* 10 + the minor binary version, so a binary version 1.3 function would return the value 13. Note that this will return a value of 10 for legacy cubins that do not have a properly-encoded binary architecture version.

#### 4.25.3.12 enum CUgraphicsMapResourceFlags\_enum

Flags for mapping and unmapping interop resources

#### 4.25.3.13 enum CUgraphicsRegisterFlags\_enum

Flags to register a graphics resource

#### 4.25.3.14 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

#### 4.25.3.15 enum CUjit\_option\_enum

Online compiler options

##### Enumerator:

**CU\_JIT\_MAX\_REGISTERS** Max number of registers that a thread may use.

Option type: unsigned int

- 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.  
 Option type: unsigned int
- CU\_JIT\_WALL\_TIME*** Returns a float value in the option of the wall clock time, in milliseconds, spent creating the cubin  
 Option type: float
- CU\_JIT\_INFO\_LOG\_BUFFER*** Pointer to a buffer in which to print any log messages from PTXAS that are informational in nature (the buffer size is specified via option [CU\\_JIT\\_INFO\\_LOG\\_BUFFER\\_SIZE\\_BYTES](#))  
 Option type: char\*
- 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  
 Option type: unsigned int
- CU\_JIT\_ERROR\_LOG\_BUFFER*** Pointer to a buffer in which to print any log messages from PTXAS that reflect errors (the buffer size is specified via option [CU\\_JIT\\_ERROR\\_LOG\\_BUFFER\\_SIZE\\_BYTES](#))  
 Option type: char\*
- 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  
 Option type: unsigned int
- CU\_JIT\_OPTIMIZATION\_LEVEL*** Level of optimizations to apply to generated code (0 - 4), with 4 being the default and highest level of optimizations.  
 Option type: unsigned int
- CU\_JIT\_TARGET\_FROM\_CUCONTEXT*** No option value required. Determines the target based on the current attached context (default)  
 Option type: No option value needed
- CU\_JIT\_TARGET*** Target is chosen based on supplied [CUjit\\_target\\_enum](#).  
 Option type: unsigned int for enumerated type [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](#).  
 Option type: unsigned int for enumerated type [CUjit\\_fallback\\_enum](#)

#### 4.25.3.16 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  
***CU\_TARGET\_COMPUTE\_20*** Compute device class 2.0  
***CU\_TARGET\_COMPUTE\_21*** Compute device class 2.1

#### 4.25.3.17 enum CUlimit\_enum

Limits

**Enumerator:**

*CU\_LIMIT\_STACK\_SIZE* GPU thread stack size  
*CU\_LIMIT\_PRINTF\_FIFO\_SIZE* GPU printf FIFO size  
*CU\_LIMIT\_MALLOC\_HEAP\_SIZE* GPU malloc heap size

#### 4.25.3.18 enum CUmemorytype\_enum

Memory types

**Enumerator:**

*CU\_MEMORYTYPE\_HOST* Host memory  
*CU\_MEMORYTYPE\_DEVICE* Device memory  
*CU\_MEMORYTYPE\_ARRAY* Array memory

## 4.26 Initialization

### Functions

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

#### 4.26.1 Detailed Description

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

#### 4.26.2 Function Documentation

##### 4.26.2.1 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.

## 4.27 Version Management

### Functions

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

*Returns the CUDA driver version.*

#### 4.27.1 Detailed Description

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

#### 4.27.2 Function Documentation

##### 4.27.2.1 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.

## 4.28 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](#) (size\_t \*bytes, CUdevice dev)  
*Returns the total amount of memory on the device.*

### 4.28.1 Detailed Description

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

### 4.28.2 Function Documentation

#### 4.28.2.1 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](#)

#### 4.28.2.2 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](#)

#### 4.28.2.3 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.
- [CU\\_DEVICE\\_ATTRIBUTE\\_CONCURRENT\\_KERNELS](#): 1 if the device supports executing multiple kernels within the same context simultaneously, or 0 if not. It is not guaranteed that multiple kernels will be resident on the device concurrently so this feature should not be relied upon for correctness;
- [CU\\_DEVICE\\_ATTRIBUTE\\_ECC\\_ENABLED](#): 1 if error correction is enabled on the device, 0 if error correction is disabled or not supported by the device.
- [CU\\_DEVICE\\_ATTRIBUTE\\_PCI\\_BUS\\_ID](#): PCI bus identifier of the device.
- [CU\\_DEVICE\\_ATTRIBUTE\\_PCI\\_DEVICE\\_ID](#): PCI device (also known as slot) identifier of the device.
- [CU\\_DEVICE\\_ATTRIBUTE\\_TCC\\_DRIVER](#): 1 if the device is using a TCC driver. TCC is only available on Tesla hardware running Windows Vista or later.

**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](#)

**4.28.2.4 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](#)

**4.28.2.5 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](#)

#### 4.28.2.6 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](#)

#### 4.28.2.7 CUresult cuDeviceTotalMem (size\_t \* 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](#),

## 4.29 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 ctx)  
*Decrement a context's usage-count.*
- **CUresult cuCtxGetApiVersion** (CUcontext ctx, unsigned int \*version)  
*Gets the context's API version.*
- **CUresult cuCtxGetCacheConfig** (CUfunc\_cache \*pconfig)  
*Returns the preferred cache configuration for the current context.*
- **CUresult cuCtxGetDevice** (CUdevice \*device)  
*Returns the device ID for the current context.*
- **CUresult cuCtxGetLimit** (size\_t \*pvalue, CUlimit limit)  
*Returns resource limits.*
- **CUresult cuCtxPopCurrent** (CUcontext \*pctx)  
*Pops the current CUDA context from the current CPU thread.*
- **CUresult cuCtxPushCurrent** (CUcontext ctx)  
*Pushes a floating context on the current CPU thread.*
- **CUresult cuCtxSetCacheConfig** (CUfunc\_cache config)  
*Sets the preferred cache configuration for the current context.*
- **CUresult cuCtxSetLimit** (CUlimit limit, size\_t value)  
*Set resource limits.*
- **CUresult cuCtxSynchronize** (void)  
*Block for a context's tasks to complete.*

### 4.29.1 Detailed Description

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

## 4.29.2 Function Documentation

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

### 4.29.2.2 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.
- [CU\\_CTX\\_LMEM\\_RESIZE\\_TO\\_MAX](#): Instruct CUDA to not reduce local memory after resizing local memory for a kernel. This can prevent thrashing by local memory allocations when launching many kernels with high local memory usage at the cost of potentially increased memory usage.

**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:**

- ptx* - 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](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#)

**4.29.2.3 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](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#)

**4.29.2.4 CUresult cuCtxDetach (CUcontext ctx)**

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

**Parameters:**

*ctx* - 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](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#)

**4.29.2.5 CUresult cuCtxGetApiVersion (CUcontext ctx, unsigned int \* version)**

Returns the API version used to create `ctx` in `version`. If `ctx` is NULL, returns the API version used to create the currently bound context.

This will return the API version used to create a context (for example, 3010 or 3020), which library developers can use to direct callers to a specific API version. Note that this API version may not be the same as returned by [cuDriverGetVersion](#).

**Parameters:**

*ctx* - Context to check

*version* - Pointer to version

**Returns:**

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

**Note:**

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

**See also:**

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

#### 4.29.2.6 CUresult cuCtxGetCacheConfig (CUfunc\_cache \* pconfig)

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

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

The supported cache configurations are:

- `CU_FUNC_CACHE_PREFER_NONE`: no preference for shared memory or L1 (default)
- `CU_FUNC_CACHE_PREFER_SHARED`: prefer larger shared memory and smaller L1 cache
- `CU_FUNC_CACHE_PREFER_L1`: prefer larger L1 cache and smaller shared memory

##### Parameters:

`pconfig` - Returned cache configuration

##### 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`, `cuCtxGetApiVersion`, `cuCtxGetDevice`, `cuCtxGetLimit`, `cuCtxPopCurrent`, `cuCtxPushCurrent`, `cuCtxSetCacheConfig`, `cuCtxSetLimit`, `cuCtxSynchronize`, `cuFuncSetCacheConfig`

#### 4.29.2.7 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`, `cuCtxGetApiVersion`, `cuCtxGetCacheConfig`, `cuCtxGetLimit`, `cuCtxPopCurrent`, `cuCtxPushCurrent`, `cuCtxSetCacheConfig`, `cuCtxSetLimit`, `cuCtxSynchronize`

#### 4.29.2.8 CUresult cuCtxGetLimit (size\_t \* pvalue, CUlimit limit)

Returns in \*pvalue the current size of limit. The supported CUlimit values are:

- [CU\\_LIMIT\\_STACK\\_SIZE](#): stack size of each GPU thread;
- [CU\\_LIMIT\\_PRINTF\\_FIFO\\_SIZE](#): size of the FIFO used by the printf() device system call.
- [CU\\_LIMIT\\_MALLOC\\_HEAP\\_SIZE](#): size of the heap used by the malloc() and free() device system calls;

##### Parameters:

*limit* - Limit to query

*pvalue* - Returned size in bytes of limit

##### Returns:

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

##### Note:

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

##### See also:

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

#### 4.29.2.9 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 old context handle in \*pctx. That 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](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetLimit](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#)

#### 4.29.2.10 CUresult cuCtxPushCurrent (CUcontext *ctx*)

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

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

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

##### Parameters:

*ctx* - 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](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#)

#### 4.29.2.11 CUresult cuCtxSetCacheConfig (CUfunc\_cache *config*)

On devices where the L1 cache and shared memory use the same hardware resources, this sets through *config* the preferred cache configuration for the current context. This is only a preference. The driver will use the requested configuration if possible, but it is free to choose a different configuration if required to execute the function. Any function preference set via [cuFuncSetCacheConfig\(\)](#) will be preferred over this context-wide setting. Setting the context-wide cache configuration to [CU\\_FUNC\\_CACHE\\_PREFER\\_NONE](#) will cause subsequent kernel launches to prefer to not change the cache configuration unless required to launch the kernel.

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

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

The supported cache configurations are:

- [CU\\_FUNC\\_CACHE\\_PREFER\\_NONE](#): no preference for shared memory or L1 (default)
- [CU\\_FUNC\\_CACHE\\_PREFER\\_SHARED](#): prefer larger shared memory and smaller L1 cache
- [CU\\_FUNC\\_CACHE\\_PREFER\\_L1](#): prefer larger L1 cache and smaller shared memory

##### Parameters:

*config* - Requested cache configuration

##### 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](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetLimit](#), [cuCtxSynchronize](#), [cuFuncSetCacheConfig](#)

**4.29.2.12 CUresult cuCtxSetLimit (CUlimit *limit*, size\_t *value*)**

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

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

- [CU\\_LIMIT\\_STACK\\_SIZE](#) controls the stack size of each GPU thread. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error [CUDA\\_ERROR\\_UNSUPPORTED\\_LIMIT](#) being returned.
- [CU\\_LIMIT\\_PRINTF\\_FIFO\\_SIZE](#) controls the size of the FIFO used by the `printf()` device system call. Setting [CU\\_LIMIT\\_PRINTF\\_FIFO\\_SIZE](#) must be performed before launching any kernel that uses the `printf()` device system call, otherwise [CUDA\\_ERROR\\_INVALID\\_VALUE](#) will be returned. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error [CUDA\\_ERROR\\_UNSUPPORTED\\_LIMIT](#) being returned.
- [CU\\_LIMIT\\_MALLOC\\_HEAP\\_SIZE](#) controls the size of the heap used by the `malloc()` and `free()` device system calls. Setting [CU\\_LIMIT\\_MALLOC\\_HEAP\\_SIZE](#) must be performed before launching any kernel that uses the `malloc()` or `free()` device system calls, otherwise [CUDA\\_ERROR\\_INVALID\\_VALUE](#) will be returned. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error [CUDA\\_ERROR\\_UNSUPPORTED\\_LIMIT](#) being returned.

**Parameters:**

*limit* - Limit to set

*value* - Size in bytes of limit

**Returns:**

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

**Note:**

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

**See also:**

[cuCtxAttach](#), [cuCtxCreate](#), [cuCtxDestroy](#), [cuCtxDetach](#), [cuCtxGetApiVersion](#), [cuCtxGetCacheConfig](#), [cuCtxGetDevice](#), [cuCtxGetLimit](#), [cuCtxPopCurrent](#), [cuCtxPushCurrent](#), [cuCtxSetCacheConfig](#), [cuCtxSynchronize](#)

#### 4.29.2.13 CUresult cuCtxSynchronize (void)

Blocks until the device has completed all preceding requested tasks. `cuCtxSynchronize()` returns an error if one of the preceding tasks failed. If the context was created with the `CU_CTX_BLOCKING_SYNC` flag, the CPU thread will block until the GPU context has finished its work.

**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`, `cuCtxGetApiVersion`, `cuCtxGetCacheConfig`, `cuCtxGetDevice`, `cuCtxGetLimit`, `cuCtxPopCurrent`, `cuCtxPushCurrent`, `cuCtxSetCacheConfig`, `cuCtxSetLimit`

## 4.30 Module Management

### Functions

- [CUresult cuModuleGetFunction](#) ([CUfunction](#) \*hfunc, [CUmodule](#) hmod, const char \*name)  
*Returns a function handle.*
- [CUresult cuModuleGetGlobal](#) ([CUdeviceptr](#) \*dptr, size\_t \*bytes, [CUmodule](#) hmod, const char \*name)  
*Returns a global pointer from a module.*
- [CUresult cuModuleGetSurfRef](#) ([CUSurfref](#) \*pSurfRef, [CUmodule](#) hmod, const char \*name)  
*Returns a handle to a surface reference.*
- [CUresult cuModuleGetTexRef](#) ([CUTexref](#) \*pTexRef, [CUmodule](#) hmod, const char \*name)  
*Returns a handle to a texture reference.*
- [CUresult cuModuleLoad](#) ([CUmodule](#) \*module, const char \*fname)  
*Loads a compute module.*
- [CUresult cuModuleLoadData](#) ([CUmodule](#) \*module, const void \*image)  
*Load a module's data.*
- [CUresult cuModuleLoadDataEx](#) ([CUmodule](#) \*module, const void \*image, unsigned int numOptions, [CUjit\\_option](#) \*options, void \*\*optionValues)  
*Load a module's data with options.*
- [CUresult cuModuleLoadFatBinary](#) ([CUmodule](#) \*module, const void \*fatCubin)  
*Load a module's data.*
- [CUresult cuModuleUnload](#) ([CUmodule](#) hmod)  
*Unloads a module.*

### 4.30.1 Detailed Description

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

### 4.30.2 Function Documentation

#### 4.30.2.1 [CUresult cuModuleGetFunction](#) ([CUfunction](#) \*hfunc, [CUmodule](#) hmod, const char \*name)

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

#### Parameters:

- hfunc* - Returned function handle
- hmod* - Module to retrieve function from
- name* - Name of function to retrieve

**Returns:**

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

**Note:**

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

**See also:**

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

**4.30.2.2 CUresult cuModuleGetGlobal (CUdeviceptr \* *dptr*, size\_t \* *bytes*, CUmodule *hmod*, const char \* *name*)**

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

**Parameters:**

*dptr* - Returned global device pointer  
*bytes* - Returned global size in bytes  
*hmod* - Module to retrieve global from  
*name* - Name of global to retrieve

**Returns:**

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

**Note:**

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

**See also:**

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

**4.30.2.3 CUresult cuModuleGetSurfRef (CUSurfref \* *pSurfRef*, CUmodule *hmod*, const char \* *name*)**

Returns in *pSurfRef* the handle of the surface reference of name *name* in the module *hmod*. If no surface reference of that name exists, [cuModuleGetSurfRef\(\)](#) returns [CUDA\\_ERROR\\_NOT\\_FOUND](#).

**Parameters:**

*pSurfRef* - Returned surface reference  
*hmod* - Module to retrieve surface reference from  
*name* - Name of surface reference to retrieve

**Returns:**

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

**Note:**

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

**See also:**

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

**4.30.2.4 CUresult cuModuleGetTexRef (CUtexref \*pTexRef, CUmodule hmod, const char \* name)**

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

**Parameters:**

*pTexRef* - Returned texture reference  
*hmod* - Module to retrieve texture reference from  
*name* - Name of texture reference to retrieve

**Returns:**

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

**Note:**

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

**See also:**

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

**4.30.2.5 CUresult cuModuleLoad (CUmodule \* module, const char \* fname)**

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

**Parameters:**

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

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_NOT\_FOUND, CUDA\_ERROR\_OUT\_OF\_MEMORY, CUDA\_ERROR\_FILE\_NOT\_FOUND, CUDA\_ERROR\_SHARED\_OBJECT\_SYMBOL\_NOT\_FOUND, CUDA\_ERROR\_SHARED\_OBJECT\_INIT\_FAILED

**Note:**

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

**See also:**

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

**4.30.2.6 CUresult cuModuleLoadData (CUmodule \* module, const void \* image)**

Takes a pointer *image* and loads the corresponding module *module* into the current context. The pointer may be obtained by mapping a *cubin* or *PTX* file, passing a *cubin* or *PTX* file as a NULL-terminated 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:**

*module* - Returned module  
*image* - Module data to load

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_OUT\_OF\_MEMORY, CUDA\_ERROR\_SHARED\_OBJECT\_SYMBOL\_NOT\_FOUND, CUDA\_ERROR\_SHARED\_OBJECT\_INIT\_FAILED

**Note:**

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

**See also:**

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

**4.30.2.7 CUresult cuModuleLoadDataEx (CUmodule \* module, const void \* image, unsigned int numOptions, CUjit\_option \* options, void \*\* optionValues)**

Takes a pointer *image* and loads the corresponding module *module* into the current context. The pointer may be obtained by mapping a *cubin* or *PTX* file, passing a *cubin* or *PTX* file as a NULL-terminated 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 (types for the option values are specified in parentheses after the option name):

- `CU_JIT_MAX_REGISTERS`: (unsigned int) input specifies the maximum number of registers per thread;
- `CU_JIT_THREADS_PER_BLOCK`: (unsigned int) 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`: (float) output returns the float value of wall clock time, in milliseconds, spent compiling the *PTX* code;
- `CU_JIT_INFO_LOG_BUFFER`: (char\*) input is a pointer to a buffer in which to print any informational log messages from *PTX* assembly (the buffer size is specified via option `CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES`);
- `CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES`: (unsigned int) input is the size in bytes of the buffer; output is the number of bytes filled with messages;
- `CU_JIT_ERROR_LOG_BUFFER`: (char\*) input is a pointer to a buffer in which to print any error log messages from *PTX* assembly (the buffer size is specified via option `CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES`);
- `CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES`: (unsigned int) input is the size in bytes of the buffer; output is the number of bytes filled with messages;
- `CU_JIT_OPTIMIZATION_LEVEL`: (unsigned int) 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`: (No option value) causes compilation target to be determined based on current attached context (default);
- `CU_JIT_TARGET`: (unsigned int for enumerated type `CUjit_target_enum`) 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`
  - `CU_TARGET_COMPUTE_20`
- `CU_JIT_FALLBACK_STRATEGY`: (unsigned int for enumerated type `CUjit_fallback_enum`) chooses fallback strategy if matching cubin is not found; possible values are:
  - `CU_PREFER_PTX`
  - `CU_PREFER_BINARY`

#### Parameters:

- module* - Returned module
- image* - Module data to load
- numOptions* - Number of options
- options* - Options for JIT
- optionValues* - Option values for JIT

#### Returns:

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_OUT_OF_MEMORY`, `CUDA_ERROR_NO_BINARY_FOR_GPU`, `CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND`, `CUDA_ERROR_SHARED_OBJECT_INIT_FAILED`

**Note:**

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

**See also:**

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

**4.30.2.8 CUresult cuModuleLoadFatBinary (CUmodule \* *module*, const void \* *fatCubin*)**

Takes a pointer *fatCubin* and loads the corresponding module *module* into the current context. The pointer represents a *fat binary* object, which is a collection of different *cubin* 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:**

*module* - Returned module  
*fatCubin* - Fat binary to load

**Returns:**

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

**Note:**

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

**See also:**

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

**4.30.2.9 CUresult cuModuleUnload (CUmodule *hmod*)**

Unloads a module *hmod* from the current context.

**Parameters:**

*hmod* - 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](#)

## 4.31 Memory Management

### Functions

- **CUresult cuArray3DCreate** (CUarray \*pHandle, const CUDA\_ARRAY3D\_DESCRIPTOR \*pAllocateArray)  
*Creates a 3D CUDA array.*
- **CUresult cuArray3DGetDescriptor** (CUDA\_ARRAY3D\_DESCRIPTOR \*pArrayDescriptor, CUarray hArray)  
*Get a 3D CUDA array descriptor.*
- **CUresult cuArrayCreate** (CUarray \*pHandle, const CUDA\_ARRAY\_DESCRIPTOR \*pAllocateArray)  
*Creates a 1D or 2D CUDA array.*
- **CUresult cuArrayDestroy** (CUarray hArray)  
*Destroys a CUDA array.*
- **CUresult cuArrayGetDescriptor** (CUDA\_ARRAY\_DESCRIPTOR \*pArrayDescriptor, CUarray hArray)  
*Get a 1D or 2D CUDA array descriptor.*
- **CUresult cuMemAlloc** (CUdeviceptr \*dptr, size\_t bytesize)  
*Allocates device memory.*
- **CUresult cuMemAllocHost** (void \*\*pp, size\_t bytesize)  
*Allocates page-locked host memory.*
- **CUresult cuMemAllocPitch** (CUdeviceptr \*dptr, size\_t \*pPitch, size\_t WidthInBytes, size\_t 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 dstArray, size\_t dstOffset, CUarray srcArray, size\_t srcOffset, size\_t ByteCount)  
*Copies memory from Array to Array.*
- **CUresult cuMemcpyAtoD** (CUdeviceptr dstDevice, CUarray srcArray, size\_t srcOffset, size\_t ByteCount)

*Copies memory from Array to Device.*

- **CUresult cuMemcpyAtoH** (void \*dstHost, **CUarray** srcArray, size\_t srcOffset, size\_t ByteCount)  
*Copies memory from Array to Host.*
- **CUresult cuMemcpyAtoHAsync** (void \*dstHost, **CUarray** srcArray, size\_t srcOffset, size\_t ByteCount, **CUstream** hStream)  
*Copies memory from Array to Host.*
- **CUresult cuMemcpyDtoA** (**CUarray** dstArray, size\_t dstOffset, **CUdeviceptr** srcDevice, size\_t ByteCount)  
*Copies memory from Device to Array.*
- **CUresult cuMemcpyDtoD** (**CUdeviceptr** dstDevice, **CUdeviceptr** srcDevice, size\_t ByteCount)  
*Copies memory from Device to Device.*
- **CUresult cuMemcpyDtoDAsync** (**CUdeviceptr** dstDevice, **CUdeviceptr** srcDevice, size\_t ByteCount, **CUstream** hStream)  
*Copies memory from Device to Device.*
- **CUresult cuMemcpyDtoH** (void \*dstHost, **CUdeviceptr** srcDevice, size\_t ByteCount)  
*Copies memory from Device to Host.*
- **CUresult cuMemcpyDtoHAsync** (void \*dstHost, **CUdeviceptr** srcDevice, size\_t ByteCount, **CUstream** hStream)  
*Copies memory from Device to Host.*
- **CUresult cuMemcpyHtoA** (**CUarray** dstArray, size\_t dstOffset, const void \*srcHost, size\_t ByteCount)  
*Copies memory from Host to Array.*
- **CUresult cuMemcpyHtoAAsync** (**CUarray** dstArray, size\_t dstOffset, const void \*srcHost, size\_t ByteCount, **CUstream** hStream)  
*Copies memory from Host to Array.*
- **CUresult cuMemcpyHtoD** (**CUdeviceptr** dstDevice, const void \*srcHost, size\_t ByteCount)  
*Copies memory from Host to Device.*
- **CUresult cuMemcpyHtoDAsync** (**CUdeviceptr** dstDevice, const void \*srcHost, size\_t ByteCount, **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** \*pbase, size\_t \*psize, **CUdeviceptr** dptr)  
*Get information on memory allocations.*
- **CUresult cuMemGetInfo** (size\_t \*free, size\_t \*total)  
*Gets free and total memory.*

- [CUresult cuMemHostAlloc](#) (void \*\*pp, size\_t bytesize, unsigned int Flags)  
*Allocates page-locked host memory.*
- [CUresult cuMemHostGetDevicePointer](#) (CUdeviceptr \*pdptr, void \*p, unsigned int Flags)  
*Passes back device pointer of mapped pinned memory.*
- [CUresult cuMemHostGetFlags](#) (unsigned int \*pFlags, void \*p)  
*Passes back flags that were used for a pinned allocation.*
- [CUresult cuMemsetD16](#) (CUdeviceptr dstDevice, unsigned short us, size\_t N)  
*Initializes device memory.*
- [CUresult cuMemsetD16Async](#) (CUdeviceptr dstDevice, unsigned short us, size\_t N, CUstream hStream)  
*Sets device memory.*
- [CUresult cuMemsetD2D16](#) (CUdeviceptr dstDevice, size\_t dstPitch, unsigned short us, size\_t Width, size\_t Height)  
*Initializes device memory.*
- [CUresult cuMemsetD2D16Async](#) (CUdeviceptr dstDevice, size\_t dstPitch, unsigned short us, size\_t Width, size\_t Height, CUstream hStream)  
*Sets device memory.*
- [CUresult cuMemsetD2D32](#) (CUdeviceptr dstDevice, size\_t dstPitch, unsigned int ui, size\_t Width, size\_t Height)  
*Initializes device memory.*
- [CUresult cuMemsetD2D32Async](#) (CUdeviceptr dstDevice, size\_t dstPitch, unsigned int ui, size\_t Width, size\_t Height, CUstream hStream)  
*Sets device memory.*
- [CUresult cuMemsetD2D8](#) (CUdeviceptr dstDevice, size\_t dstPitch, unsigned char uc, size\_t Width, size\_t Height)  
*Initializes device memory.*
- [CUresult cuMemsetD2D8Async](#) (CUdeviceptr dstDevice, size\_t dstPitch, unsigned char uc, size\_t Width, size\_t Height, CUstream hStream)  
*Sets device memory.*
- [CUresult cuMemsetD32](#) (CUdeviceptr dstDevice, unsigned int ui, size\_t N)  
*Initializes device memory.*
- [CUresult cuMemsetD32Async](#) (CUdeviceptr dstDevice, unsigned int ui, size\_t N, CUstream hStream)  
*Sets device memory.*
- [CUresult cuMemsetD8](#) (CUdeviceptr dstDevice, unsigned char uc, size\_t N)  
*Initializes device memory.*
- [CUresult cuMemsetD8Async](#) (CUdeviceptr dstDevice, unsigned char uc, size\_t N, CUstream hStream)  
*Sets device memory.*

### 4.31.1 Detailed Description

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

### 4.31.2 Function Documentation

#### 4.31.2.1 `CUresult cuArray3DCreate (CUarray * pHandle, const CUDA_ARRAY3D_DESCRIPTOR * pAllocateArray)`

Creates a CUDA array according to the `CUDA_ARRAY3D_DESCRIPTOR` structure `pAllocateArray` and returns a handle to the new CUDA array in `*pHandle`. The `CUDA_ARRAY3D_DESCRIPTOR` is defined as:

```
typedef struct {
    unsigned int Width;
    unsigned int Height;
    unsigned int Depth;
    CUarray_format Format;
    unsigned int NumChannels;
    unsigned int Flags;
} CUDA_ARRAY3D_DESCRIPTOR;
```

where:

- `Width`, `Height`, and `Depth` are the width, height, and depth of the CUDA array (in elements); the 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` may be set to `CUDA_ARRAY3D_SURFACE_LDST` to enable surface references to be bound to the CUDA array. If this flag is not set, `cuSurfRefSetArray` will fail when attempting to bind the CUDA array to a surface reference.

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  
*pAllocateArray* - 3D array descriptor

**Returns:**

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

**Note:**

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

**See also:**

[cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

**4.31.2.2 CUresult cuArray3DGetDescriptor (CUDA\_ARRAY3D\_DESCRIPTOR \* pArrayDescriptor, CUarray hArray)**

Returns in \*pArrayDescriptor a descriptor containing information on the format and dimensions of the CUDA array hArray. It is useful for subroutines that have been passed a CUDA array, but need to know the CUDA array parameters for validation or other purposes.

This function may be called on 1D and 2D arrays, in which case the Height and/or Depth members of the descriptor struct will be set to 0.

**Parameters:**

*pArrayDescriptor* - Returned 3D array descriptor  
*hArray* - 3D array to get descriptor of

**Returns:**

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

**Note:**

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

**See also:**

cuArray3DCreate, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

#### 4.31.2.3 CUresult cuArrayCreate (CUarray \* *pHandle*, const CUDA\_ARRAY\_DESCRIPTOR \* *pAllocateArray*)

Creates a CUDA array according to the [CUDA\\_ARRAY\\_DESCRIPTOR](#) structure *pAllocateArray* and returns a handle to the new CUDA array in *\*pHandle*. The [CUDA\\_ARRAY\\_DESCRIPTOR](#) is defined as:

```
typedef struct {
    unsigned int Width;
    unsigned int Height;
    CUarray_format Format;
    unsigned int NumChannels;
} CUDA_ARRAY_DESCRIPTOR;
```

where:

- *Width*, and *Height* are the width, and height of the CUDA array (in elements); the CUDA array is one-dimensional if height is 0, two-dimensional otherwise;
- *Format* specifies the format of the elements; [CUarray\\_format](#) is defined as:

```
typedef enum CUarray_format_enum {
    CU_AD_FORMAT_UNSIGNED_INT8 = 0x01,
    CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
    CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
    CU_AD_FORMAT_SIGNED_INT8 = 0x08,
    CU_AD_FORMAT_SIGNED_INT16 = 0x09,
    CU_AD_FORMAT_SIGNED_INT32 = 0x0a,
    CU_AD_FORMAT_HALF = 0x10,
    CU_AD_FORMAT_FLOAT = 0x20
} 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_INT8;
desc.NumChannels = 2;
desc.Width = width;
desc.Height = height;
```

### Parameters:

*pHandle* - Returned array

*pAllocateArray* - Array descriptor

### Returns:

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

### Note:

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

### See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

#### 4.31.2.4 CUresult cuArrayDestroy (CUarray *hArray*)

Destroys the CUDA array *hArray*.

**Parameters:**

*hArray* - Array to destroy

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_HANDLE, CUDA\_ERROR\_ARRAY\_IS\_MAPPED

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

#### 4.31.2.5 CUresult cuArrayGetDescriptor (CUDA\_ARRAY\_DESCRIPTOR \**pArrayDescriptor*, CUarray *hArray*)

Returns in *pArrayDescriptor* a descriptor containing information on the format and dimensions of the CUDA array *hArray*. It is useful for subroutines that have been passed a CUDA array, but need to know the CUDA array parameters for validation or other purposes.

**Parameters:**

*pArrayDescriptor* - Returned array descriptor

*hArray* - Array to get descriptor of

**Returns:**

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

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

#### 4.31.2.6 CUresult cuMemAlloc (CUdeviceptr \* *dptr*, size\_t *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`, `cuMemcpyDtoDAsync`, `cuMemcpyDtoH`, `cuMemcpyDtoHAsync`, `cuMemcpyHtoA`, `cuMemcpyHtoAAsync`, `cuMemcpyHtoD`, `cuMemcpyHtoDAsync`, `cuMemFree`, `cuMemFreeHost`, `cuMemGetAddressRange`, `cuMemGetInfo`, `cuMemHostAlloc`, `cuMemHostGetDevicePointer`, `cuMemsetD2D8`, `cuMemsetD2D16`, `cuMemsetD2D32`, `cuMemsetD8`, `cuMemsetD16`, `cuMemsetD32`

#### 4.31.2.7 CUresult cuMemAllocHost (void \*\* *pp*, size\_t *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](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

#### 4.31.2.8 CUresult cuMemAllocPitch (CUdeviceptr \*dptr, size\_t \*pPitch, size\_t WidthInBytes, size\_t 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).

The byte alignment of the pitch returned by [cuMemAllocPitch\(\)](#) is guaranteed to match or exceed the alignment requirement for texture binding with [cuTexRefSetAddress2D\(\)](#).

**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](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

#### 4.31.2.9 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.
- If specified, `srcPitch` must be greater than or equal to `WidthInBytes + srcXInBytes`, and `dstPitch` must be greater than or equal to `WidthInBytes + dstXInBytes`.

`cuMemcpy2D()` returns an error if any pitch is greater than the maximum allowed (`CU_DEVICE_ATTRIBUTE_MAX_PITCH`). `cuMemAllocPitch()` passes back pitches that always work with `cuMemcpy2D()`. On intra-device memory copies (device ? 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](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

**4.31.2.10 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.
- If specified, `srcPitch` must be greater than or equal to `WidthInBytes + srcXInBytes`, and `dstPitch` must be greater than or equal to `WidthInBytes + dstXInBytes`.
- If specified, `srcPitch` must be greater than or equal to `WidthInBytes + srcXInBytes`, and `dstPitch` must be greater than or equal to `WidthInBytes + dstXInBytes`.
- If specified, `srcHeight` must be greater than or equal to `Height + srcY`, and `dstHeight` must be greater than or equal to `Height + dstY`.

`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`, `cuMemcpyDtoDAsync`, `cuMemcpyDtoH`, `cuMemcpyDtoHAsync`, `cuMemcpyHtoA`, `cuMemcpyHtoAAsync`, `cuMemcpyHtoD`, `cuMemcpyHtoDAsync`, `cuMemFree`, `cuMemFreeHost`, `cuMemGetAddressRange`, `cuMemGetInfo`, `cuMemHostAlloc`, `cuMemHostGetDevicePointer`, `cuMemsetD2D8`, `cuMemsetD2D8Async`, `cuMemsetD2D16`, `cuMemsetD2D16Async`, `cuMemsetD2D32`, `cuMemsetD2D32Async`, `cuMemsetD8`, `cuMemsetD8Async`, `cuMemsetD16`, `cuMemsetD16Async`, `cuMemsetD32`, `cuMemsetD32Async`

#### 4.31.2.11 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.
- If specified, `srcPitch` must be greater than or equal to `WidthInBytes + srcXInBytes`, and `dstPitch` must be greater than or equal to `WidthInBytes + dstXInBytes`.

`cuMemcpy2D()` returns an error if any pitch is greater than the maximum allowed (`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`, `cuMemcpy2D`, `cuMemcpy2DAsync`, `cuMemcpy3D`, `cuMemcpy3DAsync`, `cuMemcpyAtoA`, `cuMemcpyAtoD`, `cuMemcpyAtoH`, `cuMemcpyAtoHAsync`, `cuMemcpyDtoA`, `cuMemcpyDtoD`, `cuMemcpyDtoDAsync`, `cuMemcpyDtoH`, `cuMemcpyDtoHAsync`, `cuMemcpyHtoA`, `cuMemcpyHtoAAsync`, `cuMemcpyHtoD`, `cuMemcpyHtoDAsync`, `cuMemFree`, `cuMemFreeHost`, `cuMemGetAddressRange`, `cuMemGetInfo`, `cuMemHostAlloc`, `cuMemHostGetDevicePointer`, `cuMemsetD2D8`, `cuMemsetD2D16`, `cuMemsetD2D32`, `cuMemsetD8`, `cuMemsetD16`, `cuMemsetD32`

#### 4.31.2.12 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.
- If specified, `srcPitch` must be greater than or equal to `WidthInBytes + srcXInBytes`, and `dstPitch` must be greater than or equal to `WidthInBytes + dstXInBytes`.
- If specified, `srcHeight` must be greater than or equal to `Height + srcY`, and `dstHeight` must be greater than or equal to `Height + dstY`.

[cuMemcpy3D\(\)](#) returns an error if any pitch is greater than the maximum allowed ([CU\\_DEVICE\\_ATTRIBUTE\\_MAX\\_PITCH](#)).

The `srcLOD` and `dstLOD` members of the [CUDA\\_MEMCPY3D](#) structure must be set to 0.

#### 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](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

#### 4.31.2.13 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.
- If specified, `srcPitch` must be greater than or equal to `WidthInBytes + srcXInBytes`, and `dstPitch` must be greater than or equal to `WidthInBytes + dstXInBytes`.
- If specified, `srcHeight` must be greater than or equal to `Height + srcY`, and `dstHeight` must be greater than or equal to `Height + dstY`.

`cuMemcpy3D()` returns an error if any pitch is greater than the maximum allowed (`CU_DEVICE_ATTRIBUTE_MAX_PITCH`).

`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, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

**4.31.2.14 CUresult cuMemcpyAtoA (CUarray dstArray, size\_t dstOffset, CUarray srcArray, size\_t srcOffset, size\_t ByteCount)**

Copies from one 1D CUDA array to another. *dstArray* and *srcArray* specify the handles of the destination and source CUDA arrays for the copy, respectively. *dstOffset* and *srcOffset* specify the destination and source offsets in bytes into the CUDA arrays. *ByteCount* is the number of bytes to be copied. The size of the elements in the CUDA arrays need not be the same format, but the elements must be the same size; and count must be evenly divisible by that size.

**Parameters:**

*dstArray* - Destination array  
*dstOffset* - Offset in bytes of destination array  
*srcArray* - Source array  
*srcOffset* - Offset in bytes of source array  
*ByteCount* - Size of memory copy in bytes

**Returns:**

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

**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, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

#### 4.31.2.15 CUresult cuMemcpyAtoD (CUdeviceptr *dstDevice*, CUarray *srcArray*, size\_t *srcOffset*, size\_t *ByteCount*)

Copies from one 1D CUDA array to device memory. *dstDevice* specifies the base pointer of the destination and must be naturally aligned with the CUDA array elements. *srcArray* and *srcOffset* specify the CUDA array handle and the offset in bytes into the array where the copy is to begin. *ByteCount* specifies the number of bytes to copy and must be evenly divisible by the array element size.

##### Parameters:

*dstDevice* - Destination device pointer  
*srcArray* - Source array  
*srcOffset* - Offset in bytes of source array  
*ByteCount* - Size of memory copy in bytes

##### Returns:

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

##### 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](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

#### 4.31.2.16 CUresult cuMemcpyAtoH (void \* *dstHost*, CUarray *srcArray*, size\_t *srcOffset*, size\_t *ByteCount*)

Copies from one 1D CUDA array to host memory. *dstHost* specifies the base pointer of the destination. *srcArray* and *srcOffset* specify the CUDA array handle and starting offset in bytes of the source data. *ByteCount* specifies the number of bytes to copy.

##### Parameters:

*dstHost* - Destination device pointer  
*srcArray* - Source array  
*srcOffset* - Offset in bytes of source array  
*ByteCount* - Size of memory copy in bytes

##### Returns:

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

##### 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](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

#### 4.31.2.17 CUresult cuMemcpyAtoHAsync (void \* *dstHost*, CUarray *srcArray*, size\_t *srcOffset*, size\_t *ByteCount*, CUstream *hStream*)

Copies from one 1D CUDA array to host memory. *dstHost* specifies the base pointer of the destination. *srcArray* and *srcOffset* specify the CUDA array handle and starting offset in bytes of the source data. *ByteCount* specifies the number of bytes to copy.

[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:**

*dstHost* - Destination pointer  
*srcArray* - Source array  
*srcOffset* - Offset in bytes of source array  
*ByteCount* - Size of memory copy in bytes  
*hStream* - Stream identifier

**Returns:**

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

**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](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#)

#### 4.31.2.18 CUresult cuMemcpyDtoA (CUarray *dstArray*, size\_t *dstOffset*, CUdeviceptr *srcDevice*, size\_t *ByteCount*)

Copies from device memory to a 1D CUDA array. *dstArray* and *dstOffset* specify the CUDA array handle and starting index of the destination data. *srcDevice* specifies the base pointer of the source. *ByteCount* specifies the number of bytes to copy.

**Parameters:**

*dstArray* - Destination array  
*dstOffset* - Offset in bytes of destination array  
*srcDevice* - Source device pointer  
*ByteCount* - Size of memory copy in bytes

**Returns:**

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

**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, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

**4.31.2.19 CUresult cuMemcpyDtoD (CUdeviceptr *dstDevice*, CUdeviceptr *srcDevice*, size\_t *ByteCount*)**

Copies from device memory to device memory. *dstDevice* and *srcDevice* are the base pointers of the destination and source, respectively. *ByteCount* specifies the number of bytes to copy. Note that this function is asynchronous.

**Parameters:**

*dstDevice* - Destination device pointer  
*srcDevice* - Source device pointer  
*ByteCount* - Size of memory copy in bytes

**Returns:**

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

**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

#### 4.31.2.20 CUresult cuMemcpyDtoDAsync (CUdeviceptr *dstDevice*, CUdeviceptr *srcDevice*, size\_t *ByteCount*, CUstream *hStream*)

Copies from device memory to device memory. *dstDevice* and *srcDevice* are the base pointers of the destination and source, respectively. *ByteCount* specifies the number of bytes to copy. Note that this function is asynchronous and can optionally be associated to a stream by passing a non-zero *hStream* argument

**Parameters:**

- dstDevice* - Destination device pointer
- srcDevice* - Source device pointer
- ByteCount* - Size of memory copy in bytes
- hStream* - Stream identifier

**Returns:**

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

**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, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

#### 4.31.2.21 CUresult cuMemcpyDtoH (void \* *dstHost*, CUdeviceptr *srcDevice*, size\_t *ByteCount*)

Copies from device to host memory. *dstHost* and *srcDevice* specify the base pointers of the destination and source, respectively. *ByteCount* specifies the number of bytes to copy. Note that this function is synchronous.

**Parameters:**

- dstHost* - Destination host pointer
- srcDevice* - Source device pointer
- ByteCount* - Size of memory copy in bytes

**Returns:**

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

**Note:**

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

#### 4.31.2.22 **CUresult cuMemcpyDtoHAsync (void \* *dstHost*, CUdeviceptr *srcDevice*, size\_t *ByteCount*, CUstream *hStream*)**

Copies from device to host memory. *dstHost* and *srcDevice* specify the base pointers of the destination and source, respectively. *ByteCount* specifies the number of bytes to copy.

[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:**

*dstHost* - Destination host pointer  
*srcDevice* - Source device pointer  
*ByteCount* - Size of memory copy in bytes  
*hStream* - Stream identifier

**Returns:**

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

**Note:**

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#)

#### 4.31.2.23 **CUresult cuMemcpyHtoA (CUarray *dstArray*, size\_t *dstOffset*, const void \* *srcHost*, size\_t *ByteCount*)**

Copies from host memory to a 1D CUDA array. *dstArray* and *dstOffset* specify the CUDA array handle and starting offset in bytes of the destination data. *srcHost* specifies the base address of the source. *ByteCount* specifies the number of bytes to copy.

**Parameters:**

*dstArray* - Destination array  
*dstOffset* - Offset in bytes of destination array  
*srcHost* - Source host pointer  
*ByteCount* - Size of memory copy in bytes

**Returns:**

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

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

**4.31.2.24 CUresult cuMemcpyHtoAAsync (CUarray *dstArray*, size\_t *dstOffset*, const void \* *srcHost*, size\_t *ByteCount*, CUstream *hStream*)**

Copies from host memory to a 1D CUDA array. *dstArray* and *dstOffset* specify the CUDA array handle and starting offset in bytes of the destination data. *srcHost* specifies the base address of the source. *ByteCount* specifies the number of bytes to copy.

`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:**

*dstArray* - Destination array  
*dstOffset* - Offset in bytes of destination array  
*srcHost* - Source host pointer  
*ByteCount* - Size of memory copy in bytes  
*hStream* - Stream identifier

**Returns:**

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

**Note:**

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#)

**4.31.2.25 CUresult cuMemcpyHtoD (CUdeviceptr *dstDevice*, const void \* *srcHost*, size\_t *ByteCount*)**

Copies from host memory to device memory. *dstDevice* and *srcHost* are the base addresses of the destination and source, respectively. *ByteCount* specifies the number of bytes to copy. Note that this function is synchronous.

**Parameters:**

*dstDevice* - Destination device pointer  
*srcHost* - Source host pointer  
*ByteCount* - Size of memory copy in bytes

**Returns:**

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

**Note:**

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

**4.31.2.26 CUresult cuMemcpyHtoDAsync (CUdeviceptr *dstDevice*, const void \* *srcHost*, size\_t *ByteCount*, CUstream *hStream*)**

Copies from host memory to device memory. *dstDevice* and *srcHost* are the base addresses of the destination and source, respectively. *ByteCount* specifies the number of bytes to copy.

[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:**

*dstDevice* - Destination device pointer

*srcHost* - Source host pointer

*ByteCount* - Size of memory copy in bytes

*hStream* - Stream identifier

**Returns:**

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

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

#### 4.31.2.27 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, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

#### 4.31.2.28 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](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

#### 4.31.2.29 CUresult cuMemGetAddressRange (CUdeviceptr \* *pbase*, size\_t \* *psize*, CUdeviceptr *dptr*)

Returns the base address in *pbase* and size in *psize* of the allocation by [cuMemAlloc\(\)](#) or [cuMemAllocPitch\(\)](#) that contains the input pointer *dptr*. Both parameters *pbase* and *psize* are optional. If one of them is NULL, it is ignored.

##### Parameters:

*pbase* - Returned base address

*psize* - Returned size of device memory allocation

*dptr* - Device pointer to query

##### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_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](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

#### 4.31.2.30 CUresult cuMemGetInfo (size\_t \* free, size\_t \* 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](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

#### 4.31.2.31 CUresult cuMemHostAlloc (void \*\* pp, size\_t bytesize, unsigned int Flags)

Allocates *bytesize* bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as [cuMemcpyHtoD\(\)](#). Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as `malloc()`. 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
- bytesize* - Requested allocation size in bytes
- Flags* - Flags for allocation request

**Returns:**

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

**Note:**

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

**See also:**

`cuArray3DCreate`, `cuArray3DGetDescriptor`, `cuArrayCreate`, `cuArrayDestroy`, `cuArrayGetDescriptor`, `cuMemAlloc`, `cuMemAllocHost`, `cuMemAllocPitch`, `cuMemcpy2D`, `cuMemcpy2DAsync`, `cuMemcpy2DUnaligned`, `cuMemcpy3D`, `cuMemcpy3DAsync`, `cuMemcpyAtoA`, `cuMemcpyAtoD`, `cuMemcpyAtoH`, `cuMemcpyAtoHAsync`, `cuMemcpyDtoA`, `cuMemcpyDtoD`, `cuMemcpyDtoDAsync`, `cuMemcpyDtoH`, `cuMemcpyDtoHAsync`, `cuMemcpyHtoA`, `cuMemcpyHtoAAsync`, `cuMemcpyHtoD`, `cuMemcpyHtoDAsync`, `cuMemFree`, `cuMemFreeHost`, `cuMemGetAddressRange`, `cuMemGetInfo`, `cuMemHostGetDevicePointer`, `cuMemsetD2D8`, `cuMemsetD2D16`, `cuMemsetD2D32`, `cuMemsetD8`, `cuMemsetD16`, `cuMemsetD32`

#### 4.31.2.32 `CUresult cuMemHostGetDevicePointer (CUdeviceptr *pdptr, void *p, unsigned int Flags)`

Passes back the device pointer `pdptr` 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:**

- pdptr* - Returned device pointer
- p* - Host pointer
- Flags* - Options (must be 0)

**Returns:**

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

**Note:**

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

**4.31.2.33 CUresult cuMemHostGetFlags (unsigned int \* *pFlags*, void \* *p*)**

Passes back the flags `pFlags` that were specified when allocating the pinned host buffer `p` allocated by [cuMemHostAlloc](#).

[cuMemHostGetFlags\(\)](#) will fail if the pointer does not reside in an allocation performed by [cuMemAllocHost\(\)](#) or [cuMemHostAlloc\(\)](#).

**Parameters:**

*pFlags* - Returned flags word

*p* - Host pointer

**Returns:**

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

**Note:**

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

**See also:**

[cuMemAllocHost](#), [cuMemHostAlloc](#)

**4.31.2.34 CUresult cuMemsetD16 (CUdeviceptr *dstDevice*, unsigned short *us*, size\_t *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](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#)

#### 4.31.2.35 CUresult cuMemsetD16Async (CUdeviceptr *dstDevice*, unsigned short *us*, size\_t *N*, CUstream *hStream*)

Sets the memory range of *N* 16-bit values to the specified value *us*.

[cuMemsetD16Async\(\)](#) is asynchronous and can optionally be associated to a stream by passing a non-zero `stream` argument.

**Parameters:**

*dstDevice* - Destination device pointer

*us* - Value to set

*N* - Number of elements

*hStream* - Stream identifier

**Returns:**

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

**Note:**

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD32](#), [cuMemsetD32Async](#)

#### 4.31.2.36 CUresult cuMemsetD2D16 (CUdeviceptr *dstDevice*, size\_t *dstPitch*, unsigned short *us*, size\_t *Width*, size\_t *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, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

**4.31.2.37 CUresult cuMemsetD2D16Async (CUdeviceptr *dstDevice*, size\_t *dstPitch*, unsigned short *us*, size\_t *Width*, size\_t *Height*, CUSTream *hStream*)**

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\(\)](#).

[cuMemsetD2D16Async\(\)](#) is asynchronous and can optionally be associated to a stream by passing a non-zero `stream` argument.

**Parameters:**

*dstDevice* - Destination device pointer  
*dstPitch* - Pitch of destination device pointer  
*us* - Value to set  
*Width* - Width of row  
*Height* - Number of rows  
*hStream* - Stream identifier

**Returns:**

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

**Note:**

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#)

#### 4.31.2.38 **CUresult cuMemsetD2D32 (CUdeviceptr *dstDevice*, size\_t *dstPitch*, unsigned int *ui*, size\_t *Width*, size\_t *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](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#)

#### 4.31.2.39 **CUresult cuMemsetD2D32Async (CUdeviceptr *dstDevice*, size\_t *dstPitch*, unsigned int *ui*, size\_t *Width*, size\_t *Height*, CUstream *hStream*)**

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\(\)](#).

[cuMemsetD2D32Async\(\)](#) is asynchronous and can optionally be associated to a stream by passing a non-zero `stream` argument.

**Parameters:**

*dstDevice* - Destination device pointer  
*dstPitch* - Pitch of destination device pointer  
*ui* - Value to set  
*Width* - Width of row  
*Height* - Number of rows  
*hStream* - Stream identifier

**Returns:**

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

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

**4.31.2.40 CUresult cuMemsetD2D8 (CUdeviceptr *dstDevice*, *size\_t* *dstPitch*, unsigned char *uc*, *size\_t* *Width*, *size\_t* *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](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#)

#### 4.31.2.41 **CUresult cuMemsetD2D8Async (CUdeviceptr *dstDevice*, size\_t *dstPitch*, unsigned char *uc*, size\_t *Width*, size\_t *Height*, CUstream *hStream*)**

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\(\)](#).

[cuMemsetD2D8Async\(\)](#) is asynchronous and can optionally be associated to a stream by passing a non-zero `stream` argument.

**Parameters:**

*dstDevice* - Destination device pointer  
*dstPitch* - Pitch of destination device pointer  
*uc* - Value to set  
*Width* - Width of row  
*Height* - Number of rows  
*hStream* - Stream identifier

**Returns:**

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

**Note:**

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#)

#### 4.31.2.42 **CUresult cuMemsetD32 (CUdeviceptr *dstDevice*, unsigned int *ui*, size\_t *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, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32Async

**4.31.2.43 CUresult cuMemsetD32Async (CUdeviceptr *dstDevice*, unsigned int *ui*, size\_t *N*, CUstream *hStream*)**

Sets the memory range of *N* 32-bit values to the specified value *ui*.

`cuMemsetD32Async()` is asynchronous and can optionally be associated to a stream by passing a non-zero `stream` argument.

**Parameters:**

*dstDevice* - Destination device pointer

*ui* - Value to set

*N* - Number of elements

*hStream* - Stream identifier

**Returns:**

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

**Note:**

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

**See also:**

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync,

[cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#)

#### 4.31.2.44 CUresult cuMemsetD8 (CUdeviceptr *dstDevice*, unsigned char *uc*, size\_t *N*)

Sets the memory range of  $N$  8-bit values to the specified value *uc*.

##### Parameters:

*dstDevice* - Destination device pointer

*uc* - Value to set

*N* - Number of elements

##### Returns:

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

##### Note:

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

##### See also:

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#)

#### 4.31.2.45 CUresult cuMemsetD8Async (CUdeviceptr *dstDevice*, unsigned char *uc*, size\_t *N*, CUstream *hStream*)

Sets the memory range of  $N$  8-bit values to the specified value *uc*.

[cuMemsetD8Async\(\)](#) is asynchronous and can optionally be associated to a stream by passing a non-zero *stream* argument.

##### Parameters:

*dstDevice* - Destination device pointer

*uc* - Value to set

*N* - Number of elements

*hStream* - Stream identifier

##### Returns:

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

**Note:**

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

**See also:**

[cuArray3DCreate](#), [cuArray3DGetDescriptor](#), [cuArrayCreate](#), [cuArrayDestroy](#), [cuArrayGetDescriptor](#), [cuMemAlloc](#), [cuMemAllocHost](#), [cuMemAllocPitch](#), [cuMemcpy2D](#), [cuMemcpy2DAsync](#), [cuMemcpy2DUnaligned](#), [cuMemcpy3D](#), [cuMemcpy3DAsync](#), [cuMemcpyAtoA](#), [cuMemcpyAtoD](#), [cuMemcpyAtoH](#), [cuMemcpyAtoHAsync](#), [cuMemcpyDtoA](#), [cuMemcpyDtoD](#), [cuMemcpyDtoDAsync](#), [cuMemcpyDtoH](#), [cuMemcpyDtoHAsync](#), [cuMemcpyHtoA](#), [cuMemcpyHtoAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#)

## 4.32 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.*
- [CUresult cuStreamWaitEvent](#) (CUstream hStream, CUevent hEvent, unsigned int Flags)  
*Make a compute stream wait on an event.*

### 4.32.1 Detailed Description

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

### 4.32.2 Function Documentation

#### 4.32.2.1 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](#), [cuStreamWaitEvent](#), [cuStreamQuery](#), [cuStreamSynchronize](#)

#### 4.32.2.2 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](#), [cuStreamWaitEvent](#), [cuStreamQuery](#), [cuStreamSynchronize](#)

#### 4.32.2.3 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](#), [cuStreamWaitEvent](#), [cuStreamDestroy](#), [cuStreamSynchronize](#)

#### 4.32.2.4 CUresult cuStreamSynchronize (CUstream *hStream*)

Waits until the device has completed all operations in the stream specified by `hStream`. If the context was created with the [CU\\_CTX\\_BLOCKING\\_SYNC](#) flag, the CPU thread will block until the stream is finished with all of its tasks.

**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](#), [cuStreamWaitEvent](#), [cuStreamQuery](#)

**4.32.2.5 CUresult cuStreamWaitEvent (CUstream *hStream*, CUevent *hEvent*, unsigned int *Flags*)**

Makes all future work submitted to *hStream* wait until *hEvent* reports completion before beginning execution. This synchronization will be performed efficiently on the device.

The stream *hStream* will wait only for the completion of the most recent host call to [cuEventRecord\(\)](#) on *hEvent*. Once this call has returned, any functions (including [cuEventRecord\(\)](#) and [cuEventDestroy\(\)](#)) may be called on *hEvent* again, and the subsequent calls will not have any effect on *hStream*.

If *hStream* is 0 (the NULL stream) any future work submitted in any stream will wait for *hEvent* to complete before beginning execution. This effectively creates a barrier for all future work submitted to the context.

If [cuEventRecord\(\)](#) has not been called on *hEvent*, this call acts as if the record has already completed, and so is a functional no-op.

**Parameters:**

*hStream* - Stream to wait

*hEvent* - Event to wait on (may not be NULL)

*Flags* - Parameters for the operation (must be 0)

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

## 4.33 Event Management

### Functions

- [CUresult cuEventCreate](#) ([CUevent](#) \*phEvent, 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.*

### 4.33.1 Detailed Description

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

### 4.33.2 Function Documentation

#### 4.33.2.1 [CUresult cuEventCreate](#) ([CUevent](#) \*phEvent, unsigned int Flags)

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

- [CU\\_EVENT\\_DEFAULT](#): Default event creation flag.
- [CU\\_EVENT\\_BLOCKING\\_SYNC](#): Specifies that the created event should use blocking synchronization. A CPU thread that uses [cuEventSynchronize\(\)](#) to wait on an event created with this flag will block until the event has actually been recorded.
- [CU\\_EVENT\\_DISABLE\\_TIMING](#): Specifies that the created event does not need to record timing data. Events created with this flag specified and the [CU\\_EVENT\\_BLOCKING\\_SYNC](#) flag not specified will provide the best performance when used with [cuStreamWaitEvent\(\)](#) and [cuEventQuery\(\)](#).

#### Parameters:

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

#### Returns:

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

**Note:**

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

**See also:**

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

**4.33.2.2 CUresult cuEventDestroy (CUevent *hEvent*)**

Destroys the event specified by *hEvent*.

**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](#)

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

If [cuEventRecord\(\)](#) has not been called on either event then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If [cuEventRecord\(\)](#) has been called on both events but one or both of them has not yet been completed (that is, [cuEventQuery\(\)](#) would return [CUDA\\_ERROR\\_NOT\\_READY](#) on at least one of the events), [CUDA\\_ERROR\\_NOT\\_READY](#) is returned. If either event was created with the [CU\\_EVENT\\_DISABLE\\_TIMING](#) flag, then this function will return [CUDA\\_ERROR\\_INVALID\\_HANDLE](#).

**Parameters:**

*pMilliseconds* - Time between *hStart* and *hEnd* in ms

*hStart* - Starting event

*hEnd* - Ending event

**Returns:**

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

**Note:**

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

**See also:**

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

**4.33.2.4 CUresult cuEventQuery (CUevent *hEvent*)**

Query the status of all device work preceding the most recent call to [cuEventRecord\(\)](#) (in the appropriate compute streams, as specified by the arguments to [cuEventRecord\(\)](#)).

If this work has successfully been completed by the device, or if [cuEventRecord\(\)](#) has not been called on `hEvent`, then [CUDA\\_SUCCESS](#) is returned. If this work has not yet been completed by the device then [CUDA\\_ERROR\\_NOT\\_READY](#) is returned.

**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](#)

**4.33.2.5 CUresult cuEventRecord (CUevent *hEvent*, CUstream *hStream*)**

Records an event. If `hStream` is non-zero, the event is recorded after all preceding operations in `hStream` 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 on `hEvent`, then this call will overwrite any existing state in `hEvent`. Any subsequent calls which examine the status of `hEvent` will only examine the completion of this most recent call to [cuEventRecord\(\)](#).

**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](#), [cuStreamWaitEvent](#), [cuEventDestroy](#), [cuEventElapsedTime](#)

**4.33.2.6 CUresult cuEventSynchronize (CUevent *hEvent*)**

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

If [cuEventRecord\(\)](#) has not been called on `hEvent`, `CUDA_SUCCESS` is returned immediately.

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

**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](#)

## 4.34 Execution Control

### Modules

- [Execution Control \[DEPRECATED\]](#)

### Functions

- [CUresult cuFuncGetAttribute](#) (int \*pi, [CUfunction\\_attribute](#) attrib, [CUfunction](#) hfunc)  
*Returns information about a function.*
- [CUresult cuFuncSetBlockShape](#) ([CUfunction](#) hfunc, int x, int y, int z)  
*Sets the block-dimensions for the function.*
- [CUresult cuFuncSetCacheConfig](#) ([CUfunction](#) hfunc, [CUfunc\\_cache](#) config)  
*Sets the preferred cache configuration for a device function.*
- [CUresult cuFuncSetSharedSize](#) ([CUfunction](#) hfunc, unsigned int bytes)  
*Sets the dynamic shared-memory size for the function.*
- [CUresult cuLaunch](#) ([CUfunction](#) f)  
*Launches a CUDA function.*
- [CUresult cuLaunchGrid](#) ([CUfunction](#) f, int grid\_width, int grid\_height)  
*Launches a CUDA function.*
- [CUresult cuLaunchGridAsync](#) ([CUfunction](#) f, int grid\_width, int grid\_height, [CUSTream](#) hStream)  
*Launches a CUDA function.*
- [CUresult cuParamSetf](#) ([CUfunction](#) hfunc, int offset, float value)  
*Adds a floating-point parameter to the function's argument list.*
- [CUresult cuParamSeti](#) ([CUfunction](#) hfunc, int offset, unsigned int value)  
*Adds an integer parameter to the function's argument list.*
- [CUresult cuParamSetSize](#) ([CUfunction](#) hfunc, unsigned int numbytes)  
*Sets the parameter size for the function.*
- [CUresult cuParamSetv](#) ([CUfunction](#) hfunc, int offset, void \*ptr, unsigned int numbytes)  
*Adds arbitrary data to the function's argument list.*

#### 4.34.1 Detailed Description

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

## 4.34.2 Function Documentation

### 4.34.2.1 CUresult cuFuncGetAttribute (int \**pi*, CUfunction\_attribute *attrib*, CUfunction *hfunc*)

Returns in \**pi* the integer value of the attribute *attrib* on the kernel given by *hfunc*. The supported attributes are:

- [CU\\_FUNC\\_ATTRIBUTE\\_MAX\\_THREADS\\_PER\\_BLOCK](#): The maximum number of threads per block, beyond which a launch of the function would fail. This number depends on both the function and the device on which the function is currently loaded.
- [CU\\_FUNC\\_ATTRIBUTE\\_SHARED\\_SIZE\\_BYTES](#): The size in bytes of statically-allocated shared memory per block required by this function. This does not include dynamically-allocated shared memory requested by the user at runtime.
- [CU\\_FUNC\\_ATTRIBUTE\\_CONST\\_SIZE\\_BYTES](#): The size in bytes of user-allocated constant memory required by this function.
- [CU\\_FUNC\\_ATTRIBUTE\\_LOCAL\\_SIZE\\_BYTES](#): The size in bytes of local memory used by each thread of this function.
- [CU\\_FUNC\\_ATTRIBUTE\\_NUM\\_REGS](#): The number of registers used by each thread of this function.
- [CU\\_FUNC\\_ATTRIBUTE\\_PTX\\_VERSION](#): The PTX virtual architecture version for which the function was compiled. This value is the major PTX version \* 10 + the minor PTX version, so a PTX version 1.3 function would return the value 13. Note that this may return the undefined value of 0 for cubins compiled prior to CUDA 3.0.
- [CU\\_FUNC\\_ATTRIBUTE\\_BINARY\\_VERSION](#): The binary architecture version for which the function was compiled. This value is the major binary version \* 10 + the minor binary version, so a binary version 1.3 function would return the value 13. Note that this will return a value of 10 for legacy cubins that do not have a properly-encoded binary architecture version.

#### Parameters:

*pi* - Returned attribute value

*attrib* - Attribute requested

*hfunc* - Function to query attribute of

#### Returns:

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

#### Note:

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

#### See also:

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuFuncSetCacheConfig](#), [cuParamSetSize](#), [cuParamSeti](#), [cuParamSetf](#), [cuParamSetv](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#)

#### 4.34.2.2 CUresult cuFuncSetBlockShape (CUfunction *hfunc*, 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 *hfunc* is launched.

**Parameters:**

- hfunc* - Kernel to specify dimensions of
- x* - X dimension
- y* - Y dimension
- z* - Z dimension

**Returns:**

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

**Note:**

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

**See also:**

[cuFuncSetSharedSize](#), [cuFuncSetCacheConfig](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSeti](#), [cuParamSetf](#), [cuParamSetv](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#)

#### 4.34.2.3 CUresult cuFuncSetCacheConfig (CUfunction *hfunc*, CUfunc\_cache *config*)

On devices where the L1 cache and shared memory use the same hardware resources, this sets through *config* the preferred cache configuration for the device function *hfunc*. This is only a preference. The driver will use the requested configuration if possible, but it is free to choose a different configuration if required to execute *hfunc*. Any context-wide preference set via [cuCtxSetCacheConfig\(\)](#) will be overridden by this per-function setting unless the per-function setting is [CU\\_FUNC\\_CACHE\\_PREFER\\_NONE](#). In that case, the current context-wide setting will be used.

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

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

The supported cache configurations are:

- [CU\\_FUNC\\_CACHE\\_PREFER\\_NONE](#): no preference for shared memory or L1 (default)
- [CU\\_FUNC\\_CACHE\\_PREFER\\_SHARED](#): prefer larger shared memory and smaller L1 cache
- [CU\\_FUNC\\_CACHE\\_PREFER\\_L1](#): prefer larger L1 cache and smaller shared memory

**Parameters:**

- hfunc* - Kernel to configure cache for
- config* - Requested cache configuration

**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:**

[cuCtxGetCacheConfig](#), [cuCtxSetCacheConfig](#), [cuFuncSetBlockShape](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSeti](#), [cuParamSetf](#), [cuParamSetv](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#)

**4.34.2.4 CUresult cuFuncSetSharedSize (CUfunction *hfunc*, 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 *hfunc* is launched.

**Parameters:**

*hfunc* - Kernel to specify dynamic shared-memory size for  
*bytes* - Dynamic shared-memory size per thread in bytes

**Returns:**

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

**Note:**

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

**See also:**

[cuFuncSetBlockShape](#), [cuFuncSetCacheConfig](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSeti](#), [cuParamSetf](#), [cuParamSetv](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#)

**4.34.2.5 CUresult cuLaunch (CUfunction *f*)**

Invokes the kernel *f* on a 1 x 1 x 1 grid of blocks. The block contains the number of threads specified by a previous call to [cuFuncSetBlockShape\(\)](#).

**Parameters:**

*f* - Kernel to launch

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_LAUNCH\\_FAILED](#), [CUDA\\_ERROR\\_LAUNCH\\_OUT\\_OF\\_RESOURCES](#), [CUDA\\_ERROR\\_LAUNCH\\_TIMEOUT](#), [CUDA\\_ERROR\\_LAUNCH\\_INCOMPATIBLE\\_TEXTURING](#), [CUDA\\_ERROR\\_SHARED\\_OBJECT\\_INIT\\_FAILED](#)

**Note:**

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

**See also:**

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSetf](#), [cuParamSeti](#), [cuParamSetv](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#)

#### 4.34.2.6 CUresult cuLaunchGrid (CUfunction *f*, int *grid\_width*, int *grid\_height*)

Invokes the kernel *f* on a *grid\_width* x *grid\_height* grid of blocks. Each block contains the number of threads specified by a previous call to [cuFuncSetBlockShape\(\)](#).

**Parameters:**

*f* - Kernel to launch  
*grid\_width* - Width of grid in blocks  
*grid\_height* - Height of grid in blocks

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_LAUNCH\\_FAILED](#), [CUDA\\_ERROR\\_LAUNCH\\_OUT\\_OF\\_RESOURCES](#), [CUDA\\_ERROR\\_LAUNCH\\_TIMEOUT](#), [CUDA\\_ERROR\\_LAUNCH\\_INCOMPATIBLE\\_TEXTURING](#), [CUDA\\_ERROR\\_SHARED\\_OBJECT\\_INIT\\_FAILED](#)

**Note:**

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

**See also:**

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSetf](#), [cuParamSeti](#), [cuParamSetv](#), [cuLaunch](#), [cuLaunchGridAsync](#)

#### 4.34.2.7 CUresult cuLaunchGridAsync (CUfunction *f*, int *grid\_width*, int *grid\_height*, CUstream *hStream*)

Invokes the kernel *f* on a *grid\_width* x *grid\_height* grid of blocks. Each block contains the number of threads specified by a previous call to [cuFuncSetBlockShape\(\)](#).

[cuLaunchGridAsync\(\)](#) can optionally be associated to a stream by passing a non-zero *hStream* argument.

**Parameters:**

*f* - Kernel to launch  
*grid\_width* - Width of grid in blocks  
*grid\_height* - Height of grid in blocks  
*hStream* - Stream identifier

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#), [CUDA\\_ERROR\\_INVALID\\_VALUE](#), [CUDA\\_ERROR\\_LAUNCH\\_FAILED](#), [CUDA\\_ERROR\\_LAUNCH\\_OUT\\_OF\\_RESOURCES](#), [CUDA\\_ERROR\\_LAUNCH\\_TIMEOUT](#), [CUDA\\_ERROR\\_LAUNCH\\_INCOMPATIBLE\\_TEXTURING](#), [CUDA\\_ERROR\\_SHARED\\_OBJECT\\_INIT\\_FAILED](#)

**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](#)

#### 4.34.2.8 CUresult cuParamSetf (CUfunction *hfunc*, int *offset*, float *value*)

Sets a floating-point parameter that will be specified the next time the kernel corresponding to `hfunc` will be invoked. `offset` is a byte offset.

**Parameters:**

- hfunc* - Kernel to add parameter to
- offset* - Offset to add parameter to argument list
- value* - Value of parameter

**Returns:**

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

**Note:**

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

**See also:**

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSeti](#), [cuParamSetv](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#)

#### 4.34.2.9 CUresult cuParamSeti (CUfunction *hfunc*, int *offset*, unsigned int *value*)

Sets an integer parameter that will be specified the next time the kernel corresponding to `hfunc` will be invoked. `offset` is a byte offset.

**Parameters:**

- hfunc* - Kernel to add parameter to
- offset* - Offset to add parameter to argument list
- value* - Value of parameter

**Returns:**

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

**Note:**

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

**See also:**

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSetf](#), [cuParamSetv](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#)

#### 4.34.2.10 CUresult cuParamSetSize (CUfunction *hfunc*, unsigned int *numbytes*)

Sets through *numbytes* the total size in bytes needed by the function parameters of the kernel corresponding to *hfunc*.

**Parameters:**

*hfunc* - Kernel to set parameter size for  
*numbytes* - Size of parameter list in bytes

**Returns:**

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

**Note:**

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

**See also:**

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuFuncGetAttribute](#), [cuParamSetf](#), [cuParamSeti](#), [cuParamSetv](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#)

#### 4.34.2.11 CUresult cuParamSetv (CUfunction *hfunc*, 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 *hfunc*. *offset* is a byte offset.

**Parameters:**

*hfunc* - Kernel to add data to  
*offset* - Offset to add data to argument list  
*ptr* - Pointer to arbitrary data  
*numbytes* - Size of data to copy in bytes

**Returns:**

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

**Note:**

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

**See also:**

[cuFuncSetBlockShape](#), [cuFuncSetSharedSize](#), [cuFuncGetAttribute](#), [cuParamSetSize](#), [cuParamSetf](#), [cuParamSeti](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#)

## 4.35 Execution Control [DEPRECATED]

### Functions

- [CUresult cuParamSetTexRef](#) ([CUfunction](#) hfunc, int texunit, [CUtexref](#) hTexRef)

*Adds a texture-reference to the function's argument list.*

### 4.35.1 Detailed Description

This section describes the deprecated execution control functions of the low-level CUDA driver application programming interface.

### 4.35.2 Function Documentation

#### 4.35.2.1 [CUresult cuParamSetTexRef](#) ([CUfunction](#) *hfunc*, int *texunit*, [CUtexref](#) *hTexRef*)

#### Deprecated

Makes the CUDA array or linear memory bound to the texture reference `hTexRef` available to a device program as a texture. In this version of CUDA, the texture-reference must be obtained via [cuModuleGetTexRef\(\)](#) and the `texunit` parameter must be set to [CU\\_PARAM\\_TR\\_DEFAULT](#).

#### Parameters:

- hfunc* - Kernel to add texture-reference to
- texunit* - Texture unit (must be [CU\\_PARAM\\_TR\\_DEFAULT](#))
- hTexRef* - Texture-reference to add to argument list

#### Returns:

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

#### Note:

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

## 4.36 Texture Reference Management

### Modules

- [Texture Reference Management \[DEPRECATED\]](#)

### Functions

- [CUresult cuTexRefGetAddress](#) (CUdeviceptr \*pdptr, CUtexref hTexRef)  
*Gets the address associated with a texture reference.*
- [CUresult cuTexRefGetAddressMode](#) (CUaddress\_mode \*pam, 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 \*pfm, 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 \*pNumChannels, CUtexref hTexRef)  
*Gets the format used by a texture reference.*
- [CUresult cuTexRefSetAddress](#) (size\_t \*ByteOffset, CUtexref hTexRef, CUdeviceptr dptr, size\_t bytes)  
*Binds an address as a texture reference.*
- [CUresult cuTexRefSetAddress2D](#) (CUtexref hTexRef, const CUDA\_ARRAY\_DESCRIPTOR \*desc, CUdeviceptr dptr, size\_t Pitch)  
*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 array 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 fmt, int NumPackedComponents)  
*Sets the format for a texture reference.*

### 4.36.1 Detailed Description

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

### 4.36.2 Function Documentation

#### 4.36.2.1 `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:**

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

#### 4.36.2.2 `CUresult cuTexRefGetAddressMode (CUaddress_mode * pam, CUTexref hTexRef, int dim)`

Returns in `*pam` the addressing mode corresponding to the dimension `dim` of the texture reference `hTexRef`. Currently, the only valid value for `dim` are 0 and 1.

**Parameters:**

*pam* - Returned addressing mode

*hTexRef* - Texture reference

*dim* - Dimension

**Returns:**

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

**See also:**

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

#### 4.36.2.3 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:**

`cuTexRefSetAddress`, `cuTexRefSetAddress2D`, `cuTexRefSetAddressMode`, `cuTexRefSetArray`, `cuTexRefSetFilterMode`, `cuTexRefSetFlags`, `cuTexRefSetFormat`, `cuTexRefGetAddress`, `cuTexRefGetAddressMode`, `cuTexRefGetFilterMode`, `cuTexRefGetFlags`, `cuTexRefGetFormat`

#### 4.36.2.4 CUresult cuTexRefGetFilterMode (CUfilter\_mode \* *pfm*, CUtexref *hTexRef*)

Returns in *pfm* the filtering mode of the texture reference *hTexRef*.

**Parameters:**

*pfm* - Returned filtering mode  
*hTexRef* - Texture reference

**Returns:**

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

**See also:**

`cuTexRefSetAddress`, `cuTexRefSetAddress2D`, `cuTexRefSetAddressMode`, `cuTexRefSetArray`, `cuTexRefSetFilterMode`, `cuTexRefSetFlags`, `cuTexRefSetFormat`, `cuTexRefGetAddress`, `cuTexRefGetAddressMode`, `cuTexRefGetArray`, `cuTexRefGetFlags`, `cuTexRefGetFormat`

#### 4.36.2.5 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:**

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFormat](#)

#### 4.36.2.6 CUresult cuTexRefGetFormat (CUarray\_format \*pFormat, int \*pNumChannels, CUtexref hTexRef)

Returns in \*pFormat and \*pNumChannels the format and number of components of the CUDA array bound to the texture reference hTexRef. If pFormat or pNumChannels is NULL, it will be ignored.

**Parameters:**

*pFormat* - Returned format  
*pNumChannels* - Returned number of components  
*hTexRef* - Texture reference

**Returns:**

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

**See also:**

[cuTexRefSetAddress](#), [cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#)

#### 4.36.2.7 CUresult cuTexRefSetAddress (size\_t \*ByteOffset, CUtexref hTexRef, CUdeviceptr dptr, size\_t 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 \*ByteOffset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the [tex1Dfetch\(\)](#) function.

If the device memory pointer was returned from [cuMemAlloc\(\)](#), the offset is guaranteed to be 0 and NULL may be passed as the ByteOffset parameter.

**Parameters:**

*ByteOffset* - Returned byte offset  
*hTexRef* - Texture reference to bind  
*dptr* - Device pointer to bind  
*bytes* - Size of memory to bind in bytes

**Returns:**

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

See also:

[cuTexRefSetAddress2D](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

#### 4.36.2.8 CUresult cuTexRefSetAddress2D (CUtexref *hTexRef*, const CUDA\_ARRAY\_DESCRIPTOR \* *desc*, CUdeviceptr *dptr*, size\_t *Pitch*)

Binds a linear address range to the texture reference `hTexRef`. Any previous address or CUDA array state associated with the texture reference is superseded by this function. Any memory previously bound to `hTexRef` is unbound.

Using a `tex2D()` function inside a kernel requires a call to either [cuTexRefSetArray\(\)](#) to bind the corresponding texture reference to an array, or [cuTexRefSetAddress2D\(\)](#) to bind the texture reference to linear memory.

Function calls to [cuTexRefSetFormat\(\)](#) cannot follow calls to [cuTexRefSetAddress2D\(\)](#) for the same texture reference.

It is required that `dptr` be aligned to the appropriate hardware-specific texture alignment. You can query this value using the device attribute `CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT`. If an unaligned `dptr` is supplied, `CUDA_ERROR_INVALID_VALUE` is returned.

**Parameters:**

*hTexRef* - Texture reference to bind

*desc* - Descriptor of CUDA array

*dptr* - Device pointer to bind

*Pitch* - Line pitch in bytes

**Returns:**

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

See also:

[cuTexRefSetAddress](#), [cuTexRefSetAddressMode](#), [cuTexRefSetArray](#), [cuTexRefSetFilterMode](#), [cuTexRefSetFlags](#), [cuTexRefSetFormat](#), [cuTexRefGetAddress](#), [cuTexRefGetAddressMode](#), [cuTexRefGetArray](#), [cuTexRefGetFilterMode](#), [cuTexRefGetFlags](#), [cuTexRefGetFormat](#)

#### 4.36.2.9 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,
    CU_TR_ADDRESS_MODE_BORDER = 3
} 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:**

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

**4.36.2.10 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:**

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

**4.36.2.11 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:**

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

#### 4.36.2.12 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:**

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

#### 4.36.2.13 CUresult cuTexRefSetFormat (CUtexref *hTexRef*, CUarray\_format *fmt*, int *NumPackedComponents*)

Specifies the format of the data to be read by the texture reference `hTexRef`. `fmt` and `NumPackedComponents` are exactly analogous to the `Format` and `NumChannels` members of the `CUDA_ARRAY_DESCRIPTOR` structure: They specify the format of each component and the number of components per array element.

**Parameters:**

*hTexRef* - Texture reference

*fmt* - Format to set

*NumPackedComponents* - Number of components per array element

**Returns:**

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

**See also:**

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

## 4.37 Texture Reference Management [DEPRECATED]

### Functions

- [CUresult cuTexRefCreate](#) (CUtexref \*pTexRef)  
*Creates a texture reference.*
- [CUresult cuTexRefDestroy](#) (CUtexref hTexRef)  
*Destroys a texture reference.*

### 4.37.1 Detailed Description

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

### 4.37.2 Function Documentation

#### 4.37.2.1 CUresult cuTexRefCreate (CUtexref \*pTexRef)

##### Deprecated

Creates a texture reference and returns its handle in \*pTexRef. Once created, the application must call [cuTexRefSetArray\(\)](#) or [cuTexRefSetAddress\(\)](#) to associate the reference with allocated memory. Other texture reference functions are used to specify the format and interpretation (addressing, filtering, etc.) to be used when the memory is read through this texture reference.

##### Parameters:

*pTexRef* - Returned texture reference

##### Returns:

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

##### See also:

[cuTexRefDestroy](#)

#### 4.37.2.2 CUresult cuTexRefDestroy (CUtexref hTexRef)

##### Deprecated

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](#)

## 4.38 Surface Reference Management

### Functions

- [CUresult cuSurfRefGetArray](#) ([CUarray \\*phArray](#), [CUSurfref hSurfRef](#))  
*Passes back the CUDA array bound to a surface reference.*
- [CUresult cuSurfRefSetArray](#) ([CUSurfref hSurfRef](#), [CUarray hArray](#), [unsigned int Flags](#))  
*Sets the CUDA array for a surface reference.*

### 4.38.1 Detailed Description

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

### 4.38.2 Function Documentation

#### 4.38.2.1 [CUresult cuSurfRefGetArray](#) ([CUarray \\*phArray](#), [CUSurfref hSurfRef](#))

Returns in [\\*phArray](#) the CUDA array bound to the surface reference [hSurfRef](#), or returns [CUDA\\_ERROR\\_INVALID\\_VALUE](#) if the surface reference is not bound to any CUDA array.

#### Parameters:

- phArray* - Surface reference handle
- hSurfRef* - Surface reference handle

#### Returns:

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

#### See also:

[cuModuleGetSurfRef](#), [cuSurfRefSetArray](#)

#### 4.38.2.2 [CUresult cuSurfRefSetArray](#) ([CUSurfref hSurfRef](#), [CUarray hArray](#), [unsigned int Flags](#))

Sets the CUDA array [hArray](#) to be read and written by the surface reference [hSurfRef](#). Any previous CUDA array state associated with the surface reference is superseded by this function. [Flags](#) must be set to 0. The [CUDA\\_ARRAY3D\\_SURFACE\\_LDST](#) flag must have been set for the CUDA array. Any CUDA array previously bound to [hSurfRef](#) is unbound.

#### Parameters:

- hSurfRef* - Surface reference handle
- hArray* - CUDA array handle
- Flags* - set to 0

**Returns:**

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

**See also:**

[cuModuleGetSurfRef](#), [cuSurfRefGetArray](#)

## 4.39 Graphics Interoperability

### Functions

- **CUresult cuGraphicsMapResources** (unsigned int count, CUgraphicsResource \*resources, CUstream hStream)  
*Map graphics resources for access by CUDA.*
- **CUresult cuGraphicsResourceGetMappedPointer** (CUdeviceptr \*pDevPtr, size\_t \*pSize, CUgraphicsResource resource)  
*Get a device pointer through which to access a mapped graphics resource.*
- **CUresult cuGraphicsResourceSetMapFlags** (CUgraphicsResource resource, unsigned int flags)  
*Set usage flags for mapping a graphics resource.*
- **CUresult cuGraphicsSubResourceGetMappedArray** (CUarray \*pArray, CUgraphicsResource resource, unsigned int arrayIndex, unsigned int mipLevel)  
*Get an array through which to access a subresource of a mapped graphics resource.*
- **CUresult cuGraphicsUnmapResources** (unsigned int count, CUgraphicsResource \*resources, CUstream hStream)  
*Unmap graphics resources.*
- **CUresult cuGraphicsUnregisterResource** (CUgraphicsResource resource)  
*Unregisters a graphics resource for access by CUDA.*

### 4.39.1 Detailed Description

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

### 4.39.2 Function Documentation

#### 4.39.2.1 CUresult cuGraphicsMapResources (unsigned int count, CUgraphicsResource \* resources, CUstream hStream)

Maps the count graphics resources in resources for access by CUDA.

The resources in resources may be accessed by CUDA until they are unmapped. The graphics API from which resources were registered should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any graphics calls issued before cuGraphicsMapResources() will complete before any subsequent CUDA work issued in stream begins.

If resources includes any duplicate entries then CUDA\_ERROR\_INVALID\_HANDLE is returned. If any of resources are presently mapped for access by CUDA then CUDA\_ERROR\_ALREADY\_MAPPED is returned.

#### Parameters:

**count** - Number of resources to map

*resources* - Resources to map for CUDA usage

*hStream* - Stream with which to synchronize

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_HANDLE, CUDA\_ERROR\_ALREADY\_MAPPED, CUDA\_ERROR\_UNKNOWN

**Note:**

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

**See also:**

[cuGraphicsResourceGetMappedPointer](#) [cuGraphicsSubResourceGetMappedArray](#) [cuGraphicsUnmapResources](#)

**4.39.2.2 CUresult cuGraphicsResourceGetMappedPointer (CUdeviceptr \*pDevPtr, size\_t \*pSize, CUgraphicsResource resource)**

Returns in *pDevPtr* a pointer through which the mapped graphics resource *resource* may be accessed. Returns in *pSize* the size of the memory in bytes which may be accessed from that pointer. The value set in *pPointer* may change every time that *resource* is mapped.

If *resource* is not a buffer then it cannot be accessed via a pointer and [CUDA\\_ERROR\\_NOT\\_MAPPED\\_AS\\_POINTER](#) is returned. If *resource* is not mapped then [CUDA\\_ERROR\\_NOT\\_MAPPED](#) is returned. \*

**Parameters:**

*pDevPtr* - Returned pointer through which *resource* may be accessed

*pSize* - Returned size of the buffer accessible starting at *pPointer*

*resource* - Mapped resource to access

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_INVALID\_HANDLE, CUDA\_ERROR\_NOT\_MAPPED, CUDA\_ERROR\_NOT\_MAPPED\_AS\_POINTER

**Note:**

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

**See also:**

[cuGraphicsMapResources](#), [cuGraphicsSubResourceGetMappedArray](#)

**4.39.2.3 CUresult cuGraphicsResourceSetMapFlags (CUgraphicsResource resource, unsigned int flags)**

Set *flags* for mapping the graphics resource *resource*.

Changes to *flags* will take effect the next time *resource* is mapped. The *flags* argument may be any of the following:

- `CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE`: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- `CU_GRAPHICS_MAP_RESOURCE_FLAGS_READONLY`: Specifies that CUDA kernels which access this resource will not write to this resource.
- `CU_GRAPHICS_MAP_RESOURCE_FLAGS_WRITEDISCARD`: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If `resource` is presently mapped for access by CUDA then `CUDA_ERROR_ALREADY_MAPPED` is returned. If `flags` is not one of the above values then `CUDA_ERROR_INVALID_VALUE` is returned.

**Parameters:**

*resource* - Registered resource to set flags for

*flags* - Parameters for resource mapping

**Returns:**

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_INVALID_HANDLE`, `CUDA_ERROR_ALREADY_MAPPED`

**Note:**

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

**See also:**

[cuGraphicsMapResources](#)

**4.39.2.4 CUresult cuGraphicsSubResourceGetMappedArray (CUarray \*pArray, CUgraphicsResource resource, unsigned int arrayIndex, unsigned int mipLevel)**

Returns in `*pArray` an array through which the subresource of the mapped graphics resource `resource` which corresponds to array index `arrayIndex` and mipmap level `mipLevel` may be accessed. The value set in `*pArray` may change every time that `resource` is mapped.

If `resource` is not a texture then it cannot be accessed via an array and `CUDA_ERROR_NOT_MAPPED_AS_ARRAY` is returned. If `arrayIndex` is not a valid array index for `resource` then `CUDA_ERROR_INVALID_VALUE` is returned. If `mipLevel` is not a valid mipmap level for `resource` then `CUDA_ERROR_INVALID_VALUE` is returned. If `resource` is not mapped then `CUDA_ERROR_NOT_MAPPED` is returned.

**Parameters:**

*pArray* - Returned array through which a subresource of `resource` may be accessed

*resource* - Mapped resource to access

*arrayIndex* - Array index for array textures or cubemap face index as defined by [CUarray\\_cubemap\\_face](#) for cubemap textures for the subresource to access

*mipLevel* - Mipmap level for the subresource to access

**Returns:**

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

**Note:**

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

**See also:**

[cuGraphicsResourceGetMappedPointer](#)

**4.39.2.5 CUresult cuGraphicsUnmapResources (unsigned int *count*, CUgraphicsResource \* *resources*, CUstream *hStream*)**

Unmaps the *count* graphics resources in *resources*.

Once unmapped, the resources in *resources* may not be accessed by CUDA until they are mapped again.

This function provides the synchronization guarantee that any CUDA work issued in *stream* before [cuGraphicsUnmapResources\(\)](#) will complete before any subsequently issued graphics work begins.

If *resources* includes any duplicate entries then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If any of *resources* are not presently mapped for access by CUDA then [CUDA\\_ERROR\\_NOT\\_MAPPED](#) is returned.

**Parameters:**

*count* - Number of resources to unmap

*resources* - Resources to unmap

*hStream* - Stream with which to synchronize

**Returns:**

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

**Note:**

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

**See also:**

[cuGraphicsMapResources](#)

**4.39.2.6 CUresult cuGraphicsUnregisterResource (CUgraphicsResource *resource*)**

Unregisters the graphics resource *resource* so it is not accessible by CUDA unless registered again.

If *resource* is invalid then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned.

**Parameters:**

*resource* - Resource to unregister

**Returns:**

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

**Note:**

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

**See also:**

cuGraphicsD3D9RegisterResource, cuGraphicsD3D10RegisterResource, cuGraphicsD3D11RegisterResource, cuGraphicsGLRegisterBuffer, cuGraphicsGLRegisterImage

## 4.40 OpenGL Interoperability

### Modules

- [OpenGL Interoperability \[DEPRECATED\]](#)

### Functions

- [CUresult cuGLCtxCreate](#) ([CUcontext](#) \*pCtx, unsigned int Flags, [CUdevice](#) device)  
*Create a CUDA context for interoperability with OpenGL.*
- [CUresult cuGraphicsGLRegisterBuffer](#) ([CUgraphicsResource](#) \*pCudaResource, GLuint buffer, unsigned int Flags)  
*Registers an OpenGL buffer object.*
- [CUresult cuGraphicsGLRegisterImage](#) ([CUgraphicsResource](#) \*pCudaResource, GLuint image, GLenum target, unsigned int Flags)  
*Register an OpenGL texture or renderbuffer object.*
- [CUresult cuWGLGetDevice](#) ([CUdevice](#) \*pDevice, HGPUNV hGpu)  
*Gets the CUDA device associated with hGpu.*

### 4.40.1 Detailed Description

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

### 4.40.2 Function Documentation

#### 4.40.2.1 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](#), [cuGLMapBufferObjectAsync](#), [cuGLUnmapBufferObjectAsync](#), [cuGLSetBufferObjectMapFlags](#), [cuWGLGetDevice](#)

**4.40.2.2 CUresult cuGraphicsGLRegisterBuffer (CUgraphicsResource \* pCudaResource, GLuint buffer, unsigned int Flags)**

Registers the buffer object specified by `buffer` for access by CUDA. A handle to the registered object is returned as `pCudaResource`. The map flags `Flags` specify the intended usage, as follows:

- `CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE`: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- `CU_GRAPHICS_MAP_RESOURCE_FLAGS_READ_ONLY`: Specifies that CUDA will not write to this resource.
- `CU_GRAPHICS_MAP_RESOURCE_FLAGS_WRITE_DISCARD`: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

**Parameters:**

*pCudaResource* - Pointer to the returned object handle

*buffer* - name of buffer object to be registered

*Flags* - Map flags

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#), [CUDA\\_ERROR\\_ALREADY\\_MAPPED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#),

**Note:**

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

**See also:**

[cuGLCtxCreate](#), [cuGraphicsUnregisterResource](#), [cuGraphicsMapResources](#), [cuGraphicsResourceGetMappedPointer](#)

**4.40.2.3 CUresult cuGraphicsGLRegisterImage (CUgraphicsResource \* pCudaResource, GLuint image, GLenum target, unsigned int Flags)**

Registers the texture or renderbuffer object specified by `image` for access by CUDA. `target` must match the type of the object. A handle to the registered object is returned as `pCudaResource`. The map flags `Flags` specify the intended usage, as follows:

- `CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE`: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- `CU_GRAPHICS_MAP_RESOURCE_FLAGS_READ_ONLY`: Specifies that CUDA will not write to this resource.

- `CU_GRAPHICS_MAP_RESOURCE_FLAGS_WRITE_DISCARD`: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

The following image classes are currently disallowed:

- Textures with borders
- Multisampled renderbuffers

**Parameters:**

*pCudaResource* - Pointer to the returned object handle

*image* - name of texture or renderbuffer object to be registered

*target* - Identifies the type of object specified by *image*, and must be one of `GL_TEXTURE_2D`, `GL_TEXTURE_RECTANGLE`, `GL_TEXTURE_CUBE_MAP`, `GL_TEXTURE_3D`, `GL_TEXTURE_2D_ARRAY`, or `GL_RENDERBUFFER`.

*Flags* - Map flags

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_INVALID\\_HANDLE](#), [CUDA\\_ERROR\\_ALREADY\\_MAPPED](#), [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#),

**Note:**

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

**See also:**

[cuGLCtxCreate](#), [cuGraphicsUnregisterResource](#), [cuGraphicsMapResources](#), [cuGraphicsSubResourceGetMappedArray](#)

#### 4.40.2.4 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](#), [cuGLUnmapBufferObjectAsync](#), [cuGLSetBufferObjectMapFlags](#)

## 4.41 OpenGL Interoperability [DEPRECATED]

### Typedefs

- typedef enum [CUGLmap\\_flags\\_enum](#) [CUGLmap\\_flags](#)

### Enumerations

- enum [CUGLmap\\_flags\\_enum](#)

### Functions

- [CUresult cuGLInit](#) (void)  
*Initializes OpenGL interoperability.*
- [CUresult cuGLMapBufferObject](#) ([CUdeviceptr](#) \*dptr, [size\\_t](#) \*size, [GLuint](#) buffer)  
*Maps an OpenGL buffer object.*
- [CUresult cuGLMapBufferObjectAsync](#) ([CUdeviceptr](#) \*dptr, [size\\_t](#) \*size, [GLuint](#) buffer, [CUstream](#) hStream)  
*Maps an OpenGL buffer object.*
- [CUresult cuGLRegisterBufferObject](#) ([GLuint](#) buffer)  
*Registers an OpenGL buffer object.*
- [CUresult cuGLSetBufferObjectMapFlags](#) ([GLuint](#) buffer, unsigned int Flags)  
*Set the map flags for an OpenGL buffer object.*
- [CUresult cuGLUnmapBufferObject](#) ([GLuint](#) buffer)  
*Unmaps an OpenGL buffer object.*
- [CUresult cuGLUnmapBufferObjectAsync](#) ([GLuint](#) buffer, [CUstream](#) hStream)  
*Unmaps an OpenGL buffer object.*
- [CUresult cuGLUnregisterBufferObject](#) ([GLuint](#) buffer)  
*Unregister an OpenGL buffer object.*

#### 4.41.1 Detailed Description

This section describes deprecated OpenGL interoperability functionality.

#### 4.41.2 Typedef Documentation

##### 4.41.2.1 typedef enum [CUGLmap\\_flags\\_enum](#) [CUGLmap\\_flags](#)

Flags to map or unmap a resource

### 4.41.3 Enumeration Type Documentation

#### 4.41.3.1 enum CUGLmap\_flags\_enum

Flags to map or unmap a resource

### 4.41.4 Function Documentation

#### 4.41.4.1 CUresult cuGLInit (void)

##### Deprecated

This function is deprecated as of Cuda 3.0.

Initializes OpenGL interoperability. This function is deprecated and calling it is no longer required. It may fail if the needed OpenGL driver facilities are not available.

##### 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](#), [cuGLMapBufferObjectAsync](#), [cuGLUnmapBufferObjectAsync](#), [cuGLSetBufferObjectMapFlags](#), [cuWGLGetDevice](#)

#### 4.41.4.2 CUresult cuGLMapBufferObject (CUdeviceptr \* *dptr*, size\_t \* *size*, GLuint *buffer*)

##### Deprecated

This function is deprecated as of Cuda 3.0.

Maps the buffer object specified by *buffer* into the address space of the current CUDA context and returns in *\*dptr* and *\*size* the base pointer and size of the resulting mapping.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

All streams in the current CUDA context are synchronized with the current GL context.

##### Parameters:

*dptr* - Returned mapped base pointer  
*size* - Returned size of mapping  
*buffer* - The name of the buffer object to map

##### Returns:

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

**Note:**

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

**See also:**

[cuGraphicsMapResources](#)

**4.41.4.3 CUresult cuGLMapBufferObjectAsync (CUdeviceptr \* *dptr*, size\_t \* *size*, GLuint *buffer*, CUstream *hStream*)****Deprecated**

This function is deprecated as of Cuda 3.0.

Maps the buffer object specified by *buffer* into the address space of the current CUDA context and returns in *\*dptr* and *\*size* the base pointer and size of the resulting mapping.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same *shareGroup*, as the context that was bound when the buffer was registered.

Stream *hStream* in the current CUDA context is synchronized with the current GL context.

**Parameters:**

*dptr* - Returned mapped base pointer

*size* - Returned size of mapping

*buffer* - The name of the buffer object to map

*hStream* - Stream to synchronize

**Returns:**

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

**Note:**

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

**See also:**

[cuGraphicsMapResources](#)

**4.41.4.4 CUresult cuGLRegisterBufferObject (GLuint *buffer*)****Deprecated**

This function is deprecated as of Cuda 3.0.

Registers the buffer object specified by *buffer* for access by CUDA. This function must be called before CUDA can map the buffer object. There must be a valid OpenGL context bound to the current thread when this function is called, and the buffer name is resolved by that context.

**Parameters:**

*buffer* - The name of the buffer object to register.

**Returns:**

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

**Note:**

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

**See also:**

[cuGraphicsGLRegisterBuffer](#)

#### 4.41.4.5 CUresult cuGLSetBufferObjectMapFlags (GLuint *buffer*, unsigned int *Flags*)

**Deprecated**

This function is deprecated as of Cuda 3.0.

Sets the map flags for the buffer object specified by *buffer*.

Changes to *Flags* will take effect the next time *buffer* is mapped. The *Flags* argument may be any of the following:

- [CU\\_GL\\_MAP\\_RESOURCE\\_FLAGS\\_NONE](#): Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- [CU\\_GL\\_MAP\\_RESOURCE\\_FLAGS\\_READ\\_ONLY](#): Specifies that CUDA kernels which access this resource will not write to this resource.
- [CU\\_GL\\_MAP\\_RESOURCE\\_FLAGS\\_WRITE\\_DISCARD](#): Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If *buffer* has not been registered for use with CUDA, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If *buffer* is presently mapped for access by CUDA, then [CUDA\\_ERROR\\_ALREADY\\_MAPPED](#) is returned.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

**Parameters:**

*buffer* - Buffer object to unmap

*Flags* - Map flags

**Returns:**

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

**Note:**

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

**See also:**

[cuGraphicsResourceSetMapFlags](#)

#### 4.41.4.6 CUresult cuGLUnmapBufferObject (GLuint *buffer*)

##### Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the buffer object specified by *buffer* for access by CUDA.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

All streams in the current CUDA context are synchronized with the current GL context.

##### Parameters:

*buffer* - 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:

[cuGraphicsUnmapResources](#)

#### 4.41.4.7 CUresult cuGLUnmapBufferObjectAsync (GLuint *buffer*, CUstream *hStream*)

##### Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the buffer object specified by *buffer* for access by CUDA.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

Stream *hStream* in the current CUDA context is synchronized with the current GL context.

##### Parameters:

*buffer* - Name of the buffer object to unmap

*hStream* - Stream to synchronize

##### Returns:

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

##### Note:

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

##### See also:

[cuGraphicsUnmapResources](#)

#### 4.41.4.8 CUresult cuGLUnregisterBufferObject (GLuint *buffer*)

##### Deprecated

This function is deprecated as of Cuda 3.0.

Unregisters the buffer object specified by `buffer`. This releases any resources associated with the registered buffer. After this call, the buffer may no longer be mapped for access by CUDA.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same `shareGroup`, as the context that was bound when the buffer was registered.

##### Parameters:

*buffer* - Name of the buffer object to unregister

##### Returns:

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

##### Note:

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

##### See also:

[cuGraphicsUnregisterResource](#)

## 4.42 Direct3D 9 Interoperability

### Modules

- [Direct3D 9 Interoperability \[DEPRECATED\]](#)

### Typedefs

- typedef enum [CUd3d9DeviceList\\_enum](#) [CUd3d9DeviceList](#)

### Enumerations

- enum [CUd3d9DeviceList\\_enum](#) {  
[CU\\_D3D9\\_DEVICE\\_LIST\\_ALL](#) = 0x01,  
[CU\\_D3D9\\_DEVICE\\_LIST\\_CURRENT\\_FRAME](#) = 0x02,  
[CU\\_D3D9\\_DEVICE\\_LIST\\_NEXT\\_FRAME](#) = 0x03 }

### Functions

- [CUresult cuD3D9CtxCreate](#) ([CUcontext](#) \*pCtx, [CUdevice](#) \*pCudaDevice, unsigned int Flags, [IDirect3DDevice9](#) \*pD3DDevice)  
*Create a CUDA context for interoperability with Direct3D 9.*
- [CUresult cuD3D9CtxCreateOnDevice](#) ([CUcontext](#) \*pCtx, unsigned int flags, [IDirect3DDevice9](#) \*pD3DDevice, [CUdevice](#) cudaDevice)  
*Create a CUDA context for interoperability with Direct3D 9.*
- [CUresult cuD3D9GetDevice](#) ([CUdevice](#) \*pCudaDevice, const char \*pszAdapterName)  
*Gets the CUDA device corresponding to a display adapter.*
- [CUresult cuD3D9GetDevices](#) (unsigned int \*pCudaDeviceCount, [CUdevice](#) \*pCudaDevices, unsigned int cudaDeviceCount, [IDirect3DDevice9](#) \*pD3D9Device, [CUd3d9DeviceList](#) deviceList)  
*Gets the CUDA devices corresponding to a Direct3D 9 device.*
- [CUresult cuD3D9GetDirect3DDevice](#) ([IDirect3DDevice9](#) \*\*ppD3DDevice)  
*Get the Direct3D 9 device against which the current CUDA context was created.*
- [CUresult cuGraphicsD3D9RegisterResource](#) ([CUgraphicsResource](#) \*pCudaResource, [IDirect3DResource9](#) \*pD3DResource, unsigned int Flags)  
*Register a Direct3D 9 resource for access by CUDA.*

#### 4.42.1 Detailed Description

This section describes the Direct3D 9 interoperability functions of the low-level CUDA driver application programming interface.

## 4.42.2 Typedef Documentation

### 4.42.2.1 typedef enum CUd3d9DeviceList\_enum CUd3d9DeviceList

CUDA devices corresponding to a D3D9 device

## 4.42.3 Enumeration Type Documentation

### 4.42.3.1 enum CUd3d9DeviceList\_enum

CUDA devices corresponding to a D3D9 device

#### Enumerator:

*CU\_D3D9\_DEVICE\_LIST\_ALL* The CUDA devices for all GPUs used by a D3D9 device

*CU\_D3D9\_DEVICE\_LIST\_CURRENT\_FRAME* The CUDA devices for the GPUs used by a D3D9 device in its currently rendering frame

*CU\_D3D9\_DEVICE\_LIST\_NEXT\_FRAME* The CUDA devices for the GPUs to be used by a D3D9 device in the next frame

## 4.42.4 Function Documentation

### 4.42.4.1 CUresult cuD3D9CtxCreate (CUcontext \* pCtx, CUdevice \* pCudaDevice, unsigned int Flags, IDirect3DDevice9 \* pD3DDevice)

Creates a new CUDA context, enables interoperability for that context with the Direct3D device *pD3DDevice*, and associates the created CUDA context with the calling thread. The created [CUcontext](#) will be returned in *\*pCtx*. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context. If *pCudaDevice* is non-NULL then the [CUdevice](#) on which this CUDA context was created will be returned in *\*pCudaDevice*.

On success, this call will increase the internal reference count on *pD3DDevice*. This reference count will be decremented upon destruction of this context through [cuCtxDestroy\(\)](#). This context will cease to function if *pD3DDevice* is destroyed or encounters an error.

#### Parameters:

*pCtx* - Returned newly created CUDA context

*pCudaDevice* - Returned pointer to the device on which the context was created

*Flags* - Context creation flags (see [cuCtxCreate\(\)](#) for details)

*pD3DDevice* - Direct3D device to create interoperability context with

#### Returns:

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

#### Note:

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

#### See also:

[cuD3D9GetDevice](#), [cuGraphicsD3D9RegisterResource](#)

#### 4.42.4.2 CUresult cuD3D9CtxCreateOnDevice (CUcontext \* *pCtx*, unsigned int *flags*, IDirect3DDevice9 \* *pD3DDevice*, CUdevice *cudaDevice*)

Creates a new CUDA context, enables interoperability for that context with the Direct3D device *pD3DDevice*, and associates the created CUDA context with the calling thread. The created [CUcontext](#) will be returned in *\*pCtx*. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context.

On success, this call will increase the internal reference count on *pD3DDevice*. This reference count will be decremented upon destruction of this context through [cuCtxDestroy\(\)](#). This context will cease to function if *pD3DDevice* is destroyed or encounters an error.

##### Parameters:

*pCtx* - Returned newly created CUDA context

*flags* - Context creation flags (see [cuCtxCreate\(\)](#) for details)

*pD3DDevice* - Direct3D device to create interoperability context with

*cudaDevice* - The CUDA device on which to create the context. This device must be among the devices returned when querying CU\_D3D9\_DEVICES\_ALL from [cuD3D9GetDevices](#).

##### Returns:

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

##### Note:

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

##### See also:

[cuD3D9GetDevices](#), [cuGraphicsD3D9RegisterResource](#)

#### 4.42.4.3 CUresult cuD3D9GetDevice (CUdevice \* *pCudaDevice*, const char \* *pszAdapterName*)

Returns in *\*pCudaDevice* the CUDA-compatible device corresponding to the adapter name *pszAdapterName* obtained from [EnumDisplayDevices\(\)](#) or [IDirect3D9::GetAdapterIdentifier\(\)](#).

If no device on the adapter with name *pszAdapterName* is CUDA-compatible, then the call will fail.

##### Parameters:

*pCudaDevice* - Returned CUDA device corresponding to *pszAdapterName*

*pszAdapterName* - Adapter name to query for device

##### Returns:

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

##### Note:

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

##### See also:

[cuD3D9CtxCreate](#)

#### 4.42.4.4 CUresult cuD3D9GetDevices (unsigned int \* pCudaDeviceCount, CUdevice \* pCudaDevices, unsigned int cudaDeviceCount, IDirect3DDevice9 \* pD3D9Device, CUd3d9DeviceList deviceList)

Returns in \*pCudaDeviceCount the number of CUDA-compatible device corresponding to the Direct3D 9 device pD3D9Device. Also returns in \*pCudaDevices at most cudaDeviceCount of the the CUDA-compatible devices corresponding to the Direct3D 9 device pD3D9Device.

If any of the GPUs being used to render pDevice are not CUDA capable then the call will return [CUDA\\_ERROR\\_NO\\_DEVICE](#).

##### Parameters:

*pCudaDeviceCount* - Returned number of CUDA devices corresponding to pD3D9Device

*pCudaDevices* - Returned CUDA devices corresponding to pD3D9Device

*cudaDeviceCount* - The size of the output device array pCudaDevices

*pD3D9Device* - Direct3D 9 device to query for CUDA devices

*deviceList* - The set of devices to return. This set may be [CU\\_D3D9\\_DEVICE\\_LIST\\_ALL](#) for all devices, [CU\\_D3D9\\_DEVICE\\_LIST\\_CURRENT\\_FRAME](#) for the devices used to render the current frame (in SLI), or [CU\\_D3D9\\_DEVICE\\_LIST\\_NEXT\\_FRAME](#) for the devices used to render the next frame (in SLI).

##### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_NO\\_DEVICE](#), [CUDA\\_ERROR\\_UNKNOWN](#)

##### Note:

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

##### See also:

[cuD3D9CtxCreate](#)

#### 4.42.4.5 CUresult cuD3D9GetDirect3DDevice (IDirect3DDevice9 \*\* ppD3DDevice)

Returns in \*ppD3DDevice the Direct3D device against which this CUDA context was created in [cuD3D9CtxCreate\(\)](#).

##### Parameters:

*ppD3DDevice* - 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:

[cuD3D9GetDevice](#)

#### 4.42.4.6 CUresult cuGraphicsD3D9RegisterResource (CUgraphicsResource \* pCudaResource, IDirect3DResource9 \* pD3DResource, unsigned int Flags)

Registers the Direct3D 9 resource `pD3DResource` for access by CUDA and returns a CUDA handle to `pD3DResource` in `pCudaResource`. The handle returned in `pCudaResource` may be used to map and unmap this resource until it is unregistered. On success this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through `cuGraphicsUnregisterResource()`.

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

The type of `pD3DResource` must be one of the following.

- `IDirect3DVertexBuffer9`: may be accessed through a device pointer
- `IDirect3DIndexBuffer9`: may be accessed through a device pointer
- `IDirect3DSurface9`: may be accessed through an array. Only stand-alone objects of type `IDirect3DSurface9` may be explicitly shared. In particular, individual mipmap levels and faces of cube maps may not be registered directly. To access individual surfaces associated with a texture, one must register the base texture object.
- `IDirect3DBaseTexture9`: individual surfaces on this texture may be accessed through an array.

The `Flags` argument may be used to specify additional parameters at register time. The only valid value for this parameter is

- `CU_GRAPHICS_REGISTER_FLAGS_NONE`

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 for this context using `cuD3D9CtxCreate` then `CUDA_ERROR_INVALID_CONTEXT` is returned. If `pD3DResource` is of incorrect type or is already registered then `CUDA_ERROR_INVALID_HANDLE` is returned. If `pD3DResource` cannot be registered then `CUDA_ERROR_UNKNOWN` is returned. If `Flags` is not one of the above specified value then `CUDA_ERROR_INVALID_VALUE` is returned.

#### Parameters:

*pCudaResource* - Returned graphics resource handle

*pD3DResource* - Direct3D resource to register

*Flags* - Parameters for resource registration

#### Returns:

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_INVALID_HANDLE`, `CUDA_ERROR_OUT_OF_MEMORY`, `CUDA_ERROR_UNKNOWN`

**Note:**

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

**See also:**

[cuD3D9CtxCreate](#), [cuGraphicsUnregisterResource](#), [cuGraphicsMapResources](#), [cuGraphicsSubResourceGetMappedArray](#), [cuGraphicsResourceGetMappedPointer](#)

## 4.43 Direct3D 9 Interoperability [DEPRECATED]

### Typedefs

- typedef enum [CUd3d9map\\_flags\\_enum](#) [CUd3d9map\\_flags](#)
- typedef enum [CUd3d9register\\_flags\\_enum](#) [CUd3d9register\\_flags](#)

### Enumerations

- enum [CUd3d9map\\_flags\\_enum](#)
- enum [CUd3d9register\\_flags\\_enum](#)

### Functions

- [CUresult cuD3D9MapResources](#) (unsigned int count, IDirect3DResource9 \*\*ppResource)  
*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, 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.*
- [CUresult cuD3D9ResourceGetMappedPitch](#) (size\_t \*pPitch, size\_t \*pPitchSlice, IDirect3DResource9 \*pResource, unsigned int Face, unsigned int Level)  
*Get the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- [CUresult cuD3D9ResourceGetMappedPointer](#) (CUdeviceptr \*pDevPtr, IDirect3DResource9 \*pResource, unsigned int Face, unsigned int Level)  
*Get the pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- [CUresult cuD3D9ResourceGetMappedSize](#) (size\_t \*pSize, IDirect3DResource9 \*pResource, unsigned int Face, unsigned int Level)  
*Get the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- [CUresult cuD3D9ResourceGetSurfaceDimensions](#) (size\_t \*pWidth, size\_t \*pHeight, size\_t \*pDepth, IDirect3DResource9 \*pResource, unsigned int Face, unsigned int Level)  
*Get the dimensions of a registered surface.*
- [CUresult cuD3D9ResourceSetMapFlags](#) (IDirect3DResource9 \*pResource, unsigned int Flags)  
*Set usage flags for mapping a Direct3D resource.*
- [CUresult cuD3D9UnmapResources](#) (unsigned int count, IDirect3DResource9 \*\*ppResource)  
*Unmaps Direct3D resources.*
- [CUresult cuD3D9UnregisterResource](#) (IDirect3DResource9 \*pResource)  
*Unregister a Direct3D resource.*

### 4.43.1 Detailed Description

This section describes deprecated Direct3D 9 interoperability functionality.

### 4.43.2 Typedef Documentation

#### 4.43.2.1 typedef enum CUd3d9map\_flags\_enum CUd3d9map\_flags

Flags to map or unmap a resource

#### 4.43.2.2 typedef enum CUd3d9register\_flags\_enum CUd3d9register\_flags

Flags to register a resource

### 4.43.3 Enumeration Type Documentation

#### 4.43.3.1 enum CUd3d9map\_flags\_enum

Flags to map or unmap a resource

#### 4.43.3.2 enum CUd3d9register\_flags\_enum

Flags to register a resource

### 4.43.4 Function Documentation

#### 4.43.4.1 CUresult cuD3D9MapResources (unsigned int *count*, IDirect3DResource9 \*\* *ppResource*)

#### Deprecated

This function is deprecated as of Cuda 3.0.

Maps the *count* Direct3D resources in *ppResource* for access by CUDA.

The resources in *ppResource* may be accessed in CUDA kernels until they are unmapped. Direct3D should not access any resources while they are mapped by CUDA. If an application does so the results are undefined.

This function provides the synchronization guarantee that any Direct3D calls issued before `cuD3D9MapResources()` will complete before any CUDA kernels issued after `cuD3D9MapResources()` begin.

If any of *ppResource* have not been registered for use with CUDA or if *ppResource* contains any duplicate entries, then `CUDA_ERROR_INVALID_HANDLE` is returned. If any of *ppResource* are presently mapped for access by CUDA, then `CUDA_ERROR_ALREADY_MAPPED` is returned.

#### Parameters:

*count* - Number of resources in *ppResource*

*ppResource* - Resources to map for CUDA usage

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_HANDLE, CUDA\_ERROR\_ALREADY\_MAPPED, CUDA\_ERROR\_UNKNOWN

**Note:**

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

**See also:**

[cuGraphicsMapResources](#)

**4.43.4.2 CUresult cuD3D9RegisterResource (IDirect3DResource9 \*pResource, unsigned int Flags)****Deprecated**

This function is deprecated as of Cuda 3.0.

Registers the Direct3D resource `pResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through [cuD3D9UnregisterResource\(\)](#). Also on success, this call will increase the internal reference count on `pResource`. This reference count will be decremented when this resource is unregistered through [cuD3D9UnregisterResource\(\)](#).

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

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

- `IDirect3DVertexBuffer9`: Cannot be used with `Flags` set to `CU_D3D9_REGISTER_FLAGS_ARRAY`.
- `IDirect3DIndexBuffer9`: Cannot be used with `Flags` set to `CU_D3D9_REGISTER_FLAGS_ARRAY`.
- `IDirect3DSurface9`: Only stand-alone objects of type `IDirect3DSurface9` may be explicitly shared. In particular, individual mipmap levels and faces of cube maps may not be registered directly. To access individual surfaces associated with a texture, one must register the base texture object. For restrictions on the `Flags` parameter, see type `IDirect3DBaseTexture9`.
- `IDirect3DBaseTexture9`: When a texture is registered, all surfaces associated with the all mipmap levels of all faces of the texture will be accessible to CUDA.

The `Flags` argument specifies the mechanism through which CUDA will access the Direct3D resource. The following values are allowed.

- `CU_D3D9_REGISTER_FLAGS_NONE`: Specifies that CUDA will access this resource through a [CUDevicePtr](#). The pointer, size, and (for textures), pitch for each subresource of this allocation may be queried through [cuD3D9ResourceGetMappedPointer\(\)](#), [cuD3D9ResourceGetMappedSize\(\)](#), and [cuD3D9ResourceGetMappedPitch\(\)](#) respectively. This option is valid for all resource types.
- `CU_D3D9_REGISTER_FLAGS_ARRAY`: Specifies that CUDA will access this resource through a [CUarray](#) queried on a sub-resource basis through [cuD3D9ResourceGetMappedArray\(\)](#). This option is only valid for resources of type `IDirect3DSurface9` and subtypes of `IDirect3DBaseTexture9`.

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

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Any resources allocated in D3DPOOL\_SYSTEMMEM or D3DPOOL\_MANAGED may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized on this context, then [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#) is returned. If `pResource` is of incorrect type (e.g. is a non-stand-alone IDirect3DSurface9) or is already registered, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If `pResource` cannot be registered then [CUDA\\_ERROR\\_UNKNOWN](#) is returned.

**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:**

[cuGraphicsD3D9RegisterResource](#)

**4.43.4.3 CUresult cuD3D9ResourceGetMappedArray (CUarray \*pArray, IDirect3DResource9 \*pResource, unsigned int Face, unsigned int Level)**

**Deprecated**

This function is deprecated as of Cuda 3.0.

Returns in `*pArray` an array through which the subresource of the mapped Direct3D resource `pResource` which corresponds to `Face` and `Level` may be accessed. The value set in `pArray` may change every time that `pResource` is mapped.

If `pResource` is not registered then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If `pResource` was not registered with usage flags `CU_D3D9_REGISTER_FLAGS_ARRAY` then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If `pResource` is not mapped then [CUDA\\_ERROR\\_NOT\\_MAPPED](#) is returned.

For usage requirements of `Face` and `Level` parameters, see [cuD3D9ResourceGetMappedPointer\(\)](#).

**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:**

[cuGraphicsSubResourceGetMappedArray](#)

**4.43.4.4 CUresult cuD3D9ResourceGetMappedPitch (size\_t \* pPitch, size\_t \* pPitchSlice, IDirect3DResource9 \* pResource, unsigned int Face, unsigned int Level)**

**Deprecated**

This function is deprecated as of Cuda 3.0.

Returns in \*pPitch and \*pPitchSlice the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource pResource, which corresponds to Face and Level. The values set in pPitch and pPitchSlice may change every time that pResource is mapped.

The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position **x**, **y** from the base pointer of the surface is:

**y \* pitch + (bytes per pixel) \* x**

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

**z\* slicePitch + y \* pitch + (bytes per pixel) \* x**

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

If pResource is not of type IDirect3DBaseTexture9 or one of its sub-types or if pResource has not been registered for use with CUDA, then [cudaErrorInvalidResourceHandle](#) is returned. If pResource was not registered with usage flags 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:**

[cuGraphicsSubResourceGetMappedArray](#)

#### 4.43.4.5 CUresult cuD3D9ResourceGetMappedPointer (CUdeviceptr \* *pDevPtr*, IDirect3DResource9 \* *pResource*, unsigned int *Face*, unsigned int *Level*)

**Deprecated**

This function is deprecated as of Cuda 3.0.

Returns in \**pDevPtr* the base pointer of the subresource of the mapped Direct3D resource *pResource*, which corresponds to *Face* and *Level*. The value set in *pDevPtr* may change every time that *pResource* is mapped.

If *pResource* is not registered, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If *pResource* was not registered with usage flags `CU_D3D9_REGISTER_FLAGS_NONE`, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If *pResource* is not mapped, then [CUDA\\_ERROR\\_NOT\\_MAPPED](#) is returned.

If *pResource* is of type `IDirect3DCubeTexture9`, then *Face* must one of the values enumerated by type `D3DCUBEMAP_FACES`. For all other types *Face* must be 0. If *Face* is invalid, then [CUDA\\_ERROR\\_INVALID\\_VALUE](#) is returned.

If *pResource* is of type `IDirect3DBaseTexture9`, then *Level* must correspond to a valid mipmap level. At present only mipmap level 0 is supported. For all other types *Level* must be 0. If *Level* is invalid, then [CUDA\\_ERROR\\_INVALID\\_VALUE](#) is returned.

**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:**

[cuGraphicsResourceGetMappedPointer](#)

#### 4.43.4.6 CUresult cuD3D9ResourceGetMappedSize (size\_t \* *pSize*, IDirect3DResource9 \* *pResource*, unsigned int *Face*, unsigned int *Level*)

**Deprecated**

This function is deprecated as of Cuda 3.0.

Returns in *pSize* the size of the subresource of the mapped Direct3D resource *pResource*, which corresponds to *Face* and *Level*. The value set in *pSize* may change every time that *pResource* is mapped.

If *pResource* has not been registered for use with CUDA, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If *pResource* was not registered with usage flags `CU_D3D9_REGISTER_FLAGS_NONE`, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If *pResource* is not mapped for access by CUDA, then [CUDA\\_ERROR\\_NOT\\_MAPPED](#) is returned.

For usage requirements of *Face* and *Level* parameters, see [cuD3D9ResourceGetMappedPointer](#).

**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:**

[cuGraphicsResourceGetMappedPointer](#)

#### 4.43.4.7 CUresult cuD3D9ResourceGetSurfaceDimensions (size\_t \* *pWidth*, size\_t \* *pHeight*, size\_t \* *pDepth*, IDirect3DResource9 \* *pResource*, unsigned int *Face*, unsigned int *Level*)

**Deprecated**

This function is deprecated as of Cuda 3.0.

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

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

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

If *pResource* is not of type `IDirect3DBaseTexture9` or `IDirect3DSurface9` or if *pResource* has not been registered for use with CUDA, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned.

For usage requirements of *Face* and *Level* parameters, see [cuD3D9ResourceGetMappedPointer\(\)](#).

**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:**

[cuGraphicsSubResourceGetMappedArray](#)

#### 4.43.4.8 CUresult cuD3D9ResourceSetMapFlags (IDirect3DResource9 \*pResource, unsigned int Flags)

**Deprecated**

This function is deprecated as of Cuda 3.0.

Set `Flags` for mapping the Direct3D resource `pResource`.

Changes to `Flags` will take effect the next time `pResource` is mapped. The `Flags` argument may be any of the following:

- `CU_D3D9_MAPRESOURCE_FLAGS_NONE`: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- `CU_D3D9_MAPRESOURCE_FLAGS_READONLY`: Specifies that CUDA kernels which access this resource will not write to this resource.
- `CU_D3D9_MAPRESOURCE_FLAGS_WRITEDISCARD`: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If `pResource` has not been registered for use with CUDA, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If `pResource` is presently mapped for access by CUDA, then [CUDA\\_ERROR\\_ALREADY\\_MAPPED](#) is returned.

**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:**

[cuGraphicsResourceSetMapFlags](#)

**4.43.4.9 CUresult cuD3D9UnmapResources (unsigned int *count*, IDirect3DResource9 \*\* *ppResource*)****Deprecated**

This function is deprecated as of Cuda 3.0.

Unmaps the *count* Direct3D resources in *ppResource*.

This function provides the synchronization guarantee that any CUDA kernels issued before [cuD3D9UnmapResources\(\)](#) will complete before any Direct3D calls issued after [cuD3D9UnmapResources\(\)](#) begin.

If any of *ppResource* have not been registered for use with CUDA or if *ppResource* contains any duplicate entries, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If any of *ppResource* are not presently mapped for access by CUDA, then [CUDA\\_ERROR\\_NOT\\_MAPPED](#) is returned.

**Parameters:**

*count* - Number of resources to unmap for CUDA

*ppResource* - Resources to unmap for CUDA

**Returns:**

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

**Note:**

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

**See also:**

[cuGraphicsUnmapResources](#)

**4.43.4.10 CUresult cuD3D9UnregisterResource (IDirect3DResource9 \* *pResource*)****Deprecated**

This function is deprecated as of Cuda 3.0.

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

If *pResource* is not registered, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned.

**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:**

[cuGraphicsUnregisterResource](#)

## 4.44 Direct3D 10 Interoperability

### Modules

- [Direct3D 10 Interoperability \[DEPRECATED\]](#)

### Typedefs

- typedef enum [CUd3d10DeviceList\\_enum](#) [CUd3d10DeviceList](#)

### Enumerations

- enum [CUd3d10DeviceList\\_enum](#) {  
[CU\\_D3D10\\_DEVICE\\_LIST\\_ALL](#) = 0x01,  
[CU\\_D3D10\\_DEVICE\\_LIST\\_CURRENT\\_FRAME](#) = 0x02,  
[CU\\_D3D10\\_DEVICE\\_LIST\\_NEXT\\_FRAME](#) = 0x03 }

### Functions

- [CUresult cuD3D10CtxCreate](#) ([CUcontext](#) \*pCtx, [CUdevice](#) \*pCudaDevice, unsigned int Flags, [ID3D10Device](#) \*pD3DDevice)  
*Create a CUDA context for interoperability with Direct3D 10.*
- [CUresult cuD3D10CtxCreateOnDevice](#) ([CUcontext](#) \*pCtx, unsigned int flags, [ID3D10Device](#) \*pD3DDevice, [CUdevice](#) cudaDevice)  
*Create a CUDA context for interoperability with Direct3D 10.*
- [CUresult cuD3D10GetDevice](#) ([CUdevice](#) \*pCudaDevice, [IDXGIAdapter](#) \*pAdapter)  
*Gets the CUDA device corresponding to a display adapter.*
- [CUresult cuD3D10GetDevices](#) (unsigned int \*pCudaDeviceCount, [CUdevice](#) \*pCudaDevices, unsigned int cudaDeviceCount, [ID3D10Device](#) \*pD3D10Device, [CUd3d10DeviceList](#) deviceList)  
*Gets the CUDA devices corresponding to a Direct3D 10 device.*
- [CUresult cuD3D10GetDirect3DDevice](#) ([ID3D10Device](#) \*\*ppD3DDevice)  
*Get the Direct3D 10 device against which the current CUDA context was created.*
- [CUresult cuGraphicsD3D10RegisterResource](#) ([CUgraphicsResource](#) \*pCudaResource, [ID3D10Resource](#) \*pD3DResource, unsigned int Flags)  
*Register a Direct3D 10 resource for access by CUDA.*

#### 4.44.1 Detailed Description

This section describes the Direct3D 10 interoperability functions of the low-level CUDA driver application programming interface.

## 4.44.2 Typedef Documentation

### 4.44.2.1 typedef enum CUd3d10DeviceList\_enum CUd3d10DeviceList

CUDA devices corresponding to a D3D10 device

## 4.44.3 Enumeration Type Documentation

### 4.44.3.1 enum CUd3d10DeviceList\_enum

CUDA devices corresponding to a D3D10 device

#### Enumerator:

*CU\_D3D10\_DEVICE\_LIST\_ALL* The CUDA devices for all GPUs used by a D3D10 device

*CU\_D3D10\_DEVICE\_LIST\_CURRENT\_FRAME* The CUDA devices for the GPUs used by a D3D10 device in its currently rendering frame

*CU\_D3D10\_DEVICE\_LIST\_NEXT\_FRAME* The CUDA devices for the GPUs to be used by a D3D10 device in the next frame

## 4.44.4 Function Documentation

### 4.44.4.1 CUresult cuD3D10CtxCreate (CUcontext \* *pCtx*, CUdevice \* *pCudaDevice*, unsigned int *Flags*, ID3D10Device \* *pD3DDevice*)

Creates a new CUDA context, enables interoperability for that context with the Direct3D device *pD3DDevice*, and associates the created CUDA context with the calling thread. The created [CUcontext](#) will be returned in *\*pCtx*. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context. If *pCudaDevice* is non-NULL then the [CUdevice](#) on which this CUDA context was created will be returned in *\*pCudaDevice*.

On success, this call will increase the internal reference count on *pD3DDevice*. This reference count will be decremented upon destruction of this context through [cuCtxDestroy\(\)](#). This context will cease to function if *pD3DDevice* is destroyed or encounters an error.

#### Parameters:

*pCtx* - Returned newly created CUDA context

*pCudaDevice* - Returned pointer to the device on which the context was created

*Flags* - Context creation flags (see [cuCtxCreate\(\)](#) for details)

*pD3DDevice* - Direct3D device to create interoperability context with

#### Returns:

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

#### Note:

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

#### See also:

[cuD3D10GetDevice](#), [cuGraphicsD3D10RegisterResource](#)

#### 4.44.4.2 CUresult cuD3D10CtxCreateOnDevice (CUcontext \* *pCtx*, unsigned int *flags*, ID3D10Device \* *pD3DDevice*, CUdevice *cudaDevice*)

Creates a new CUDA context, enables interoperability for that context with the Direct3D device *pD3DDevice*, and associates the created CUDA context with the calling thread. The created [CUcontext](#) will be returned in *\*pCtx*. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context.

On success, this call will increase the internal reference count on *pD3DDevice*. This reference count will be decremented upon destruction of this context through [cuCtxDestroy\(\)](#). This context will cease to function if *pD3DDevice* is destroyed or encounters an error.

##### Parameters:

*pCtx* - Returned newly created CUDA context

*flags* - Context creation flags (see [cuCtxCreate\(\)](#) for details)

*pD3DDevice* - Direct3D device to create interoperability context with

*cudaDevice* - The CUDA device on which to create the context. This device must be among the devices returned when querying CU\_D3D10\_DEVICES\_ALL from [cuD3D10GetDevices](#).

##### Returns:

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

##### Note:

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

##### See also:

[cuD3D10GetDevices](#), [cuGraphicsD3D10RegisterResource](#)

#### 4.44.4.3 CUresult cuD3D10GetDevice (CUdevice \* *pCudaDevice*, IDXGIAdapter \* *pAdapter*)

Returns in *\*pCudaDevice* the CUDA-compatible device corresponding to the adapter *pAdapter* obtained from `IDXGIFactory::EnumAdapters`.

If no device on *pAdapter* is CUDA-compatible then the call will fail.

##### Parameters:

*pCudaDevice* - Returned CUDA device corresponding to *pAdapter*

*pAdapter* - Adapter to query for CUDA device

##### Returns:

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

##### Note:

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

##### See also:

[cuD3D10CtxCreate](#)

**4.44.4.4 CUresult cuD3D10GetDevices (unsigned int \* *pCudaDeviceCount*, CUdevice \* *pCudaDevices*, unsigned int *cudaDeviceCount*, ID3D10Device \* *pD3D10Device*, CUd3d10DeviceList *deviceList*)**

Returns in *\*pCudaDeviceCount* the number of CUDA-compatible device corresponding to the Direct3D 10 device *pD3D10Device*. Also returns in *\*pCudaDevices* at most *cudaDeviceCount* of the the CUDA-compatible devices corresponding to the Direct3D 10 device *pD3D10Device*.

If any of the GPUs being used to render *pDevice* are not CUDA capable then the call will return [CUDA\\_ERROR\\_NO\\_DEVICE](#).

**Parameters:**

*pCudaDeviceCount* - Returned number of CUDA devices corresponding to *pD3D10Device*

*pCudaDevices* - Returned CUDA devices corresponding to *pD3D10Device*

*cudaDeviceCount* - The size of the output device array *pCudaDevices*

*pD3D10Device* - Direct3D 10 device to query for CUDA devices

*deviceList* - The set of devices to return. This set may be [CU\\_D3D10\\_DEVICE\\_LIST\\_ALL](#) for all devices, [CU\\_D3D10\\_DEVICE\\_LIST\\_CURRENT\\_FRAME](#) for the devices used to render the current frame (in SLI), or [CU\\_D3D10\\_DEVICE\\_LIST\\_NEXT\\_FRAME](#) for the devices used to render the next frame (in SLI).

**Returns:**

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_NO\\_DEVICE](#), [CUDA\\_ERROR\\_UNKNOWN](#)

**Note:**

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

**See also:**

[cuD3D10CtxCreate](#)

**4.44.4.5 CUresult cuD3D10GetDirect3DDevice (ID3D10Device \*\* *ppD3DDevice*)**

Returns in *\*ppD3DDevice* the Direct3D device against which this CUDA context was created in [cuD3D10CtxCreate\(\)](#).

**Parameters:**

*ppD3DDevice* - 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:**

[cuD3D10GetDevice](#)

#### 4.44.4.6 CUresult cuGraphicsD3D10RegisterResource (CUgraphicsResource \* pCudaResource, ID3D10Resource \* pD3DResource, unsigned int Flags)

Registers the Direct3D 10 resource `pD3DResource` for access by CUDA and returns a CUDA handle to `pD3DResource` in `pCudaResource`. The handle returned in `pCudaResource` may be used to map and unmap this resource until it is unregistered. On success this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through `cuGraphicsUnregisterResource()`.

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

The type of `pD3DResource` must be one of the following.

- `ID3D10Buffer`: may be accessed through a device pointer.
- `ID3D10Texture1D`: individual subresources of the texture may be accessed via arrays
- `ID3D10Texture2D`: individual subresources of the texture may be accessed via arrays
- `ID3D10Texture3D`: individual subresources of the texture may be accessed via arrays

The `Flags` argument may be used to specify additional parameters at register time. The only valid value for this parameter is

- `CU_GRAPHICS_REGISTER_FLAGS_NONE`

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 for this context using `cuD3D10CtxCreate` then `CUDA_ERROR_INVALID_CONTEXT` is returned. If `pD3DResource` is of incorrect type or is already registered then `CUDA_ERROR_INVALID_HANDLE` is returned. If `pD3DResource` cannot be registered then `CUDA_ERROR_UNKNOWN` is returned. If `Flags` is not one of the above specified value then `CUDA_ERROR_INVALID_VALUE` is returned.

#### Parameters:

*pCudaResource* - Returned graphics resource handle

*pD3DResource* - Direct3D resource to register

*Flags* - Parameters for resource registration

#### Returns:

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_INVALID_HANDLE`, `CUDA_ERROR_OUT_OF_MEMORY`, `CUDA_ERROR_UNKNOWN`

#### Note:

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

**See also:**

[cuD3D10CtxCreate](#), [cuGraphicsUnregisterResource](#), [cuGraphicsMapResources](#), [cuGraphicsSubResourceGetMappedArray](#), [cuGraphicsResourceGetMappedPointer](#)

## 4.45 Direct3D 10 Interoperability [DEPRECATED]

### Typedefs

- typedef enum [CUDA3D10map\\_flags\\_enum](#) [CUDA3D10map\\_flags](#)
- typedef enum [CUDA3D10register\\_flags\\_enum](#) [CUDA3D10register\\_flags](#)

### Enumerations

- enum [CUDA3D10map\\_flags\\_enum](#)
- enum [CUDA3D10register\\_flags\\_enum](#)

### Functions

- [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, unsigned int SubResource)  
*Get an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- [CUresult cuD3D10ResourceGetMappedPitch](#) (size\_t \*pPitch, size\_t \*pPitchSlice, ID3D10Resource \*pResource, unsigned int SubResource)  
*Get the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- [CUresult cuD3D10ResourceGetMappedPointer](#) (CUdeviceptr \*pDevPtr, ID3D10Resource \*pResource, unsigned int SubResource)  
*Get a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- [CUresult cuD3D10ResourceGetMappedSize](#) (size\_t \*pSize, ID3D10Resource \*pResource, unsigned int SubResource)  
*Get the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.*
- [CUresult cuD3D10ResourceGetSurfaceDimensions](#) (size\_t \*pWidth, size\_t \*pHeight, size\_t \*pDepth, ID3D10Resource \*pResource, unsigned int 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.*

## 4.45.1 Detailed Description

This section describes deprecated Direct3D 10 interoperability functionality.

## 4.45.2 Typedef Documentation

### 4.45.2.1 typedef enum CUD3D10map\_flags\_enum CUD3D10map\_flags

Flags to map or unmap a resource

### 4.45.2.2 typedef enum CUD3D10register\_flags\_enum CUD3D10register\_flags

Flags to register a resource

## 4.45.3 Enumeration Type Documentation

### 4.45.3.1 enum CUD3D10map\_flags\_enum

Flags to map or unmap a resource

### 4.45.3.2 enum CUD3D10register\_flags\_enum

Flags to register a resource

## 4.45.4 Function Documentation

### 4.45.4.1 CUresult cuD3D10MapResources (unsigned int *count*, ID3D10Resource \*\* *ppResources*)

#### Deprecated

This function is deprecated as of Cuda 3.0.

Maps the *count* Direct3D resources in *ppResources* for access by CUDA.

The resources in *ppResources* may be accessed in CUDA kernels until they are unmapped. Direct3D should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any Direct3D calls issued before [cuD3D10MapResources\(\)](#) will complete before any CUDA kernels issued after [cuD3D10MapResources\(\)](#) begin.

If any of *ppResources* have not been registered for use with CUDA or if *ppResources* contains any duplicate entries, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If any of *ppResources* are presently mapped for access by CUDA, then [CUDA\\_ERROR\\_ALREADY\\_MAPPED](#) is returned.

#### 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:**

[cuGraphicsMapResources](#)

**4.45.4.2 CUresult cuD3D10RegisterResource (ID3D10Resource \*pResource, unsigned int Flags)****Deprecated**

This function is deprecated as of Cuda 3.0.

Registers the Direct3D resource `pResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through [cuD3D10UnregisterResource\(\)](#). Also on success, this call will increase the internal reference count on `pResource`. This reference count will be decremented when this resource is unregistered through [cuD3D10UnregisterResource\(\)](#).

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

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

- ID3D10Buffer: Cannot be used with `Flags` set to `CU_D3D10_REGISTER_FLAGS_ARRAY`.
- ID3D10Texture1D: No restrictions.
- ID3D10Texture2D: No restrictions.
- ID3D10Texture3D: No restrictions.

The `Flags` argument specifies the mechanism through which CUDA will access the Direct3D resource. The following values are allowed.

- `CU_D3D10_REGISTER_FLAGS_NONE`: Specifies that CUDA will access this resource through a [CUdeviceptr](#). The pointer, size, and (for textures), pitch for each subresource of this allocation may be queried through [cuD3D10ResourceGetMappedPointer\(\)](#), [cuD3D10ResourceGetMappedSize\(\)](#), and [cuD3D10ResourceGetMappedPitch\(\)](#) respectively. This option is valid for all resource types.
- `CU_D3D10_REGISTER_FLAGS_ARRAY`: Specifies that CUDA will access this resource through a [CUarray](#) queried on a sub-resource basis through [cuD3D10ResourceGetMappedArray\(\)](#). This option is only valid for resources of type `ID3D10Texture1D`, `ID3D10Texture2D`, and `ID3D10Texture3D`.

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

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.

- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized on this context then [CUDA\\_ERROR\\_INVALID\\_CONTEXT](#) is returned. If `pResource` is of incorrect type or is already registered, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If `pResource` cannot be registered, then [CUDA\\_ERROR\\_UNKNOWN](#) is returned.

**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:**

[cuGraphicsD3D10RegisterResource](#)

#### 4.45.4.3 CUresult cuD3D10ResourceGetMappedArray (CUarray \* pArray, ID3D10Resource \* pResource, unsigned int SubResource)

**Deprecated**

This function is deprecated as of Cuda 3.0.

Returns in `*pArray` an array through which the subresource of the mapped Direct3D resource `pResource`, which corresponds to `SubResource` may be accessed. The value set in `pArray` may change every time that `pResource` is mapped.

If `pResource` is not registered, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If `pResource` was not registered with usage flags `CU_D3D10_REGISTER_FLAGS_ARRAY`, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If `pResource` is not mapped, then [CUDA\\_ERROR\\_NOT\\_MAPPED](#) is returned.

For usage requirements of the `SubResource` parameter, see [cuD3D10ResourceGetMappedPointer\(\)](#).

**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:**

[cuGraphicsSubResourceGetMappedArray](#)

#### 4.45.4.4 CUresult cuD3D10ResourceGetMappedPitch (size\_t \* pPitch, size\_t \* pPitchSlice, ID3D10Resource \* pResource, unsigned int SubResource)

**Deprecated**

This function is deprecated as of Cuda 3.0.

Returns in *pPitch* and *pPitchSlice* the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource *pResource*, which corresponds to *SubResource*. The values set in *pPitch* and *pPitchSlice* may change every time that *pResource* is mapped.

The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position *x*, *y* from the base pointer of the surface is:

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

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

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

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

If *pResource* is not of type `IDirect3DBaseTexture10` or one of its sub-types or if *pResource* has not been registered for use with CUDA, then `CUDA_ERROR_INVALID_HANDLE` is returned. If *pResource* was not registered with usage flags `CU_D3D10_REGISTER_FLAGS_NONE`, then `CUDA_ERROR_INVALID_HANDLE` is returned. If *pResource* is not mapped for access by CUDA, then `CUDA_ERROR_NOT_MAPPED` is returned.

For usage requirements of the *SubResource* parameter, see [cuD3D10ResourceGetMappedPointer\(\)](#).

**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:**

[cuGraphicsSubResourceGetMappedArray](#)

#### 4.45.4.5 CUresult cuD3D10ResourceGetMappedPointer (CUdeviceptr \* *pDevPtr*, ID3D10Resource \* *pResource*, unsigned int *SubResource*)

##### Deprecated

This function is deprecated as of Cuda 3.0.

Returns in \**pDevPtr* the base pointer of the subresource of the mapped Direct3D resource *pResource*, which corresponds to *SubResource*. The value set in *pDevPtr* may change every time that *pResource* is mapped.

If *pResource* is not registered, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If *pResource* was not registered with usage flags `CU_D3D10_REGISTER_FLAGS_NONE`, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If *pResource* is not mapped, then [CUDA\\_ERROR\\_NOT\\_MAPPED](#) is returned.

If *pResource* is of type `ID3D10Buffer`, then *SubResource* must be 0. If *pResource* is of any other type, then the value of *SubResource* must come from the subresource calculation in `D3D10CalcSubResource()`.

##### 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:

[cuGraphicsResourceGetMappedPointer](#)

#### 4.45.4.6 CUresult cuD3D10ResourceGetMappedSize (size\_t \* *pSize*, ID3D10Resource \* *pResource*, unsigned int *SubResource*)

##### Deprecated

This function is deprecated as of Cuda 3.0.

Returns in \**pSize* the size of the subresource of the mapped Direct3D resource *pResource*, which corresponds to *SubResource*. The value set in *pSize* may change every time that *pResource* is mapped.

If *pResource* has not been registered for use with CUDA, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If *pResource* was not registered with usage flags `CU_D3D10_REGISTER_FLAGS_NONE`, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned. If *pResource* is not mapped for access by CUDA, then [CUDA\\_ERROR\\_NOT\\_MAPPED](#) is returned.

For usage requirements of the *SubResource* parameter, see [cuD3D10ResourceGetMappedPointer\(\)](#).

##### 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:**

[cuGraphicsResourceGetMappedPointer](#)

**4.45.4.7 CUresult cuD3D10ResourceGetSurfaceDimensions (size\_t \* pWidth, size\_t \* pHeight, size\_t \* pDepth, IDirect3D10Resource \* pResource, unsigned int SubResource)**

**Deprecated**

This function is deprecated as of Cuda 3.0.

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

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

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

If pResource is not of type IDirect3DBaseTexture10 or IDirect3DSurface10 or if pResource has not been registered for use with CUDA, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned.

For usage requirements of the SubResource parameter, see [cuD3D10ResourceGetMappedPointer\(\)](#).

**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:**

[cuGraphicsSubResourceGetMappedArray](#)

#### 4.45.4.8 CUresult cuD3D10ResourceSetMapFlags (ID3D10Resource \* pResource, unsigned int Flags)

##### Deprecated

This function is deprecated as of Cuda 3.0.

Set flags for mapping the Direct3D resource pResource.

Changes to flags will take effect the next time pResource is mapped. The Flags argument may be any of the following.

- `CU_D3D10_MAPRESOURCE_FLAGS_NONE`: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- `CU_D3D10_MAPRESOURCE_FLAGS_READONLY`: Specifies that CUDA kernels which access this resource will not write to this resource.
- `CU_D3D10_MAPRESOURCE_FLAGS_WRITEDISCARD`: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If pResource has not been registered for use with CUDA, then `CUDA_ERROR_INVALID_HANDLE` is returned. If pResource is presently mapped for access by CUDA then `CUDA_ERROR_ALREADY_MAPPED` is returned.

##### 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:

[cuGraphicsResourceSetMapFlags](#)

#### 4.45.4.9 CUresult cuD3D10UnmapResources (unsigned int count, ID3D10Resource \*\* ppResources)

##### Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the count Direct3D resources in ppResources.

This function provides the synchronization guarantee that any CUDA kernels issued before `cuD3D10UnmapResources()` will complete before any Direct3D calls issued after `cuD3D10UnmapResources()` begin.

If any of ppResources have not been registered for use with CUDA or if ppResources contains any duplicate entries, then `CUDA_ERROR_INVALID_HANDLE` is returned. If any of ppResources are not presently mapped for access by CUDA, then `CUDA_ERROR_NOT_MAPPED` is returned.

**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:**

[cuGraphicsUnmapResources](#)

**4.45.4.10 CUresult cuD3D10UnregisterResource (ID3D10Resource \* pResource)****Deprecated**

This function is deprecated as of Cuda 3.0.

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

If *pResource* is not registered, then [CUDA\\_ERROR\\_INVALID\\_HANDLE](#) is returned.

**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:**

[cuGraphicsUnregisterResource](#)

## 4.46 Direct3D 11 Interoperability

### Typedefs

- typedef enum [CUd3d11DeviceList\\_enum](#) [CUd3d11DeviceList](#)

### Enumerations

- enum [CUd3d11DeviceList\\_enum](#) {  
[CU\\_D3D11\\_DEVICE\\_LIST\\_ALL](#) = 0x01,  
[CU\\_D3D11\\_DEVICE\\_LIST\\_CURRENT\\_FRAME](#) = 0x02,  
[CU\\_D3D11\\_DEVICE\\_LIST\\_NEXT\\_FRAME](#) = 0x03 }

### Functions

- [CUresult cuD3D11CtxCreate](#) ([CUcontext](#) \*pCtx, [CUdevice](#) \*pCudaDevice, unsigned int Flags, [ID3D11Device](#) \*pD3DDevice)  
*Create a CUDA context for interoperability with Direct3D 11.*
- [CUresult cuD3D11CtxCreateOnDevice](#) ([CUcontext](#) \*pCtx, unsigned int flags, [ID3D11Device](#) \*pD3DDevice, [CUdevice](#) cudaDevice)  
*Create a CUDA context for interoperability with Direct3D 11.*
- [CUresult cuD3D11GetDevice](#) ([CUdevice](#) \*pCudaDevice, [IDXGIAdapter](#) \*pAdapter)  
*Gets the CUDA device corresponding to a display adapter.*
- [CUresult cuD3D11GetDevices](#) (unsigned int \*pCudaDeviceCount, [CUdevice](#) \*pCudaDevices, unsigned int cudaDeviceCount, [ID3D11Device](#) \*pD3D11Device, [CUd3d11DeviceList](#) deviceList)  
*Gets the CUDA devices corresponding to a Direct3D 11 device.*
- [CUresult cuD3D11GetDirect3DDevice](#) ([ID3D11Device](#) \*\*ppD3DDevice)  
*Get the Direct3D 11 device against which the current CUDA context was created.*
- [CUresult cuGraphicsD3D11RegisterResource](#) ([CUgraphicsResource](#) \*pCudaResource, [ID3D11Resource](#) \*pD3DResource, unsigned int Flags)  
*Register a Direct3D 11 resource for access by CUDA.*

#### 4.46.1 Detailed Description

This section describes the Direct3D 11 interoperability functions of the low-level CUDA driver application programming interface.

#### 4.46.2 Typedef Documentation

##### 4.46.2.1 typedef enum [CUd3d11DeviceList\\_enum](#) [CUd3d11DeviceList](#)

CUDA devices corresponding to a D3D11 device

### 4.46.3 Enumeration Type Documentation

#### 4.46.3.1 enum CUd3d11DeviceList\_enum

CUDA devices corresponding to a D3D11 device

##### Enumerator:

*CU\_D3D11\_DEVICE\_LIST\_ALL* The CUDA devices for all GPUs used by a D3D11 device

*CU\_D3D11\_DEVICE\_LIST\_CURRENT\_FRAME* The CUDA devices for the GPUs used by a D3D11 device in its currently rendering frame

*CU\_D3D11\_DEVICE\_LIST\_NEXT\_FRAME* The CUDA devices for the GPUs to be used by a D3D11 device in the next frame

### 4.46.4 Function Documentation

#### 4.46.4.1 CUresult cuD3D11CtxCreate (CUcontext \* *pCtx*, CUdevice \* *pCudaDevice*, unsigned int *Flags*, ID3D11Device \* *pD3DDevice*)

Creates a new CUDA context, enables interoperability for that context with the Direct3D device *pD3DDevice*, and associates the created CUDA context with the calling thread. The created [CUcontext](#) will be returned in *\*pCtx*. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context. If *pCudaDevice* is non-NULL then the [CUdevice](#) on which this CUDA context was created will be returned in *\*pCudaDevice*.

On success, this call will increase the internal reference count on *pD3DDevice*. This reference count will be decremented upon destruction of this context through [cuCtxDestroy\(\)](#). This context will cease to function if *pD3DDevice* is destroyed or encounters an error.

##### Parameters:

*pCtx* - Returned newly created CUDA context

*pCudaDevice* - Returned pointer to the device on which the context was created

*Flags* - Context creation flags (see [cuCtxCreate\(\)](#) for details)

*pD3DDevice* - Direct3D device to create interoperability context with

##### Returns:

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

##### Note:

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

##### See also:

[cuD3D11GetDevice](#), [cuGraphicsD3D11RegisterResource](#)

#### 4.46.4.2 CUresult cuD3D11CtxCreateOnDevice (CUcontext \* *pCtx*, unsigned int *flags*, ID3D11Device \* *pD3DDevice*, CUdevice *cudaDevice*)

Creates a new CUDA context, enables interoperability for that context with the Direct3D device *pD3DDevice*, and associates the created CUDA context with the calling thread. The created [CUcontext](#) will be returned in *\*pCtx*. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context.

On success, this call will increase the internal reference count on `pD3DDevice`. This reference count will be decremented upon destruction of this context through `cuCtxDestroy()`. This context will cease to function if `pD3DDevice` is destroyed or encounters an error.

**Parameters:**

*pCtx* - Returned newly created CUDA context

*flags* - Context creation flags (see `cuCtxCreate()` for details)

*pD3DDevice* - Direct3D device to create interoperability context with

*cudaDevice* - The CUDA device on which to create the context. This device must be among the devices returned when querying `CU_D3D11_DEVICES_ALL` from `cuD3D11GetDevices`.

**Returns:**

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_OUT_OF_MEMORY`, `CUDA_ERROR_UNKNOWN`

**Note:**

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

**See also:**

`cuD3D11GetDevices`, `cuGraphicsD3D11RegisterResource`

**4.46.4.3 CUresult cuD3D11GetDevice (CUdevice \*pCudaDevice, IDXGIAdapter \*pAdapter)**

Returns in `*pCudaDevice` the CUDA-compatible device corresponding to the adapter `pAdapter` obtained from `IDXGIFactory::EnumAdapters`.

If no device on `pAdapter` is CUDA-compatible the call will return `CUDA_ERROR_NO_DEVICE`.

**Parameters:**

*pCudaDevice* - Returned CUDA device corresponding to `pAdapter`

*pAdapter* - Adapter to query for CUDA device

**Returns:**

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_NO_DEVICE`, `CUDA_ERROR_UNKNOWN`

**Note:**

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

**See also:**

`cuD3D11CtxCreate`

#### 4.46.4.4 CUresult cuD3D11GetDevices (unsigned int \*pCudaDeviceCount, CUdevice \*pCudaDevices, unsigned int cudaDeviceCount, ID3D11Device \*pD3D11Device, CUd3d11DeviceList deviceList)

Returns in \*pCudaDeviceCount the number of CUDA-compatible device corresponding to the Direct3D 11 device pD3D11Device. Also returns in \*pCudaDevices at most cudaDeviceCount of the the CUDA-compatible devices corresponding to the Direct3D 11 device pD3D11Device.

If any of the GPUs being used to render pDevice are not CUDA capable then the call will return [CUDA\\_ERROR\\_NO\\_DEVICE](#).

##### Parameters:

*pCudaDeviceCount* - Returned number of CUDA devices corresponding to pD3D11Device

*pCudaDevices* - Returned CUDA devices corresponding to pD3D11Device

*cudaDeviceCount* - The size of the output device array pCudaDevices

*pD3D11Device* - Direct3D 11 device to query for CUDA devices

*deviceList* - The set of devices to return. This set may be [CU\\_D3D11\\_DEVICE\\_LIST\\_ALL](#) for all devices, [CU\\_D3D11\\_DEVICE\\_LIST\\_CURRENT\\_FRAME](#) for the devices used to render the current frame (in SLI), or [CU\\_D3D11\\_DEVICE\\_LIST\\_NEXT\\_FRAME](#) for the devices used to render the next frame (in SLI).

##### Returns:

[CUDA\\_SUCCESS](#), [CUDA\\_ERROR\\_DEINITIALIZED](#), [CUDA\\_ERROR\\_NOT\\_INITIALIZED](#), [CUDA\\_ERROR\\_NO\\_DEVICE](#), [CUDA\\_ERROR\\_UNKNOWN](#)

##### Note:

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

##### See also:

[cuD3D11CtxCreate](#)

#### 4.46.4.5 CUresult cuD3D11GetDirect3DDevice (ID3D11Device \*\*ppD3DDevice)

Returns in \*ppD3DDevice the Direct3D device against which this CUDA context was created in [cuD3D11CtxCreate\(\)](#).

##### Parameters:

*ppD3DDevice* - 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:

[cuD3D11GetDevice](#)

#### 4.46.4.6 `CUresult cuGraphicsD3D11RegisterResource (CUgraphicsResource * pCudaResource, ID3D11Resource * pD3DResource, unsigned int Flags)`

Registers the Direct3D 11 resource `pD3DResource` for access by CUDA and returns a CUDA handle to `pD3DResource` in `pCudaResource`. The handle returned in `pCudaResource` may be used to map and unmap this resource until it is unregistered. On success this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through `cuGraphicsUnregisterResource()`.

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

The type of `pD3DResource` must be one of the following.

- `ID3D11Buffer`: may be accessed through a device pointer.
- `ID3D11Texture1D`: individual subresources of the texture may be accessed via arrays
- `ID3D11Texture2D`: individual subresources of the texture may be accessed via arrays
- `ID3D11Texture3D`: individual subresources of the texture may be accessed via arrays

The `Flags` argument may be used to specify additional parameters at register time. The only valid value for this parameter is

- `CU_GRAPHICS_REGISTER_FLAGS_NONE`

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 for this context using `cuD3D11CtxCreate` then `CUDA_ERROR_INVALID_CONTEXT` is returned. If `pD3DResource` is of incorrect type or is already registered then `CUDA_ERROR_INVALID_HANDLE` is returned. If `pD3DResource` cannot be registered then `CUDA_ERROR_UNKNOWN` is returned. If `Flags` is not one of the above specified value then `CUDA_ERROR_INVALID_VALUE` is returned.

#### Parameters:

*pCudaResource* - Returned graphics resource handle

*pD3DResource* - Direct3D resource to register

*Flags* - Parameters for resource registration

#### Returns:

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_INVALID_HANDLE`, `CUDA_ERROR_OUT_OF_MEMORY`, `CUDA_ERROR_UNKNOWN`

#### Note:

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

**See also:**

[cuD3D11CtxCreate](#), [cuGraphicsUnregisterResource](#), [cuGraphicsMapResources](#), [cuGraphicsSubResourceGetMappedArray](#), [cuGraphicsResourceGetMappedPointer](#)

## 4.47 VDPAU Interoperability

### Functions

- **CUresult cuGraphicsVDPAURegisterOutputSurface** (**CUgraphicsResource** \*pCudaResource, VdpOutputSurface vdpSurface, unsigned int flags)  
*Registers a VDPAU VdpOutputSurface object.*
- **CUresult cuGraphicsVDPAURegisterVideoSurface** (**CUgraphicsResource** \*pCudaResource, VdpVideoSurface vdpSurface, unsigned int flags)  
*Registers a VDPAU VdpVideoSurface object.*
- **CUresult cuVDPAUCtxCreate** (**CUcontext** \*pCtx, unsigned int flags, **CUdevice** device, VdpDevice vdpDevice, VdpGetProcAddress \*vdpGetProcAddress)  
*Create a CUDA context for interoperability with VDPAU.*
- **CUresult cuVDPAUGetDevice** (**CUdevice** \*pDevice, VdpDevice vdpDevice, VdpGetProcAddress \*vdpGetProcAddress)  
*Gets the CUDA device associated with a VDPAU device.*

### 4.47.1 Detailed Description

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

### 4.47.2 Function Documentation

#### 4.47.2.1 CUresult cuGraphicsVDPAURegisterOutputSurface (CUgraphicsResource \*pCudaResource, VdpOutputSurface vdpSurface, unsigned int flags)

Registers the VdpOutputSurface specified by vdpSurface for access by CUDA. A handle to the registered object is returned as pCudaResource. The surface's intended usage is specified using flags, as follows:

- **CU\_GRAPHICS\_MAP\_RESOURCE\_FLAGS\_NONE**: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- **CU\_GRAPHICS\_MAP\_RESOURCE\_FLAGS\_READ\_ONLY**: Specifies that CUDA will not write to this resource.
- **CU\_GRAPHICS\_MAP\_RESOURCE\_FLAGS\_WRITE\_DISCARD**: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

The VdpOutputSurface is presented as an array of subresources that may be accessed using pointers returned by [cuGraphicsSubResourceGetMappedArray](#). The exact number of valid arrayIndex values depends on the VDPAU surface format. The mapping is shown in the table below. mipLevel must be 0.

VdpRGBAFormat	arrayIndex	Size	Format	Content
VDP_RGBA_FORMAT_B8G8R8A8	0	w x h	ARGB8	Entire surface
VDP_RGBA_FORMAT_R10G10B10A2	0	w x h	A2BGR10	Entire surface

**Parameters:**

*pCudaResource* - Pointer to the returned object handle

*vdpSurface* - The VdpOutputSurface to be registered

*flags* - Map flags

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_INVALID\_HANDLE, CUDA\_ERROR\_ALREADY\_MAPPED, CUDA\_ERROR\_INVALID\_CONTEXT,

**Note:**

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

**See also:**

[cuCtxCreate](#), [cuVDPACtxCreate](#), [cuGraphicsVDPAResourceRegisterVideoSurface](#), [cuGraphicsUnregisterResource](#), [cuGraphicsResourceSetMapFlags](#), [cuGraphicsMapResources](#), [cuGraphicsUnmapResources](#), [cuGraphicsSubResourceGetMappedArray](#), [cuVDPAGetDevice](#)

#### 4.47.2.2 CUresult cuGraphicsVDPAResourceRegisterVideoSurface (CUgraphicsResource \* pCudaResource, VdpVideoSurface vdpSurface, unsigned int flags)

Registers the VdpVideoSurface specified by vdpSurface for access by CUDA. A handle to the registered object is returned as pCudaResource. The surface's intended usage is specified using flags, as follows:

- CU\_GRAPHICS\_MAP\_RESOURCE\_FLAGS\_NONE: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- CU\_GRAPHICS\_MAP\_RESOURCE\_FLAGS\_READ\_ONLY: Specifies that CUDA will not write to this resource.
- CU\_GRAPHICS\_MAP\_RESOURCE\_FLAGS\_WRITE\_DISCARD: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

The VdpVideoSurface is presented as an array of subresources that may be accessed using pointers returned by [cuGraphicsSubResourceGetMappedArray](#). The exact number of valid arrayIndex values depends on the VDPAResource surface format. The mapping is shown in the table below. mipLevel must be 0.

VdpChromaType	arrayIndex	Size	Format	Content
VDP_CHROMA_TYPE_420	0	w x h/2	R8	Top-field luma
	1	w x h/2	R8	Bottom-field luma
	2	w/2 x h/4	R8G8	Top-field chroma
	3	w/2 x h/4	R8G8	Bottom-field chroma
VDP_CHROMA_TYPE_422	0	w x h/2	R8	Top-field luma
	1	w x h/2	R8	Bottom-field luma
	2	w/2 x h/2	R8G8	Top-field chroma
	3	w/2 x h/2	R8G8	Bottom-field chroma

**Parameters:**

*pCudaResource* - Pointer to the returned object handle

*vdpSurface* - The VdpVideoSurface to be registered

*flags* - Map flags

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_INVALID\_HANDLE, CUDA\_ERROR\_ALREADY\_MAPPED, CUDA\_ERROR\_INVALID\_CONTEXT,

**Note:**

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

**See also:**

cuCtxCreate, cuVDPAUCtxCreate, cuGraphicsVDPAURegisterOutputSurface, cuGraphicsUnregisterResource, cuGraphicsResourceSetMapFlags, cuGraphicsMapResources, cuGraphicsUnmapResources, cuGraphicsSubResourceGetMappedArray, cuVDPAUGetDevice

**4.47.2.3 CUresult cuVDPAUCtxCreate (CUcontext \* *pCtx*, unsigned int *flags*, CUdevice *device*, VdpDevice *vdpDevice*, VdpGetProcAddress \* *vdpGetProcAddress*)**

Creates a new CUDA context, initializes VDPAU interoperability, and associates the CUDA context with the calling thread. It must be called before performing any other VDPAU interoperability operations. It may fail if the needed VDPAU driver facilities are not available. For usage of the *flags* parameter, see [cuCtxCreate\(\)](#).

**Parameters:**

*pCtx* - Returned CUDA context

*flags* - Options for CUDA context creation

*device* - Device on which to create the context

*vdpDevice* - The VdpDevice to interop with

*vdpGetProcAddress* - VDPAU's VdpGetProcAddress function pointer

**Returns:**

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

**Note:**

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

**See also:**

cuCtxCreate, cuGraphicsVDPAURegisterVideoSurface, cuGraphicsVDPAURegisterOutputSurface, cuGraphicsUnregisterResource, cuGraphicsResourceSetMapFlags, cuGraphicsMapResources, cuGraphicsUnmapResources, cuGraphicsSubResourceGetMappedArray, cuVDPAUGetDevice

**4.47.2.4 CUresult cuVDPAUGetDevice (CUdevice \* *pDevice*, VdpDevice *vdpDevice*, VdpGetProcAddress \* *vdpGetProcAddress*)**

Returns in *\*pDevice* the CUDA device associated with a *vdpDevice*, if applicable.

**Parameters:**

*pDevice* - Device associated with vdpDevice

*vdpDevice* - A VdpDevice handle

*vdpGetProcAddress* - VDPAU's VdpGetProcAddress function pointer

**Returns:**

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

**Note:**

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

**See also:**

[cuCtxCreate](#), [cuVDPAUCtxCreate](#), [cuGraphicsVDPAURegisterVideoSurface](#), [cuGraphicsVDPAURegisterOutputSurface](#), [cuGraphicsUnregisterResource](#), [cuGraphicsResourceSetMapFlags](#), [cuGraphicsMapResources](#), [cuGraphicsUnmapResources](#), [cuGraphicsSubResourceGetMappedArray](#)

# Chapter 5

## Data Structure Documentation

### 5.1 CUDA\_ARRAY3D\_DESCRIPTOR\_st Struct Reference

#### Data Fields

- [size\\_t Depth](#)
- [unsigned int Flags](#)
- [CUarray\\_format Format](#)
- [size\\_t Height](#)
- [unsigned int NumChannels](#)
- [size\\_t Width](#)

#### 5.1.1 Detailed Description

3D array descriptor

#### 5.1.2 Field Documentation

##### 5.1.2.1 `size_t CUDA_ARRAY3D_DESCRIPTOR_st::Depth`

Depth of 3D array

##### 5.1.2.2 `unsigned int CUDA_ARRAY3D_DESCRIPTOR_st::Flags`

Flags

##### 5.1.2.3 `CUarray_format CUDA_ARRAY3D_DESCRIPTOR_st::Format`

Array format

##### 5.1.2.4 `size_t CUDA_ARRAY3D_DESCRIPTOR_st::Height`

Height of 3D array

**5.1.2.5 unsigned int CUDA\_ARRAY3D\_DESCRIPTOR\_st::NumChannels**

Channels per array element

**5.1.2.6 size\_t CUDA\_ARRAY3D\_DESCRIPTOR\_st::Width**

Width of 3D array

## 5.2 CUDA\_ARRAY\_DESCRIPTOR\_st Struct Reference

### Data Fields

- [CUarray\\_format](#) Format
- [size\\_t](#) Height
- [unsigned int](#) NumChannels
- [size\\_t](#) Width

### 5.2.1 Detailed Description

Array descriptor

### 5.2.2 Field Documentation

#### 5.2.2.1 [CUarray\\_format](#) CUDA\_ARRAY\_DESCRIPTOR\_st::Format

Array format

#### 5.2.2.2 [size\\_t](#) CUDA\_ARRAY\_DESCRIPTOR\_st::Height

Height of array

#### 5.2.2.3 [unsigned int](#) CUDA\_ARRAY\_DESCRIPTOR\_st::NumChannels

Channels per array element

#### 5.2.2.4 [size\\_t](#) CUDA\_ARRAY\_DESCRIPTOR\_st::Width

Width of array

## 5.3 CUDA\_MEMCPY2D\_st Struct Reference

### Data Fields

- [CUarray dstArray](#)
- [CUdeviceptr dstDevice](#)
- [void \\* dstHost](#)
- [CUmemorytype dstMemoryType](#)
- [size\\_t dstPitch](#)
- [size\\_t dstXInBytes](#)
- [size\\_t dstY](#)
- [size\\_t Height](#)
- [CUarray srcArray](#)
- [CUdeviceptr srcDevice](#)
- [const void \\* srcHost](#)
- [CUmemorytype srcMemoryType](#)
- [size\\_t srcPitch](#)
- [size\\_t srcXInBytes](#)
- [size\\_t srcY](#)
- [size\\_t WidthInBytes](#)

### 5.3.1 Detailed Description

2D memory copy parameters

### 5.3.2 Field Documentation

#### 5.3.2.1 CUarray CUDA\_MEMCPY2D\_st::dstArray

Destination array reference

#### 5.3.2.2 CUdeviceptr CUDA\_MEMCPY2D\_st::dstDevice

Destination device pointer

#### 5.3.2.3 void\* CUDA\_MEMCPY2D\_st::dstHost

Destination host pointer

#### 5.3.2.4 CUmemorytype CUDA\_MEMCPY2D\_st::dstMemoryType

Destination memory type (host, device, array)

#### 5.3.2.5 size\_t CUDA\_MEMCPY2D\_st::dstPitch

Destination pitch (ignored when dst is array)

**5.3.2.6 size\_t CUDA\_MEMCPY2D\_st::dstXInBytes**

Destination X in bytes

**5.3.2.7 size\_t CUDA\_MEMCPY2D\_st::dstY**

Destination Y

**5.3.2.8 size\_t CUDA\_MEMCPY2D\_st::Height**

Height of 2D memory copy

**5.3.2.9 CUarray CUDA\_MEMCPY2D\_st::srcArray**

Source array reference

**5.3.2.10 CUdeviceptr CUDA\_MEMCPY2D\_st::srcDevice**

Source device pointer

**5.3.2.11 const void\* CUDA\_MEMCPY2D\_st::srcHost**

Source host pointer

**5.3.2.12 CUmemorytype CUDA\_MEMCPY2D\_st::srcMemoryType**

Source memory type (host, device, array)

**5.3.2.13 size\_t CUDA\_MEMCPY2D\_st::srcPitch**

Source pitch (ignored when src is array)

**5.3.2.14 size\_t CUDA\_MEMCPY2D\_st::srcXInBytes**

Source X in bytes

**5.3.2.15 size\_t CUDA\_MEMCPY2D\_st::srcY**

Source Y

**5.3.2.16 size\_t CUDA\_MEMCPY2D\_st::WidthInBytes**

Width of 2D memory copy in bytes

## 5.4 CUDA\_MEMCPY3D\_st Struct Reference

### Data Fields

- [size\\_t Depth](#)
- [CUarray dstArray](#)
- [CUdeviceptr dstDevice](#)
- [size\\_t dstHeight](#)
- [void \\* dstHost](#)
- [size\\_t dstLOD](#)
- [CUmemorytype dstMemoryType](#)
- [size\\_t dstPitch](#)
- [size\\_t dstXInBytes](#)
- [size\\_t dstY](#)
- [size\\_t dstZ](#)
- [size\\_t Height](#)
- [void \\* reserved0](#)
- [void \\* reserved1](#)
- [CUarray srcArray](#)
- [CUdeviceptr srcDevice](#)
- [size\\_t srcHeight](#)
- [const void \\* srcHost](#)
- [size\\_t srcLOD](#)
- [CUmemorytype srcMemoryType](#)
- [size\\_t srcPitch](#)
- [size\\_t srcXInBytes](#)
- [size\\_t srcY](#)
- [size\\_t srcZ](#)
- [size\\_t WidthInBytes](#)

### 5.4.1 Detailed Description

3D memory copy parameters

### 5.4.2 Field Documentation

#### 5.4.2.1 [size\\_t CUDA\\_MEMCPY3D\\_st::Depth](#)

Depth of 3D memory copy

#### 5.4.2.2 [CUarray CUDA\\_MEMCPY3D\\_st::dstArray](#)

Destination array reference

#### 5.4.2.3 [CUdeviceptr CUDA\\_MEMCPY3D\\_st::dstDevice](#)

Destination device pointer

**5.4.2.4 size\_t CUDA\_MEMCPY3D\_st::dstHeight**

Destination height (ignored when dst is array; may be 0 if Depth==1)

**5.4.2.5 void\* CUDA\_MEMCPY3D\_st::dstHost**

Destination host pointer

**5.4.2.6 size\_t CUDA\_MEMCPY3D\_st::dstLOD**

Destination LOD

**5.4.2.7 CUmemorytype CUDA\_MEMCPY3D\_st::dstMemoryType**

Destination memory type (host, device, array)

**5.4.2.8 size\_t CUDA\_MEMCPY3D\_st::dstPitch**

Destination pitch (ignored when dst is array)

**5.4.2.9 size\_t CUDA\_MEMCPY3D\_st::dstXInBytes**

Destination X in bytes

**5.4.2.10 size\_t CUDA\_MEMCPY3D\_st::dstY**

Destination Y

**5.4.2.11 size\_t CUDA\_MEMCPY3D\_st::dstZ**

Destination Z

**5.4.2.12 size\_t CUDA\_MEMCPY3D\_st::Height**

Height of 3D memory copy

**5.4.2.13 void\* CUDA\_MEMCPY3D\_st::reserved0**

Must be NULL

**5.4.2.14 void\* CUDA\_MEMCPY3D\_st::reserved1**

Must be NULL

**5.4.2.15 CUarray CUDA\_MEMCPY3D\_st::srcArray**

Source array reference

**5.4.2.16 CUdeviceptr CUDA\_MEMCPY3D\_st::srcDevice**

Source device pointer

**5.4.2.17 size\_t CUDA\_MEMCPY3D\_st::srcHeight**

Source height (ignored when src is array; may be 0 if Depth==1)

**5.4.2.18 const void\* CUDA\_MEMCPY3D\_st::srcHost**

Source host pointer

**5.4.2.19 size\_t CUDA\_MEMCPY3D\_st::srcLOD**

Source LOD

**5.4.2.20 CUmemorytype CUDA\_MEMCPY3D\_st::srcMemoryType**

Source memory type (host, device, array)

**5.4.2.21 size\_t CUDA\_MEMCPY3D\_st::srcPitch**

Source pitch (ignored when src is array)

**5.4.2.22 size\_t CUDA\_MEMCPY3D\_st::srcXInBytes**

Source X in bytes

**5.4.2.23 size\_t CUDA\_MEMCPY3D\_st::srcY**

Source Y

**5.4.2.24 size\_t CUDA\_MEMCPY3D\_st::srcZ**

Source Z

**5.4.2.25 size\_t CUDA\_MEMCPY3D\_st::WidthInBytes**

Width of 3D memory copy in bytes

## 5.5 cudaChannelFormatDesc Struct Reference

### Data Fields

- enum [cudaChannelFormatKind](#) f
- int [w](#)
- int [x](#)
- int [y](#)
- int [z](#)

### 5.5.1 Detailed Description

CUDA Channel format descriptor

### 5.5.2 Field Documentation

#### 5.5.2.1 enum [cudaChannelFormatKind](#) [cudaChannelFormatDesc::f](#)

Channel format kind

#### 5.5.2.2 int [cudaChannelFormatDesc::w](#)

w

#### 5.5.2.3 int [cudaChannelFormatDesc::x](#)

x

#### 5.5.2.4 int [cudaChannelFormatDesc::y](#)

y

#### 5.5.2.5 int [cudaChannelFormatDesc::z](#)

z

## 5.6 cudaDeviceProp Struct Reference

### Data Fields

- int [canMapHostMemory](#)
- int [clockRate](#)
- int [computeMode](#)
- int [concurrentKernels](#)
- int [deviceOverlap](#)
- int [ECCEnabled](#)
- int [integrated](#)
- int [kernelExecTimeoutEnabled](#)
- int [major](#)
- int [maxGridSize](#) [3]
- int [maxTexture1D](#)
- int [maxTexture2D](#) [2]
- int [maxTexture2DArray](#) [3]
- int [maxTexture3D](#) [3]
- int [maxThreadsDim](#) [3]
- int [maxThreadsPerBlock](#)
- size\_t [memPitch](#)
- int [minor](#)
- int [multiProcessorCount](#)
- char [name](#) [256]
- int [pciBusID](#)
- int [pciDeviceID](#)
- int [regsPerBlock](#)
- size\_t [sharedMemPerBlock](#)
- size\_t [surfaceAlignment](#)
- int [tccDriver](#)
- size\_t [textureAlignment](#)
- size\_t [totalConstMem](#)
- size\_t [totalGlobalMem](#)
- int [warpSize](#)

### 5.6.1 Detailed Description

CUDA device properties

### 5.6.2 Field Documentation

#### 5.6.2.1 int cudaDeviceProp::canMapHostMemory

Device can map host memory with `cudaHostAlloc/cudaHostGetDevicePointer`

#### 5.6.2.2 int cudaDeviceProp::clockRate

Clock frequency in kilohertz

**5.6.2.3 int cudaDeviceProp::computeMode**

Compute mode (See [cudaComputeMode](#))

**5.6.2.4 int cudaDeviceProp::concurrentKernels**

Device can possibly execute multiple kernels concurrently

**5.6.2.5 int cudaDeviceProp::deviceOverlap**

Device can concurrently copy memory and execute a kernel

**5.6.2.6 int cudaDeviceProp::ECCEnabled**

Device has ECC support enabled

**5.6.2.7 int cudaDeviceProp::integrated**

Device is integrated as opposed to discrete

**5.6.2.8 int cudaDeviceProp::kernelExecTimeoutEnabled**

Specified whether there is a run time limit on kernels

**5.6.2.9 int cudaDeviceProp::major**

Major compute capability

**5.6.2.10 int cudaDeviceProp::maxGridSize[3]**

Maximum size of each dimension of a grid

**5.6.2.11 int cudaDeviceProp::maxTexture1D**

Maximum 1D texture size

**5.6.2.12 int cudaDeviceProp::maxTexture2D[2]**

Maximum 2D texture dimensions

**5.6.2.13 int cudaDeviceProp::maxTexture2DArray[3]**

Maximum 2D texture array dimensions

**5.6.2.14 int cudaDeviceProp::maxTexture3D[3]**

Maximum 3D texture dimensions

**5.6.2.15 int cudaDeviceProp::maxThreadsDim[3]**

Maximum size of each dimension of a block

**5.6.2.16 int cudaDeviceProp::maxThreadsPerBlock**

Maximum number of threads per block

**5.6.2.17 size\_t cudaDeviceProp::memPitch**

Maximum pitch in bytes allowed by memory copies

**5.6.2.18 int cudaDeviceProp::minor**

Minor compute capability

**5.6.2.19 int cudaDeviceProp::multiProcessorCount**

Number of multiprocessors on device

**5.6.2.20 char cudaDeviceProp::name[256]**

ASCII string identifying device

**5.6.2.21 int cudaDeviceProp::pciBusID**

PCI bus ID of the device

**5.6.2.22 int cudaDeviceProp::pciDeviceID**

PCI device ID of the device

**5.6.2.23 int cudaDeviceProp::regsPerBlock**

32-bit registers available per block

**5.6.2.24 size\_t cudaDeviceProp::sharedMemPerBlock**

Shared memory available per block in bytes

**5.6.2.25 size\_t cudaDeviceProp::surfaceAlignment**

Alignment requirements for surfaces

**5.6.2.26 int cudaDeviceProp::tccDriver**

1 if device is a Tesla device using TCC driver, 0 otherwise

**5.6.2.27 size\_t cudaDeviceProp::textureAlignment**

Alignment requirement for textures

**5.6.2.28 size\_t cudaDeviceProp::totalConstMem**

Constant memory available on device in bytes

**5.6.2.29 size\_t cudaDeviceProp::totalGlobalMem**

Global memory available on device in bytes

**5.6.2.30 int cudaDeviceProp::warpSize**

Warp size in threads

## 5.7 cudaExtent Struct Reference

### Data Fields

- [size\\_t depth](#)
- [size\\_t height](#)
- [size\\_t width](#)

### 5.7.1 Detailed Description

CUDA extent

See also:

[make\\_cudaExtent](#)

### 5.7.2 Field Documentation

#### 5.7.2.1 `size_t cudaExtent::depth`

Depth in elements

#### 5.7.2.2 `size_t cudaExtent::height`

Height in elements

#### 5.7.2.3 `size_t cudaExtent::width`

Width in elements when referring to array memory, in bytes when referring to linear memory

## 5.8 cudaFuncAttributes Struct Reference

### Data Fields

- int [binaryVersion](#)
- size\_t [constSizeBytes](#)
- size\_t [localSizeBytes](#)
- int [maxThreadsPerBlock](#)
- int [numRegs](#)
- int [ptxVersion](#)
- size\_t [sharedSizeBytes](#)

### 5.8.1 Detailed Description

CUDA function attributes

### 5.8.2 Field Documentation

#### 5.8.2.1 int cudaFuncAttributes::binaryVersion

The binary architecture version for which the function was compiled. This value is the major binary version \* 10 + the minor binary version, so a binary version 1.3 function would return the value 13.

#### 5.8.2.2 size\_t cudaFuncAttributes::constSizeBytes

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

#### 5.8.2.3 size\_t cudaFuncAttributes::localSizeBytes

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

#### 5.8.2.4 int cudaFuncAttributes::maxThreadsPerBlock

The maximum number of threads per block, beyond which a launch of the function would fail. This number depends on both the function and the device on which the function is currently loaded.

#### 5.8.2.5 int cudaFuncAttributes::numRegs

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

#### 5.8.2.6 int cudaFuncAttributes::ptxVersion

The PTX virtual architecture version for which the function was compiled. This value is the major PTX version \* 10 + the minor PTX version, so a PTX version 1.3 function would return the value 13.

### 5.8.2.7 `size_t cudaFuncAttributes::sharedSizeBytes`

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

## 5.9 cudaMemcpy3DParms Struct Reference

### Data Fields

- struct [cudaArray](#) \* [dstArray](#)
- struct [cudaPos](#) [dstPos](#)
- struct [cudaPitchedPtr](#) [dstPtr](#)
- struct [cudaExtent](#) [extent](#)
- enum [cudaMemcpyKind](#) [kind](#)
- struct [cudaArray](#) \* [srcArray](#)
- struct [cudaPos](#) [srcPos](#)
- struct [cudaPitchedPtr](#) [srcPtr](#)

### 5.9.1 Detailed Description

CUDA 3D memory copying parameters

### 5.9.2 Field Documentation

#### 5.9.2.1 struct [cudaArray](#)\* [cudaMemcpy3DParms::dstArray](#) [read]

Destination memory address

#### 5.9.2.2 struct [cudaPos](#) [cudaMemcpy3DParms::dstPos](#) [read]

Destination position offset

#### 5.9.2.3 struct [cudaPitchedPtr](#) [cudaMemcpy3DParms::dstPtr](#) [read]

Pitched destination memory address

#### 5.9.2.4 struct [cudaExtent](#) [cudaMemcpy3DParms::extent](#) [read]

Requested memory copy size

#### 5.9.2.5 enum [cudaMemcpyKind](#) [cudaMemcpy3DParms::kind](#)

Type of transfer

#### 5.9.2.6 struct [cudaArray](#)\* [cudaMemcpy3DParms::srcArray](#) [read]

Source memory address

#### 5.9.2.7 struct [cudaPos](#) [cudaMemcpy3DParms::srcPos](#) [read]

Source position offset

**5.9.2.8 struct cudaPitchedPtr cudaMemcpy3DParms::srcPtr** [read]

Pitched source memory address

## 5.10 cudaPitchedPtr Struct Reference

### Data Fields

- [size\\_t pitch](#)
- [void \\* ptr](#)
- [size\\_t xsize](#)
- [size\\_t ysize](#)

### 5.10.1 Detailed Description

CUDA Pitched memory pointer

See also:

[make\\_cudaPitchedPtr](#)

### 5.10.2 Field Documentation

#### 5.10.2.1 `size_t cudaPitchedPtr::pitch`

Pitch of allocated memory in bytes

#### 5.10.2.2 `void* cudaPitchedPtr::ptr`

Pointer to allocated memory

#### 5.10.2.3 `size_t cudaPitchedPtr::xsize`

Logical width of allocation in elements

#### 5.10.2.4 `size_t cudaPitchedPtr::ysize`

Logical height of allocation in elements

## 5.11 cudaPos Struct Reference

### Data Fields

- [size\\_t x](#)
- [size\\_t y](#)
- [size\\_t z](#)

### 5.11.1 Detailed Description

CUDA 3D position

See also:

[make\\_cudaPos](#)

### 5.11.2 Field Documentation

#### 5.11.2.1 size\_t cudaPos::x

x

#### 5.11.2.2 size\_t cudaPos::y

y

#### 5.11.2.3 size\_t cudaPos::z

z

## 5.12 CUdevprop\_st Struct Reference

### Data Fields

- int [clockRate](#)
- int [maxGridSize](#) [3]
- int [maxThreadsDim](#) [3]
- int [maxThreadsPerBlock](#)
- int [memPitch](#)
- int [regsPerBlock](#)
- int [sharedMemPerBlock](#)
- int [SIMDWidth](#)
- int [textureAlign](#)
- int [totalConstantMemory](#)

### 5.12.1 Detailed Description

Legacy device properties

### 5.12.2 Field Documentation

#### 5.12.2.1 int CUdevprop\_st::clockRate

Clock frequency in kilohertz

#### 5.12.2.2 int CUdevprop\_st::maxGridSize[3]

Maximum size of each dimension of a grid

#### 5.12.2.3 int CUdevprop\_st::maxThreadsDim[3]

Maximum size of each dimension of a block

#### 5.12.2.4 int CUdevprop\_st::maxThreadsPerBlock

Maximum number of threads per block

#### 5.12.2.5 int CUdevprop\_st::memPitch

Maximum pitch in bytes allowed by memory copies

#### 5.12.2.6 int CUdevprop\_st::regsPerBlock

32-bit registers available per block

#### 5.12.2.7 int CUdevprop\_st::sharedMemPerBlock

Shared memory available per block in bytes

**5.12.2.8 int CUdevprop\_st::SIMDWidth**

Warp size in threads

**5.12.2.9 int CUdevprop\_st::textureAlign**

Alignment requirement for textures

**5.12.2.10 int CUdevprop\_st::totalConstantMemory**

Constant memory available on device in bytes

## 5.13 surfaceReference Struct Reference

### Data Fields

- struct [cudaChannelFormatDesc](#) `channelDesc`

### 5.13.1 Detailed Description

CUDA Surface reference

### 5.13.2 Field Documentation

**5.13.2.1** struct `cudaChannelFormatDesc` `surfaceReference::channelDesc` [read]

Channel descriptor for surface reference

## 5.14 textureReference Struct Reference

### Data Fields

- enum `cudaTextureAddressMode` `addressMode` [3]
- struct `cudaChannelFormatDesc` `channelDesc`
- enum `cudaTextureFilterMode` `filterMode`
- int `normalized`

### 5.14.1 Detailed Description

CUDA texture reference

### 5.14.2 Field Documentation

#### 5.14.2.1 enum `cudaTextureAddressMode` `textureReference::addressMode`[3]

Texture address mode for up to 3 dimensions

#### 5.14.2.2 struct `cudaChannelFormatDesc` `textureReference::channelDesc` [read]

Channel descriptor for the texture reference

#### 5.14.2.3 enum `cudaTextureFilterMode` `textureReference::filterMode`

Texture filter mode

#### 5.14.2.4 int `textureReference::normalized`

Indicates whether texture reads are normalized or not

# Index

addressMode  
    textureReference, 334

binaryVersion  
    cudaFuncAttributes, 325

C++ API Routines, 99

canMapHostMemory  
    cudaDeviceProp, 320

channelDesc  
    surfaceReference, 333  
    textureReference, 334

clockRate  
    cudaDeviceProp, 320  
    CUdevprop\_st, 331

computeMode  
    cudaDeviceProp, 320

concurrentKernels  
    cudaDeviceProp, 321

constSizeBytes  
    cudaFuncAttributes, 325

Context Management, 173

CU\_AD\_FORMAT\_FLOAT  
    CUDA\_TYPES, 157

CU\_AD\_FORMAT\_HALF  
    CUDA\_TYPES, 157

CU\_AD\_FORMAT\_SIGNED\_INT16  
    CUDA\_TYPES, 157

CU\_AD\_FORMAT\_SIGNED\_INT32  
    CUDA\_TYPES, 157

CU\_AD\_FORMAT\_SIGNED\_INT8  
    CUDA\_TYPES, 157

CU\_AD\_FORMAT\_UNSIGNED\_INT16  
    CUDA\_TYPES, 157

CU\_AD\_FORMAT\_UNSIGNED\_INT32  
    CUDA\_TYPES, 157

CU\_AD\_FORMAT\_UNSIGNED\_INT8  
    CUDA\_TYPES, 157

CU\_COMPUTEMODE\_DEFAULT  
    CUDA\_TYPES, 157

CU\_COMPUTEMODE\_EXCLUSIVE  
    CUDA\_TYPES, 157

CU\_COMPUTEMODE\_PROHIBITED  
    CUDA\_TYPES, 158

CU\_CTX\_BLOCKING\_SYNC  
    CUDA\_TYPES, 158

CU\_CTX\_LMEM\_RESIZE\_TO\_MAX  
    CUDA\_TYPES, 158

CU\_CTX\_MAP\_HOST  
    CUDA\_TYPES, 158

CU\_CTX\_SCHED\_AUTO  
    CUDA\_TYPES, 158

CU\_CTX\_SCHED\_SPIN  
    CUDA\_TYPES, 158

CU\_CTX\_SCHED\_YIELD  
    CUDA\_TYPES, 158

CU\_CUBEMAP\_FACE\_NEGATIVE\_X  
    CUDA\_TYPES, 157

CU\_CUBEMAP\_FACE\_NEGATIVE\_Y  
    CUDA\_TYPES, 157

CU\_CUBEMAP\_FACE\_NEGATIVE\_Z  
    CUDA\_TYPES, 157

CU\_CUBEMAP\_FACE\_POSITIVE\_X  
    CUDA\_TYPES, 157

CU\_CUBEMAP\_FACE\_POSITIVE\_Y  
    CUDA\_TYPES, 157

CU\_CUBEMAP\_FACE\_POSITIVE\_Z  
    CUDA\_TYPES, 157

CU\_D3D10\_DEVICE\_LIST\_ALL  
    CUDA\_D3D10, 287

CU\_D3D10\_DEVICE\_LIST\_CURRENT\_FRAME  
    CUDA\_D3D10, 287

CU\_D3D10\_DEVICE\_LIST\_NEXT\_FRAME  
    CUDA\_D3D10, 287

CU\_D3D11\_DEVICE\_LIST\_ALL  
    CUDA\_D3D11, 302

CU\_D3D11\_DEVICE\_LIST\_CURRENT\_FRAME  
    CUDA\_D3D11, 302

CU\_D3D11\_DEVICE\_LIST\_NEXT\_FRAME  
    CUDA\_D3D11, 302

CU\_D3D9\_DEVICE\_LIST\_ALL  
    CUDA\_D3D9, 272

CU\_D3D9\_DEVICE\_LIST\_CURRENT\_FRAME  
    CUDA\_D3D9, 272

CU\_D3D9\_DEVICE\_LIST\_NEXT\_FRAME  
    CUDA\_D3D9, 272

CU\_DEVICE\_ATTRIBUTE\_CAN\_MAP\_HOST\_MEMORY  
    CUDA\_TYPES, 160

CU\_DEVICE\_ATTRIBUTE\_CLOCK\_RATE

- CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_COMPUTE\_MODE
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_CONCURRENT\_KERNELS
  - CUDA\_TYPES, 161
- CU\_DEVICE\_ATTRIBUTE\_ECC\_ENABLED
  - CUDA\_TYPES, 161
- CU\_DEVICE\_ATTRIBUTE\_GPU\_OVERLAP
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_INTEGRATED
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_KERNEL\_EXEC\_TIMEOUT
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_MAX\_BLOCK\_DIM\_X
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_MAX\_BLOCK\_DIM\_Y
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_MAX\_BLOCK\_DIM\_Z
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_MAX\_GRID\_DIM\_X
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_MAX\_GRID\_DIM\_Y
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_MAX\_GRID\_DIM\_Z
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_MAX\_PITCH
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_MAX\_REGISTERS\_PER\_BLOCK
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_MAX\_SHARED\_MEMORY\_PER\_BLOCK
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_MAX\_THREADS\_PER\_BLOCK
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE1D\_WIDTH
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE2D\_ARRAY\_HEIGHT
  - CUDA\_TYPES, 161
- CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE2D\_ARRAY\_NUMSLICES
  - CUDA\_TYPES, 161
- CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE2D\_ARRAY\_WIDTH
  - CUDA\_TYPES, 161
- CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE2D\_HEIGHT
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE2D\_WIDTH
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE3D\_DEPTH
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE3D\_HEIGHT
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE3D\_WIDTH
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_MULTIPROCESSOR\_COUNT
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_PCI\_BUS\_ID
  - CUDA\_TYPES, 161
- CU\_DEVICE\_ATTRIBUTE\_PCI\_DEVICE\_ID
  - CUDA\_TYPES, 161
- CU\_DEVICE\_ATTRIBUTE\_REGISTERS\_PER\_BLOCK
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_SHARED\_MEMORY\_PER\_BLOCK
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_SURFACE\_ALIGNMENT
  - CUDA\_TYPES, 161
- CU\_DEVICE\_ATTRIBUTE\_TCC\_DRIVER
  - CUDA\_TYPES, 161
- CU\_DEVICE\_ATTRIBUTE\_TEXTURE\_ALIGNMENT
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_TOTAL\_CONSTANT\_MEMORY
  - CUDA\_TYPES, 160
- CU\_DEVICE\_ATTRIBUTE\_WARP\_SIZE
  - CUDA\_TYPES, 160
- CU\_EVENT\_BLOCKING\_SYNC
  - CUDA\_TYPES, 161
- CU\_EVENT\_DEFAULT
  - CUDA\_TYPES, 161
- CU\_EVENT\_DISABLE\_TIMING
  - CUDA\_TYPES, 161
- CU\_FUNC\_ATTRIBUTE\_BINARY\_VERSION
  - CUDA\_TYPES, 162
- CU\_FUNC\_ATTRIBUTE\_CONST\_SIZE\_BYTES
  - CUDA\_TYPES, 162
- CU\_FUNC\_ATTRIBUTE\_LOCAL\_SIZE\_BYTES
  - CUDA\_TYPES, 162
- CU\_FUNC\_ATTRIBUTE\_MAX\_THREADS\_PER\_BLOCK
  - CUDA\_TYPES, 162
- CU\_FUNC\_ATTRIBUTE\_NUM\_REGS
  - CUDA\_TYPES, 162
- CU\_FUNC\_ATTRIBUTE\_PTX\_VERSION

- CUDA\_TYPES, 162
- CU\_FUNC\_ATTRIBUTE\_SHARED\_SIZE\_BYTES
  - CUDA\_TYPES, 162
- CU\_FUNC\_CACHE\_PREFER\_L1
  - CUDA\_TYPES, 161
- CU\_FUNC\_CACHE\_PREFER\_NONE
  - CUDA\_TYPES, 161
- CU\_FUNC\_CACHE\_PREFER\_SHARED
  - CUDA\_TYPES, 161
- CU\_JIT\_ERROR\_LOG\_BUFFER
  - CUDA\_TYPES, 163
- CU\_JIT\_ERROR\_LOG\_BUFFER\_SIZE\_BYTES
  - CUDA\_TYPES, 163
- CU\_JIT\_FALLBACK\_STRATEGY
  - CUDA\_TYPES, 163
- CU\_JIT\_INFO\_LOG\_BUFFER
  - CUDA\_TYPES, 163
- CU\_JIT\_INFO\_LOG\_BUFFER\_SIZE\_BYTES
  - CUDA\_TYPES, 163
- CU\_JIT\_MAX\_REGISTERS
  - CUDA\_TYPES, 162
- CU\_JIT\_OPTIMIZATION\_LEVEL
  - CUDA\_TYPES, 163
- CU\_JIT\_TARGET
  - CUDA\_TYPES, 163
- CU\_JIT\_TARGET\_FROM\_CUCONTEXT
  - CUDA\_TYPES, 163
- CU\_JIT\_THREADS\_PER\_BLOCK
  - CUDA\_TYPES, 162
- CU\_JIT\_WALL\_TIME
  - CUDA\_TYPES, 163
- CU\_LIMIT\_MALLOC\_HEAP\_SIZE
  - CUDA\_TYPES, 164
- CU\_LIMIT\_PRINTF\_FIFO\_SIZE
  - CUDA\_TYPES, 164
- CU\_LIMIT\_STACK\_SIZE
  - CUDA\_TYPES, 164
- CU\_MEMORYTYPE\_ARRAY
  - CUDA\_TYPES, 164
- CU\_MEMORYTYPE\_DEVICE
  - CUDA\_TYPES, 164
- CU\_MEMORYTYPE\_HOST
  - CUDA\_TYPES, 164
- CU\_PREFER\_BINARY
  - CUDA\_TYPES, 162
- CU\_PREFER\_PTX
  - CUDA\_TYPES, 162
- CU\_TARGET\_COMPUTE\_10
  - CUDA\_TYPES, 163
- CU\_TARGET\_COMPUTE\_11
  - CUDA\_TYPES, 163
- CU\_TARGET\_COMPUTE\_12
  - CUDA\_TYPES, 163
- CU\_TARGET\_COMPUTE\_13
  - CUDA\_TYPES, 163
- CU\_TARGET\_COMPUTE\_20
  - CUDA\_TYPES, 163
- CU\_TARGET\_COMPUTE\_21
  - CUDA\_TYPES, 163
- CU\_TR\_ADDRESS\_MODE\_BORDER
  - CUDA\_TYPES, 157
- CU\_TR\_ADDRESS\_MODE\_CLAMP
  - CUDA\_TYPES, 157
- CU\_TR\_ADDRESS\_MODE\_MIRROR
  - CUDA\_TYPES, 157
- CU\_TR\_ADDRESS\_MODE\_WRAP
  - CUDA\_TYPES, 157
- CU\_TR\_FILTER\_MODE\_LINEAR
  - CUDA\_TYPES, 161
- CU\_TR\_FILTER\_MODE\_POINT
  - CUDA\_TYPES, 161
- CU\_MEMHOSTALLOC\_DEVICEMAP
  - CUDA\_TYPES, 153
- CU\_MEMHOSTALLOC\_PORTABLE
  - CUDA\_TYPES, 153
- CU\_MEMHOSTALLOC\_WRITECOMBINED
  - CUDA\_TYPES, 153
- CU\_PARAM\_TR\_DEFAULT
  - CUDA\_TYPES, 153
- CU\_TRSA\_OVERRIDE\_FORMAT
  - CUDA\_TYPES, 153
- CU\_TRSF\_NORMALIZED\_COORDINATES
  - CUDA\_TYPES, 153
- CU\_TRSF\_READ\_AS\_INTEGER
  - CUDA\_TYPES, 153
- CU\_TRSF\_SRGB
  - CUDA\_TYPES, 153
- CUaddress\_mode
  - CUDA\_TYPES, 154
- CUaddress\_mode\_enum
  - CUDA\_TYPES, 157
- CUarray
  - CUDA\_TYPES, 154
- cuArray3DCreate
  - CUDA\_MEM, 191
- cuArray3DGetDescriptor
  - CUDA\_MEM, 192
- CUarray\_cubemap\_face
  - CUDA\_TYPES, 154
- CUarray\_cubemap\_face\_enum
  - CUDA\_TYPES, 157
- CUarray\_format
  - CUDA\_TYPES, 154
- CUarray\_format\_enum
  - CUDA\_TYPES, 157
- cuArrayCreate
  - CUDA\_MEM, 193
- cuArrayDestroy

- CUDA\_MEM, [194](#)
- cuArrayGetDescriptor
  - CUDA\_MEM, [195](#)
- CUcomputemode
  - CUDA\_TYPES, [154](#)
- CUcomputemode\_enum
  - CUDA\_TYPES, [157](#)
- CUcontext
  - CUDA\_TYPES, [154](#)
- CUctx\_flags
  - CUDA\_TYPES, [154](#)
- CUctx\_flags\_enum
  - CUDA\_TYPES, [158](#)
- cuCtxAttach
  - CUDA\_CTX, [174](#)
- cuCtxCreate
  - CUDA\_CTX, [174](#)
- cuCtxDestroy
  - CUDA\_CTX, [175](#)
- cuCtxDetach
  - CUDA\_CTX, [176](#)
- cuCtxGetApiVersion
  - CUDA\_CTX, [176](#)
- cuCtxGetCacheConfig
  - CUDA\_CTX, [176](#)
- cuCtxGetDevice
  - CUDA\_CTX, [177](#)
- cuCtxGetLimit
  - CUDA\_CTX, [177](#)
- cuCtxPopCurrent
  - CUDA\_CTX, [178](#)
- cuCtxPushCurrent
  - CUDA\_CTX, [178](#)
- cuCtxSetCacheConfig
  - CUDA\_CTX, [179](#)
- cuCtxSetLimit
  - CUDA\_CTX, [180](#)
- cuCtxSynchronize
  - CUDA\_CTX, [180](#)
- cuD3D10CtxCreate
  - CUDA\_D3D10, [287](#)
- cuD3D10CtxCreateOnDevice
  - CUDA\_D3D10, [287](#)
- CUd3d10DeviceList
  - CUDA\_D3D10, [287](#)
- CUd3d10DeviceList\_enum
  - CUDA\_D3D10, [287](#)
- cuD3D10GetDevice
  - CUDA\_D3D10, [288](#)
- cuD3D10GetDevices
  - CUDA\_D3D10, [288](#)
- cuD3D10GetDirect3DDevice
  - CUDA\_D3D10, [289](#)
- CUD3D10map\_flags
  - CUDA\_D3D10\_DEPRECATED, [293](#)
- CUD3D10map\_flags\_enum
  - CUDA\_D3D10\_DEPRECATED, [293](#)
- cuD3D10MapResources
  - CUDA\_D3D10\_DEPRECATED, [293](#)
- CUD3D10register\_flags
  - CUDA\_D3D10\_DEPRECATED, [293](#)
- CUD3D10register\_flags\_enum
  - CUDA\_D3D10\_DEPRECATED, [293](#)
- cuD3D10RegisterResource
  - CUDA\_D3D10\_DEPRECATED, [294](#)
- cuD3D10ResourceGetMappedArray
  - CUDA\_D3D10\_DEPRECATED, [295](#)
- cuD3D10ResourceGetMappedPitch
  - CUDA\_D3D10\_DEPRECATED, [296](#)
- cuD3D10ResourceGetMappedPointer
  - CUDA\_D3D10\_DEPRECATED, [296](#)
- cuD3D10ResourceGetMappedSize
  - CUDA\_D3D10\_DEPRECATED, [297](#)
- cuD3D10ResourceGetSurfaceDimensions
  - CUDA\_D3D10\_DEPRECATED, [298](#)
- cuD3D10ResourceSetMapFlags
  - CUDA\_D3D10\_DEPRECATED, [298](#)
- cuD3D10UnmapResources
  - CUDA\_D3D10\_DEPRECATED, [299](#)
- cuD3D10UnregisterResource
  - CUDA\_D3D10\_DEPRECATED, [300](#)
- cuD3D11CtxCreate
  - CUDA\_D3D11, [302](#)
- cuD3D11CtxCreateOnDevice
  - CUDA\_D3D11, [302](#)
- CUd3d11DeviceList
  - CUDA\_D3D11, [301](#)
- CUd3d11DeviceList\_enum
  - CUDA\_D3D11, [302](#)
- cuD3D11GetDevice
  - CUDA\_D3D11, [303](#)
- cuD3D11GetDevices
  - CUDA\_D3D11, [303](#)
- cuD3D11GetDirect3DDevice
  - CUDA\_D3D11, [304](#)
- cuD3D9CtxCreate
  - CUDA\_D3D9, [272](#)
- cuD3D9CtxCreateOnDevice
  - CUDA\_D3D9, [272](#)
- CUd3d9DeviceList
  - CUDA\_D3D9, [272](#)
- CUd3d9DeviceList\_enum
  - CUDA\_D3D9, [272](#)
- cuD3D9GetDevice
  - CUDA\_D3D9, [273](#)
- cuD3D9GetDevices
  - CUDA\_D3D9, [273](#)
- cuD3D9GetDirect3DDevice

- CUDA\_D3D9, 274
- CuD3d9map\_flags
  - CUDA\_D3D9\_DEPRECATED, 278
- CuD3d9map\_flags\_enum
  - CUDA\_D3D9\_DEPRECATED, 278
- cuD3D9MapResources
  - CUDA\_D3D9\_DEPRECATED, 278
- CuD3d9register\_flags
  - CUDA\_D3D9\_DEPRECATED, 278
- CuD3d9register\_flags\_enum
  - CUDA\_D3D9\_DEPRECATED, 278
- cuD3D9RegisterResource
  - CUDA\_D3D9\_DEPRECATED, 279
- cuD3D9ResourceGetMappedArray
  - CUDA\_D3D9\_DEPRECATED, 280
- cuD3D9ResourceGetMappedPitch
  - CUDA\_D3D9\_DEPRECATED, 281
- cuD3D9ResourceGetMappedPointer
  - CUDA\_D3D9\_DEPRECATED, 282
- cuD3D9ResourceGetMappedSize
  - CUDA\_D3D9\_DEPRECATED, 282
- cuD3D9ResourceGetSurfaceDimensions
  - CUDA\_D3D9\_DEPRECATED, 283
- cuD3D9ResourceSetMapFlags
  - CUDA\_D3D9\_DEPRECATED, 284
- cuD3D9UnmapResources
  - CUDA\_D3D9\_DEPRECATED, 284
- cuD3D9UnregisterResource
  - CUDA\_D3D9\_DEPRECATED, 285
- CUDA Driver API, 147
- CUDA Runtime API, 9
- CUDA\_D3D10
  - CU\_D3D10\_DEVICE\_LIST\_ALL, 287
  - CU\_D3D10\_DEVICE\_LIST\_CURRENT\_FRAME, 287
  - CU\_D3D10\_DEVICE\_LIST\_NEXT\_FRAME, 287
- CUDA\_D3D11
  - CU\_D3D11\_DEVICE\_LIST\_ALL, 302
  - CU\_D3D11\_DEVICE\_LIST\_CURRENT\_FRAME, 302
  - CU\_D3D11\_DEVICE\_LIST\_NEXT\_FRAME, 302
- CUDA\_D3D9
  - CU\_D3D9\_DEVICE\_LIST\_ALL, 272
  - CU\_D3D9\_DEVICE\_LIST\_CURRENT\_FRAME, 272
  - CU\_D3D9\_DEVICE\_LIST\_NEXT\_FRAME, 272
- CUDA\_ERROR\_ALREADY\_ACQUIRED
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_ALREADY\_MAPPED
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_ARRAY\_IS\_MAPPED
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_CONTEXT\_ALREADY\_CURRENT
  - CUDA\_TYPES, 158
- CUDA\_ERROR\_DEINITIALIZED
  - CUDA\_TYPES, 158
- CUDA\_ERROR\_ECC\_UNCORRECTABLE
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_FILE\_NOT\_FOUND
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_INVALID\_CONTEXT
  - CUDA\_TYPES, 158
- CUDA\_ERROR\_INVALID\_DEVICE
  - CUDA\_TYPES, 158
- CUDA\_ERROR\_INVALID\_HANDLE
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_INVALID\_IMAGE
  - CUDA\_TYPES, 158
- CUDA\_ERROR\_INVALID\_SOURCE
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_INVALID\_VALUE
  - CUDA\_TYPES, 158
- CUDA\_ERROR\_LAUNCH\_FAILED
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_LAUNCH\_INCOMPATIBLE\_-TEXTURING
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_LAUNCH\_OUT\_OF\_RESOURCES
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_LAUNCH\_TIMEOUT
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_MAP\_FAILED
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_NO\_BINARY\_FOR\_GPU
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_NO\_DEVICE
  - CUDA\_TYPES, 158
- CUDA\_ERROR\_NOT\_FOUND
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_NOT\_INITIALIZED
  - CUDA\_TYPES, 158
- CUDA\_ERROR\_NOT\_MAPPED
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_NOT\_MAPPED\_AS\_ARRAY
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_NOT\_MAPPED\_AS\_POINTER
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_NOT\_READY
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_OPERATING\_SYSTEM
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_OUT\_OF\_MEMORY
  - CUDA\_TYPES, 158
- CUDA\_ERROR\_SHARED\_OBJECT\_INIT\_FAILED
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_SHARED\_OBJECT\_SYMBOL\_-NOT\_FOUND
  - CUDA\_TYPES, 159

- CUDA\_ERROR\_UNKNOWN
  - CUDA\_TYPES, 160
- CUDA\_ERROR\_UNMAP\_FAILED
  - CUDA\_TYPES, 159
- CUDA\_ERROR\_UNSUPPORTED\_LIMIT
  - CUDA\_TYPES, 159
- CUDA\_SUCCESS
  - CUDA\_TYPES, 158
- CUDA\_TYPES
  - CU\_AD\_FORMAT\_FLOAT, 157
  - CU\_AD\_FORMAT\_HALF, 157
  - CU\_AD\_FORMAT\_SIGNED\_INT16, 157
  - CU\_AD\_FORMAT\_SIGNED\_INT32, 157
  - CU\_AD\_FORMAT\_SIGNED\_INT8, 157
  - CU\_AD\_FORMAT\_UNSIGNED\_INT16, 157
  - CU\_AD\_FORMAT\_UNSIGNED\_INT32, 157
  - CU\_AD\_FORMAT\_UNSIGNED\_INT8, 157
  - CU\_COMPUTEMODE\_DEFAULT, 157
  - CU\_COMPUTEMODE\_EXCLUSIVE, 157
  - CU\_COMPUTEMODE\_PROHIBITED, 158
  - CU\_CTX\_BLOCKING\_SYNC, 158
  - CU\_CTX\_LMEM\_RESIZE\_TO\_MAX, 158
  - CU\_CTX\_MAP\_HOST, 158
  - CU\_CTX\_SCHED\_AUTO, 158
  - CU\_CTX\_SCHED\_SPIN, 158
  - CU\_CTX\_SCHED\_YIELD, 158
  - CU\_CUBEMAP\_FACE\_NEGATIVE\_X, 157
  - CU\_CUBEMAP\_FACE\_NEGATIVE\_Y, 157
  - CU\_CUBEMAP\_FACE\_NEGATIVE\_Z, 157
  - CU\_CUBEMAP\_FACE\_POSITIVE\_X, 157
  - CU\_CUBEMAP\_FACE\_POSITIVE\_Y, 157
  - CU\_CUBEMAP\_FACE\_POSITIVE\_Z, 157
  - CU\_DEVICE\_ATTRIBUTE\_CAN\_MAP\_HOST\_MEMORY, 160
  - CU\_DEVICE\_ATTRIBUTE\_CLOCK\_RATE, 160
  - CU\_DEVICE\_ATTRIBUTE\_COMPUTE\_MODE, 160
  - CU\_DEVICE\_ATTRIBUTE\_CONCURRENT\_KERNELS, 161
  - CU\_DEVICE\_ATTRIBUTE\_ECC\_ENABLED, 161
  - CU\_DEVICE\_ATTRIBUTE\_GPU\_OVERLAP, 160
  - CU\_DEVICE\_ATTRIBUTE\_INTEGRATED, 160
  - CU\_DEVICE\_ATTRIBUTE\_KERNEL\_EXEC\_TIMEOUT, 160
  - CU\_DEVICE\_ATTRIBUTE\_MAX\_BLOCK\_DIM\_X, 160
  - CU\_DEVICE\_ATTRIBUTE\_MAX\_BLOCK\_DIM\_Y, 160
  - CU\_DEVICE\_ATTRIBUTE\_MAX\_BLOCK\_DIM\_Z, 160
  - CU\_DEVICE\_ATTRIBUTE\_MAX\_GRID\_DIM\_X, 160
  - CU\_DEVICE\_ATTRIBUTE\_MAX\_GRID\_DIM\_Y, 160
  - CU\_DEVICE\_ATTRIBUTE\_MAX\_GRID\_DIM\_Z, 160
  - CU\_DEVICE\_ATTRIBUTE\_MAX\_PITCH, 160
  - CU\_DEVICE\_ATTRIBUTE\_MAX\_REGISTERS\_PER\_BLOCK, 160
  - CU\_DEVICE\_ATTRIBUTE\_MAX\_SHARED\_MEMORY\_PER\_BLOCK, 160
  - CU\_DEVICE\_ATTRIBUTE\_MAX\_THREADS\_PER\_BLOCK, 160
  - CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE1D\_WIDTH, 160
  - CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE2D\_ARRAY\_HEIGHT, 161
  - CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE2D\_ARRAY\_NUMSLICES, 161
  - CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE2D\_ARRAY\_WIDTH, 161
  - CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE2D\_HEIGHT, 160
  - CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE2D\_WIDTH, 160
  - CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE3D\_DEPTH, 160
  - CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE3D\_HEIGHT, 160
  - CU\_DEVICE\_ATTRIBUTE\_MAXIMUM\_TEXTURE3D\_WIDTH, 160
  - CU\_DEVICE\_ATTRIBUTE\_MULTIPROCESSOR\_COUNT, 160
  - CU\_DEVICE\_ATTRIBUTE\_PCI\_BUS\_ID, 161
  - CU\_DEVICE\_ATTRIBUTE\_PCI\_DEVICE\_ID, 161
  - CU\_DEVICE\_ATTRIBUTE\_REGISTERS\_PER\_BLOCK, 160
  - CU\_DEVICE\_ATTRIBUTE\_SHARED\_MEMORY\_PER\_BLOCK, 160
  - CU\_DEVICE\_ATTRIBUTE\_SURFACE\_ALIGNMENT, 161
  - CU\_DEVICE\_ATTRIBUTE\_TCC\_DRIVER, 161
  - CU\_DEVICE\_ATTRIBUTE\_TEXTURE\_ALIGNMENT, 160
  - CU\_DEVICE\_ATTRIBUTE\_TOTAL\_CONSTANT\_MEMORY, 160
  - CU\_DEVICE\_ATTRIBUTE\_WARP\_SIZE, 160
  - CU\_EVENT\_BLOCKING\_SYNC, 161
  - CU\_EVENT\_DEFAULT, 161
  - CU\_EVENT\_DISABLE\_TIMING, 161
  - CU\_FUNC\_ATTRIBUTE\_BINARY\_VERSION, 162
  - CU\_FUNC\_ATTRIBUTE\_CONST\_SIZE\_BYTES, 162
  - CU\_FUNC\_ATTRIBUTE\_LOCAL\_SIZE\_BYTES, 162

- 162
- CU\_FUNC\_ATTRIBUTE\_MAX\_THREADS\_-  
PER\_BLOCK, 162
- CU\_FUNC\_ATTRIBUTE\_NUM\_REGS, 162
- CU\_FUNC\_ATTRIBUTE\_PTX\_VERSION, 162
- CU\_FUNC\_ATTRIBUTE\_SHARED\_SIZE\_-  
BYTES, 162
- CU\_FUNC\_CACHE\_PREFER\_L1, 161
- CU\_FUNC\_CACHE\_PREFER\_NONE, 161
- CU\_FUNC\_CACHE\_PREFER\_SHARED, 161
- CU\_JIT\_ERROR\_LOG\_BUFFER, 163
- CU\_JIT\_ERROR\_LOG\_BUFFER\_SIZE\_BYTES,  
163
- CU\_JIT\_FALLBACK\_STRATEGY, 163
- CU\_JIT\_INFO\_LOG\_BUFFER, 163
- CU\_JIT\_INFO\_LOG\_BUFFER\_SIZE\_BYTES,  
163
- CU\_JIT\_MAX\_REGISTERS, 162
- CU\_JIT\_OPTIMIZATION\_LEVEL, 163
- CU\_JIT\_TARGET, 163
- CU\_JIT\_TARGET\_FROM\_CUCONTEXT, 163
- CU\_JIT\_THREADS\_PER\_BLOCK, 162
- CU\_JIT\_WALL\_TIME, 163
- CU\_LIMIT\_MALLOC\_HEAP\_SIZE, 164
- CU\_LIMIT\_PRINTF\_FIFO\_SIZE, 164
- CU\_LIMIT\_STACK\_SIZE, 164
- CU\_MEMORYTYPE\_ARRAY, 164
- CU\_MEMORYTYPE\_DEVICE, 164
- CU\_MEMORYTYPE\_HOST, 164
- CU\_PREFER\_BINARY, 162
- CU\_PREFER\_PTX, 162
- CU\_TARGET\_COMPUTE\_10, 163
- CU\_TARGET\_COMPUTE\_11, 163
- CU\_TARGET\_COMPUTE\_12, 163
- CU\_TARGET\_COMPUTE\_13, 163
- CU\_TARGET\_COMPUTE\_20, 163
- CU\_TARGET\_COMPUTE\_21, 163
- CU\_TR\_ADDRESS\_MODE\_BORDER, 157
- CU\_TR\_ADDRESS\_MODE\_CLAMP, 157
- CU\_TR\_ADDRESS\_MODE\_MIRROR, 157
- CU\_TR\_ADDRESS\_MODE\_WRAP, 157
- CU\_TR\_FILTER\_MODE\_LINEAR, 161
- CU\_TR\_FILTER\_MODE\_POINT, 161
- CUDA\_ERROR\_ALREADY\_ACQUIRED, 159
- CUDA\_ERROR\_ALREADY\_MAPPED, 159
- CUDA\_ERROR\_ARRAY\_IS\_MAPPED, 159
- CUDA\_ERROR\_CONTEXT\_ALREADY\_-  
CURRENT, 158
- CUDA\_ERROR\_DEINITIALIZED, 158
- CUDA\_ERROR\_ECC\_UNCORRECTABLE, 159
- CUDA\_ERROR\_FILE\_NOT\_FOUND, 159
- CUDA\_ERROR\_INVALID\_CONTEXT, 158
- CUDA\_ERROR\_INVALID\_DEVICE, 158
- CUDA\_ERROR\_INVALID\_HANDLE, 159
- CUDA\_ERROR\_INVALID\_IMAGE, 158
- CUDA\_ERROR\_INVALID\_SOURCE, 159
- CUDA\_ERROR\_INVALID\_VALUE, 158
- CUDA\_ERROR\_LAUNCH\_FAILED, 159
- CUDA\_ERROR\_LAUNCH\_INCOMPATIBLE\_-  
TEXTURING, 159
- CUDA\_ERROR\_LAUNCH\_OUT\_OF\_-  
RESOURCES, 159
- CUDA\_ERROR\_LAUNCH\_TIMEOUT, 159
- CUDA\_ERROR\_MAP\_FAILED, 159
- CUDA\_ERROR\_NO\_BINARY\_FOR\_GPU, 159
- CUDA\_ERROR\_NO\_DEVICE, 158
- CUDA\_ERROR\_NOT\_FOUND, 159
- CUDA\_ERROR\_NOT\_INITIALIZED, 158
- CUDA\_ERROR\_NOT\_MAPPED, 159
- CUDA\_ERROR\_NOT\_MAPPED\_AS\_ARRAY,  
159
- CUDA\_ERROR\_NOT\_MAPPED\_AS\_POINTER,  
159
- CUDA\_ERROR\_NOT\_READY, 159
- CUDA\_ERROR\_OPERATING\_SYSTEM, 159
- CUDA\_ERROR\_OUT\_OF\_MEMORY, 158
- CUDA\_ERROR\_SHARED\_OBJECT\_INIT\_-  
FAILED, 159
- CUDA\_ERROR\_SHARED\_OBJECT\_SYMBOL\_-  
NOT\_FOUND, 159
- CUDA\_ERROR\_UNKNOWN, 160
- CUDA\_ERROR\_UNMAP\_FAILED, 159
- CUDA\_ERROR\_UNSUPPORTED\_LIMIT, 159
- CUDA\_SUCCESS, 158
- CUDA\_ARRAY3D\_2DARRAY  
CUDA\_TYPES, 153
- CUDA\_ARRAY3D\_DESCRIPTOR  
CUDA\_TYPES, 154
- CUDA\_ARRAY3D\_DESCRIPTOR\_st, 311  
Depth, 311  
Flags, 311  
Format, 311  
Height, 311  
NumChannels, 311  
Width, 312
- CUDA\_ARRAY3D\_SURFACE\_LDST  
CUDA\_TYPES, 153
- CUDA\_ARRAY\_DESCRIPTOR  
CUDA\_TYPES, 154
- CUDA\_ARRAY\_DESCRIPTOR\_st, 313  
Format, 313  
Height, 313  
NumChannels, 313  
Width, 313
- CUDA\_CTX  
cuCtxAttach, 174  
cuCtxCreate, 174  
cuCtxDestroy, 175

- cuCtxDetach, 176
- cuCtxGetApiVersion, 176
- cuCtxGetCacheConfig, 176
- cuCtxGetDevice, 177
- cuCtxGetLimit, 177
- cuCtxPopCurrent, 178
- cuCtxPushCurrent, 178
- cuCtxSetCacheConfig, 179
- cuCtxSetLimit, 180
- cuCtxSynchronize, 180
- CUDA\_D3D10
  - cuD3D10CtxCreate, 287
  - cuD3D10CtxCreateOnDevice, 287
  - CUd3d10DeviceList, 287
  - CUd3d10DeviceList\_enum, 287
  - cuD3D10GetDevice, 288
  - cuD3D10GetDevices, 288
  - cuD3D10GetDirect3DDevice, 289
  - cuGraphicsD3D10RegisterResource, 289
- CUDA\_D3D10\_DEPRECATED
  - CUD3D10map\_flags, 293
  - CUD3D10map\_flags\_enum, 293
  - cuD3D10MapResources, 293
  - CUD3D10register\_flags, 293
  - CUD3D10register\_flags\_enum, 293
  - cuD3D10RegisterResource, 294
  - cuD3D10ResourceGetMappedArray, 295
  - cuD3D10ResourceGetMappedPitch, 296
  - cuD3D10ResourceGetMappedPointer, 296
  - cuD3D10ResourceGetMappedSize, 297
  - cuD3D10ResourceGetSurfaceDimensions, 298
  - cuD3D10ResourceSetMapFlags, 298
  - cuD3D10UnmapResources, 299
  - cuD3D10UnregisterResource, 300
- CUDA\_D3D11
  - cuD3D11CtxCreate, 302
  - cuD3D11CtxCreateOnDevice, 302
  - CUd3d11DeviceList, 301
  - CUd3d11DeviceList\_enum, 302
  - cuD3D11GetDevice, 303
  - cuD3D11GetDevices, 303
  - cuD3D11GetDirect3DDevice, 304
  - cuGraphicsD3D11RegisterResource, 304
- CUDA\_D3D9
  - cuD3D9CtxCreate, 272
  - cuD3D9CtxCreateOnDevice, 272
  - CUd3d9DeviceList, 272
  - CUd3d9DeviceList\_enum, 272
  - cuD3D9GetDevice, 273
  - cuD3D9GetDevices, 273
  - cuD3D9GetDirect3DDevice, 274
  - cuGraphicsD3D9RegisterResource, 274
- CUDA\_D3D9\_DEPRECATED
  - CUd3d9map\_flags, 278
  - CUd3d9map\_flags\_enum, 278
  - cuD3D9MapResources, 278
  - CUd3d9register\_flags, 278
  - CUd3d9register\_flags\_enum, 278
  - cuD3D9RegisterResource, 279
  - cuD3D9ResourceGetMappedArray, 280
  - cuD3D9ResourceGetMappedPitch, 281
  - cuD3D9ResourceGetMappedPointer, 282
  - cuD3D9ResourceGetMappedSize, 282
  - cuD3D9ResourceGetSurfaceDimensions, 283
  - cuD3D9ResourceSetMapFlags, 284
  - cuD3D9UnmapResources, 284
  - cuD3D9UnregisterResource, 285
- CUDA\_DEVICE
  - cuDeviceComputeCapability, 167
  - cuDeviceGet, 168
  - cuDeviceGetAttribute, 168
  - cuDeviceGetCount, 170
  - cuDeviceGetName, 170
  - cuDeviceGetProperties, 170
  - cuDeviceTotalMem, 171
- CUDA\_EVENT
  - cuEventCreate, 233
  - cuEventDestroy, 234
  - cuEventElapsedTime, 234
  - cuEventQuery, 235
  - cuEventRecord, 235
  - cuEventSynchronize, 236
- CUDA\_EXEC
  - cuFuncGetAttribute, 238
  - cuFuncSetBlockShape, 238
  - cuFuncSetCacheConfig, 239
  - cuFuncSetSharedSize, 240
  - cuLaunch, 240
  - cuLaunchGrid, 240
  - cuLaunchGridAsync, 241
  - cuParamSetf, 241
  - cuParamSeti, 242
  - cuParamSetSize, 242
  - cuParamSetv, 243
- CUDA\_EXEC\_DEPRECATED
  - cuParamSetTexRef, 244
- CUDA\_GL
  - cuGLCtxCreate, 262
  - cuGraphicsGLRegisterBuffer, 263
  - cuGraphicsGLRegisterImage, 263
  - cuWGLGetDevice, 264
- CUDA\_GL\_DEPRECATED
  - cuGLInit, 266
  - CUGLmap\_flags, 265
  - CUGLmap\_flags\_enum, 266
  - cuGLMapBufferObject, 266
  - cuGLMapBufferObjectAsync, 267
  - cuGLRegisterBufferObject, 267

- cuGLSetBufferObjectMapFlags, 268
- cuGLUnmapBufferObject, 268
- cuGLUnmapBufferObjectAsync, 269
- cuGLUnregisterBufferObject, 269
- CUDA\_GRAPHICS
  - cuGraphicsMapResources, 257
  - cuGraphicsResourceGetMappedPointer, 258
  - cuGraphicsResourceSetMapFlags, 258
  - cuGraphicsSubResourceGetMappedArray, 259
  - cuGraphicsUnmapResources, 260
  - cuGraphicsUnregisterResource, 260
- CUDA\_INITIALIZE
  - cuInit, 165
- CUDA\_MEM
  - cuArray3DCreate, 191
  - cuArray3DGetDescriptor, 192
  - cuArrayCreate, 193
  - cuArrayDestroy, 194
  - cuArrayGetDescriptor, 195
  - cuMemAlloc, 195
  - cuMemAllocHost, 196
  - cuMemAllocPitch, 197
  - cuMemcpy2D, 198
  - cuMemcpy2DAsync, 200
  - cuMemcpy2DUnaligned, 202
  - cuMemcpy3D, 204
  - cuMemcpy3DAsync, 206
  - cuMemcpyAtoA, 209
  - cuMemcpyAtoD, 209
  - cuMemcpyAtoH, 210
  - cuMemcpyAtoHAsync, 211
  - cuMemcpyDtoA, 211
  - cuMemcpyDtoD, 212
  - cuMemcpyDtoDAsync, 212
  - cuMemcpyDtoH, 213
  - cuMemcpyDtoHAsync, 214
  - cuMemcpyHtoA, 214
  - cuMemcpyHtoAAsync, 215
  - cuMemcpyHtoD, 216
  - cuMemcpyHtoDAsync, 216
  - cuMemFree, 217
  - cuMemFreeHost, 217
  - cuMemGetAddressRange, 218
  - cuMemGetInfo, 218
  - cuMemHostAlloc, 219
  - cuMemHostGetDevicePointer, 220
  - cuMemHostGetFlags, 221
  - cuMemsetD16, 221
  - cuMemsetD16Async, 222
  - cuMemsetD2D16, 222
  - cuMemsetD2D16Async, 223
  - cuMemsetD2D32, 224
  - cuMemsetD2D32Async, 224
  - cuMemsetD2D8, 225
  - cuMemsetD2D8Async, 226
  - cuMemsetD32, 226
  - cuMemsetD32Async, 227
  - cuMemsetD8, 228
  - cuMemsetD8Async, 228
- CUDA\_MEMCPY2D
  - CUDA\_TYPES, 154
- CUDA\_MEMCPY2D\_st, 314
  - dstArray, 314
  - dstDevice, 314
  - dstHost, 314
  - dstMemoryType, 314
  - dstPitch, 314
  - dstXInBytes, 314
  - dstY, 315
  - Height, 315
  - srcArray, 315
  - srcDevice, 315
  - srcHost, 315
  - srcMemoryType, 315
  - srcPitch, 315
  - srcXInBytes, 315
  - srcY, 315
  - WidthInBytes, 315
- CUDA\_MEMCPY3D
  - CUDA\_TYPES, 154
- CUDA\_MEMCPY3D\_st, 316
  - Depth, 316
  - dstArray, 316
  - dstDevice, 316
  - dstHeight, 316
  - dstHost, 317
  - dstLOD, 317
  - dstMemoryType, 317
  - dstPitch, 317
  - dstXInBytes, 317
  - dstY, 317
  - dstZ, 317
  - Height, 317
  - reserved0, 317
  - reserved1, 317
  - srcArray, 317
  - srcDevice, 317
  - srcHeight, 318
  - srcHost, 318
  - srcLOD, 318
  - srcMemoryType, 318
  - srcPitch, 318
  - srcXInBytes, 318
  - srcY, 318
  - srcZ, 318
  - WidthInBytes, 318
- CUDA\_MODULE
  - cuModuleGetFunction, 182

- cuModuleGetGlobal, 183
- cuModuleGetSurfRef, 183
- cuModuleGetTexRef, 184
- cuModuleLoad, 184
- cuModuleLoadData, 185
- cuModuleLoadDataEx, 185
- cuModuleLoadFatBinary, 187
- cuModuleUnload, 187
- CUDA\_STREAM
  - cuStreamCreate, 230
  - cuStreamDestroy, 230
  - cuStreamQuery, 231
  - cuStreamSynchronize, 231
  - cuStreamWaitEvent, 232
- CUDA\_SURFREF
  - cuSurfRefGetArray, 255
  - cuSurfRefSetArray, 255
- CUDA\_TEXREF
  - cuTexRefGetAddress, 246
  - cuTexRefGetAddressMode, 246
  - cuTexRefGetArray, 246
  - cuTexRefGetFilterMode, 247
  - cuTexRefGetFlags, 247
  - cuTexRefGetFormat, 248
  - cuTexRefSetAddress, 248
  - cuTexRefSetAddress2D, 249
  - cuTexRefSetAddressMode, 249
  - cuTexRefSetArray, 250
  - cuTexRefSetFilterMode, 250
  - cuTexRefSetFlags, 251
  - cuTexRefSetFormat, 251
- CUDA\_TEXREF\_DEPRECATED
  - cuTexRefCreate, 253
  - cuTexRefDestroy, 253
- CUDA\_TYPES
  - CU\_MEMHOSTALLOC\_DEVICEMAP, 153
  - CU\_MEMHOSTALLOC\_PORTABLE, 153
  - CU\_MEMHOSTALLOC\_WRITECOMBINED, 153
  - CU\_PARAM\_TR\_DEFAULT, 153
  - CU\_TRSA\_OVERRIDE\_FORMAT, 153
  - CU\_TRSF\_NORMALIZED\_COORDINATES, 153
  - CU\_TRSF\_READ\_AS\_INTEGER, 153
  - CU\_TRSF\_SRGB, 153
  - CUaddress\_mode, 154
  - CUaddress\_mode\_enum, 157
  - CUarray, 154
  - CUarray\_cubemap\_face, 154
  - CUarray\_cubemap\_face\_enum, 157
  - CUarray\_format, 154
  - CUarray\_format\_enum, 157
  - CUcomputemode, 154
  - CUcomputemode\_enum, 157
  - CUcontext, 154
  - CUctx\_flags, 154
  - CUctx\_flags\_enum, 158
  - CUDA\_ARRAY3D\_2DARRAY, 153
  - CUDA\_ARRAY3D\_DESCRIPTOR, 154
  - CUDA\_ARRAY3D\_SURFACE\_LDST, 153
  - CUDA\_ARRAY\_DESCRIPTOR, 154
  - CUDA\_MEMCPY2D, 154
  - CUDA\_MEMCPY3D, 154
  - CUDA\_VERSION, 153
  - cudaError\_enum, 158
  - CUdevice, 155
  - CUdevice\_attribute, 155
  - CUdevice\_attribute\_enum, 160
  - CUdeviceptr, 155
  - CUdevprop, 155
  - CUevent, 155
  - CUevent\_flags, 155
  - CUevent\_flags\_enum, 161
  - CUfilter\_mode, 155
  - CUfilter\_mode\_enum, 161
  - CUfunc\_cache, 155
  - CUfunc\_cache\_enum, 161
  - CUfunction, 155
  - CUfunction\_attribute, 155
  - CUfunction\_attribute\_enum, 161
  - CUgraphicsMapResourceFlags, 155
  - CUgraphicsMapResourceFlags\_enum, 162
  - CUgraphicsRegisterFlags, 155
  - CUgraphicsRegisterFlags\_enum, 162
  - CUgraphicsResource, 156
  - CUjit\_fallback, 156
  - CUjit\_fallback\_enum, 162
  - CUjit\_option, 156
  - CUjit\_option\_enum, 162
  - CUjit\_target, 156
  - CUjit\_target\_enum, 163
  - CUlimit, 156
  - CUlimit\_enum, 163
  - CUmemorytype, 156
  - CUmemorytype\_enum, 164
  - CUmodule, 156
  - CUresult, 156
  - CUstream, 156
  - CUsurfref, 156
  - CUtexref, 156
- CUDA\_VDPAU
  - cuGraphicsVDPAURegisterOutputSurface, 307
  - cuGraphicsVDPAURegisterVideoSurface, 308
  - cuVDPAUCtxCreate, 309
  - cuVDPAUGetDevice, 309
- CUDA\_VERSION
  - CUDA\_TYPES, 153
  - cuDriverGetVersion, 166
- cudaAddressModeBorder

- CUDART\_TYPES, [145](#)
- cudaAddressModeClamp
  - CUDART\_TYPES, [145](#)
- cudaAddressModeMirror
  - CUDART\_TYPES, [145](#)
- cudaAddressModeWrap
  - CUDART\_TYPES, [145](#)
- cudaArrayDefault
  - CUDART\_TYPES, [138](#)
- cudaArraySurfaceLoadStore
  - CUDART\_TYPES, [138](#)
- cudaBindSurfaceToArray
  - CUDART\_HIGHLEVEL, [100](#), [101](#)
  - CUDART\_SURFACE, [96](#)
- cudaBindTexture
  - CUDART\_HIGHLEVEL, [101](#), [102](#)
  - CUDART\_TEXTURE, [91](#)
- cudaBindTexture2D
  - CUDART\_HIGHLEVEL, [102](#), [103](#)
  - CUDART\_TEXTURE, [92](#)
- cudaBindTextureToArray
  - CUDART\_HIGHLEVEL, [104](#)
  - CUDART\_TEXTURE, [93](#)
- cudaBoundaryModeClamp
  - CUDART\_TYPES, [145](#)
- cudaBoundaryModeTrap
  - CUDART\_TYPES, [145](#)
- cudaBoundaryModeZero
  - CUDART\_TYPES, [145](#)
- cudaChannelFormatDesc, [319](#)
  - f, [319](#)
  - w, [319](#)
  - x, [319](#)
  - y, [319](#)
  - z, [319](#)
- cudaChannelFormatKind
  - CUDART\_TYPES, [140](#)
- cudaChannelFormatKindFloat
  - CUDART\_TYPES, [140](#)
- cudaChannelFormatKindNone
  - CUDART\_TYPES, [140](#)
- cudaChannelFormatKindSigned
  - CUDART\_TYPES, [140](#)
- cudaChannelFormatKindUnsigned
  - CUDART\_TYPES, [140](#)
- cudaChooseDevice
  - CUDART\_DEVICE, [17](#)
- cudaComputeMode
  - CUDART\_TYPES, [140](#)
- cudaComputeModeDefault
  - CUDART\_TYPES, [140](#)
- cudaComputeModeExclusive
  - CUDART\_TYPES, [140](#)
- cudaComputeModeProhibited
  - CUDART\_TYPES, [140](#)
- cudaConfigureCall
  - CUDART\_EXECUTION, [30](#)
- cudaCreateChannelDesc
  - CUDART\_HIGHLEVEL, [105](#)
  - CUDART\_TEXTURE, [93](#)
- cudaD3D10DeviceList
  - CUDART\_D3D10, [76](#)
- cudaD3D10DeviceListAll
  - CUDART\_D3D10, [76](#)
- cudaD3D10DeviceListCurrentFrame
  - CUDART\_D3D10, [76](#)
- cudaD3D10DeviceListNextFrame
  - CUDART\_D3D10, [76](#)
- cudaD3D10GetDevice
  - CUDART\_D3D10, [76](#)
- cudaD3D10GetDevices
  - CUDART\_D3D10, [77](#)
- cudaD3D10GetDirect3DDevice
  - CUDART\_D3D10, [77](#)
- cudaD3D10MapFlags
  - CUDART\_D3D10, [76](#)
- cudaD3D10MapFlagsNone
  - CUDART\_D3D10, [76](#)
- cudaD3D10MapFlagsReadOnly
  - CUDART\_D3D10, [76](#)
- cudaD3D10MapFlagsWriteDiscard
  - CUDART\_D3D10, [76](#)
- cudaD3D10MapResources
  - CUDART\_D3D10\_DEPRECATED, [122](#)
- cudaD3D10RegisterFlags
  - CUDART\_D3D10, [76](#)
- cudaD3D10RegisterFlagsArray
  - CUDART\_D3D10, [76](#)
- cudaD3D10RegisterFlagsNone
  - CUDART\_D3D10, [76](#)
- cudaD3D10RegisterResource
  - CUDART\_D3D10\_DEPRECATED, [123](#)
- cudaD3D10ResourceGetMappedArray
  - CUDART\_D3D10\_DEPRECATED, [124](#)
- cudaD3D10ResourceGetMappedPitch
  - CUDART\_D3D10\_DEPRECATED, [125](#)
- cudaD3D10ResourceGetMappedPointer
  - CUDART\_D3D10\_DEPRECATED, [126](#)
- cudaD3D10ResourceGetMappedSize
  - CUDART\_D3D10\_DEPRECATED, [126](#)
- cudaD3D10ResourceGetSurfaceDimensions
  - CUDART\_D3D10\_DEPRECATED, [127](#)
- cudaD3D10ResourceSetMapFlags
  - CUDART\_D3D10\_DEPRECATED, [127](#)
- cudaD3D10SetDirect3DDevice
  - CUDART\_D3D10, [77](#)
- cudaD3D10UnmapResources
  - CUDART\_D3D10\_DEPRECATED, [128](#)

- cudaD3D10UnregisterResource
  - CUDART\_D3D10\_DEPRECATED, 129
- cudaD3D11DeviceList
  - CUDART\_D3D11, 80
- cudaD3D11DeviceListAll
  - CUDART\_D3D11, 80
- cudaD3D11DeviceListCurrentFrame
  - CUDART\_D3D11, 80
- cudaD3D11DeviceListNextFrame
  - CUDART\_D3D11, 80
- cudaD3D11GetDevice
  - CUDART\_D3D11, 81
- cudaD3D11GetDevices
  - CUDART\_D3D11, 81
- cudaD3D11GetDirect3DDevice
  - CUDART\_D3D11, 81
- cudaD3D11SetDirect3DDevice
  - CUDART\_D3D11, 82
- cudaD3D9DeviceList
  - CUDART\_D3D9, 71
- cudaD3D9DeviceListAll
  - CUDART\_D3D9, 71
- cudaD3D9DeviceListCurrentFrame
  - CUDART\_D3D9, 71
- cudaD3D9DeviceListNextFrame
  - CUDART\_D3D9, 71
- cudaD3D9GetDevice
  - CUDART\_D3D9, 71
- cudaD3D9GetDevices
  - CUDART\_D3D9, 72
- cudaD3D9GetDirect3DDevice
  - CUDART\_D3D9, 72
- cudaD3D9MapFlags
  - CUDART\_D3D9, 71
- cudaD3D9MapFlagsNone
  - CUDART\_D3D9, 71
- cudaD3D9MapFlagsReadOnly
  - CUDART\_D3D9, 71
- cudaD3D9MapFlagsWriteDiscard
  - CUDART\_D3D9, 71
- cudaD3D9MapResources
  - CUDART\_D3D9\_DEPRECATED, 114
- cudaD3D9RegisterFlags
  - CUDART\_D3D9, 71
- cudaD3D9RegisterFlagsArray
  - CUDART\_D3D9, 71
- cudaD3D9RegisterFlagsNone
  - CUDART\_D3D9, 71
- cudaD3D9RegisterResource
  - CUDART\_D3D9\_DEPRECATED, 115
- cudaD3D9ResourceGetMappedArray
  - CUDART\_D3D9\_DEPRECATED, 116
- cudaD3D9ResourceGetMappedPitch
  - CUDART\_D3D9\_DEPRECATED, 117
- cudaD3D9ResourceGetMappedPointer
  - CUDART\_D3D9\_DEPRECATED, 118
- cudaD3D9ResourceGetMappedSize
  - CUDART\_D3D9\_DEPRECATED, 118
- cudaD3D9ResourceGetSurfaceDimensions
  - CUDART\_D3D9\_DEPRECATED, 119
- cudaD3D9ResourceSetMapFlags
  - CUDART\_D3D9\_DEPRECATED, 120
- cudaD3D9SetDirect3DDevice
  - CUDART\_D3D9, 72
- cudaD3D9UnmapResources
  - CUDART\_D3D9\_DEPRECATED, 120
- cudaD3D9UnregisterResource
  - CUDART\_D3D9\_DEPRECATED, 121
- cudaDeviceBlockingSync
  - CUDART\_TYPES, 138
- cudaDeviceLmemResizeToMax
  - CUDART\_TYPES, 138
- cudaDeviceMapHost
  - CUDART\_TYPES, 138
- cudaDeviceMask
  - CUDART\_TYPES, 138
- cudaDeviceProp, 320
  - canMapHostMemory, 320
  - clockRate, 320
  - computeMode, 320
  - concurrentKernels, 321
  - deviceOverlap, 321
  - ECCEnabled, 321
  - integrated, 321
  - kernelExecTimeoutEnabled, 321
  - major, 321
  - maxGridSize, 321
  - maxTexture1D, 321
  - maxTexture2D, 321
  - maxTexture2DArray, 321
  - maxTexture3D, 321
  - maxThreadsDim, 321
  - maxThreadsPerBlock, 322
  - memPitch, 322
  - minor, 322
  - multiProcessorCount, 322
  - name, 322
  - pciBusID, 322
  - pciDeviceID, 322
  - regsPerBlock, 322
  - sharedMemPerBlock, 322
  - surfaceAlignment, 322
  - tccDriver, 322
  - textureAlignment, 322
  - totalConstMem, 323
  - totalGlobalMem, 323
  - warpSize, 323
- cudaDevicePropDontCare

- CUDART\_TYPES, 138
- cudaDeviceScheduleAuto
  - CUDART\_TYPES, 139
- cudaDeviceScheduleSpin
  - CUDART\_TYPES, 139
- cudaDeviceScheduleYield
  - CUDART\_TYPES, 139
- cudaDriverGetVersion
  - CUDART\_\_VERSION, 98
- cudaError
  - CUDART\_TYPES, 140
- cudaError\_enum
  - CUDA\_TYPES, 158
- cudaError\_t
  - CUDART\_TYPES, 140
- cudaErrorAddressOfConstant
  - CUDART\_TYPES, 142
- cudaErrorApiFailureBase
  - CUDART\_TYPES, 144
- cudaErrorCudartUnloading
  - CUDART\_TYPES, 142
- cudaErrorDevicesUnavailable
  - CUDART\_TYPES, 143
- cudaErrorDuplicateSurfaceName
  - CUDART\_TYPES, 143
- cudaErrorDuplicateTextureName
  - CUDART\_TYPES, 143
- cudaErrorDuplicateVariableName
  - CUDART\_TYPES, 143
- cudaErrorECCUncorrectable
  - CUDART\_TYPES, 143
- cudaErrorIncompatibleDriverContext
  - CUDART\_TYPES, 143
- cudaErrorInitializationError
  - CUDART\_TYPES, 141
- cudaErrorInsufficientDriver
  - CUDART\_TYPES, 143
- cudaErrorInvalidChannelDescriptor
  - CUDART\_TYPES, 142
- cudaErrorInvalidConfiguration
  - CUDART\_TYPES, 141
- cudaErrorInvalidDevice
  - CUDART\_TYPES, 141
- cudaErrorInvalidDeviceFunction
  - CUDART\_TYPES, 141
- cudaErrorInvalidDevicePointer
  - CUDART\_TYPES, 141
- cudaErrorInvalidFilterSetting
  - CUDART\_TYPES, 142
- cudaErrorInvalidHostPointer
  - CUDART\_TYPES, 141
- cudaErrorInvalidKernelImage
  - CUDART\_TYPES, 143
- cudaErrorInvalidMemcpyDirection
  - CUDART\_TYPES, 142
- cudaErrorInvalidNormSetting
  - CUDART\_TYPES, 142
- cudaErrorInvalidPitchValue
  - CUDART\_TYPES, 141
- cudaErrorInvalidResourceHandle
  - CUDART\_TYPES, 143
- cudaErrorInvalidSurface
  - CUDART\_TYPES, 143
- cudaErrorInvalidSymbol
  - CUDART\_TYPES, 141
- cudaErrorInvalidTexture
  - CUDART\_TYPES, 142
- cudaErrorInvalidTextureBinding
  - CUDART\_TYPES, 142
- cudaErrorInvalidValue
  - CUDART\_TYPES, 141
- cudaErrorLaunchFailure
  - CUDART\_TYPES, 141
- cudaErrorLaunchOutOfResources
  - CUDART\_TYPES, 141
- cudaErrorLaunchTimeout
  - CUDART\_TYPES, 141
- cudaErrorMapBufferObjectFailed
  - CUDART\_TYPES, 141
- cudaErrorMemoryAllocation
  - CUDART\_TYPES, 141
- cudaErrorMemoryValueTooLarge
  - CUDART\_TYPES, 143
- cudaErrorMissingConfiguration
  - CUDART\_TYPES, 141
- cudaErrorMixedDeviceExecution
  - CUDART\_TYPES, 142
- cudaErrorNoDevice
  - CUDART\_TYPES, 143
- cudaErrorNoKernelImageForDevice
  - CUDART\_TYPES, 143
- cudaErrorNotReady
  - CUDART\_TYPES, 143
- cudaErrorNotYetImplemented
  - CUDART\_TYPES, 142
- cudaErrorPriorLaunchFailure
  - CUDART\_TYPES, 141
- cudaErrorSetOnActiveProcess
  - CUDART\_TYPES, 143
- cudaErrorSharedObjectInitFailed
  - CUDART\_TYPES, 143
- cudaErrorSharedObjectSymbolNotFound
  - CUDART\_TYPES, 143
- cudaErrorStartupFailure
  - CUDART\_TYPES, 143
- cudaErrorSynchronizationError
  - CUDART\_TYPES, 142
- cudaErrorTextureFetchFailed

- CUDART\_TYPES, 142
- cudaErrorTextureNotBound
  - CUDART\_TYPES, 142
- cudaErrorUnknown
  - CUDART\_TYPES, 142
- cudaErrorUnmapBufferObjectFailed
  - CUDART\_TYPES, 141
- cudaErrorUnsupportedLimit
  - CUDART\_TYPES, 143
- cudaEvent\_t
  - CUDART\_TYPES, 140
- cudaEventBlockingSync
  - CUDART\_TYPES, 139
- cudaEventCreate
  - CUDART\_EVENT, 26
  - CUDART\_HIGHLEVEL, 105
- cudaEventCreateWithFlags
  - CUDART\_EVENT, 26
- cudaEventDefault
  - CUDART\_TYPES, 139
- cudaEventDestroy
  - CUDART\_EVENT, 27
- cudaEventDisableTiming
  - CUDART\_TYPES, 139
- cudaEventElapsedTime
  - CUDART\_EVENT, 27
- cudaEventQuery
  - CUDART\_EVENT, 28
- cudaEventRecord
  - CUDART\_EVENT, 28
- cudaEventSynchronize
  - CUDART\_EVENT, 29
- cudaExtent, 324
  - depth, 324
  - height, 324
  - width, 324
- cudaFilterModeLinear
  - CUDART\_TYPES, 146
- cudaFilterModePoint
  - CUDART\_TYPES, 146
- cudaFormatModeAuto
  - CUDART\_TYPES, 145
- cudaFormatModeForced
  - CUDART\_TYPES, 145
- cudaFree
  - CUDART\_MEMORY, 37
- cudaFreeArray
  - CUDART\_MEMORY, 37
- cudaFreeHost
  - CUDART\_MEMORY, 38
- cudaFuncAttributes, 325
  - binaryVersion, 325
  - constSizeBytes, 325
  - localSizeBytes, 325
  - maxThreadsPerBlock, 325
  - numRegs, 325
  - ptxVersion, 325
  - sharedSizeBytes, 325
- cudaFuncCache
  - CUDART\_TYPES, 144
- cudaFuncCachePreferL1
  - CUDART\_TYPES, 144
- cudaFuncCachePreferNone
  - CUDART\_TYPES, 144
- cudaFuncCachePreferShared
  - CUDART\_TYPES, 144
- cudaFuncGetAttributes
  - CUDART\_EXECUTION, 31
  - CUDART\_HIGHLEVEL, 106
- cudaFuncSetCacheConfig
  - CUDART\_EXECUTION, 31
  - CUDART\_HIGHLEVEL, 107
- cudaGetChannelDesc
  - CUDART\_TEXTURE, 94
- cudaGetDevice
  - CUDART\_DEVICE, 17
- cudaGetDeviceCount
  - CUDART\_DEVICE, 18
- cudaGetDeviceProperties
  - CUDART\_DEVICE, 18
- cudaGetErrorString
  - CUDART\_ERROR, 15
- cudaGetLastError
  - CUDART\_ERROR, 15
- cudaGetSurfaceReference
  - CUDART\_SURFACE, 96
- cudaGetSymbolAddress
  - CUDART\_HIGHLEVEL, 107
  - CUDART\_MEMORY, 38
- cudaGetSymbolSize
  - CUDART\_HIGHLEVEL, 108
  - CUDART\_MEMORY, 38
- cudaGetTextureAlignmentOffset
  - CUDART\_HIGHLEVEL, 108
  - CUDART\_TEXTURE, 94
- cudaGetTextureReference
  - CUDART\_TEXTURE, 95
- cudaGLMapBufferObject
  - CUDART\_OPENGL\_DEPRECATED, 130
- cudaGLMapBufferObjectAsync
  - CUDART\_OPENGL\_DEPRECATED, 131
- cudaGLMapFlags
  - CUDART\_OPENGL, 66
- cudaGLMapFlagsNone
  - CUDART\_OPENGL, 66
- cudaGLMapFlagsReadOnly
  - CUDART\_OPENGL, 66
- cudaGLMapFlagsWriteDiscard

- CUDART\_OPENGL, 66
- cudaGLRegisterBufferObject
  - CUDART\_OPENGL\_DEPRECATED, 131
- cudaGLSetBufferObjectMapFlags
  - CUDART\_OPENGL\_DEPRECATED, 132
- cudaGLSetGLDevice
  - CUDART\_OPENGL, 67
- cudaGLUnmapBufferObject
  - CUDART\_OPENGL\_DEPRECATED, 132
- cudaGLUnmapBufferObjectAsync
  - CUDART\_OPENGL\_DEPRECATED, 133
- cudaGLUnregisterBufferObject
  - CUDART\_OPENGL\_DEPRECATED, 133
- cudaGraphicsCubeFace
  - CUDART\_TYPES, 144
- cudaGraphicsCubeFaceNegativeX
  - CUDART\_TYPES, 144
- cudaGraphicsCubeFaceNegativeY
  - CUDART\_TYPES, 144
- cudaGraphicsCubeFaceNegativeZ
  - CUDART\_TYPES, 144
- cudaGraphicsCubeFacePositiveX
  - CUDART\_TYPES, 144
- cudaGraphicsCubeFacePositiveY
  - CUDART\_TYPES, 144
- cudaGraphicsCubeFacePositiveZ
  - CUDART\_TYPES, 144
- cudaGraphicsD3D10RegisterResource
  - CUDART\_D3D10, 78
- cudaGraphicsD3D11RegisterResource
  - CUDART\_D3D11, 82
- cudaGraphicsD3D9RegisterResource
  - CUDART\_D3D9, 73
- cudaGraphicsGLRegisterBuffer
  - CUDART\_OPENGL, 67
- cudaGraphicsGLRegisterImage
  - CUDART\_OPENGL, 67
- cudaGraphicsMapFlags
  - CUDART\_TYPES, 144
- cudaGraphicsMapFlagsNone
  - CUDART\_TYPES, 144
- cudaGraphicsMapFlagsReadOnly
  - CUDART\_TYPES, 144
- cudaGraphicsMapFlagsWriteDiscard
  - CUDART\_TYPES, 144
- cudaGraphicsMapResources
  - CUDART\_INTEROP, 87
- cudaGraphicsRegisterFlags
  - CUDART\_TYPES, 144
- cudaGraphicsRegisterFlagsNone
  - CUDART\_TYPES, 144
- cudaGraphicsResource\_t
  - CUDART\_TYPES, 140
- cudaGraphicsResourceGetMappedPointer
  - CUDART\_INTEROP, 88
- cudaGraphicsResourceSetMapFlags
  - CUDART\_INTEROP, 88
- cudaGraphicsSubResourceGetMappedArray
  - CUDART\_INTEROP, 89
- cudaGraphicsUnmapResources
  - CUDART\_INTEROP, 89
- cudaGraphicsUnregisterResource
  - CUDART\_INTEROP, 90
- cudaGraphicsVDPAAURegisterOutputSurface
  - CUDART\_VDPAU, 84
- cudaGraphicsVDPAAURegisterVideoSurface
  - CUDART\_VDPAU, 85
- cudaHostAlloc
  - CUDART\_MEMORY, 39
- cudaHostAllocDefault
  - CUDART\_TYPES, 139
- cudaHostAllocMapped
  - CUDART\_TYPES, 139
- cudaHostAllocPortable
  - CUDART\_TYPES, 139
- cudaHostAllocWriteCombined
  - CUDART\_TYPES, 139
- cudaHostGetDevicePointer
  - CUDART\_MEMORY, 40
- cudaHostGetFlags
  - CUDART\_MEMORY, 40
- cudaLaunch
  - CUDART\_EXECUTION, 32
  - CUDART\_HIGHLEVEL, 109
- cudaLimit
  - CUDART\_TYPES, 144
- cudaLimitMallocHeapSize
  - CUDART\_TYPES, 145
- cudaLimitPrintfFifoSize
  - CUDART\_TYPES, 145
- cudaLimitStackSize
  - CUDART\_TYPES, 145
- cudaMalloc
  - CUDART\_MEMORY, 41
- cudaMalloc3D
  - CUDART\_MEMORY, 41
- cudaMalloc3DArray
  - CUDART\_MEMORY, 42
- cudaMallocArray
  - CUDART\_MEMORY, 43
- cudaMallocHost
  - CUDART\_HIGHLEVEL, 109
  - CUDART\_MEMORY, 43
- cudaMallocPitch
  - CUDART\_MEMORY, 44
- cudaMemcpy
  - CUDART\_MEMORY, 45
- cudaMemcpy2D

- CUDART\_MEMORY, 45
- cudaMemcpy2DArrayToArray
  - CUDART\_MEMORY, 46
- cudaMemcpy2DAsync
  - CUDART\_MEMORY, 47
- cudaMemcpy2DFromArray
  - CUDART\_MEMORY, 47
- cudaMemcpy2DFromArrayAsync
  - CUDART\_MEMORY, 48
- cudaMemcpy2DToArray
  - CUDART\_MEMORY, 49
- cudaMemcpy2DToArrayAsync
  - CUDART\_MEMORY, 50
- cudaMemcpy3D
  - CUDART\_MEMORY, 51
- cudaMemcpy3DAsync
  - CUDART\_MEMORY, 52
- cudaMemcpy3DParms, 327
  - dstArray, 327
  - dstPos, 327
  - dstPtr, 327
  - extent, 327
  - kind, 327
  - srcArray, 327
  - srcPos, 327
  - srcPtr, 327
- cudaMemcpyArrayToArray
  - CUDART\_MEMORY, 53
- cudaMemcpyAsync
  - CUDART\_MEMORY, 54
- cudaMemcpyDeviceToDevice
  - CUDART\_TYPES, 145
- cudaMemcpyDeviceToHost
  - CUDART\_TYPES, 145
- cudaMemcpyFromArray
  - CUDART\_MEMORY, 55
- cudaMemcpyFromArrayAsync
  - CUDART\_MEMORY, 55
- cudaMemcpyFromSymbol
  - CUDART\_MEMORY, 56
- cudaMemcpyFromSymbolAsync
  - CUDART\_MEMORY, 57
- cudaMemcpyHostToDevice
  - CUDART\_TYPES, 145
- cudaMemcpyHostToHost
  - CUDART\_TYPES, 145
- cudaMemcpyKind
  - CUDART\_TYPES, 145
- cudaMemcpyToArray
  - CUDART\_MEMORY, 58
- cudaMemcpyToArrayAsync
  - CUDART\_MEMORY, 58
- cudaMemcpyToSymbol
  - CUDART\_MEMORY, 59
- cudaMemcpyToSymbolAsync
  - CUDART\_MEMORY, 59
- cudaMemGetInfo
  - CUDART\_MEMORY, 60
- cudaMemset
  - CUDART\_MEMORY, 60
- cudaMemset2D
  - CUDART\_MEMORY, 61
- cudaMemset2DAsync
  - CUDART\_MEMORY, 61
- cudaMemset3D
  - CUDART\_MEMORY, 62
- cudaMemset3DAsync
  - CUDART\_MEMORY, 63
- cudaMemsetAsync
  - CUDART\_MEMORY, 63
- cudaPeekAtLastError
  - CUDART\_ERROR, 16
- cudaPitchedPtr, 329
  - pitch, 329
  - ptr, 329
  - xsize, 329
  - ysize, 329
- cudaPos, 330
  - x, 330
  - y, 330
  - z, 330
- cudaReadModeElementType
  - CUDART\_TYPES, 146
- cudaReadModeNormalizedFloat
  - CUDART\_TYPES, 146
- CUDART
  - CUDART\_VERSION, 10
- CUDART\_D3D10
  - cudaD3D10DeviceListAll, 76
  - cudaD3D10DeviceListCurrentFrame, 76
  - cudaD3D10DeviceListNextFrame, 76
  - cudaD3D10MapFlagsNone, 76
  - cudaD3D10MapFlagsReadOnly, 76
  - cudaD3D10MapFlagsWriteDiscard, 76
  - cudaD3D10RegisterFlagsArray, 76
  - cudaD3D10RegisterFlagsNone, 76
- CUDART\_D3D11
  - cudaD3D11DeviceListAll, 80
  - cudaD3D11DeviceListCurrentFrame, 80
  - cudaD3D11DeviceListNextFrame, 80
- CUDART\_D3D9
  - cudaD3D9DeviceListAll, 71
  - cudaD3D9DeviceListCurrentFrame, 71
  - cudaD3D9DeviceListNextFrame, 71
  - cudaD3D9MapFlagsNone, 71
  - cudaD3D9MapFlagsReadOnly, 71
  - cudaD3D9MapFlagsWriteDiscard, 71
  - cudaD3D9RegisterFlagsArray, 71

- cudaD3D9RegisterFlagsNone, 71
- CUDART\_OPENGL
  - cudaGLMapFlagsNone, 66
  - cudaGLMapFlagsReadOnly, 66
  - cudaGLMapFlagsWriteDiscard, 66
- CUDART\_TYPES
  - cudaAddressModeBorder, 145
  - cudaAddressModeClamp, 145
  - cudaAddressModeMirror, 145
  - cudaAddressModeWrap, 145
  - cudaBoundaryModeClamp, 145
  - cudaBoundaryModeTrap, 145
  - cudaBoundaryModeZero, 145
  - cudaChannelFormatKindFloat, 140
  - cudaChannelFormatKindNone, 140
  - cudaChannelFormatKindSigned, 140
  - cudaChannelFormatKindUnsigned, 140
  - cudaComputeModeDefault, 140
  - cudaComputeModeExclusive, 140
  - cudaComputeModeProhibited, 140
  - cudaErrorAddressOfConstant, 142
  - cudaErrorApiFailureBase, 144
  - cudaErrorCudartUnloading, 142
  - cudaErrorDevicesUnavailable, 143
  - cudaErrorDuplicateSurfaceName, 143
  - cudaErrorDuplicateTextureName, 143
  - cudaErrorDuplicateVariableName, 143
  - cudaErrorECCUncorrectable, 143
  - cudaErrorIncompatibleDriverContext, 143
  - cudaErrorInitializationError, 141
  - cudaErrorInsufficientDriver, 143
  - cudaErrorInvalidChannelDescriptor, 142
  - cudaErrorInvalidConfiguration, 141
  - cudaErrorInvalidDevice, 141
  - cudaErrorInvalidDeviceFunction, 141
  - cudaErrorInvalidDevicePointer, 141
  - cudaErrorInvalidFilterSetting, 142
  - cudaErrorInvalidHostPointer, 141
  - cudaErrorInvalidKernelImage, 143
  - cudaErrorInvalidMemcpyDirection, 142
  - cudaErrorInvalidNormSetting, 142
  - cudaErrorInvalidPitchValue, 141
  - cudaErrorInvalidResourceHandle, 143
  - cudaErrorInvalidSurface, 143
  - cudaErrorInvalidSymbol, 141
  - cudaErrorInvalidTexture, 142
  - cudaErrorInvalidTextureBinding, 142
  - cudaErrorInvalidValue, 141
  - cudaErrorLaunchFailure, 141
  - cudaErrorLaunchOutOfResources, 141
  - cudaErrorLaunchTimeout, 141
  - cudaErrorMapBufferObjectFailed, 141
  - cudaErrorMemoryAllocation, 141
  - cudaErrorMemoryValueTooLarge, 143
  - cudaErrorMissingConfiguration, 141
  - cudaErrorMixedDeviceExecution, 142
  - cudaErrorNoDevice, 143
  - cudaErrorNoKernelImageForDevice, 143
  - cudaErrorNotReady, 143
  - cudaErrorNotYetImplemented, 142
  - cudaErrorPriorLaunchFailure, 141
  - cudaErrorSetOnActiveProcess, 143
  - cudaErrorSharedObjectInitFailed, 143
  - cudaErrorSharedObjectSymbolNotFound, 143
  - cudaErrorStartupFailure, 143
  - cudaErrorSynchronizationError, 142
  - cudaErrorTextureFetchFailed, 142
  - cudaErrorTextureNotBound, 142
  - cudaErrorUnknown, 142
  - cudaErrorUnmapBufferObjectFailed, 141
  - cudaErrorUnsupportedLimit, 143
  - cudaFilterModeLinear, 146
  - cudaFilterModePoint, 146
  - cudaFormatModeAuto, 145
  - cudaFormatModeForced, 145
  - cudaFuncCachePreferL1, 144
  - cudaFuncCachePreferNone, 144
  - cudaFuncCachePreferShared, 144
  - cudaGraphicsCubeFaceNegativeX, 144
  - cudaGraphicsCubeFaceNegativeY, 144
  - cudaGraphicsCubeFaceNegativeZ, 144
  - cudaGraphicsCubeFacePositiveX, 144
  - cudaGraphicsCubeFacePositiveY, 144
  - cudaGraphicsCubeFacePositiveZ, 144
  - cudaGraphicsMapFlagsNone, 144
  - cudaGraphicsMapFlagsReadOnly, 144
  - cudaGraphicsMapFlagsWriteDiscard, 144
  - cudaGraphicsRegisterFlagsNone, 144
  - cudaLimitMallocHeapSize, 145
  - cudaLimitPrintfFifoSize, 145
  - cudaLimitStackSize, 145
  - cudaMemcpyDeviceToDevice, 145
  - cudaMemcpyDeviceToHost, 145
  - cudaMemcpyHostToDevice, 145
  - cudaMemcpyHostToHost, 145
  - cudaReadModeElementType, 146
  - cudaReadModeNormalizedFloat, 146
  - cudaSuccess, 141
- CUDART\_\_VERSION
  - cudaDriverGetVersion, 98
  - cudaRuntimeGetVersion, 98
- CUDART\_D3D10
  - cudaD3D10DeviceList, 76
  - cudaD3D10GetDevice, 76
  - cudaD3D10GetDevices, 77
  - cudaD3D10GetDirect3DDevice, 77
  - cudaD3D10MapFlags, 76
  - cudaD3D10RegisterFlags, 76

- cudaD3D10SetDirect3DDevice, 77
  - cudaGraphicsD3D10RegisterResource, 78
- CUDART\_D3D10\_DEPRECATED
  - cudaD3D10MapResources, 122
  - cudaD3D10RegisterResource, 123
  - cudaD3D10ResourceGetMappedArray, 124
  - cudaD3D10ResourceGetMappedPitch, 125
  - cudaD3D10ResourceGetMappedPointer, 126
  - cudaD3D10ResourceGetMappedSize, 126
  - cudaD3D10ResourceGetSurfaceDimensions, 127
  - cudaD3D10ResourceSetMapFlags, 127
  - cudaD3D10UnmapResources, 128
  - cudaD3D10UnregisterResource, 129
- CUDART\_D3D11
  - cudaD3D11DeviceList, 80
  - cudaD3D11GetDevice, 81
  - cudaD3D11GetDevices, 81
  - cudaD3D11GetDirect3DDevice, 81
  - cudaD3D11SetDirect3DDevice, 82
  - cudaGraphicsD3D11RegisterResource, 82
- CUDART\_D3D9
  - cudaD3D9DeviceList, 71
  - cudaD3D9GetDevice, 71
  - cudaD3D9GetDevices, 72
  - cudaD3D9GetDirect3DDevice, 72
  - cudaD3D9MapFlags, 71
  - cudaD3D9RegisterFlags, 71
  - cudaD3D9SetDirect3DDevice, 72
  - cudaGraphicsD3D9RegisterResource, 73
- CUDART\_D3D9\_DEPRECATED
  - cudaD3D9MapResources, 114
  - cudaD3D9RegisterResource, 115
  - cudaD3D9ResourceGetMappedArray, 116
  - cudaD3D9ResourceGetMappedPitch, 117
  - cudaD3D9ResourceGetMappedPointer, 118
  - cudaD3D9ResourceGetMappedSize, 118
  - cudaD3D9ResourceGetSurfaceDimensions, 119
  - cudaD3D9ResourceSetMapFlags, 120
  - cudaD3D9UnmapResources, 120
  - cudaD3D9UnregisterResource, 121
- CUDART\_DEVICE
  - cudaChooseDevice, 17
  - cudaGetDevice, 17
  - cudaGetDeviceCount, 18
  - cudaGetDeviceProperties, 18
  - cudaSetDevice, 20
  - cudaSetDeviceFlags, 20
  - cudaSetValidDevices, 21
- CUDART\_ERROR
  - cudaGetErrorString, 15
  - cudaGetLastError, 15
  - cudaPeekAtLastError, 16
- CUDART\_EVENT
  - cudaEventCreate, 26
  - cudaEventCreateWithFlags, 26
  - cudaEventDestroy, 27
  - cudaEventElapsedTime, 27
  - cudaEventQuery, 28
  - cudaEventRecord, 28
  - cudaEventSynchronize, 29
- CUDART\_EXECUTION
  - cudaConfigureCall, 30
  - cudaFuncGetAttributes, 31
  - cudaFuncSetCacheConfig, 31
  - cudaLaunch, 32
  - cudaSetDoubleForDevice, 32
  - cudaSetDoubleForHost, 33
  - cudaSetupArgument, 33
- CUDART\_HIGHLEVEL
  - cudaBindSurfaceToArray, 100, 101
  - cudaBindTexture, 101, 102
  - cudaBindTexture2D, 102, 103
  - cudaBindTextureToArray, 104
  - cudaCreateChannelDesc, 105
  - cudaEventCreate, 105
  - cudaFuncGetAttributes, 106
  - cudaFuncSetCacheConfig, 107
  - cudaGetSymbolAddress, 107
  - cudaGetSymbolSize, 108
  - cudaGetTextureAlignmentOffset, 108
  - cudaLaunch, 109
  - cudaMallocHost, 109
  - cudaSetupArgument, 110
  - cudaUnbindTexture, 111
- CUDART\_INTEROP
  - cudaGraphicsMapResources, 87
  - cudaGraphicsResourceGetMappedPointer, 88
  - cudaGraphicsResourceSetMapFlags, 88
  - cudaGraphicsSubResourceGetMappedArray, 89
  - cudaGraphicsUnmapResources, 89
  - cudaGraphicsUnregisterResource, 90
- CUDART\_MEMORY
  - cudaFree, 37
  - cudaFreeArray, 37
  - cudaFreeHost, 38
  - cudaGetSymbolAddress, 38
  - cudaGetSymbolSize, 38
  - cudaHostAlloc, 39
  - cudaHostGetDevicePointer, 40
  - cudaHostGetFlags, 40
  - cudaMalloc, 41
  - cudaMalloc3D, 41
  - cudaMalloc3DArray, 42
  - cudaMallocArray, 43
  - cudaMallocHost, 43
  - cudaMallocPitch, 44
  - cudaMemcpy, 45
  - cudaMemcpy2D, 45

- cudaMemcpy2DArrayToArray, 46
- cudaMemcpy2DAsync, 47
- cudaMemcpy2DFromArray, 47
- cudaMemcpy2DFromArrayAsync, 48
- cudaMemcpy2DToArray, 49
- cudaMemcpy2DToArrayAsync, 50
- cudaMemcpy3D, 51
- cudaMemcpy3DAsync, 52
- cudaMemcpyArrayToArray, 53
- cudaMemcpyAsync, 54
- cudaMemcpyFromArray, 55
- cudaMemcpyFromArrayAsync, 55
- cudaMemcpyFromSymbol, 56
- cudaMemcpyFromSymbolAsync, 57
- cudaMemcpyToArray, 58
- cudaMemcpyToArrayAsync, 58
- cudaMemcpyToSymbol, 59
- cudaMemcpyToSymbolAsync, 59
- cudaMemGetInfo, 60
- cudaMemset, 60
- cudaMemset2D, 61
- cudaMemset2DAsync, 61
- cudaMemset3D, 62
- cudaMemset3DAsync, 63
- cudaMemsetAsync, 63
- make\_cudaExtent, 64
- make\_cudaPitchedPtr, 64
- make\_cudaPos, 65
- CUDART\_OPENGL
  - cudaGLMapFlags, 66
  - cudaGLSetGLDevice, 67
  - cudaGraphicsGLRegisterBuffer, 67
  - cudaGraphicsGLRegisterImage, 67
  - cudaWGLGetDevice, 68
- CUDART\_OPENGL\_DEPRECATED
  - cudaGLMapBufferObject, 130
  - cudaGLMapBufferObjectAsync, 131
  - cudaGLRegisterBufferObject, 131
  - cudaGLSetBufferObjectMapFlags, 132
  - cudaGLUnmapBufferObject, 132
  - cudaGLUnmapBufferObjectAsync, 133
  - cudaGLUnregisterBufferObject, 133
- CUDART\_STREAM
  - cudaStreamCreate, 23
  - cudaStreamDestroy, 23
  - cudaStreamQuery, 24
  - cudaStreamSynchronize, 24
  - cudaStreamWaitEvent, 24
- CUDART\_SURFACE
  - cudaBindSurfaceToArray, 96
  - cudaGetSurfaceReference, 96
- CUDART\_TEXTURE
  - cudaBindTexture, 91
  - cudaBindTexture2D, 92
  - cudaBindTextureToArray, 93
  - cudaCreateChannelDesc, 93
  - cudaGetChannelDesc, 94
  - cudaGetTextureAlignmentOffset, 94
  - cudaGetTextureReference, 95
  - cudaUnbindTexture, 95
- CUDART\_THREAD
  - cudaThreadExit, 11
  - cudaThreadGetCacheConfig, 11
  - cudaThreadGetLimit, 12
  - cudaThreadSetCacheConfig, 12
  - cudaThreadSetLimit, 13
  - cudaThreadSynchronize, 14
- CUDART\_TYPES
  - cudaArrayDefault, 138
  - cudaArraySurfaceLoadStore, 138
  - cudaChannelFormatKind, 140
  - cudaComputeMode, 140
  - cudaDeviceBlockingSync, 138
  - cudaDeviceLmemResizeToMax, 138
  - cudaDeviceMapHost, 138
  - cudaDeviceMask, 138
  - cudaDevicePropDontCare, 138
  - cudaDeviceScheduleAuto, 139
  - cudaDeviceScheduleSpin, 139
  - cudaDeviceScheduleYield, 139
  - cudaError, 140
  - cudaError\_t, 140
  - cudaEvent\_t, 140
  - cudaEventBlockingSync, 139
  - cudaEventDefault, 139
  - cudaEventDisableTiming, 139
  - cudaFuncCache, 144
  - cudaGraphicsCubeFace, 144
  - cudaGraphicsMapFlags, 144
  - cudaGraphicsRegisterFlags, 144
  - cudaGraphicsResource\_t, 140
  - cudaHostAllocDefault, 139
  - cudaHostAllocMapped, 139
  - cudaHostAllocPortable, 139
  - cudaHostAllocWriteCombined, 139
  - cudaLimit, 144
  - cudaMemcpyKind, 145
  - cudaStream\_t, 140
  - cudaSurfaceBoundaryMode, 145
  - cudaSurfaceFormatMode, 145
  - cudaTextureAddressMode, 145
  - cudaTextureFilterMode, 145
  - cudaTextureReadMode, 146
  - cudaUUID\_t, 140
- CUDART\_VDPAU
  - cudaGraphicsVDPAURegisterOutputSurface, 84
  - cudaGraphicsVDPAURegisterVideoSurface, 85
  - cudaVDPAUGetDevice, 85

- cudaVDPAUSetVDPAUDevice, 86
- CUDART\_VERSION
  - CUDART, 10
- cudaRuntimeGetVersion
  - CUDART\_\_VERSION, 98
- cudaSetDevice
  - CUDART\_DEVICE, 20
- cudaSetDeviceFlags
  - CUDART\_DEVICE, 20
- cudaSetDoubleForDevice
  - CUDART\_EXECUTION, 32
- cudaSetDoubleForHost
  - CUDART\_EXECUTION, 33
- cudaSetupArgument
  - CUDART\_EXECUTION, 33
  - CUDART\_HIGHLEVEL, 110
- cudaSetValidDevices
  - CUDART\_DEVICE, 21
- cudaStream\_t
  - CUDART\_TYPES, 140
- cudaStreamCreate
  - CUDART\_STREAM, 23
- cudaStreamDestroy
  - CUDART\_STREAM, 23
- cudaStreamQuery
  - CUDART\_STREAM, 24
- cudaStreamSynchronize
  - CUDART\_STREAM, 24
- cudaStreamWaitEvent
  - CUDART\_STREAM, 24
- cudaSuccess
  - CUDART\_TYPES, 141
- cudaSurfaceBoundaryMode
  - CUDART\_TYPES, 145
- cudaSurfaceFormatMode
  - CUDART\_TYPES, 145
- cudaTextureAddressMode
  - CUDART\_TYPES, 145
- cudaTextureFilterMode
  - CUDART\_TYPES, 145
- cudaTextureReadMode
  - CUDART\_TYPES, 146
- cudaThreadExit
  - CUDART\_THREAD, 11
- cudaThreadGetCacheConfig
  - CUDART\_THREAD, 11
- cudaThreadGetLimit
  - CUDART\_THREAD, 12
- cudaThreadSetCacheConfig
  - CUDART\_THREAD, 12
- cudaThreadSetLimit
  - CUDART\_THREAD, 13
- cudaThreadSynchronize
  - CUDART\_THREAD, 14
- cudaUnbindTexture
  - CUDART\_HIGHLEVEL, 111
  - CUDART\_TEXTURE, 95
- cudaUUID\_t
  - CUDART\_TYPES, 140
- cudaVDPAUGetDevice
  - CUDART\_VDPAU, 85
- cudaVDPAUSetVDPAUDevice
  - CUDART\_VDPAU, 86
- cudaWGLGetDevice
  - CUDART\_OPENGL, 68
- CUdevice
  - CUDA\_TYPES, 155
- CUdevice\_attribute
  - CUDA\_TYPES, 155
- CUdevice\_attribute\_enum
  - CUDA\_TYPES, 160
- cuDeviceComputeCapability
  - CUDA\_DEVICE, 167
- cuDeviceGet
  - CUDA\_DEVICE, 168
- cuDeviceGetAttribute
  - CUDA\_DEVICE, 168
- cuDeviceGetCount
  - CUDA\_DEVICE, 170
- cuDeviceGetName
  - CUDA\_DEVICE, 170
- cuDeviceGetProperties
  - CUDA\_DEVICE, 170
- CUdeviceptr
  - CUDA\_TYPES, 155
- cuDeviceTotalMem
  - CUDA\_DEVICE, 171
- CUdevprop
  - CUDA\_TYPES, 155
- CUdevprop\_st, 331
  - clockRate, 331
  - maxGridSize, 331
  - maxThreadsDim, 331
  - maxThreadsPerBlock, 331
  - memPitch, 331
  - regsPerBlock, 331
  - sharedMemPerBlock, 331
  - SIMDWidth, 331
  - textureAlign, 332
  - totalConstantMemory, 332
- cuDriverGetVersion
  - CUDA\_VERSION, 166
- CUevent
  - CUDA\_TYPES, 155
- CUevent\_flags
  - CUDA\_TYPES, 155
- CUevent\_flags\_enum
  - CUDA\_TYPES, 161

- cuEventCreate
  - CUDA\_EVENT, 233
- cuEventDestroy
  - CUDA\_EVENT, 234
- cuEventElapsedTime
  - CUDA\_EVENT, 234
- cuEventQuery
  - CUDA\_EVENT, 235
- cuEventRecord
  - CUDA\_EVENT, 235
- cuEventSynchronize
  - CUDA\_EVENT, 236
- CUfilter\_mode
  - CUDA\_TYPES, 155
- CUfilter\_mode\_enum
  - CUDA\_TYPES, 161
- CUfunc\_cache
  - CUDA\_TYPES, 155
- CUfunc\_cache\_enum
  - CUDA\_TYPES, 161
- cuFuncGetAttribute
  - CUDA\_EXEC, 238
- cuFuncSetBlockShape
  - CUDA\_EXEC, 238
- cuFuncSetCacheConfig
  - CUDA\_EXEC, 239
- cuFuncSetSharedSize
  - CUDA\_EXEC, 240
- CUfunction
  - CUDA\_TYPES, 155
- CUfunction\_attribute
  - CUDA\_TYPES, 155
- CUfunction\_attribute\_enum
  - CUDA\_TYPES, 161
- cuGLCtxCreate
  - CUDA\_GL, 262
- cuGLInit
  - CUDA\_GL\_DEPRECATED, 266
- CUGLmap\_flags
  - CUDA\_GL\_DEPRECATED, 265
- CUGLmap\_flags\_enum
  - CUDA\_GL\_DEPRECATED, 266
- cuGLMapBufferObject
  - CUDA\_GL\_DEPRECATED, 266
- cuGLMapBufferObjectAsync
  - CUDA\_GL\_DEPRECATED, 267
- cuGLRegisterBufferObject
  - CUDA\_GL\_DEPRECATED, 267
- cuGLSetBufferObjectMapFlags
  - CUDA\_GL\_DEPRECATED, 268
- cuGLUnmapBufferObject
  - CUDA\_GL\_DEPRECATED, 268
- cuGLUnmapBufferObjectAsync
  - CUDA\_GL\_DEPRECATED, 269
- cuGLUnregisterBufferObject
  - CUDA\_GL\_DEPRECATED, 269
- cuGraphicsD3D10RegisterResource
  - CUDA\_D3D10, 289
- cuGraphicsD3D11RegisterResource
  - CUDA\_D3D11, 304
- cuGraphicsD3D9RegisterResource
  - CUDA\_D3D9, 274
- cuGraphicsGLRegisterBuffer
  - CUDA\_GL, 263
- cuGraphicsGLRegisterImage
  - CUDA\_GL, 263
- CUgraphicsMapResourceFlags
  - CUDA\_TYPES, 155
- CUgraphicsMapResourceFlags\_enum
  - CUDA\_TYPES, 162
- cuGraphicsMapResources
  - CUDA\_GRAPHICS, 257
- CUgraphicsRegisterFlags
  - CUDA\_TYPES, 155
- CUgraphicsRegisterFlags\_enum
  - CUDA\_TYPES, 162
- CUgraphicsResource
  - CUDA\_TYPES, 156
- cuGraphicsResourceGetMappedPointer
  - CUDA\_GRAPHICS, 258
- cuGraphicsResourceSetMapFlags
  - CUDA\_GRAPHICS, 258
- cuGraphicsSubResourceGetMappedArray
  - CUDA\_GRAPHICS, 259
- cuGraphicsUnmapResources
  - CUDA\_GRAPHICS, 260
- cuGraphicsUnregisterResource
  - CUDA\_GRAPHICS, 260
- cuGraphicsVDPAURegisterOutputSurface
  - CUDA\_VDPAU, 307
- cuGraphicsVDPAURegisterVideoSurface
  - CUDA\_VDPAU, 308
- cuInit
  - CUDA\_INITIALIZE, 165
- CUjit\_fallback
  - CUDA\_TYPES, 156
- CUjit\_fallback\_enum
  - CUDA\_TYPES, 162
- CUjit\_option
  - CUDA\_TYPES, 156
- CUjit\_option\_enum
  - CUDA\_TYPES, 162
- CUjit\_target
  - CUDA\_TYPES, 156
- CUjit\_target\_enum
  - CUDA\_TYPES, 163
- cuLaunch
  - CUDA\_EXEC, 240

- cuLaunchGrid
  - CUDA\_EXEC, 240
- cuLaunchGridAsync
  - CUDA\_EXEC, 241
- CUlimit
  - CUDA\_TYPES, 156
- CUlimit\_enum
  - CUDA\_TYPES, 163
- cuMemAlloc
  - CUDA\_MEM, 195
- cuMemAllocHost
  - CUDA\_MEM, 196
- cuMemAllocPitch
  - CUDA\_MEM, 197
- cuMemcpy2D
  - CUDA\_MEM, 198
- cuMemcpy2DAsync
  - CUDA\_MEM, 200
- cuMemcpy2DUnaligned
  - CUDA\_MEM, 202
- cuMemcpy3D
  - CUDA\_MEM, 204
- cuMemcpy3DAsync
  - CUDA\_MEM, 206
- cuMemcpyAtoA
  - CUDA\_MEM, 209
- cuMemcpyAtoD
  - CUDA\_MEM, 209
- cuMemcpyAtoH
  - CUDA\_MEM, 210
- cuMemcpyAtoHAsync
  - CUDA\_MEM, 211
- cuMemcpyDtoA
  - CUDA\_MEM, 211
- cuMemcpyDtoD
  - CUDA\_MEM, 212
- cuMemcpyDtoDAsync
  - CUDA\_MEM, 212
- cuMemcpyDtoH
  - CUDA\_MEM, 213
- cuMemcpyDtoHAsync
  - CUDA\_MEM, 214
- cuMemcpyHtoA
  - CUDA\_MEM, 214
- cuMemcpyHtoAAsync
  - CUDA\_MEM, 215
- cuMemcpyHtoD
  - CUDA\_MEM, 216
- cuMemcpyHtoDAsync
  - CUDA\_MEM, 216
- cuMemFree
  - CUDA\_MEM, 217
- cuMemFreeHost
  - CUDA\_MEM, 217
- cuMemGetAddressRange
  - CUDA\_MEM, 218
- cuMemGetInfo
  - CUDA\_MEM, 218
- cuMemHostAlloc
  - CUDA\_MEM, 219
- cuMemHostGetDevicePointer
  - CUDA\_MEM, 220
- cuMemHostGetFlags
  - CUDA\_MEM, 221
- CUmemorytype
  - CUDA\_TYPES, 156
- CUmemorytype\_enum
  - CUDA\_TYPES, 164
- cuMemsetD16
  - CUDA\_MEM, 221
- cuMemsetD16Async
  - CUDA\_MEM, 222
- cuMemsetD2D16
  - CUDA\_MEM, 222
- cuMemsetD2D16Async
  - CUDA\_MEM, 223
- cuMemsetD2D32
  - CUDA\_MEM, 224
- cuMemsetD2D32Async
  - CUDA\_MEM, 224
- cuMemsetD2D8
  - CUDA\_MEM, 225
- cuMemsetD2D8Async
  - CUDA\_MEM, 226
- cuMemsetD32
  - CUDA\_MEM, 226
- cuMemsetD32Async
  - CUDA\_MEM, 227
- cuMemsetD8
  - CUDA\_MEM, 228
- cuMemsetD8Async
  - CUDA\_MEM, 228
- CUmodule
  - CUDA\_TYPES, 156
- cuModuleGetFunction
  - CUDA\_MODULE, 182
- cuModuleGetGlobal
  - CUDA\_MODULE, 183
- cuModuleGetSurfRef
  - CUDA\_MODULE, 183
- cuModuleGetTexRef
  - CUDA\_MODULE, 184
- cuModuleLoad
  - CUDA\_MODULE, 184
- cuModuleLoadData
  - CUDA\_MODULE, 185
- cuModuleLoadDataEx
  - CUDA\_MODULE, 185

- cuModuleLoadFatBinary
    - CUDA\_MODULE, 187
  - cuModuleUnload
    - CUDA\_MODULE, 187
  - cuParamSetf
    - CUDA\_EXEC, 241
  - cuParamSeti
    - CUDA\_EXEC, 242
  - cuParamSetSize
    - CUDA\_EXEC, 242
  - cuParamSetTexRef
    - CUDA\_EXEC\_DEPRECATED, 244
  - cuParamSetv
    - CUDA\_EXEC, 243
  - CUresult
    - CUDA\_TYPES, 156
  - CUstream
    - CUDA\_TYPES, 156
  - cuStreamCreate
    - CUDA\_STREAM, 230
  - cuStreamDestroy
    - CUDA\_STREAM, 230
  - cuStreamQuery
    - CUDA\_STREAM, 231
  - cuStreamSynchronize
    - CUDA\_STREAM, 231
  - cuStreamWaitEvent
    - CUDA\_STREAM, 232
  - CUsurfref
    - CUDA\_TYPES, 156
  - cuSurfRefGetArray
    - CUDA\_SURFREF, 255
  - cuSurfRefSetArray
    - CUDA\_SURFREF, 255
  - CUtexref
    - CUDA\_TYPES, 156
  - cuTexRefCreate
    - CUDA\_TEXREF\_DEPRECATED, 253
  - cuTexRefDestroy
    - CUDA\_TEXREF\_DEPRECATED, 253
  - cuTexRefGetAddress
    - CUDA\_TEXREF, 246
  - cuTexRefGetAddressMode
    - CUDA\_TEXREF, 246
  - cuTexRefGetArray
    - CUDA\_TEXREF, 246
  - cuTexRefGetFilterMode
    - CUDA\_TEXREF, 247
  - cuTexRefGetFlags
    - CUDA\_TEXREF, 247
  - cuTexRefGetFormat
    - CUDA\_TEXREF, 248
  - cuTexRefSetAddress
    - CUDA\_TEXREF, 248
  - cuTexRefSetAddress2D
    - CUDA\_TEXREF, 249
  - cuTexRefSetAddressMode
    - CUDA\_TEXREF, 249
  - cuTexRefSetArray
    - CUDA\_TEXREF, 250
  - cuTexRefSetFilterMode
    - CUDA\_TEXREF, 250
  - cuTexRefSetFlags
    - CUDA\_TEXREF, 251
  - cuTexRefSetFormat
    - CUDA\_TEXREF, 251
  - cuVDPAUCtxCreate
    - CUDA\_VDPAU, 309
  - cuVDPAUGetDevice
    - CUDA\_VDPAU, 309
  - cuWGLGetDevice
    - CUDA\_GL, 264
- Data types used by CUDA driver, 148
- Data types used by CUDA Runtime, 135
- Depth
  - CUDA\_ARRAY3D\_DESCRIPTOR\_st, 311
  - CUDA\_MEMCPY3D\_st, 316
- depth
  - cudaExtent, 324
- Device Management, 17, 167
- deviceOverlap
  - cudaDeviceProp, 321
- Direct3D 10 Interoperability, 75, 286
- Direct3D 11 Interoperability, 80, 301
- Direct3D 9 Interoperability, 70, 271
- dstArray
  - CUDA\_MEMCPY2D\_st, 314
  - CUDA\_MEMCPY3D\_st, 316
  - cudaMemcpy3DParms, 327
- dstDevice
  - CUDA\_MEMCPY2D\_st, 314
  - CUDA\_MEMCPY3D\_st, 316
- dstHeight
  - CUDA\_MEMCPY3D\_st, 316
- dstHost
  - CUDA\_MEMCPY2D\_st, 314
  - CUDA\_MEMCPY3D\_st, 317
- dstLOD
  - CUDA\_MEMCPY3D\_st, 317
- dstMemoryType
  - CUDA\_MEMCPY2D\_st, 314
  - CUDA\_MEMCPY3D\_st, 317
- dstPitch
  - CUDA\_MEMCPY2D\_st, 314
  - CUDA\_MEMCPY3D\_st, 317
- dstPos
  - cudaMemcpy3DParms, 327

- dstPtr
  - cudaMemcpy3DParms, 327
- dstXInBytes
  - CUDA\_MEMCPY2D\_st, 314
  - CUDA\_MEMCPY3D\_st, 317
- dstY
  - CUDA\_MEMCPY2D\_st, 315
  - CUDA\_MEMCPY3D\_st, 317
- dstZ
  - CUDA\_MEMCPY3D\_st, 317
- ECCEEnabled
  - cudaDeviceProp, 321
- Error Handling, 15
- Event Management, 26, 233
- Execution Control, 30, 237
- extent
  - cudaMemcpy3DParms, 327
- f
  - cudaChannelFormatDesc, 319
- filterMode
  - textureReference, 334
- Flags
  - CUDA\_ARRAY3D\_DESCRIPTOR\_st, 311
- Format
  - CUDA\_ARRAY3D\_DESCRIPTOR\_st, 311
  - CUDA\_ARRAY\_DESCRIPTOR\_st, 313
- Graphics Interoperability, 87, 257
- Height
  - CUDA\_ARRAY3D\_DESCRIPTOR\_st, 311
  - CUDA\_ARRAY\_DESCRIPTOR\_st, 313
  - CUDA\_MEMCPY2D\_st, 315
  - CUDA\_MEMCPY3D\_st, 317
- height
  - cudaExtent, 324
- Initialization, 165
- integrated
  - cudaDeviceProp, 321
- Interactions with the CUDA Driver API, 112
- kernelExecTimeoutEnabled
  - cudaDeviceProp, 321
- kind
  - cudaMemcpy3DParms, 327
- localSizeBytes
  - cudaFuncAttributes, 325
- major
  - cudaDeviceProp, 321
- make\_cudaExtent
  - CUDART\_MEMORY, 64
- make\_cudaPitchedPtr
  - CUDART\_MEMORY, 64
- make\_cudaPos
  - CUDART\_MEMORY, 65
- maxGridSize
  - cudaDeviceProp, 321
  - CUdevprop\_st, 331
- maxTexture1D
  - cudaDeviceProp, 321
- maxTexture2D
  - cudaDeviceProp, 321
- maxTexture2DArray
  - cudaDeviceProp, 321
- maxTexture3D
  - cudaDeviceProp, 321
- maxThreadsDim
  - cudaDeviceProp, 321
  - CUdevprop\_st, 331
- maxThreadsPerBlock
  - cudaDeviceProp, 322
  - cudaFuncAttributes, 325
  - CUdevprop\_st, 331
- Memory Management, 34, 188
- memPitch
  - cudaDeviceProp, 322
  - CUdevprop\_st, 331
- minor
  - cudaDeviceProp, 322
- Module Management, 182
- multiProcessorCount
  - cudaDeviceProp, 322
- name
  - cudaDeviceProp, 322
- normalized
  - textureReference, 334
- NumChannels
  - CUDA\_ARRAY3D\_DESCRIPTOR\_st, 311
  - CUDA\_ARRAY\_DESCRIPTOR\_st, 313
- numRegs
  - cudaFuncAttributes, 325
- OpenGL Interoperability, 66, 262
- pciBusID
  - cudaDeviceProp, 322
- pciDeviceID
  - cudaDeviceProp, 322
- pitch
  - cudaPitchedPtr, 329
- ptr
  - cudaPitchedPtr, 329
- ptxVersion
  - cudaFuncAttributes, 325

- regsPerBlock
  - cudaDeviceProp, 322
  - CUdevprop\_st, 331
- reserved0
  - CUDA\_MEMCPY3D\_st, 317
- reserved1
  - CUDA\_MEMCPY3D\_st, 317
- sharedMemPerBlock
  - cudaDeviceProp, 322
  - CUdevprop\_st, 331
- sharedSizeBytes
  - cudaFuncAttributes, 325
- SIMDWidth
  - CUdevprop\_st, 331
- srcArray
  - CUDA\_MEMCPY2D\_st, 315
  - CUDA\_MEMCPY3D\_st, 317
  - cudaMemcpy3DParms, 327
- srcDevice
  - CUDA\_MEMCPY2D\_st, 315
  - CUDA\_MEMCPY3D\_st, 317
- srcHeight
  - CUDA\_MEMCPY3D\_st, 318
- srcHost
  - CUDA\_MEMCPY2D\_st, 315
  - CUDA\_MEMCPY3D\_st, 318
- srcLOD
  - CUDA\_MEMCPY3D\_st, 318
- srcMemoryType
  - CUDA\_MEMCPY2D\_st, 315
  - CUDA\_MEMCPY3D\_st, 318
- srcPitch
  - CUDA\_MEMCPY2D\_st, 315
  - CUDA\_MEMCPY3D\_st, 318
- srcPos
  - cudaMemcpy3DParms, 327
- srcPtr
  - cudaMemcpy3DParms, 327
- srcXInBytes
  - CUDA\_MEMCPY2D\_st, 315
  - CUDA\_MEMCPY3D\_st, 318
- srcY
  - CUDA\_MEMCPY2D\_st, 315
  - CUDA\_MEMCPY3D\_st, 318
- srcZ
  - CUDA\_MEMCPY3D\_st, 318
- Stream Management, 23, 230
- Surface Reference Management, 96, 255
- surfaceAlignment
  - cudaDeviceProp, 322
- surfaceReference, 333
  - channelDesc, 333
- tccDriver
  - cudaDeviceProp, 322
- Texture Reference Management, 91, 245
- textureAlign
  - CUdevprop\_st, 332
- textureAlignment
  - cudaDeviceProp, 322
- textureReference, 334
  - addressMode, 334
  - channelDesc, 334
  - filterMode, 334
  - normalized, 334
- Thread Management, 11
- totalConstantMemory
  - CUdevprop\_st, 332
- totalConstMem
  - cudaDeviceProp, 323
- totalGlobalMem
  - cudaDeviceProp, 323
- VDPAU Interoperability, 84, 307
- Version Management, 98, 166
- w
  - cudaChannelFormatDesc, 319
- warpSize
  - cudaDeviceProp, 323
- Width
  - CUDA\_ARRAY3D\_DESCRIPTOR\_st, 312
  - CUDA\_ARRAY\_DESCRIPTOR\_st, 313
- width
  - cudaExtent, 324
- WidthInBytes
  - CUDA\_MEMCPY2D\_st, 315
  - CUDA\_MEMCPY3D\_st, 318
- x
  - cudaChannelFormatDesc, 319
  - cudaPos, 330
- xsize
  - cudaPitchedPtr, 329
- y
  - cudaChannelFormatDesc, 319
  - cudaPos, 330
- ysize
  - cudaPitchedPtr, 329
- z
  - cudaChannelFormatDesc, 319
  - cudaPos, 330



## **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**

© 2007-2010 NVIDIA Corporation. All rights reserved.



**NVIDIA**

NVIDIA Corporation  
2701 San Tomas Expressway  
Santa Clara, CA 95050  
[www.nvidia.com](http://www.nvidia.com)