



NVIDIA CUDA

Reference Manual

Version 4.0

February 2011

Contents

1	Deprecated List	1
2	Module Index	7
2.1	Modules	7
3	Data Structure Index	9
3.1	Data Structures	9
4	Module Documentation	11
4.1	CUDA Runtime API	11
4.1.1	Detailed Description	12
4.1.2	Define Documentation	12
4.1.2.1	CUDART_VERSION	12
4.2	Device Management	13
4.2.1	Detailed Description	14
4.2.2	Function Documentation	14
4.2.2.1	cudaChooseDevice	14
4.2.2.2	cudaDeviceGetCacheConfig	14
4.2.2.3	cudaDeviceGetLimit	15
4.2.2.4	cudaDeviceReset	15
4.2.2.5	cudaDeviceSetCacheConfig	15
4.2.2.6	cudaDeviceSetLimit	16
4.2.2.7	cudaDeviceSynchronize	17
4.2.2.8	cudaGetDevice	17
4.2.2.9	cudaGetDeviceCount	18
4.2.2.10	cudaGetDeviceProperties	18
4.2.2.11	cudaSetDevice	20
4.2.2.12	cudaSetDeviceFlags	21
4.2.2.13	cudaSetValidDevices	21

4.3	Thread Management [DEPRECATED]	23
4.3.1	Detailed Description	23
4.3.2	Function Documentation	23
4.3.2.1	cudaThreadExit	23
4.3.2.2	cudaThreadGetCacheConfig	24
4.3.2.3	cudaThreadGetLimit	24
4.3.2.4	cudaThreadSetCacheConfig	25
4.3.2.5	cudaThreadSetLimit	26
4.3.2.6	cudaThreadSynchronize	27
4.4	Error Handling	28
4.4.1	Detailed Description	28
4.4.2	Function Documentation	28
4.4.2.1	cudaGetErrorString	28
4.4.2.2	cudaGetLastError	28
4.4.2.3	cudaPeekAtLastError	29
4.5	Stream Management	30
4.5.1	Detailed Description	30
4.5.2	Function Documentation	30
4.5.2.1	cudaStreamCreate	30
4.5.2.2	cudaStreamDestroy	30
4.5.2.3	cudaStreamQuery	31
4.5.2.4	cudaStreamSynchronize	31
4.5.2.5	cudaStreamWaitEvent	32
4.6	Event Management	33
4.6.1	Detailed Description	33
4.6.2	Function Documentation	33
4.6.2.1	cudaEventCreate	33
4.6.2.2	cudaEventCreateWithFlags	34
4.6.2.3	cudaEventDestroy	34
4.6.2.4	cudaEventElapsedTime	35
4.6.2.5	cudaEventQuery	35
4.6.2.6	cudaEventRecord	36
4.6.2.7	cudaEventSynchronize	36
4.7	Execution Control	37
4.7.1	Detailed Description	37
4.7.2	Function Documentation	37
4.7.2.1	cudaConfigureCall	37

4.7.2.2	cudaFuncGetAttributes	38
4.7.2.3	cudaFuncSetCacheConfig	38
4.7.2.4	cudaLaunch	39
4.7.2.5	cudaSetDoubleForDevice	39
4.7.2.6	cudaSetDoubleForHost	40
4.7.2.7	cudaSetupArgument	40
4.8	Memory Management	41
4.8.1	Detailed Description	44
4.8.2	Function Documentation	44
4.8.2.1	cudaFree	44
4.8.2.2	cudaFreeArray	45
4.8.2.3	cudaFreeHost	45
4.8.2.4	cudaGetSymbolAddress	46
4.8.2.5	cudaGetSymbolSize	46
4.8.2.6	cudaHostAlloc	46
4.8.2.7	cudaHostGetDevicePointer	47
4.8.2.8	cudaHostGetFlags	48
4.8.2.9	cudaHostRegister	48
4.8.2.10	cudaHostUnregister	49
4.8.2.11	cudaMalloc	50
4.8.2.12	cudaMalloc3D	50
4.8.2.13	cudaMalloc3DArray	51
4.8.2.14	cudaMallocArray	52
4.8.2.15	cudaMallocHost	53
4.8.2.16	cudaMallocPitch	53
4.8.2.17	cudaMemcpy	54
4.8.2.18	cudaMemcpy2D	54
4.8.2.19	cudaMemcpy2DToArray	55
4.8.2.20	cudaMemcpy2DAsync	56
4.8.2.21	cudaMemcpy2DFromArray	57
4.8.2.22	cudaMemcpy2DFromArrayAsync	57
4.8.2.23	cudaMemcpy2DToArray	58
4.8.2.24	cudaMemcpy2DToArrayAsync	59
4.8.2.25	cudaMemcpy3D	60
4.8.2.26	cudaMemcpy3DAsync	61
4.8.2.27	cudaMemcpy3DPeer	63
4.8.2.28	cudaMemcpy3DPeerAsync	63

4.8.2.29	cudaMemcpyToArray	63
4.8.2.30	cudaMemcpyAsync	64
4.8.2.31	cudaMemcpyFromArray	65
4.8.2.32	cudaMemcpyFromArrayAsync	65
4.8.2.33	cudaMemcpyFromSymbol	66
4.8.2.34	cudaMemcpyFromSymbolAsync	67
4.8.2.35	cudaMemcpyPeer	67
4.8.2.36	cudaMemcpyPeerAsync	68
4.8.2.37	cudaMemcpyToArray	69
4.8.2.38	cudaMemcpyToArrayAsync	69
4.8.2.39	cudaMemcpyToSymbol	70
4.8.2.40	cudaMemcpyToSymbolAsync	71
4.8.2.41	cudaMemGetInfo	71
4.8.2.42	cudaMemset	72
4.8.2.43	cudaMemset2D	72
4.8.2.44	cudaMemset2DAsync	73
4.8.2.45	cudaMemset3D	73
4.8.2.46	cudaMemset3DAsync	74
4.8.2.47	cudaMemsetAsync	75
4.8.2.48	make_cudaExtent	75
4.8.2.49	make_cudaPitchedPtr	75
4.8.2.50	make_cudaPos	76
4.9	Unified Addressing	77
4.9.1	Detailed Description	77
4.9.2	Overview	77
4.9.3	Supported Platforms	77
4.9.4	Looking Up Information from Pointer Values	77
4.9.5	Automatic Mapping of Host Allocated Host Memory	77
4.9.6	Automatic Registration of Peer Memory	78
4.9.7	Exceptions, Disjoint Addressing	78
4.9.8	Function Documentation	78
4.9.8.1	cudaPointerGetAttributes	78
4.10	Peer Device Memory Access	80
4.10.1	Detailed Description	80
4.10.2	Function Documentation	80
4.10.2.1	cudaDeviceCanAccessPeer	80
4.10.2.2	cudaDeviceDisablePeerAccess	81

4.10.2.3	cudaDeviceEnablePeerAccess	81
4.10.2.4	cudaPeerGetDevicePointer	82
4.10.2.5	cudaPeerRegister	82
4.10.2.6	cudaPeerUnregister	83
4.11	OpenGL Interoperability	85
4.11.1	Detailed Description	85
4.11.2	Function Documentation	85
4.11.2.1	cudaGraphicsGLRegisterBuffer	85
4.11.2.2	cudaGraphicsGLRegisterImage	86
4.11.2.3	cudaWGLGetDevice	87
4.12	Direct3D 9 Interoperability	88
4.12.1	Detailed Description	88
4.12.2	Enumeration Type Documentation	88
4.12.2.1	cudaD3D9DeviceList	88
4.12.3	Function Documentation	89
4.12.3.1	cudaD3D9GetDevice	89
4.12.3.2	cudaD3D9GetDevices	89
4.12.3.3	cudaD3D9GetDirect3DDevice	90
4.12.3.4	cudaD3D9SetDirect3DDevice	90
4.12.3.5	cudaGraphicsD3D9RegisterResource	91
4.13	Direct3D 10 Interoperability	93
4.13.1	Detailed Description	93
4.13.2	Enumeration Type Documentation	93
4.13.2.1	cudaD3D10DeviceList	93
4.13.3	Function Documentation	94
4.13.3.1	cudaD3D10GetDevice	94
4.13.3.2	cudaD3D10GetDevices	94
4.13.3.3	cudaD3D10GetDirect3DDevice	95
4.13.3.4	cudaD3D10SetDirect3DDevice	95
4.13.3.5	cudaGraphicsD3D10RegisterResource	96
4.14	Direct3D 11 Interoperability	98
4.14.1	Detailed Description	98
4.14.2	Enumeration Type Documentation	98
4.14.2.1	cudaD3D11DeviceList	98
4.14.3	Function Documentation	99
4.14.3.1	cudaD3D11GetDevice	99
4.14.3.2	cudaD3D11GetDevices	99

4.14.3.3	cudaD3D11GetDirect3DDevice	100
4.14.3.4	cudaD3D11SetDirect3DDevice	100
4.14.3.5	cudaGraphicsD3D11RegisterResource	101
4.15	VDPAU Interoperability	103
4.15.1	Detailed Description	103
4.15.2	Function Documentation	103
4.15.2.1	cudaGraphicsVDPAURegisterOutputSurface	103
4.15.2.2	cudaGraphicsVDPAURegisterVideoSurface	104
4.15.2.3	cudaVDPAUGetDevice	104
4.15.2.4	cudaVDPAUSetVDPAUDevice	105
4.16	Graphics Interoperability	106
4.16.1	Detailed Description	106
4.16.2	Function Documentation	106
4.16.2.1	cudaGraphicsMapResources	106
4.16.2.2	cudaGraphicsResourceGetMappedPointer	107
4.16.2.3	cudaGraphicsResourceSetMapFlags	107
4.16.2.4	cudaGraphicsSubResourceGetMappedArray	108
4.16.2.5	cudaGraphicsUnmapResources	109
4.16.2.6	cudaGraphicsUnregisterResource	109
4.17	Texture Reference Management	110
4.17.1	Detailed Description	110
4.17.2	Function Documentation	110
4.17.2.1	cudaBindTexture	110
4.17.2.2	cudaBindTexture2D	111
4.17.2.3	cudaBindTextureToArray	112
4.17.2.4	cudaCreateChannelDesc	112
4.17.2.5	cudaGetChannelDesc	113
4.17.2.6	cudaGetTextureAlignmentOffset	113
4.17.2.7	cudaGetTextureReference	114
4.17.2.8	cudaUnbindTexture	114
4.18	Surface Reference Management	115
4.18.1	Detailed Description	115
4.18.2	Function Documentation	115
4.18.2.1	cudaBindSurfaceToArray	115
4.18.2.2	cudaGetSurfaceReference	115
4.19	Version Management	117
4.19.1	Function Documentation	117

4.19.1.1	cudaDriverGetVersion	117
4.19.1.2	cudaRuntimeGetVersion	117
4.20	C++ API Routines	118
4.20.1	Detailed Description	119
4.20.2	Function Documentation	119
4.20.2.1	cudaBindSurfaceToArray	119
4.20.2.2	cudaBindSurfaceToArray	120
4.20.2.3	cudaBindTexture	120
4.20.2.4	cudaBindTexture	121
4.20.2.5	cudaBindTexture2D	122
4.20.2.6	cudaBindTexture2D	122
4.20.2.7	cudaBindTextureToArray	123
4.20.2.8	cudaBindTextureToArray	124
4.20.2.9	cudaCreateChannelDesc	124
4.20.2.10	cudaEventCreate	125
4.20.2.11	cudaFuncGetAttributes	125
4.20.2.12	cudaFuncSetCacheConfig	126
4.20.2.13	cudaGetSymbolAddress	126
4.20.2.14	cudaGetSymbolSize	127
4.20.2.15	cudaGetTextureAlignmentOffset	127
4.20.2.16	cudaLaunch	128
4.20.2.17	cudaMallocHost	128
4.20.2.18	cudaSetupArgument	129
4.20.2.19	cudaUnbindTexture	130
4.21	Interactions with the CUDA Driver API	131
4.21.1	Primary Contexts	131
4.21.2	Initialization and Tear-Down	131
4.21.3	Context Interoperability	131
4.21.4	Interactions between CUstream and cudaStream_t	132
4.21.5	Interactions between CUevent and cudaEvent_t	132
4.21.6	Interactions between CUarray and struct cudaArray *	132
4.21.7	Interactions between CUgraphicsResource and cudaGraphicsResource_t	132
4.22	Direct3D 9 Interoperability [DEPRECATED]	133
4.22.1	Detailed Description	134
4.22.2	Enumeration Type Documentation	134
4.22.2.1	cudaD3D9MapFlags	134
4.22.2.2	cudaD3D9RegisterFlags	134

4.22.3	Function Documentation	134
4.22.3.1	cudaD3D9MapResources	134
4.22.3.2	cudaD3D9RegisterResource	135
4.22.3.3	cudaD3D9ResourceGetMappedArray	136
4.22.3.4	cudaD3D9ResourceGetMappedPitch	137
4.22.3.5	cudaD3D9ResourceGetMappedPointer	137
4.22.3.6	cudaD3D9ResourceGetMappedSize	138
4.22.3.7	cudaD3D9ResourceGetSurfaceDimensions	139
4.22.3.8	cudaD3D9ResourceSetMapFlags	140
4.22.3.9	cudaD3D9UnmapResources	140
4.22.3.10	cudaD3D9UnregisterResource	141
4.23	Direct3D 10 Interoperability [DEPRECATED]	142
4.23.1	Detailed Description	143
4.23.2	Enumeration Type Documentation	143
4.23.2.1	cudaD3D10MapFlags	143
4.23.2.2	cudaD3D10RegisterFlags	143
4.23.3	Function Documentation	143
4.23.3.1	cudaD3D10MapResources	143
4.23.3.2	cudaD3D10RegisterResource	144
4.23.3.3	cudaD3D10ResourceGetMappedArray	145
4.23.3.4	cudaD3D10ResourceGetMappedPitch	146
4.23.3.5	cudaD3D10ResourceGetMappedPointer	146
4.23.3.6	cudaD3D10ResourceGetMappedSize	147
4.23.3.7	cudaD3D10ResourceGetSurfaceDimensions	148
4.23.3.8	cudaD3D10ResourceSetMapFlags	148
4.23.3.9	cudaD3D10UnmapResources	149
4.23.3.10	cudaD3D10UnregisterResource	150
4.24	OpenGL Interoperability [DEPRECATED]	151
4.24.1	Detailed Description	151
4.24.2	Enumeration Type Documentation	151
4.24.2.1	cudaGLMapFlags	151
4.24.3	Function Documentation	152
4.24.3.1	cudaGLMapBufferObject	152
4.24.3.2	cudaGLMapBufferObjectAsync	152
4.24.3.3	cudaGLRegisterBufferObject	153
4.24.3.4	cudaGLSetBufferObjectMapFlags	153
4.24.3.5	cudaGLUnmapBufferObject	154

4.24.3.6	cudaGLUnmapBufferObjectAsync	154
4.24.3.7	cudaGLUnregisterBufferObject	155
4.25	Data types used by CUDA Runtime	156
4.25.1	Define Documentation	160
4.25.1.1	cudaArrayDefault	160
4.25.1.2	cudaArrayLayered	160
4.25.1.3	cudaArraySurfaceLoadStore	160
4.25.1.4	cudaDeviceBlockingSync	160
4.25.1.5	cudaDeviceLmemResizeToMax	160
4.25.1.6	cudaDeviceMapHost	160
4.25.1.7	cudaDeviceMask	160
4.25.1.8	cudaDevicePropDontCare	160
4.25.1.9	cudaDeviceScheduleAuto	161
4.25.1.10	cudaDeviceScheduleBlockingSync	161
4.25.1.11	cudaDeviceScheduleSpin	161
4.25.1.12	cudaDeviceScheduleYield	161
4.25.1.13	cudaEventBlockingSync	161
4.25.1.14	cudaEventDefault	161
4.25.1.15	cudaEventDisableTiming	161
4.25.1.16	cudaHostAllocDefault	161
4.25.1.17	cudaHostAllocMapped	161
4.25.1.18	cudaHostAllocPortable	161
4.25.1.19	cudaHostAllocWriteCombined	161
4.25.1.20	cudaHostRegisterDefault	161
4.25.1.21	cudaHostRegisterMapped	162
4.25.1.22	cudaHostRegisterPortable	162
4.25.1.23	cudaPeerAccessDefault	162
4.25.1.24	cudaPeerDevicePointerDefault	162
4.25.1.25	cudaPeerRegisterDefault	162
4.25.1.26	cudaPeerRegisterMapped	162
4.25.2	Typedef Documentation	162
4.25.2.1	cudaError_t	162
4.25.2.2	cudaEvent_t	162
4.25.2.3	cudaGraphicsResource_t	162
4.25.2.4	cudaOutputMode_t	162
4.25.2.5	cudaStream_t	162
4.25.2.6	cudaUUID_t	163

4.25.3 Enumeration Type Documentation	163
4.25.3.1 cudaChannelFormatKind	163
4.25.3.2 cudaComputeMode	163
4.25.3.3 cudaError	163
4.25.3.4 cudaFuncCache	167
4.25.3.5 cudaGraphicsCubeFace	167
4.25.3.6 cudaGraphicsMapFlags	167
4.25.3.7 cudaGraphicsRegisterFlags	168
4.25.3.8 cudaLimit	168
4.25.3.9 cudaMemcpyKind	168
4.25.3.10 cudaMemcpyType	168
4.25.3.11 cudaMemcpyMode	168
4.25.3.12 cudaSurfaceBoundaryMode	169
4.25.3.13 cudaSurfaceFormatMode	169
4.25.3.14 cudaTextureAddressMode	169
4.25.3.15 cudaTextureFilterMode	169
4.25.3.16 cudaTextureReadMode	169
4.26 CUDA Driver API	170
4.26.1 Detailed Description	170
4.27 Data types used by CUDA driver	171
4.27.1 Define Documentation	177
4.27.1.1 CU_LAUNCH_PARAM_BUFFER_POINTER	177
4.27.1.2 CU_LAUNCH_PARAM_BUFFER_SIZE	177
4.27.1.3 CU_LAUNCH_PARAM_END	177
4.27.1.4 CU_MEMHOSTALLOC_DEVICEMAP	177
4.27.1.5 CU_MEMHOSTALLOC_PORTABLE	177
4.27.1.6 CU_MEMHOSTALLOC_WRITECOMBINED	177
4.27.1.7 CU_MEMHOSTREGISTER_DEVICEMAP	177
4.27.1.8 CU_MEMHOSTREGISTER_PORTABLE	178
4.27.1.9 CU_MEMPEERREGISTER_DEVICEMAP	178
4.27.1.10 CU_PARAM_TR_DEFAULT	178
4.27.1.11 CU_TRSA_OVERRIDE_FORMAT	178
4.27.1.12 CU_TRSF_NORMALIZED_COORDINATES	178
4.27.1.13 CU_TRSF_READ_AS_INTEGER	178
4.27.1.14 CU_TRSF_SRGB	178
4.27.1.15 CUDA_ARRAY3D_2DARRAY	178
4.27.1.16 CUDA_ARRAY3D_LAYERED	178

4.27.1.17 CUDA_ARRAY3D_SURFACE_LDST	178
4.27.1.18 CUDA_VERSION	178
4.27.2 Typedef Documentation	179
4.27.2.1 CUaddress_mode	179
4.27.2.2 CUarray	179
4.27.2.3 CUarray_cubemap_face	179
4.27.2.4 CUarray_format	179
4.27.2.5 CUcomputemode	179
4.27.2.6 CUcontext	179
4.27.2.7 CUctx_flags	179
4.27.2.8 CUDA_ARRAY3D_DESCRIPTOR	179
4.27.2.9 CUDA_ARRAY_DESCRIPTOR	179
4.27.2.10 CUDA_MEMCPY2D	179
4.27.2.11 CUDA_MEMCPY3D	179
4.27.2.12 CUDA_MEMCPY3D_PEER	180
4.27.2.13 CUdevice	180
4.27.2.14 CUdevice_attribute	180
4.27.2.15 CUdeviceptr	180
4.27.2.16 CUdevprop	180
4.27.2.17 CUevent	180
4.27.2.18 CUevent_flags	180
4.27.2.19 CUfilter_mode	180
4.27.2.20 CUfunc_cache	180
4.27.2.21 CUfunction	180
4.27.2.22 CUfunction_attribute	180
4.27.2.23 CUgraphicsMapResourceFlags	180
4.27.2.24 CUgraphicsRegisterFlags	181
4.27.2.25 CUgraphicsResource	181
4.27.2.26 CUjit_fallback	181
4.27.2.27 CUjit_option	181
4.27.2.28 CUjit_target	181
4.27.2.29 CULimit	181
4.27.2.30 CUMemorytype	181
4.27.2.31 CUmodule	181
4.27.2.32 CUOutputMode	181
4.27.2.33 CUpointer_attribute	181
4.27.2.34 CUresult	181

4.27.2.35 <code>CUstream</code>	181
4.27.2.36 <code>CUsurfref</code>	182
4.27.2.37 <code>CUtexref</code>	182
4.27.3 Enumeration Type Documentation	182
4.27.3.1 <code>CUaddress_mode_enum</code>	182
4.27.3.2 <code>CUarray_cubemap_face_enum</code>	182
4.27.3.3 <code>CUarray_format_enum</code>	182
4.27.3.4 <code>CUcomputemode_enum</code>	183
4.27.3.5 <code>CUctx_flags_enum</code>	183
4.27.3.6 <code>cudaError_enum</code>	183
4.27.3.7 <code>CUdevice_attribute_enum</code>	185
4.27.3.8 <code>CUevent_flags_enum</code>	187
4.27.3.9 <code>CUfilter_mode_enum</code>	187
4.27.3.10 <code>CUfunc_cache_enum</code>	188
4.27.3.11 <code>CUfunction_attribute_enum</code>	188
4.27.3.12 <code>CUgraphicsMapResourceFlags_enum</code>	188
4.27.3.13 <code>CUgraphicsRegisterFlags_enum</code>	188
4.27.3.14 <code>CUjit_fallback_enum</code>	188
4.27.3.15 <code>CUjit_option_enum</code>	189
4.27.3.16 <code>CUjit_target_enum</code>	190
4.27.3.17 <code>CUlimit_enum</code>	190
4.27.3.18 <code>CUmemorytype_enum</code>	190
4.27.3.19 <code>CUOutputMode_st</code>	190
4.27.3.20 <code>CUpointer_attribute_enum</code>	190
4.28 Initialization	191
4.28.1 Detailed Description	191
4.28.2 Function Documentation	191
4.28.2.1 <code>cuInit</code>	191
4.29 Version Management	192
4.29.1 Detailed Description	192
4.29.2 Function Documentation	192
4.29.2.1 <code>cuDriverGetVersion</code>	192
4.30 Device Management	193
4.30.1 Detailed Description	193
4.30.2 Function Documentation	193
4.30.2.1 <code>cuDeviceComputeCapability</code>	193
4.30.2.2 <code>cuDeviceGet</code>	194

4.30.2.3 cuDeviceGetAttribute	194
4.30.2.4 cuDeviceGetCount	196
4.30.2.5 cuDeviceGetName	197
4.30.2.6 cuDeviceGetProperties	197
4.30.2.7 cuDeviceTotalMem	198
4.31 Context Management	199
4.31.1 Detailed Description	200
4.31.2 Function Documentation	200
4.31.2.1 cuCtxCreate	200
4.31.2.2 cuCtxDestroy	201
4.31.2.3 cuCtxGetApiVersion	202
4.31.2.4 cuCtxGetCacheConfig	202
4.31.2.5 cuCtxGetCurrent	203
4.31.2.6 cuCtxGetDevice	203
4.31.2.7 cuCtxGetLimit	203
4.31.2.8 cuCtxPopCurrent	204
4.31.2.9 cuCtxPushCurrent	204
4.31.2.10 cuCtxSetCacheConfig	205
4.31.2.11 cuCtxSetCurrent	206
4.31.2.12 cuCtxSetLimit	206
4.31.2.13 cuCtxSynchronize	207
4.32 Context Management [DEPRECATED]	208
4.32.1 Detailed Description	208
4.32.2 Function Documentation	208
4.32.2.1 cuCtxAttach	208
4.32.2.2 cuCtxDetach	209
4.33 Module Management	210
4.33.1 Detailed Description	210
4.33.2 Function Documentation	210
4.33.2.1 cuModuleGetFunction	210
4.33.2.2 cuModuleGetGlobal	211
4.33.2.3 cuModuleGetSurfRef	211
4.33.2.4 cuModuleGetTexRef	212
4.33.2.5 cuModuleLoad	212
4.33.2.6 cuModuleLoadData	213
4.33.2.7 cuModuleLoadDataEx	213
4.33.2.8 cuModuleLoadFatBinary	215

4.33.2.9 cuModuleUnload	215
4.34 Memory Management	217
4.34.1 Detailed Description	220
4.34.2 Function Documentation	220
4.34.2.1 cuArray3DCreate	220
4.34.2.2 cuArray3DGetDescriptor	222
4.34.2.3 cuArrayCreate	223
4.34.2.4 cuArrayDestroy	224
4.34.2.5 cuArrayGetDescriptor	225
4.34.2.6 cuMemAlloc	225
4.34.2.7 cuMemAllocHost	226
4.34.2.8 cuMemAllocPitch	227
4.34.2.9 cuMemcpy	227
4.34.2.10 cuMemcpy2D	228
4.34.2.11 cuMemcpy2DAsync	230
4.34.2.12 cuMemcpy2DUnaligned	233
4.34.2.13 cuMemcpy3D	235
4.34.2.14 cuMemcpy3DAsync	238
4.34.2.15 cuMemcpy3DPeer	240
4.34.2.16 cuMemcpy3DPeerAsync	241
4.34.2.17 cuMemcpyAsync	241
4.34.2.18 cuMemcpyAtoA	242
4.34.2.19 cuMemcpyAtoD	243
4.34.2.20 cuMemcpyAtoH	243
4.34.2.21 cuMemcpyAtoHAsync	244
4.34.2.22 cuMemcpyDtoA	245
4.34.2.23 cuMemcpyDtoD	245
4.34.2.24 cuMemcpyDtoDAsync	246
4.34.2.25 cuMemcpyDtoH	246
4.34.2.26 cuMemcpyDtoHAsync	247
4.34.2.27 cuMemcpyHtoA	248
4.34.2.28 cuMemcpyHtoAAAsync	248
4.34.2.29 cuMemcpyHtoD	249
4.34.2.30 cuMemcpyHtoDAsync	249
4.34.2.31 cuMemcpyPeer	250
4.34.2.32 cuMemcpyPeerAsync	251
4.34.2.33 cuMemFree	251

4.34.2.34 cuMemFreeHost	252
4.34.2.35 cuMemGetAddressRange	252
4.34.2.36 cuMemGetInfo	253
4.34.2.37 cuMemHostAlloc	253
4.34.2.38 cuMemHostGetDevicePointer	255
4.34.2.39 cuMemHostGetFlags	255
4.34.2.40 cuMemHostRegister	256
4.34.2.41 cuMemHostUnregister	256
4.34.2.42 cuMemsetD16	257
4.34.2.43 cuMemsetD16Async	257
4.34.2.44 cuMemsetD2D16	258
4.34.2.45 cuMemsetD2D16Async	259
4.34.2.46 cuMemsetD2D32	259
4.34.2.47 cuMemsetD2D32Async	260
4.34.2.48 cuMemsetD2D8	261
4.34.2.49 cuMemsetD2D8Async	261
4.34.2.50 cuMemsetD32	262
4.34.2.51 cuMemsetD32Async	263
4.34.2.52 cuMemsetD8	263
4.34.2.53 cuMemsetD8Async	264
4.34.2.54 cuProfilerInitialize	264
4.34.2.55 cuProfilerStart	265
4.34.2.56 cuProfilerStop	265
4.35 Unified Addressing	267
4.35.1 Detailed Description	267
4.35.2 Overview	267
4.35.3 Supported Platforms	267
4.35.4 Looking Up Information from Pointer Values	267
4.35.5 Automatic Mapping of Host Allocated Host Memory	267
4.35.6 Automatic Registration of Peer Memory	268
4.35.7 Exceptions, Disjoint Addressing	268
4.35.8 Function Documentation	268
4.35.8.1 cuPointerGetAttribute	268
4.36 Stream Management	271
4.36.1 Detailed Description	271
4.36.2 Function Documentation	271
4.36.2.1 cuStreamCreate	271

4.36.2.2 cuStreamDestroy	272
4.36.2.3 cuStreamQuery	272
4.36.2.4 cuStreamSynchronize	272
4.36.2.5 cuStreamWaitEvent	273
4.37 Event Management	274
4.37.1 Detailed Description	274
4.37.2 Function Documentation	274
4.37.2.1 cuEventCreate	274
4.37.2.2 cuEventDestroy	275
4.37.2.3 cuEventElapsedTime	275
4.37.2.4 cuEventQuery	276
4.37.2.5 cuEventRecord	276
4.37.2.6 cuEventSynchronize	277
4.38 Execution Control	278
4.38.1 Detailed Description	278
4.38.2 Function Documentation	278
4.38.2.1 cuFuncGetAttribute	278
4.38.2.2 cuFuncSetCacheConfig	279
4.38.2.3 cuLaunchKernel	280
4.39 Execution Control [DEPRECATED]	282
4.39.1 Detailed Description	282
4.39.2 Function Documentation	282
4.39.2.1 cuFuncSetBlockShape	282
4.39.2.2 cuFuncSetSharedSize	283
4.39.2.3 cuLaunch	283
4.39.2.4 cuLaunchGrid	284
4.39.2.5 cuLaunchGridAsync	285
4.39.2.6 cuParamSetf	285
4.39.2.7 cuParamSeti	286
4.39.2.8 cuParamSetSize	286
4.39.2.9 cuParamSetTexRef	287
4.39.2.10 cuParamSetv	287
4.40 Texture Reference Management	289
4.40.1 Detailed Description	290
4.40.2 Function Documentation	290
4.40.2.1 cuTexRefGetAddress	290
4.40.2.2 cuTexRefGetAddressMode	290

4.40.2.3 cuTexRefGetArray	291
4.40.2.4 cuTexRefGetFilterMode	291
4.40.2.5 cuTexRefGetFlags	291
4.40.2.6 cuTexRefGetFormat	292
4.40.2.7 cuTexRefSetAddress	292
4.40.2.8 cuTexRefSetAddress2D	293
4.40.2.9 cuTexRefSetAddressMode	293
4.40.2.10 cuTexRefSetArray	294
4.40.2.11 cuTexRefSetFilterMode	294
4.40.2.12 cuTexRefSetFlags	295
4.40.2.13 cuTexRefSetFormat	295
4.41 Texture Reference Management [DEPRECATED]	297
4.41.1 Detailed Description	297
4.41.2 Function Documentation	297
4.41.2.1 cuTexRefCreate	297
4.41.2.2 cuTexRefDestroy	297
4.42 Surface Reference Management	299
4.42.1 Detailed Description	299
4.42.2 Function Documentation	299
4.42.2.1 cuSurfRefGetArray	299
4.42.2.2 cuSurfRefSetArray	299
4.43 Peer Context Memory Access	301
4.43.1 Detailed Description	301
4.43.2 Function Documentation	301
4.43.2.1 cuCtxDisablePeerAccess	301
4.43.2.2 cuCtxEnablePeerAccess	302
4.43.2.3 cuDeviceCanAccessPeer	303
4.43.2.4 cuMemPeerGetDevicePointer	303
4.43.2.5 cuMemPeerRegister	304
4.43.2.6 cuMemPeerUnregister	305
4.44 Graphics Interoperability	306
4.44.1 Detailed Description	306
4.44.2 Function Documentation	306
4.44.2.1 cuGraphicsMapResources	306
4.44.2.2 cuGraphicsResourceGetMappedPointer	307
4.44.2.3 cuGraphicsResourceSetMapFlags	307
4.44.2.4 cuGraphicsSubResourceGetMappedArray	308

4.44.2.5 cuGraphicsUnmapResources	309
4.44.2.6 cuGraphicsUnregisterResource	309
4.45 OpenGL Interoperability	311
4.45.1 Detailed Description	311
4.45.2 Function Documentation	311
4.45.2.1 cuGLCtxCreate	311
4.45.2.2 cuGraphicsGLRegisterBuffer	312
4.45.2.3 cuGraphicsGLRegisterImage	312
4.45.2.4 cuWGLGetDevice	313
4.46 OpenGL Interoperability [DEPRECATED]	314
4.46.1 Detailed Description	314
4.46.2 Typedef Documentation	314
4.46.2.1 CUGLmap_flags	314
4.46.3 Enumeration Type Documentation	315
4.46.3.1 CUGLmap_flags_enum	315
4.46.4 Function Documentation	315
4.46.4.1 cuGLInit	315
4.46.4.2 cuGLMapBufferObject	315
4.46.4.3 cuGLMapBufferObjectAsync	316
4.46.4.4 cuGLRegisterBufferObject	316
4.46.4.5 cuGLSetBufferObjectMapFlags	317
4.46.4.6 cuGLUnmapBufferObject	318
4.46.4.7 cuGLUnmapBufferObjectAsync	318
4.46.4.8 cuGLUnregisterBufferObject	319
4.47 Direct3D 9 Interoperability	320
4.47.1 Detailed Description	320
4.47.2 Typedef Documentation	321
4.47.2.1 CUd3d9DeviceList	321
4.47.3 Enumeration Type Documentation	321
4.47.3.1 CUd3d9DeviceList_enum	321
4.47.4 Function Documentation	321
4.47.4.1 cuD3D9CtxCreate	321
4.47.4.2 cuD3D9CtxCreateOnDevice	322
4.47.4.3 cuD3D9GetDevice	322
4.47.4.4 cuD3D9GetDevices	323
4.47.4.5 cuD3D9GetDirect3DDevice	323
4.47.4.6 cuGraphicsD3D9RegisterResource	324

4.48 Direct3D 9 Interoperability [DEPRECATED]	326
4.48.1 Detailed Description	327
4.48.2 Typedef Documentation	327
4.48.2.1 CUd3d9map_flags	327
4.48.2.2 CUd3d9register_flags	327
4.48.3 Enumeration Type Documentation	327
4.48.3.1 CUd3d9map_flags_enum	327
4.48.3.2 CUd3d9register_flags_enum	327
4.48.4 Function Documentation	327
4.48.4.1 cuD3D9MapResources	327
4.48.4.2 cuD3D9RegisterResource	328
4.48.4.3 cuD3D9ResourceGetMappedArray	329
4.48.4.4 cuD3D9ResourceGetMappedPitch	330
4.48.4.5 cuD3D9ResourceGetMappedPointer	331
4.48.4.6 cuD3D9ResourceGetMappedSize	331
4.48.4.7 cuD3D9ResourceGetSurfaceDimensions	332
4.48.4.8 cuD3D9ResourceSetMapFlags	333
4.48.4.9 cuD3D9UnmapResources	334
4.48.4.10 cuD3D9UnregisterResource	334
4.49 Direct3D 10 Interoperability	335
4.49.1 Detailed Description	335
4.49.2 Typedef Documentation	336
4.49.2.1 CUd3d10DeviceList	336
4.49.3 Enumeration Type Documentation	336
4.49.3.1 CUd3d10DeviceList_enum	336
4.49.4 Function Documentation	336
4.49.4.1 cuD3D10CtxCreate	336
4.49.4.2 cuD3D10CtxCreateOnDevice	337
4.49.4.3 cuD3D10GetDevice	337
4.49.4.4 cuD3D10GetDevices	338
4.49.4.5 cuD3D10GetDirect3DDevice	338
4.49.4.6 cuGraphicsD3D10RegisterResource	339
4.50 Direct3D 10 Interoperability [DEPRECATED]	341
4.50.1 Detailed Description	342
4.50.2 Typedef Documentation	342
4.50.2.1 CUD3D10map_flags	342
4.50.2.2 CUD3D10register_flags	342

4.50.3 Enumeration Type Documentation	342
4.50.3.1 CUD3D10map_flags_enum	342
4.50.3.2 CUD3D10register_flags_enum	342
4.50.4 Function Documentation	342
4.50.4.1 cuD3D10MapResources	342
4.50.4.2 cuD3D10RegisterResource	343
4.50.4.3 cuD3D10ResourceGetMappedArray	344
4.50.4.4 cuD3D10ResourceGetMappedPitch	345
4.50.4.5 cuD3D10ResourceGetMappedPointer	346
4.50.4.6 cuD3D10ResourceGetMappedSize	346
4.50.4.7 cuD3D10ResourceGetSurfaceDimensions	347
4.50.4.8 cuD3D10ResourceSetMapFlags	348
4.50.4.9 cuD3D10UnmapResources	348
4.50.4.10 cuD3D10UnregisterResource	349
4.51 Direct3D 11 Interoperability	350
4.51.1 Detailed Description	350
4.51.2 Typedef Documentation	350
4.51.2.1 CUd3d11DeviceList	350
4.51.3 Enumeration Type Documentation	351
4.51.3.1 CUd3d11DeviceList_enum	351
4.51.4 Function Documentation	351
4.51.4.1 cuD3D11CtxCreate	351
4.51.4.2 cuD3D11CtxCreateOnDevice	351
4.51.4.3 cuD3D11GetDevice	352
4.51.4.4 cuD3D11GetDevices	353
4.51.4.5 cuD3D11GetDirect3DDevice	353
4.51.4.6 cuGraphicsD3D11RegisterResource	354
4.52 VDPAU Interoperability	356
4.52.1 Detailed Description	356
4.52.2 Function Documentation	356
4.52.2.1 cuGraphicsVDPAURegisterOutputSurface	356
4.52.2.2 cuGraphicsVDPAURegisterVideoSurface	357
4.52.2.3 cuVDPAUCtxCreate	358
4.52.2.4 cuVDPAUGetDevice	358
5 Data Structure Documentation	361
5.1 CUDA_ARRAY3D_DESCRIPTOR_st Struct Reference	361

5.1.1	Detailed Description	361
5.1.2	Field Documentation	361
5.1.2.1	Depth	361
5.1.2.2	Flags	361
5.1.2.3	Format	361
5.1.2.4	Height	361
5.1.2.5	NumChannels	362
5.1.2.6	Width	362
5.2	CUDA_ARRAY_DESCRIPTOR_st Struct Reference	363
5.2.1	Detailed Description	363
5.2.2	Field Documentation	363
5.2.2.1	Format	363
5.2.2.2	Height	363
5.2.2.3	NumChannels	363
5.2.2.4	Width	363
5.3	CUDA_MEMCPY2D_st Struct Reference	364
5.3.1	Detailed Description	364
5.3.2	Field Documentation	364
5.3.2.1	dstArray	364
5.3.2.2	dstDevice	364
5.3.2.3	dstHost	364
5.3.2.4	dstMemoryType	364
5.3.2.5	dstPitch	364
5.3.2.6	dstXInBytes	365
5.3.2.7	dstY	365
5.3.2.8	Height	365
5.3.2.9	srcArray	365
5.3.2.10	srcDevice	365
5.3.2.11	srcHost	365
5.3.2.12	srcMemoryType	365
5.3.2.13	srcPitch	365
5.3.2.14	srcXInBytes	365
5.3.2.15	srcY	365
5.3.2.16	WidthInBytes	365
5.4	CUDA_MEMCPY3D_PEER_st Struct Reference	366
5.4.1	Detailed Description	366
5.4.2	Field Documentation	366

5.4.2.1	Depth	366
5.4.2.2	dstArray	366
5.4.2.3	dstContext	366
5.4.2.4	dstDevice	367
5.4.2.5	dstHeight	367
5.4.2.6	dstHost	367
5.4.2.7	dstLOD	367
5.4.2.8	dstMemoryType	367
5.4.2.9	dstPitch	367
5.4.2.10	dstXInBytes	367
5.4.2.11	dstY	367
5.4.2.12	dstZ	367
5.4.2.13	Height	367
5.4.2.14	srcArray	367
5.4.2.15	srcContext	367
5.4.2.16	srcDevice	368
5.4.2.17	srcHeight	368
5.4.2.18	srcHost	368
5.4.2.19	srcLOD	368
5.4.2.20	srcMemoryType	368
5.4.2.21	srcPitch	368
5.4.2.22	srcXInBytes	368
5.4.2.23	srcY	368
5.4.2.24	srcZ	368
5.4.2.25	WidthInBytes	368
5.5	CUDA_MEMCPY3D_st Struct Reference	369
5.5.1	Detailed Description	369
5.5.2	Field Documentation	369
5.5.2.1	Depth	369
5.5.2.2	dstArray	369
5.5.2.3	dstDevice	369
5.5.2.4	dstHeight	370
5.5.2.5	dstHost	370
5.5.2.6	dstLOD	370
5.5.2.7	dstMemoryType	370
5.5.2.8	dstPitch	370
5.5.2.9	dstXInBytes	370

5.5.2.10	dstY	370
5.5.2.11	dstZ	370
5.5.2.12	Height	370
5.5.2.13	reserved0	370
5.5.2.14	reserved1	370
5.5.2.15	srcArray	370
5.5.2.16	srcDevice	371
5.5.2.17	srcHeight	371
5.5.2.18	srcHost	371
5.5.2.19	srcLOD	371
5.5.2.20	srcMemoryType	371
5.5.2.21	srcPitch	371
5.5.2.22	srcXInBytes	371
5.5.2.23	srcY	371
5.5.2.24	srcZ	371
5.5.2.25	WidthInBytes	371
5.6	cudaChannelFormatDesc Struct Reference	372
5.6.1	Detailed Description	372
5.6.2	Field Documentation	372
5.6.2.1	f	372
5.6.2.2	w	372
5.6.2.3	x	372
5.6.2.4	y	372
5.6.2.5	z	372
5.7	cudaDeviceProp Struct Reference	373
5.7.1	Detailed Description	373
5.7.2	Field Documentation	373
5.7.2.1	asyncEngineCount	373
5.7.2.2	canMapHostMemory	374
5.7.2.3	clockRate	374
5.7.2.4	computeMode	374
5.7.2.5	concurrentKernels	374
5.7.2.6	deviceOverlap	374
5.7.2.7	ECCEnabled	374
5.7.2.8	integrated	374
5.7.2.9	kernelExecTimeoutEnabled	374
5.7.2.10	major	374

5.7.2.11	maxGridSize	374
5.7.2.12	maxTexture1D	374
5.7.2.13	maxTexture1DLayered	374
5.7.2.14	maxTexture2D	375
5.7.2.15	maxTexture2DLayered	375
5.7.2.16	maxTexture3D	375
5.7.2.17	maxThreadsDim	375
5.7.2.18	maxThreadsPerBlock	375
5.7.2.19	memPitch	375
5.7.2.20	minor	375
5.7.2.21	multiProcessorCount	375
5.7.2.22	name	375
5.7.2.23	pciBusID	375
5.7.2.24	pciDeviceID	375
5.7.2.25	regsPerBlock	375
5.7.2.26	sharedMemPerBlock	376
5.7.2.27	surfaceAlignment	376
5.7.2.28	tccDriver	376
5.7.2.29	textureAlignment	376
5.7.2.30	totalConstMem	376
5.7.2.31	totalGlobalMem	376
5.7.2.32	unifiedAddressing	376
5.7.2.33	warpSize	376
5.8	cudaExtent Struct Reference	377
5.8.1	Detailed Description	377
5.8.2	Field Documentation	377
5.8.2.1	depth	377
5.8.2.2	height	377
5.8.2.3	width	377
5.9	cudaFuncAttributes Struct Reference	378
5.9.1	Detailed Description	378
5.9.2	Field Documentation	378
5.9.2.1	binaryVersion	378
5.9.2.2	constSizeBytes	378
5.9.2.3	localSizeBytes	378
5.9.2.4	maxThreadsPerBlock	378
5.9.2.5	numRegs	378

5.9.2.6	ptxVersion	378
5.9.2.7	sharedSizeBytes	379
5.10	cudaMemcpy3DParms Struct Reference	380
5.10.1	Detailed Description	380
5.10.2	Field Documentation	380
5.10.2.1	dstArray	380
5.10.2.2	dstPos	380
5.10.2.3	dstPtr	380
5.10.2.4	extent	380
5.10.2.5	kind	380
5.10.2.6	srcArray	380
5.10.2.7	srcPos	380
5.10.2.8	srcPtr	381
5.11	cudaMemcpy3DPeerParms Struct Reference	382
5.11.1	Detailed Description	382
5.11.2	Field Documentation	382
5.11.2.1	dstArray	382
5.11.2.2	dstDevice	382
5.11.2.3	dstPos	382
5.11.2.4	dstPtr	382
5.11.2.5	extent	382
5.11.2.6	srcArray	382
5.11.2.7	srcDevice	382
5.11.2.8	srcPos	383
5.11.2.9	srcPtr	383
5.12	cudaPitchedPtr Struct Reference	384
5.12.1	Detailed Description	384
5.12.2	Field Documentation	384
5.12.2.1	pitch	384
5.12.2.2	ptr	384
5.12.2.3	xsize	384
5.12.2.4	ysize	384
5.13	cudaPointerAttributes Struct Reference	385
5.13.1	Detailed Description	385
5.13.2	Field Documentation	385
5.13.2.1	device	385
5.13.2.2	devicePointer	385

5.13.2.3 hostPointer	385
5.13.2.4 memoryType	385
5.14 cudaPos Struct Reference	386
5.14.1 Detailed Description	386
5.14.2 Field Documentation	386
5.14.2.1 x	386
5.14.2.2 y	386
5.14.2.3 z	386
5.15 CUdevprop_st Struct Reference	387
5.15.1 Detailed Description	387
5.15.2 Field Documentation	387
5.15.2.1 clockRate	387
5.15.2.2 maxGridSize	387
5.15.2.3 maxThreadsDim	387
5.15.2.4 maxThreadsPerBlock	387
5.15.2.5 memPitch	387
5.15.2.6 regsPerBlock	387
5.15.2.7 sharedMemPerBlock	387
5.15.2.8 SIMDWidth	388
5.15.2.9 textureAlign	388
5.15.2.10 totalConstantMemory	388
5.16 surfaceReference Struct Reference	389
5.16.1 Detailed Description	389
5.16.2 Field Documentation	389
5.16.2.1 channelDesc	389
5.17 textureReference Struct Reference	390
5.17.1 Detailed Description	390
5.17.2 Field Documentation	390
5.17.2.1 addressMode	390
5.17.2.2 channelDesc	390
5.17.2.3 filterMode	390
5.17.2.4 normalized	390
5.17.2.5 sRGB	390

Chapter 1

Deprecated List

Global `cudaSetDeviceFlags` This flag was deprecated as of CUDA 4.0 and replaced with `cudaDeviceScheduleBlockingSync`.

- `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.

Global `cudaThreadExit`

Global `cudaThreadGetCacheConfig`

Global `cudaThreadGetLimit`

Global `cudaThreadSetCacheConfig`

Global `cudaThreadSetLimit`

Global `cudaThreadSynchronize`

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 `cudaDeviceBlockingSync`

Global `CU_CTX_BLOCKING_SYNC`

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

Global `cuCtxAttach`

Global `cuCtxDetach`

Global `cuFuncSetBlockShape`

Global `cuFuncSetSharedSize`

Global `cuLaunch`

Global `cuLaunchGrid`

Global `cuLaunchGridAsync`

Global `cuParamSetf`

Global `cuParamSeti`

Global `cuParamSetSize`

Global `cuParamSetTexRef`

Global `cuParamSetv`

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	11
Device Management	13
Thread Management [DEPRECATED]	23
Error Handling	28
Stream Management	30
Event Management	33
Execution Control	37
Memory Management	41
Unified Addressing	77
Peer Device Memory Access	80
OpenGL Interoperability	85
OpenGL Interoperability [DEPRECATED]	151
Direct3D 9 Interoperability	88
Direct3D 9 Interoperability [DEPRECATED]	133
Direct3D 10 Interoperability	93
Direct3D 10 Interoperability [DEPRECATED]	142
Direct3D 11 Interoperability	98
VDPAU Interoperability	103
Graphics Interoperability	106
Texture Reference Management	110
Surface Reference Management	115
Version Management	117
C++ API Routines	118
Interactions with the CUDA Driver API	131
Data types used by CUDA Runtime	156
CUDA Driver API	170
Data types used by CUDA driver	171
Initialization	191
Version Management	192
Device Management	193
Context Management	199
Context Management [DEPRECATED]	208
Module Management	210

Memory Management	217
Unified Addressing	267
Stream Management	271
Event Management	274
Execution Control	278
Execution Control [DEPRECATED]	282
Texture Reference Management	289
Texture Reference Management [DEPRECATED]	297
Surface Reference Management	299
Peer Context Memory Access	301
Graphics Interoperability	306
OpenGL Interoperability	311
OpenGL Interoperability [DEPRECATED]	314
Direct3D 9 Interoperability	320
Direct3D 9 Interoperability [DEPRECATED]	326
Direct3D 10 Interoperability	335
Direct3D 10 Interoperability [DEPRECATED]	341
Direct3D 11 Interoperability	350
VDPAU Interoperability	356

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

CUDA_ARRAY3D_DESCRIPTOR_st	361
CUDA_ARRAY_DESCRIPTOR_st	363
CUDA_MEMCPY2D_st	364
CUDA_MEMCPY3D_PEER_st	366
CUDA_MEMCPY3D_st	369
cudaChannelFormatDesc	372
cudaDeviceProp	373
cudaExtent	377
cudaFuncAttributes	378
cudaMemcpy3DParms	380
cudaMemcpy3DPeerParms	382
cudaPitchedPtr	384
cudaPointerAttributes	385
cudaPos	386
CUdevprop_st	387
surfaceReference	389
textureReference	390

Chapter 4

Module Documentation

4.1 CUDA Runtime API

Modules

- Device Management
- Error Handling
- Stream Management
- Event Management
- Execution Control
- Memory Management
- Unified Addressing
- Peer Device Memory Access
- 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 4000

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 4000

CUDA Runtime API Version 4.0

4.2 Device Management

Modules

- Thread Management [DEPRECATED]

Functions

- `cudaError_t cudaChooseDevice (int *device, const struct cudaDeviceProp *prop)`
Select compute-device which best matches criteria.
- `cudaError_t cudaDeviceGetCacheConfig (enum cudaFuncCache *pCacheConfig)`
Returns the preferred cache configuration for the current device.
- `cudaError_t cudaDeviceGetLimit (size_t *pValue, enum cudaLimit limit)`
Returns resource limits.
- `cudaError_t cudaDeviceReset (void)`
Destroy all allocations and reset all state on the current device in the current process.
- `cudaError_t cudaDeviceSetCacheConfig (enum cudaFuncCache cacheConfig)`
Sets the preferred cache configuration for the current device.
- `cudaError_t cudaDeviceSetLimit (enum cudaLimit limit, size_t value)`
Set resource limits.
- `cudaError_t cudaDeviceSynchronize (void)`
Wait for compute device to finish.
- `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.

4.2.1 Detailed Description

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

4.2.2 Function Documentation

4.2.2.1 `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](#)

4.2.2.2 `cudaError_t cudaDeviceGetCacheConfig (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 device. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute functions.

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

The supported cache configurations are:

- `cudaFuncCacheNone`: no preference for shared memory or L1 (default)
- `cudaFuncCacheShared`: prefer larger shared memory and smaller L1 cache
- `cudaFuncCacheL1`: 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:

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

4.2.2.3 `cudaError_t cudaDeviceGetLimit (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:

[cudaDeviceSetLimit](#)

4.2.2.4 `cudaError_t cudaDeviceReset (void)`

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

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

Returns:

`cudaSuccess`

Note:

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

See also:

[cudaDeviceSynchronize](#)

4.2.2.5 `cudaError_t cudaDeviceSetCacheConfig (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 device. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute the function. Any function preference set via [cudaFuncSetCacheConfig \(C API\)](#) or [cudaFuncSetCacheConfig \(C++ API\)](#) will be preferred over this device-wide setting. Setting the device-wide cache configuration to `cudaFuncCacheNone` 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:

- `cudaFuncCacheNone`: no preference for shared memory or L1 (default)
- `cudaFuncCacheShared`: prefer larger shared memory and smaller L1 cache
- `cudaFuncCacheL1`: 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:

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

4.2.2.6 `cudaError_t cudaDeviceSetLimit (enum cudaLimit limit, size_t value)`

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

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

- `cudaLimitStackSize` controls the stack size 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:

[cudaDeviceGetLimit](#)

4.2.2.7 `cudaError_t cudaDeviceSynchronize (void)`

Blocks until the device has completed all preceding requested tasks. [cudaDeviceSynchronize\(\)](#) returns an error if one of the preceding tasks has failed. If the [cudaDeviceScheduleBlockingSync](#) flag was set for this device, the host thread will block until the device has finished its work.

Returns:

[cudaSuccess](#)

Note:

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

See also:

[cudaDeviceSynchronize](#)

4.2.2.8 `cudaError_t cudaGetDevice (int * device)`

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

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](#)

4.2.2.9 `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](#)

4.2.2.10 `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];
    int clockRate;
    size_t totalConstMem;
    int major;
    int minor;
    size_t textureAlignment;
    int deviceOverlap;
    int multiProcessorCount;
    int kernelExecTimeoutEnabled;
    int integrated;
    int canMapHostMemory;
    int computeMode;
    int maxTextureID;
    int maxTexture2D[2];
    int maxTexture3D[3];
    int maxTexture1DLayered[2];
    int maxTexture2DLayered[3];
    size_t surfaceAlignment;
    int concurrentKernels;
    int ECCEnabled;
    int pciBusID;
    int pciDeviceID;
    int tccDriver;
    int asyncEngineCount;
    int unifiedAddressing;
}
```

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. Deprecated, use instead `asyncEngineCount`.
- `multiProcessorCount` is the number of multiprocessors on the device;
- `kernelExecTimeoutEnabled` is 1 if there is a run time limit for kernels executed on the device, or 0 if not.
- `integrated` is 1 if the device is an integrated (motherboard) GPU and 0 if it is a discrete (card) component.
- `canMapHostMemory` is 1 if the device can map host memory into the CUDA address space for use with `cudaHostAlloc()`/`cudaHostGetDevicePointer()`, or 0 if not;
- `computeMode` is the compute mode that the device is currently in. Available modes are as follows:
 - `cudaComputeModeDefault`: Default mode - Device is not restricted and multiple threads can use `cudaSetDevice()` with this device.
 - `cudaComputeModeExclusive`: Compute-exclusive mode - Only one thread will be able to use `cudaSetDevice()` with this device.
 - `cudaComputeModeProhibited`: Compute-prohibited mode - No threads can use `cudaSetDevice()` with this device. 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.
- `maxTexture1D` is the maximum 1D texture size.
- `maxTexture2D[2]` contains the maximum 2D texture dimensions.
- `maxTexture3D[3]` contains the maximum 3D texture dimensions.
- `maxTexture1DLayered[2]` contains the maximum 1D layered texture dimensions.

- `maxTexture2DLayered[3]` contains the maximum 2D layered texture dimensions.
- `surfaceAlignment` specifies the alignment requirements for surfaces.
- `concurrentKernels` is 1 if the device supports executing multiple kernels within the same context simultaneously, or 0 if not. It is not guaranteed that multiple kernels will be resident on the device concurrently so this feature should not be relied upon for correctness;
- `ECCEnabled` is 1 if the device has ECC support turned on, or 0 if not.
- `pciBusID` is the PCI bus identifier of the device.
- `pciDeviceID` is the PCI device (sometimes called slot) identifier of the device.
- `tccDriver` is 1 if the device is using a TCC driver or 0 if not.
- `asyncEngineCount` is 1 when the device can concurrently copy memory between host and device while executing a kernel. It is 2 when the device can concurrently copy memory between host and device in both directions and execute a kernel at the same time. It is 0 if neither of these is supported.
- `unifiedAddressing` is 1 if the device shares a unified address space with the host and 0 otherwise.

Parameters:

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

Returns:

`cudaSuccess`, `cudaErrorInvalidDevice`

See also:

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

4.2.2.11 `cudaError_t cudaSetDevice (int device)`

Sets `device` as the current device for the calling host thread.

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

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

Parameters:

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

Returns:

`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorSetOnActiveProcess`

Note:

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

See also:

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

4.2.2.12 `cudaError_t cudaSetDeviceFlags (unsigned int flags)`

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

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

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

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

Deprecated

This flag was deprecated as of CUDA 4.0 and replaced with `cudaDeviceScheduleBlockingSync`.

Parameters:

`flags` - Parameters for device operation

Returns:

`cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorSetOnActiveProcess`

See also:

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

4.2.2.13 `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](#)

4.3 Thread Management [DEPRECATED]

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 device.
- `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 device.
- `cudaError_t cudaThreadSetLimit (enum cudaLimit limit, size_t value)`
Set resource limits.
- `cudaError_t cudaThreadSynchronize (void)`
Wait for compute device to finish.

4.3.1 Detailed Description

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

4.3.2 Function Documentation

4.3.2.1 `cudaError_t cudaThreadExit (void)`

Deprecated

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

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

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

Returns:

`cudaSuccess`

Note:

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

See also:[cudaDeviceReset](#)**4.3.2.2 cudaError_t cudaThreadGetCacheConfig (enum cudaFuncCache * *pCacheConfig*)****Deprecated**

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

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

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

The supported cache configurations are:

- [cudaFuncCacheNone](#): no preference for shared memory or L1 (default)
- [cudaFuncCacheShared](#): prefer larger shared memory and smaller L1 cache
- [cudaFuncCacheL1](#): 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:[cudaDeviceGetCacheConfig](#)**4.3.2.3 cudaError_t cudaThreadGetLimit (size_t * *pValue*, enum cudaLimit *limit*)****Deprecated**

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

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

- [cudaLimitStackSize](#): stack size of each GPU thread;

- `cudaLimitPrintfFifoSize`: size of the 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:

`cudaDeviceGetLimit`

4.3.2.4 `cudaError_t cudaThreadSetCacheConfig (enum cudaFuncCache cacheConfig)`

Deprecated

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

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

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

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

The supported cache configurations are:

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

Parameters:

cacheConfig - Requested cache configuration

Returns:

`cudaSuccess`, `cudaErrorInitializationError`

Note:

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

See also:

[cudaDeviceSetCacheConfig](#)

4.3.2.5 cudaError_t cudaThreadSetLimit (enum cudaLimit *limit*, size_t *value*)**Deprecated**

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

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

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

- [cudaLimitStackSize](#) controls the stack size of each GPU thread. 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:

[cudaDeviceSetLimit](#)

4.3.2.6 `cudaError_t cudaThreadSynchronize (void)`

Deprecated

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

Blocks until the device has completed all preceding requested tasks. [cudaThreadSynchronize\(\)](#) returns an error if one of the preceding tasks has failed. If the [cudaDeviceScheduleBlockingSync](#) flag was set for this device, the host thread will block until the device has finished its work.

Returns:

[cudaSuccess](#)

Note:

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

See also:

[cudaDeviceSynchronize](#)

4.4 Error Handling

Functions

- `const char * cudaGetString (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.

4.4.1 Detailed Description

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

4.4.2 Function Documentation

4.4.2.1 `const char* cudaGetString (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](#)

4.4.2.2 `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](#)

4.4.2.3 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](#)

4.5 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`.

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

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 [cudaDeviceScheduleBlockingSync](#) 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 event `event` may be from a different context than `stream`, in which case this function will perform cross-device synchronization.

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

If `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*.

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

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`, `cudaErrorLaunchFailure`

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`, `cudaErrorLaunchFailure`

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:

- `cudaFuncCacheNone`: 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\(\)](#) or registered by [cudaHostRegister\(\)](#).
- **cudaError_t cudaHostGetFlags** (unsigned int *pFlags, void *pHost)
Passes back flags used to allocate pinned host memory allocated by [cudaHostAlloc\(\)](#).
- **cudaError_t cudaHostRegister** (void *ptr, size_t size, unsigned int flags)
Registers an existing host memory range for use by CUDA.
- **cudaError_t cudaHostUnregister** (void *ptr)
Unregisters a memory range that was registered with [cuMemHostRegister\(\)](#).
- **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 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 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 cudaMemcpy3DPeer` (const struct `cudaMemcpy3DPeerParms` *p)
Copies memory between devices.
- `cudaError_t cudaMemcpy3DPeerAsync` (const struct `cudaMemcpy3DPeerParms` *p, `cudaStream_t` stream=0)
Copies memory between devices asynchronously.
- `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 cudaMemcpyPeer` (void *dst, int dstDevice, const void *src, int srcDevice, size_t count)

Copies memory between two devices.

- `cudaError_t cudaMemcpyPeerAsync` (void *dst, int dstDevice, const void *src, int srcDevice, size_t count, `cudaStream_t` stream=0)

Copies memory between two devices asynchronously.

- `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](#), [cudaAlloc3D](#), [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()` or registered by `cudaHostRegister()`.

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

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

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

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

- [cudaHostRegisterPortable](#): The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.

- [cudaHostRegisterMapped](#): Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling [cuMemHostGetDevicePointer\(\)](#). This feature is available only on GPUs with compute capability greater than or equal to 1.1.

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

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

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

The pointer `ptr` and size `size` must be aligned to the host page size (4 KB).

The memory page-locked by this function must be unregistered with [cudaHostUnregister\(\)](#).

Parameters:

`ptr` - Host pointer to memory to page-lock
`size` - Size in bytes of the address range to page-lock in bytes
`flags` - Flags for allocation request

Returns:

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

Note:

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

See also:

[cudaHostUnregister](#), [cudaHostGetFlags](#), [cudaHostGetDevicePointer](#)

4.8.2.10 `cudaError_t cudaHostUnregister (void *ptr)`

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

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

Parameters:

`ptr` - Host pointer to memory to unregister

Returns:

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

Note:

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

See also:

[cudaHostUnregister](#)

4.8.2.11 `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`

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

4.8.2.13 `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()` can allocate the following:

- A 1D array is allocated if the height and depth extent are both zero. For 1D arrays, valid extent ranges are $\{(1, \text{maxTexture1D}), 0, 0\}$.
- A 2D array is allocated if only the depth extent is zero. For 2D arrays, valid extent ranges are $\{(1, \text{maxTexture2D}[0]), (1, \text{maxTexture2D}[1]), 0\}$.
- A 3D array is allocated if all three extents are non-zero. For 3D arrays, valid extent ranges are $\{(1, \text{maxTexture3D}[0]), (1, \text{maxTexture3D}[1]), (1, \text{maxTexture3D}[2])\}$.
- A 1D layered texture is allocated if only the height extent is zero and the `cudaArrayLayered` flag is set. The number of layers is determined by the depth extent. For 1D layered textures, valid extent ranges are $\{(1, \text{maxTexture1DLayered}[0]), 0, (1, \text{maxTexture1DLayered}[1])\}$.
- A 2D layered texture is allocated if all three extents are non-zero and the `cudaArrayLayered` flag is set. The number of layers is determined by the depth extent. For 1D layered textures, valid extent ranges are $\{(1, \text{maxTexture2DLayered}[0]), (1, \text{maxTexture2DLayered}[1]), (1, \text{maxTexture2DLayered}[2])\}$.

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.

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

- `cudaArrayDefault`: This flag's value is defined to be 0 and provides default array allocation
- `cudaArrayLayered`: Allocates a layered texture, with the depth extent indicating the number of layers

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](#)

4.8.2.14 `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](#)

4.8.2.15 `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`

4.8.2.16 `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](#)

4.8.2.17 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](#)

4.8.2.18 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](#)

4.8.2.19 `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](#)

4.8.2.20 `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](#)

4.8.2.21 `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 `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. `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`, `cudaMemcpyArrayToArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`

4.8.2.22 `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 `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. `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`

4.8.2.23 `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`, `cudaMemcpyArrayToArray`, `cudaMemcpy2DArrayToArray`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpy2DAsync`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`

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

4.8.2.25 cudaError_t 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`

4.8.2.26 `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](#), [cudaMemcpyArrayToArray](#), [cudaMemcpy2DArrayToArray](#), [cudaMemcpyToSymbol](#), [cudaMemcpyFromSymbol](#), [cudaMemcpyAsync](#), [cudaMemcpy2DAsync](#), [cudaMemcpyToArrayAsync](#), [cudaMemcpy2DToArrayAsync](#), [cudaMemcpyFromArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#), [make_cudaExtent](#), [make_cudaPos](#)

4.8.2.27 cudaError_t cudaMemcpy3DPeer (const struct cudaMemcpy3DPeerParms **p*)

Perform a 3D memory copy according to the parameters specified in *p*. See the definition of the [cudaMemcpy3DPeerParms](#) structure for documentation of its parameters.

Note that this function is synchronous with respect to the host only if the source or destination of the transfer is host memory. Note also that this copy is serialized with respect to all pending and future asynchronous work in to the current device, the copy's source device, and the copy's destination device (use [cudaMemcpy3DPeerAsync](#) to avoid this synchronization).

Parameters:

p - Parameters for the memory copy

Returns:

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

Note:

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

See also:

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

4.8.2.28 cudaError_t cudaMemcpy3DPeerAsync (const struct cudaMemcpy3DPeerParms **p*, cudaStream_t *stream* = 0)

Perform a 3D memory copy according to the parameters specified in *p*. See the definition of the [cudaMemcpy3DPeerParms](#) structure for documentation of its parameters.

Parameters:

p - Parameters for the memory copy

stream - Stream identifier

Returns:

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

Note:

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

See also:

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

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

4.8.2.30 `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](#), [cudaMemcpy2DToAsync](#), [cudaMemcpyFromArrayAsync](#), [cudaMemcpy2DFromArrayAsync](#), [cudaMemcpyToSymbolAsync](#), [cudaMemcpyFromSymbolAsync](#)

4.8.2.31 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](#)

4.8.2.32 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`

4.8.2.33 `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](#)

4.8.2.34 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](#)

4.8.2.35 cudaError_t cudaMemcpyPeer (void * *dst*, int *dstDevice*, const void * *src*, int *srcDevice*, size_t *count*)

Copies memory from one device to memory on another device. *dst* is the base device pointer of the destination memory and *dstDevice* is the destination device. *src* is the base device pointer of the source memory and *srcDevice* is the source device. *count* specifies the number of bytes to copy.

Note that this function is asynchronous with respect to the host, but serialized with respect all pending and future asynchronous work in to the current device, `srcDevice`, and `dstDevice` (use `cudaMemcpyPeerAsync` to avoid this synchronization).

Parameters:

- `dst` - Destination device pointer
- `dstDevice` - Destination device
- `src` - Source device pointer
- `srcDevice` - Source device
- `count` - Size of memory copy in bytes

Returns:

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`

Note:

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

See also:

`cudaMemcpy`, `cudaMemcpyPeer3D`, `cudaMemcpyAsync`, `cudaMemcpyPeerAsync`, `cudaMemcpy3DPeerAsync`

4.8.2.36 `cudaError_t cudaMemcpyPeerAsync (void * dst, int dstDevice, const void * src, int srcDevice, size_t count, cudaStream_t stream = 0)`

Copies memory from one device to memory on another device. `dst` is the base device pointer of the destination memory and `dstDevice` is the destination device. `src` is the base device pointer of the source memory and `srcDevice` is the source device. `count` specifies the number of bytes to copy.

Note that this function is asynchronous with respect to the host and all work in other streams and other devices.

Parameters:

- `dst` - Destination device pointer
- `dstDevice` - Destination device
- `src` - Source device pointer
- `srcDevice` - Source device
- `count` - Size of memory copy in bytes
- `stream` - Stream identifier

Returns:

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`

Note:

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

See also:

`cudaMemcpy`, `cudaMemcpyPeer`, `cudaMemcpyPeer3D`, `cudaMemcpyAsync`, `cudaMemcpy3DPeerAsync`

4.8.2.37 `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`

4.8.2.38 `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](#)

4.8.2.39 `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](#)

4.8.2.40 `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`

4.8.2.41 `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.

4.8.2.42 **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](#)

4.8.2.43 **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](#)

4.8.2.44 `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](#)

4.8.2.45 `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`. Secondarily, 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`

4.8.2.46 `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*. Secondarily, 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`

4.8.2.47 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`

4.8.2.48 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`

4.8.2.49 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](#)

4.8.2.50 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 Unified Addressing

Functions

- `cudaError_t cudaPointerGetAttributes` (struct `cudaPointerAttributes` *`attributes`, void *`ptr`)

Returns attributes about a specified pointer.

4.9.1 Detailed Description

This section describes the unified addressing functions of the CUDA runtime application programming interface.

4.9.2 Overview

CUDA devices can share a unified address space with the host. For these devices there is no distinction between a device pointer and a host pointer – the same pointer value may be used to access memory from the host program and from a kernel running on the device (with exceptions enumerated below).

4.9.3 Supported Platforms

Whether or not a device supports unified addressing may be queried by calling `cudaGetDeviceProperties()` with the device property `cudaDeviceProp::unifiedAddressing`.

Unified addressing is automatically enabled in 64-bit processes on devices with compute capability greater than or equal to 2.0.

Unified addressing is not yet supported on Windows Vista or Windows 7 for devices that do not use the TCC driver model.

4.9.4 Looking Up Information from Pointer Values

It is possible to look up information about the memory which backs a pointer value. For instance, one may want to know if a pointer points to host or device memory. As another example, in the case of device memory, one may want to know on which CUDA device the memory resides. These properties may be queried using the function `cudaPointerGetAttributes()`

Because pointers are unique, it is not necessary to specify information about the pointers specified to `cudaMemcpy()` and other copy functions. The copy direction `cudaMemcpyDefault` may be used to specify that the CUDA runtime should infer the location of the pointer from its value.

4.9.5 Automatic Mapping of Host Allocated Host Memory

All host memory allocated through all devices using `cudaMallocHost()` and `cudaHostAlloc()` is always directly accessible from all devices that support unified addressing. This is the case regardless of whether or not the flags `cudaHostAllocPortable` and `cudaHostAllocMapped` are specified.

The pointer value through which allocated host memory may be accessed in kernels on all devices that support unified addressing is the same as the pointer value through which that memory is accessed on the host. It is not necessary to call `cuMemHostGetDevicePointer()` to get the device pointer for these allocations.

Note that this is not the case for memory allocated using the flag `cudaHostAllocWriteCombined`, as discussed below.

4.9.6 Automatic Registration of Peer Memory

Upon enabling direct access from a device that supports unified addressing to another peer device that supports unified addressing using `cudaDeviceEnablePeerAccess()` all memory allocated in the peer device using `cudaMalloc()` and `cudaMallocPitch()` will immediately be accessible by the current device. In this case, explicitly registering allocations using `cudaPeerRegister()` is not necessary. The device pointer value through which any peer's memory may be accessed in the current device is the same pointer value through which that memory may be accessed from the peer device, so it is not necessary to call `cuMemPeerGetDevicePointer()` to get the device pointer for these allocations.

4.9.7 Exceptions, Disjoint Addressing

Not all memory may be accessed on devices through the same pointer value through which they are accessed on the host. These exceptions are host memory registered using `cudaHostRegister()` and host memory allocated using the flag `cudaHostAllocWriteCombined`. For these exceptions, there exists a distinct host and device address for the memory. The device address is guaranteed to not overlap any valid host pointer range and is guaranteed to have the same value across all devices that support unified addressing.

This device address may be queried using `cudaHostGetDevicePointer()` when a device using unified addressing is current. Either the host or the unified device pointer value may be used to refer to this memory in `cudaMemcpy()` and similar functions using the `cudaMemcpyDefault` memory direction.

4.9.8 Function Documentation

4.9.8.1 `cudaError_t cudaPointerGetAttributes (struct cudaPointerAttributes * attributes, void * ptr)`

Returns in `*attributes` the attributes of the pointer `ptr`.

The `cudaPointerAttributes` structure is defined as:

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

In this structure, the individual fields mean

- `memoryType` identifies the physical location of the memory associated with pointer `ptr`. It can be `cudaMemoryTypeHost` for host memory or `cudaMemoryTypeDevice` for device memory.
- `device` is the device against which `ptr` was allocated. If `ptr` has memory type `cudaMemoryTypeDevice` then this identifies the device on which the memory referred to by `ptr` physically resides. If `ptr` has memory type `cudaMemoryTypeHost` then this identifies the device which was current when the allocation was made (and if that device is deinitialized then this allocation will vanish with that device's state).
- `devicePointer` is the device pointer alias through which the memory referred to by `ptr` may be accessed on the current device. If the memory referred to by `ptr` cannot be accessed directly by the current device then this is NULL.
- `hostPointer` is the host pointer alias through which the memory referred to by `ptr` may be accessed on the host. If the memory referred to by `ptr` cannot be accessed directly by the host then this is NULL.

Parameters:

attributes - Attributes for the specified pointer

ptr - Pointer to get attributes for

Returns:

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

See also:

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

4.10 Peer Device Memory Access

Functions

- [cudaError_t cudaDeviceCanAccessPeer \(int *canAccessPeer, int device, int peerDevice\)](#)
Queries if a device may directly access a peer device's memory.
- [cudaError_t cudaDeviceDisablePeerAccess \(int peerDevice\)](#)
Disables direct access to memory allocations on a peer device and unregisters any registered allocations from that device.
- [cudaError_t cudaDeviceEnablePeerAccess \(int peerDevice, unsigned int flags\)](#)
Enables direct access to memory allocations on a peer device.
- [cudaError_t cudaPeerGetDevicePointer \(void **pDevice, void *peerDevicePointer, int peerDevice, unsigned int flags\)](#)
Retrieves a device pointer through which a peer device's registered memory may be directly accessed.
- [cudaError_t cudaPeerRegister \(void *peerDevicePointer, int peerDevice, unsigned int flags\)](#)
Registers an existing memory allocation on a peer device for direct access from the current device.
- [cudaError_t cudaPeerUnregister \(void *peerDevicePointer, int peerDevice\)](#)
Unregisters a peer device's memory allocation, disabling direct access.

4.10.1 Detailed Description

This section describes the peer device memory access functions of the CUDA runtime application programming interface.

4.10.2 Function Documentation

4.10.2.1 [cudaError_t cudaDeviceCanAccessPeer \(int * canAccessPeer, int device, int peerDevice\)](#)

Returns in `*canAccessPeer` a value of 1 if device `device` is capable of directly accessing memory from `peerDevice` and 0 otherwise. If direct access of `peerDevice` from `device` is possible, then access may be enabled by calling [cudaDeviceEnablePeerAccess\(\)](#).

Parameters:

`canAccessPeer` - Returned access capability

`device` - Device from which allocations on `peerDevice` are to be directly accessed.

`peerDevice` - Device on which the allocations to be directly accessed by `device` reside.

Returns:

`cudaSuccess`, `cudaErrorInvalidDevice`

Note:

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

See also:

[cudaDeviceEnablePeerAccess](#), [cudaDeviceDisablePeerAccess](#), [cudaPeerRegister](#), [cudaPeerUnregister](#), [cudaPeerGetDevicePointer](#)

4.10.2.2 `cudaError_t cudaDeviceDisablePeerAccess (int peerDevice)`

Disables registering memory on `peerDevice` for direct access from the current device. If there are any allocations on `peerDevice` which were registered in the current device using [cudaPeerRegister\(\)](#) then these allocations will be automatically unregistered.

Returns [cudaErrorPeerAccessNotEnabled](#) if direct access to memory on `peerDevice` has not yet been enabled from the current device.

Parameters:

peerDevice - Peer device to disable direct access to

Returns:

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

Note:

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

See also:

[cudaDeviceCanAccessPeer](#), [cudaDeviceEnablePeerAccess](#), [cudaPeerRegister](#), [cudaPeerUnregister](#), [cudaPeerGetDevicePointer](#)

4.10.2.3 `cudaError_t cudaDeviceEnablePeerAccess (int peerDevice, unsigned int flags)`

Enables registering memory on `peerDevice` for direct access from the current device. On success, allocations on `peerDevice` may be registered for access from the current device using [cudaPeerRegister\(\)](#). Registering peer memory will be possible until it is explicitly disabled using [cudaDeviceDisablePeerAccess\(\)](#), or either the current device or `peerDevice` is reset using [cudaDeviceReset\(\)](#).

If both the current device and `peerDevice` support unified addressing then all allocations from `peerDevice` will immediately be accessible by the current device upon success. In this case, explicitly sharing allocations using [cudaPeerRegister\(\)](#) is not necessary.

Note that access granted by this call is unidirectional and that in order to access memory on the current device from `peerDevice`, a separate symmetric call to [cudaDeviceEnablePeerAccess\(\)](#) is required.

Returns [cudaErrorInvalidDevice](#) if [cudaDeviceCanAccessPeer\(\)](#) indicates that the current device cannot directly access memory from `peerDevice`.

Returns [cudaErrorPeerAccessAlreadyEnabled](#) if direct access of `peerDevice` from the current device has already been enabled.

Returns [cudaErrorInvalidValue](#) if `flags` is not 0.

Parameters:

peerDevice - Peer device to enable direct access to from the current device

flags - Reserved for future use and must be set to 0

Returns:

`cudaSuccess`, `cudaErrorInvalidDevice` `cudaErrorPeerAccessAlreadyEnabled`, `cudaErrorInvalidValue`

Note:

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

See also:

`cudaDeviceCanAccessPeer`, `cudaDeviceDisablePeerAccess`, `cudaPeerRegister`, `cudaPeerUnregister`, `cudaPeerGetDevicePointer`

4.10.2.4 `cudaError_t cudaPeerGetDevicePointer (void **pDevice, void *peerDevicePointer, int peerDevice, unsigned int flags)`

Retrieves a device pointer through which memory registered using `cudaPeerRegister()` may be directly accessed from the current device. On success, returns in `*pDevice` a device pointer through which the memory at `peerPointer` on `peerDevice` may be accessed through kernels and copies in the current device.

If both the current device and `peerDevice` support unified addressing then the pointer to access `peerPointer` on `peerDevice` from the current device will be `peerPointer` itself, and querying this pointer value is not necessary.

Returns `cudaErrorInvalidValue` if `peerPointer` on `peerDevice` was not registered with the flag `cudaPeerRegisterMapped`.

Returns `cudaErrorPeerMemoryNotRegistered` if `peerPointer` on `peerDevice` has not been registered yet.

Returns `cudaErrorInvalidValue` if `peerPointer` is not a pointer value returned by `cudaMalloc()` or its variance on `peerDevice`, or if `flags` is not 0.

Returns `cudaErrorInvalidDevice` if `peerDevice` is not a valid device.

Parameters:

pDevice - Returned device pointer on the current device.

peerDevicePointer - Pointer on `peerDevice` to query.

peerDevice - Peer device on which `peerPointer` was allocated.

flags - Reserved for future use and must be set to 0

Returns:

`cudaSuccess`, `cudaErrorPeerMemoryNotRegistered`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`

Note:

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

See also:

`cudaDeviceCanAccessPeer`, `cudaDeviceEnablePeerAccess`, `cudaDeviceDisablePeerAccess`, `cudaPeerRegister`, `cudaPeerUnregister`

4.10.2.5 `cudaError_t cudaPeerRegister (void *peerDevicePointer, int peerDevice, unsigned int flags)`

Registers the memory associated with `peerPointer` on `peerDevice` for direct access from the current device. Memory that has been registered will be read and written directly when accessed through `cudaMemcpyPeer()` and

its variants (as opposed to being staged through host memory). Memory which has been registered with the flag `cudaPeerRegisterMapped` may also be accessed directly via pointer dereferences in kernels.

This memory will remain accessible until explicitly unregistered via `cudaPeerUnregister()` or implicitly unregistered when peer access to `peerDevice` is disabled by `cudaDeviceDisablePeerAccess()` or either the current device or `peerDevice` is reset by `cudaDeviceReset()`.

Note that explicitly registering memory is not required for direct access between devices which support unified addressing. In such cases all memory allocated in `peerDevice` is automatically made accessible to the current device by `cudaDeviceEnablePeerAccess()`.

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

- `cudaPeerRegisterDefault`: Allow direct access only when copying to and from the allocation using `cudaMemcpyPeer()` and its variants.
- `cudaPeerRegisterMapped`: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling `cudaPeerGetDevicePointer()`.

Returns `cudaErrorPeerAccessNotEnabled` if direct access has not yet been enabled from `peerDevice` to the current device.

Returns `cudaErrorPeerMemoryAlreadyRegistered` if `peerPointer` from `peerDevice` has already been registered for direct access by the current device.

Returns `cudaErrorInvalidValue` if `peerPointer` is not a pointer value returned by `cudaMalloc()` or its variants on `peerDevice`.

Returns `cudaErrorInvalidDevice` if `peerDevice` is not a valid device.

Parameters:

`peerDevicePointer` - Pointer in peer device to register for direct access from the current device.

`peerDevice` - Peer device on which `peerPointer` was allocated.

`flags` - Flags affecting allocation registration.

Returns:

`cudaSuccess`, `cudaErrorPeerAccessNotEnabled`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`

Note:

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

See also:

`cudaDeviceCanAccessPeer`, `cudaDeviceEnablePeerAccess`, `cudaDeviceDisablePeerAccess`, `cudaPeerUnregister`, `cudaPeerGetDevicePointer`

4.10.2.6 `cudaError_t cudaPeerUnregister (void *peerDevicePointer, int peerDevice)`

Unregisters peer memory which was registered with the current device using `cudaPeerRegister()`.

Returns `cudaErrorPeerMemoryNotRegistered` if `peerPointer` on `peerDevice` has not been registered yet.

Returns `cudaErrorInvalidValue` if `peerPointer` is not a pointer value returned by `cudaMalloc()` or its variants on `peerDevice`.

Returns `cudaErrorInvalidDevice` if `peerDevice` is not a valid device.

Parameters:

peerDevicePointer - Pointer in *peerDevice* to unregister.
peerDevice - Peer device on which *peerPointer* was allocated.

Returns:

`cudaSuccess`, `cudaErrorPeerMemoryNotRegistered`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`

Note:

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

See also:

`cudaDeviceCanAccessPeer`, `cudaDeviceEnablePeerAccess`, `cudaDeviceDisablePeerAccess`, `cudaPeerRegister`,
`cudaPeerGetDevicePointer`

4.11 OpenGL Interoperability

Modules

- OpenGL Interoperability [DEPRECATED]

Functions

- **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.

4.11.1 Detailed Description

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

4.11.2 Function Documentation

4.11.2.1 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 register flags `flags` specify the intended usage, as follows:

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

Parameters:

`resource` - Pointer to the returned object handle

`buffer` - name of buffer object to be registered

`flags` - Register flags

Returns:

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

Note:

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

See also:

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

4.11.2.2 `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 register flags `flags` specify the intended usage, as follows:

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

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` - Register 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](#)

4.11.2.3 `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`

4.12 Direct3D 9 Interoperability

Modules

- Direct3D 9 Interoperability [DEPRECATED]

Enumerations

- enum `cudaD3D9DeviceList` {

`cudaD3D9DeviceListAll` = 1,

`cudaD3D9DeviceListCurrentFrame` = 2,

`cudaD3D9DeviceListNextFrame` = 3 }

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 9 device to use for interoperability with a CUDA device.
- `cudaError_t cudaGraphicsD3D9RegisterResource` (struct `cudaGraphicsResource` **resource, IDirect3DResource9 *pD3DResource, unsigned int flags)

Register a Direct3D 9 resource for access by CUDA.

4.12.1 Detailed Description

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

4.12.2 Enumeration Type Documentation

4.12.2.1 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

4.12.3 Function Documentation

4.12.3.1 ***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](#),

4.12.3.2 ***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](#)

4.12.3.3 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](#)

4.12.3.4 cudaError_t cudaD3D9SetDirect3DDevice (IDirect3DDevice9 * *pD3D9Device*, int *device* = -1)

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

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

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

Parameters:

pD3D9Device - Direct3D device to use for this thread

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

Returns:

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

Note:

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

See also:

[cudaD3D9GetDevice](#), [cudaGraphicsD3D9RegisterResource](#), [cudaDeviceReset](#)

4.12.3.5 `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 valid values for this parameter are

- `cudaGraphicsRegisterFlagsNone`: Specifies no hints about how this resource will be used.
- `cudaGraphicsRegisterFlagsSurfaceLoadStore`: Specifies that CUDA will bind this resource to a surface reference.

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](#)

4.13 Direct3D 10 Interoperability

Modules

- Direct3D 10 Interoperability [DEPRECATED]

Enumerations

- enum `cudaD3D10DeviceList` {
 `cudaD3D10DeviceListAll` = 1,
 `cudaD3D10DeviceListCurrentFrame` = 2,
 `cudaD3D10DeviceListNextFrame` = 3 }

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 with a CUDA device.
- `cudaError_t cudaGraphicsD3D10RegisterResource` (struct `cudaGraphicsResource` **resource, ID3D10Resource *pD3DResource, unsigned int flags)
Register a Direct3D 10 resource for access by CUDA.

4.13.1 Detailed Description

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

4.13.2 Enumeration Type Documentation

4.13.2.1 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

4.13.3 Function Documentation

4.13.3.1 **cudaError_t cudaD3D10GetDevice (int *device, IDXGIAdapter *pAdapter)**

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

Parameters:

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

Returns:

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

Note:

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

See also:

[cudaD3D10SetDirect3DDevice](#), [cudaGraphicsD3D10RegisterResource](#),

4.13.3.2 **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](#)

4.13.3.3 `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](#)

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

Records `pD3D10Device` as the Direct3D 10 device to use for Direct3D 10 interoperability with the CUDA device `device` and sets `device` as the current device for the calling host thread.

If `device` has already been initialized then this call will fail with the error [cudaErrorSetOnActiveProcess](#). In this case it is necessary to reset `device` using [cudaDeviceReset\(\)](#) before Direct3D 10 interoperability on `device` may be enabled.

Successfully initializing CUDA interoperability with `pD3D10Device` will increase the internal reference count on `pD3D10Device`. This reference count will be decremented when `device` is reset using [cudaDeviceReset\(\)](#).

Parameters:

pD3D10Device - Direct3D device to use for interoperability

device - The CUDA device to use. This device must be among the devices returned when querying [cudaD3D10DeviceListAll](#) from [cudaD3D10GetDevices](#), may be set to -1 to automatically select an appropriate CUDA device.

Returns:

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

Note:

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

See also:

[cudaD3D10GetDevice](#), [cudaGraphicsD3D10RegisterResource](#), [cudaDeviceReset](#)

4.13.3.5 `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 valid values for this parameter are

- `cudaGraphicsRegisterFlagsNone`: Specifies no hints about how this resource will be used.
- `cudaGraphicsRegisterFlagsSurfaceLoadStore`: Specifies that CUDA will bind this resource to a surface reference.

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](#)

4.14 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 cud-aDeviceCount, 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 with a CUDA device.
- `cudaError_t cudaGraphicsD3D11RegisterResource` (struct cudaGraphicsResource **resource, ID3D11Resource *pD3DResource, unsigned int flags)

Register a Direct3D 11 resource for access by CUDA.

4.14.1 Detailed Description

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

4.14.2 Enumeration Type Documentation

4.14.2.1 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

4.14.3 Function Documentation

4.14.3.1 `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](#)

4.14.3.2 `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](#)

4.14.3.3 `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](#)

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

Records `pD3D11Device` as the Direct3D 11 device to use for Direct3D 11 interoperability with the CUDA device `device` and sets `device` as the current device for the calling host thread.

If `device` has already been initialized then this call will fail with the error [cudaErrorSetOnActiveProcess](#). In this case it is necessary to reset `device` using [cudaDeviceReset\(\)](#) before Direct3D 11 interoperability on `device` may be enabled.

Successfully initializing CUDA interoperability with `pD3D11Device` will increase the internal reference count on `pD3D11Device`. This reference count will be decremented when `device` is reset using [cudaDeviceReset\(\)](#).

Parameters:

`pD3D11Device` - Direct3D device to use for interoperability

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

Returns:

`cudaSuccess`, `cudaErrorInitializationError`, `cudaErrorInvalidValue`, `cudaErrorSetOnActiveProcess`

Note:

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

See also:

[cudaD3D11GetDevice](#), [cudaGraphicsD3D11RegisterResource](#), [cudaDeviceReset](#)

4.14.3.5 **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 valid values for this parameter are

- `cudaGraphicsRegisterFlagsNone`: Specifies no hints about how this resource will be used.
- `cudaGraphicsRegisterFlagsSurfaceLoadStore`: Specifies that CUDA will bind this resource to a surface reference.

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](#)

4.15 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 a CUDA device to use VDPAU interoperability.

4.15.1 Detailed Description

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

4.15.2 Function Documentation

4.15.2.1 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](#)

4.15.2.2 `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](#)

4.15.2.3 `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](#)

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

Records `vdpDevice` as the `VdpDevice` for VDPAU interoperability with the CUDA device `device` and sets `device` as the current device for the calling host thread.

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

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](#), [cudaDeviceReset](#)

4.16 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.

4.16.1 Detailed Description

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

4.16.2 Function Documentation

4.16.2.1 `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](#)

4.16.2.2 **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](#)

4.16.2.3 **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](#)

4.16.2.4 `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](#)

4.16.2.5 `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](#)

4.16.2.6 `cudaError_t cudaGraphicsUnregisterResource (cudaGraphicsResource_t resource)`

Registers 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](#)

4.17 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.

4.17.1 Detailed Description

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

4.17.2 Function Documentation

4.17.2.1 `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)

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

4.17.2.3 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 \(C API\)](#)

4.17.2.4 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 \(C API\)](#)

4.17.2.5 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 \(C API\)](#)

4.17.2.6 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 \(C++ API\)](#)

4.17.2.7 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 \(C API\)](#)

4.17.2.8 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 \(C API\)](#)

4.18 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.

4.18.1 Detailed Description

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

4.18.2 Function Documentation

4.18.2.1 `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](#)

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

4.19 Version Management

Functions

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

4.19.1 Function Documentation

4.19.1.1 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](#)

4.19.1.2 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](#)

4.20 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

4.20.1 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.

4.20.2 Function Documentation

4.20.2.1 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 \(C++ API\)](#)

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

4.20.2.3 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 \(C++ API\)](#)

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

**4.20.2.5 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)

**4.20.2.6 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)

4.20.2.7 `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)

**4.20.2.8 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)

4.20.2.9 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)

4.20.2.10 `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`

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

4.20.2.12 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:

- [cudaFuncCacheNone](#): no preference for shared memory or L1 (default)
- [cudaFuncCacheShared](#): prefer larger shared memory and smaller L1 cache
- [cudaFuncCacheL1](#): 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](#)

4.20.2.13 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 \(C++ API\)](#)

4.20.2.14 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 \(C API\)](#)

4.20.2.15 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 \(C API\)](#)

4.20.2.16 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](#)

4.20.2.17 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](#)

4.20.2.18 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](#), [cudaASetDoubleForHost](#), [cudaSetupArgument \(C API\)](#)

**4.20.2.19 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 \(C++ API\)](#)

4.21 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

4.21.1 Primary Contexts

There exists a one to one relationship between CUDA devices in the CUDA Runtime API and [CUcontext](#)s in the CUDA Driver API within a process. The specific context which the CUDA Runtime API uses for a device is called the device's primary context. From the perspective of the CUDA Runtime API, a device and its primary context are synonymous.

4.21.2 Initialization and Tear-Down

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

The function [cudaSetDevice\(\)](#) makes the primary context for the specified device current to the calling thread by calling [cuCtxsetCurrent\(\)](#).

The CUDA Runtime API will automatically initialize the primary context for a device at the first CUDA Runtime API call which requires an active context. If no [CUcontext](#) is current to the calling thread when a CUDA Runtime API call which requires an active context is made, then the primary context for a device will be selected, made current to the calling thread, and initialized.

The context which the CUDA Runtime API initializes will be initialized using the parameters specified by the CUDA Runtime API functions [cudaSetDeviceFlags\(\)](#), [cudaD3D9SetDirect3DDevice\(\)](#), [cudaD3D10SetDirect3DDevice\(\)](#), [cudaD3D11SetDirect3DDevice\(\)](#), [cudaGLSetGLDevice\(\)](#), and [cudaVDPAUSetVDPAUDevice\(\)](#). Note that these functions will fail with [cudaErrorSetOnActiveProcess](#) if they are called when the primary context for the specified device has already been initialized. (or if the current device has already been initialized, in the case of [cudaSetDeviceFlags\(\)](#)).

Primary contexts will remain active until they are explicitly deinitialized using [cudaDeviceReset\(\)](#). The function [cudaDeviceReset\(\)](#) will deinitialize the primary context for the calling thread's current device immediately. The context will remain current to all of the threads that it was current to. The next CUDA Runtime API call on any thread which requires an active context will trigger the reinitialization of that device's primary context.

Note that there is no reference counting of the primary context's lifetime. It is recommended that the primary context not be deinitialized except just before exit or to recover from an unspecified launch failure.

4.21.3 Context Interoperability

Note that the use of multiple [CUcontext](#)s per device within a single process will substantially degrade performance and is strongly discouraged. Instead, it is highly recommended that the implicit one-to-one device-to-context mapping for the process provided by the CUDA Runtime API be used.

If a non-primary [CUcontext](#) created by the CUDA Driver API is current to a thread then the CUDA Runtime API calls to that thread will operate on that [CUcontext](#), with some exceptions listed below. Interoperability between data types is discussed in the following sections.

The function [cudaPointerGetAttributes\(\)](#) will return the error [cudaErrorIncompatibleDriverContext](#) if the pointer being queried was allocated by a non-primary context. The function [cudaDeviceEnablePeerAccess\(\)](#) and the rest of the peer access API may not be called when a non-primary [CUcontext](#) is current. To use the pointer query and peer access APIs with a context created using the CUDA Driver API, it is necessary that the CUDA Driver API be used to access these features.

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

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

4.21.4 Interactions between [CUstream](#) and [cudaStream_t](#)

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

4.21.5 Interactions between [CUevent](#) and [cudaEvent_t](#)

The types [CUevent](#) and [cudaEvent_t](#) are identical and may be used interchangeably.

4.21.6 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](#).

4.21.7 Interactions between [CUgraphicsResource](#) and [cudaGraphicsResource_t](#)

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

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

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

4.22 Direct3D 9 Interoperability [DEPRECATED]

Enumerations

- enum `cudaD3D9MapFlags` {
 `cudaD3D9MapFlagsNone` = 0,
 `cudaD3D9MapFlagsReadOnly` = 1,
 `cudaD3D9MapFlagsWriteDiscard` = 2 }
- enum `cudaD3D9RegisterFlags` {
 `cudaD3D9RegisterFlagsNone` = 0,
 `cudaD3D9RegisterFlagsArray` = 1 }

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.

4.22.1 Detailed Description

This section describes deprecated Direct3D 9 interoperability functions.

4.22.2 Enumeration Type Documentation

4.22.2.1 enum cudaD3D9MapFlags

CUDA D3D9 Map Flags

Enumerator:

cudaD3D9MapFlagsNone Default; Assume resource can be read/written

cudaD3D9MapFlagsReadOnly CUDA kernels will not write to this resource

cudaD3D9MapFlagsWriteDiscard CUDA kernels will only write to and will not read from this resource

4.22.2.2 enum cudaD3D9RegisterFlags

CUDA D3D9 Register Flags

Enumerator:

cudaD3D9RegisterFlagsNone Default; Resource can be accessed through a void*

cudaD3D9RegisterFlagsArray Resource can be accessed through a CUarray*

4.22.3 Function Documentation

4.22.3.1 `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](#)

4.22.3.2 `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](#)

4.22.3.3 `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 `pArray` may change every time that `pResource` is mapped.

If `pResource` is not registered then `cudaErrorInvalidResourceHandle` is returned. If `pResource` was not registered with usage flags `cudaD3D9RegisterFlagsArray`, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` is not mapped, then `cudaErrorUnknown` is returned.

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

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](#)

**4.22.3.4 `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 * pitch + (bytes per pixel) * x`

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

`z * slicePitch + y * pitch + (bytes per pixel) * x`

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

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

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

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](#)

**4.22.3.5 `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](#)

4.22.3.6 `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](#)

4.22.3.7 `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](#)

4.22.3.8 `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`

4.22.3.9 `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](#)

4.22.3.10 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](#)

4.23 Direct3D 10 Interoperability [DEPRECATED]

Enumerations

- enum `cudaD3D10MapFlags` {

`cudaD3D10MapFlagsNone` = 0,

`cudaD3D10MapFlagsReadOnly` = 1,

`cudaD3D10MapFlagsWriteDiscard` = 2
 }
- enum `cudaD3D10RegisterFlags` {

`cudaD3D10RegisterFlagsNone` = 0,

`cudaD3D10RegisterFlagsArray` = 1
 }

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.

4.23.1 Detailed Description

This section describes deprecated Direct3D 10 interoperability functions.

4.23.2 Enumeration Type Documentation

4.23.2.1 enum cudaD3D10MapFlags

CUDA D3D10 Map Flags

Enumerator:

cudaD3D10MapFlagsNone Default; Assume resource can be read/written

cudaD3D10MapFlagsReadOnly CUDA kernels will not write to this resource

cudaD3D10MapFlagsWriteDiscard CUDA kernels will only write to and will not read from this resource

4.23.2.2 enum cudaD3D10RegisterFlags

CUDA D3D10 Register Flags

Enumerator:

cudaD3D10RegisterFlagsNone Default; Resource can be accessed through a void*

cudaD3D10RegisterFlagsArray Resource can be accessed through a CUarray*

4.23.3 Function Documentation

4.23.3.1 `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](#)

4.23.3.2 `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](#)

4.23.3.3 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](#)

4.23.3.4 `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](#)

4.23.3.5 `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](#)

4.23.3.6 `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](#)

4.23.3.7 `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](#)

4.23.3.8 `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`

4.23.3.9 `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`

4.23.3.10 `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](#)

4.24 OpenGL Interoperability [DEPRECATED]

Enumerations

- enum `cudaGLMapFlags` {
 `cudaGLMapFlagsNone` = 0,
 `cudaGLMapFlagsReadOnly` = 1,
 `cudaGLMapFlagsWriteDiscard` = 2 }

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.

4.24.1 Detailed Description

This section describes deprecated OpenGL interoperability functionality.

4.24.2 Enumeration Type Documentation

4.24.2.1 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

4.24.3 Function Documentation

4.24.3.1 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](#)

4.24.3.2 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](#)

4.24.3.3 `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](#)

4.24.3.4 `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](#)

4.24.3.5 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](#)

4.24.3.6 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](#)

4.24.3.7 `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](#)

4.25 Data types used by CUDA Runtime

Data Structures

- struct `cudaChannelFormatDesc`
- struct `cudaDeviceProp`
- struct `cudaExtent`
- struct `cudaFuncAttributes`
- struct `cudaMemcpy3DParms`
- struct `cudaMemcpy3DPeerParms`
- struct `cudaPitchedPtr`
- struct `cudaPointerAttributes`
- 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,
 `cudaComputeModeExclusiveProcess` = 3 }
- 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,
cudaErrorPeerAccessAlreadyEnabled = 50,
cudaErrorPeerAccessNotEnabled = 51,
cudaErrorPeerMemoryAlreadyRegistered = 52,
cudaErrorPeerMemoryNotRegistered = 53,
cudaErrorDeviceAlreadyInUse = 54,
cudaErrorProfilerDisabled = 55,
cudaErrorProfilerNotInitialized = 56,
cudaErrorProfilerAlreadyStarted = 57,
cudaErrorProfilerAlreadyStopped = 58,
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,
 `cudaGraphicsRegisterFlagsReadOnly` = 1,
 `cudaGraphicsRegisterFlagsWriteDiscard` = 2,
 `cudaGraphicsRegisterFlagsSurfaceLoadStore` = 4 }
- enum `cudaLimit` {
 `cudaLimitStackSize` = 0x00,
 `cudaLimitPrintfFifoSize` = 0x01,
 `cudaLimitMallocHeapSize` = 0x02 }
- enum `cudaMemcpyKind` {
 `cudaMemcpyHostToHost` = 0,
 `cudaMemcpyHostToDevice` = 1,
 `cudaMemcpyDeviceToHost` = 2,
 `cudaMemcpyDeviceToDevice` = 3,
 `cudaMemcpyDefault` = 4 }
- enum `cudaMemoryType` {
 `cudaMemoryTypeHost` = 1,
 `cudaMemoryTypeDevice` = 2 }
- enum `cudaOutputMode`
- typedef enum `cudaError` `cudaError_t`
- typedef struct `CUevent_st` * `cudaEvent_t`
- typedef struct `cudaGraphicsResource` * `cudaGraphicsResource_t`
- typedef enum `cudaOutputMode` `cudaOutputMode_t`
- typedef struct `CUstream_st` * `cudaStream_t`
- typedef struct `CUuid_st` `cudaUUID_t`
- #define `cudaArrayDefault` 0x00
- #define `cudaArrayLayered` 0x01
- #define `cudaArraySurfaceLoadStore` 0x02
- #define `cudaDeviceBlockingSync` 0x04
- #define `cudaDeviceLmemResizeToMax` 0x10
- #define `cudaDeviceMapHost` 0x08
- #define `cudaDeviceMask` 0x1f
- #define `cudaDevicePropDontCare`
- #define `cudaDeviceScheduleAuto` 0x00
- #define `cudaDeviceScheduleBlockingSync` 0x04
- #define `cudaDeviceScheduleSpin` 0x01
- #define `cudaDeviceScheduleYield` 0x02
- #define `cudaEventBlockingSync` 0x01
- #define `cudaEventDefault` 0x00
- #define `cudaEventDisableTiming` 0x02
- #define `cudaHostAllocDefault` 0x00
- #define `cudaHostAllocMapped` 0x02

- #define `cudaHostAllocPortable` 0x01
- #define `cudaHostAllocWriteCombined` 0x04
- #define `cudaHostRegisterDefault` 0x00
- #define `cudaHostRegisterMapped` 0x02
- #define `cudaHostRegisterPortable` 0x01
- #define `cudaPeerAccessDefault` 0x00
- #define `cudaPeerDevicePointerDefault` 0x00
- #define `cudaPeerRegisterDefault` 0x00
- #define `cudaPeerRegisterMapped` 0x02

4.25.1 Define Documentation

4.25.1.1 #define `cudaArrayDefault` 0x00

Default CUDA array allocation flag

4.25.1.2 #define `cudaArrayLayered` 0x01

Must be set in `cudaMalloc3DArray` to create a layered texture

4.25.1.3 #define `cudaArraySurfaceLoadStore` 0x02

Must be set in `cudaMallocArray` in order to bind surfaces to the CUDA array

4.25.1.4 #define `cudaDeviceBlockingSync` 0x04

Device flag - Use blocking synchronization

Deprecated

4.25.1.5 #define `cudaDeviceLmemResizeToMax` 0x10

Device flag - Keep local memory allocation after launch

4.25.1.6 #define `cudaDeviceMapHost` 0x08

Device flag - Support mapped pinned allocations

4.25.1.7 #define `cudaDeviceMask` 0x1f

Device flags mask

4.25.1.8 #define `cudaDevicePropDontCare`

Empty device properties

4.25.1.9 #define cudaDeviceScheduleAuto 0x00

Device flag - Automatic scheduling

4.25.1.10 #define cudaDeviceScheduleBlockingSync 0x04

Device flag - Use blocking synchronization

4.25.1.11 #define cudaDeviceScheduleSpin 0x01

Device flag - Spin default scheduling

4.25.1.12 #define cudaDeviceScheduleYield 0x02

Device flag - Yield default scheduling

4.25.1.13 #define cudaEventBlockingSync 0x01

Event uses blocking synchronization

4.25.1.14 #define cudaEventDefault 0x00

Default event flag

4.25.1.15 #define cudaEventDisableTiming 0x02

Event will not record timing data

4.25.1.16 #define cudaHostAllocDefault 0x00

Default page-locked allocation flag

4.25.1.17 #define cudaHostAllocMapped 0x02

Map allocation into device space

4.25.1.18 #define cudaHostAllocPortable 0x01

Pinned memory accessible by all CUDA contexts

4.25.1.19 #define cudaHostAllocWriteCombined 0x04

Write-combined memory

4.25.1.20 #define cudaHostRegisterDefault 0x00

Default host memory registration flag

4.25.1.21 #define cudaHostRegisterMapped 0x02

Map registered memory into device space

4.25.1.22 #define cudaHostRegisterPortable 0x01

Pinned memory accessible by all CUDA contexts

4.25.1.23 #define cudaPeerAccessDefault 0x00

Default peer addressing enable flag

4.25.1.24 #define cudaPeerDevicePointerDefault 0x00

Default peer device pointer query flag

4.25.1.25 #define cudaPeerRegisterDefault 0x00

Default peer memory registration flag

4.25.1.26 #define cudaPeerRegisterMapped 0x02

Map registered memory into device space

4.25.2 Typedef Documentation

4.25.2.1 typedef enum cudaError cudaError_t

CUDA Error types

4.25.2.2 typedef struct CUevent_st* cudaEvent_t

CUDA event types

4.25.2.3 typedef struct cudaGraphicsResource* cudaGraphicsResource_t

CUDA graphics resource types

4.25.2.4 typedef enum cudaOutputMode cudaOutputMode_t

CUDA output file modes

4.25.2.5 typedef struct CUstream_st* cudaStream_t

CUDA stream

4.25.2.6 `typedef struct CUuid_st cudaUUID_t`

CUDA UUID types

4.25.3 Enumeration Type Documentation

4.25.3.1 `enum cudaChannelFormatKind`

Channel format kind

Enumerator:

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

4.25.3.2 `enum cudaComputeMode`

CUDA device compute modes

Enumerator:

- cudaComputeModeDefault* Default compute mode (Multiple threads can use [cudaSetDevice\(\)](#) with this device)
- cudaComputeModeExclusive* Compute-exclusive-thread mode (Only one thread in one process will be able to use [cudaSetDevice\(\)](#) with this device)
- cudaComputeModeProhibited* Compute-prohibited mode (No threads can use [cudaSetDevice\(\)](#) with this device)
- cudaComputeModeExclusiveProcess* Compute-exclusive-process mode (Many threads in one process will be able to use [cudaSetDevice\(\)](#) with this device)

4.25.3.3 `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 the CUDA Runtime. This can only occur if you are using CUDA Runtime/Driver interoperability and have created an existing Driver context using the driver API. The Driver context may be incompatible either because the Driver context was created using an older version of the API, because the Runtime API call expects a primary driver context and the Driver context is not primary, or because the Driver context has been destroyed. Please see [Interactions](#) with the CUDA Driver API" for more information.

cudaErrorPeerAccessAlreadyEnabled This error indicates that a call to `cudaDeviceEnablePeerAccess()` is trying to re-enable peer addressing on from a context which has already had peer addressing enabled.

cudaErrorPeerAccessNotEnabled This error indicates that a call to `cudaPeerRegister` is trying to register memory from a context which has not had peer addressing enabled yet via `cudaDeviceEnablePeerAccess()`, or that `cudaDeviceDisablePeerAccess()` is trying to disable peer addressing which has not been enabled yet.

cudaErrorPeerMemoryAlreadyRegistered This error indicates that a call to `cudaPeerRegister` is trying to register already-registered memory.

cudaErrorPeerMemoryNotRegistered This error indicates that a call to `cudaPeerUnregister` is trying to unregister memory that has not been registered.

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

cudaErrorProfilerDisabled This indicates profiler has been disabled for this run and thus runtime APIs cannot be used to profile subsets of the program. This can happen when the application is running with external profiling tools like visual profiler.

cudaErrorProfilerNotInitialized This indicates profiler has not been initialized yet. `cudaProfilerInitialize()` must be called before calling `cudaProfilerStart` and `cudaProfilerStop` to initialize profiler.

cudaErrorProfilerAlreadyStarted This indicates profiler is already started. This error can be returned if `cudaProfilerStart()` is called multiple times without subsequent call to `cudaProfilerStop()`.

cudaErrorProfilerAlreadyStopped This indicates profiler is already stopped. This error can be returned if `cudaProfilerStop()` is called without starting profiler using `cudaProfilerStart()`.

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.

4.25.3.4 enum cudaFuncCache

CUDA function cache configurations

Enumerator:

cudaFuncCacheNone Default function cache configuration, no preference

cudaFuncCacheShared Prefer larger shared memory and smaller L1 cache

cudaFuncCacheL1 Prefer larger L1 cache and smaller shared memory

4.25.3.5 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

4.25.3.6 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

4.25.3.7 enum cudaGraphicsRegisterFlags

CUDA graphics interop register flags

Enumerator:

- cudaGraphicsRegisterFlagsNone* Default
- cudaGraphicsRegisterFlagsReadOnly* CUDA will not write to this resource
- cudaGraphicsRegisterFlagsWriteDiscard* CUDA will only write to and will not read from this resource
- cudaGraphicsRegisterFlagsSurfaceLoadStore* CUDA will bind this resource to a surface reference

4.25.3.8 enum cudaLimit

CUDA Limits

Enumerator:

- cudaLimitStackSize* GPU thread stack size
- cudaLimitPrintfFifoSize* GPU printf FIFO size
- cudaLimitMallocHeapSize* GPU malloc heap size

4.25.3.9 enum cudaMemcpyKind

CUDA memory copy types

Enumerator:

- cudaMemcpyHostToHost* Host -> Host
- cudaMemcpyHostToDevice* Host -> Device
- cudaMemcpyDeviceToHost* Device -> Host
- cudaMemcpyDeviceToDevice* Device -> Device
- cudaMemcpyDefault* Default based unified virtual address space

4.25.3.10 enum cudaMemoryType

CUDA memory types

Enumerator:

- cudaMemoryTypeHost* Host memory
- cudaMemoryTypeDevice* Device memory

4.25.3.11 enum cudaOutputMode

CUDA Profiler Output modes

4.25.3.12 enum cudaSurfaceBoundaryMode

CUDA Surface boundary modes

Enumerator:

- cudaBoundaryModeZero* Zero boundary mode
- cudaBoundaryModeClamp* Clamp boundary mode
- cudaBoundaryModeTrap* Trap boundary mode

4.25.3.13 enum cudaSurfaceFormatMode

CUDA Surface format modes

Enumerator:

- cudaFormatModeForced* Forced format mode
- cudaFormatModeAuto* Auto format mode

4.25.3.14 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

4.25.3.15 enum cudaTextureFilterMode

CUDA texture filter modes

Enumerator:

- cudaFilterModePoint* Point filter mode
- cudaFilterModeLinear* Linear filter mode

4.25.3.16 enum cudaTextureReadMode

CUDA texture read modes

Enumerator:

- cudaReadModeElementType* Read texture as specified element type
- cudaReadModeNormalizedFloat* Read texture as normalized float

4.26 CUDA Driver API

Modules

- Data types used by CUDA driver
- Initialization
- Version Management
- Device Management
- Context Management
- Module Management
- Memory Management
- Unified Addressing
- Stream Management
- Event Management
- Execution Control
- Texture Reference Management
- Surface Reference Management
- Peer Context Memory Access
- Graphics Interoperability
- OpenGL Interoperability
- Direct3D 9 Interoperability
- Direct3D 10 Interoperability
- Direct3D 11 Interoperability
- VDPAU Interoperability

4.26.1 Detailed Description

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

4.27 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_PEER_st`
- struct `CUDA_MEMCPY3D_st`
- struct `CUdevprop_st`

Defines

- `#define CU_LAUNCH_PARAM_BUFFER_POINTER ((void*)0x01)`
- `#define CU_LAUNCH_PARAM_BUFFER_SIZE ((void*)0x02)`
- `#define CU_LAUNCH_PARAM_END ((void*)0x00)`
- `#define CU_MEMHOSTALLOC_DEVICEMAP 0x02`
- `#define CU_MEMHOSTALLOC_PORTABLE 0x01`
- `#define CU_MEMHOSTALLOC_WRITECOMBINED 0x04`
- `#define CU_MEMHOSTREGISTER_DEVICEMAP 0x02`
- `#define CU_MEMHOSTREGISTER_PORTABLE 0x01`
- `#define CU_MEMPEERREGISTER_DEVICEMAP 0x02`
- `#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_LAYERED 0x01`
- `#define CUDA_ARRAY3D_SURFACE_LDST 0x02`
- `#define CUDA_VERSION 4000`

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 struct CUDA_MEMCPY3D_PEER_st CUDA_MEMCPY3D_PEER`
- `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 CUjitFallback_enum CUjitFallback`
- `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 CUOutputMode_st CUOutputMode`
- `typedef enum CUpointer_attribute_enum CUpointer_attribute`
- `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,
 `CU_COMPUTEMODE_EXCLUSIVE_PROCESS` = 3 }
- enum `CUctx_flags_enum` {
 `CU_CTX_SCHED_AUTO` = 0x00,
 `CU_CTX_SCHED_SPIN` = 0x01,
 `CU_CTX_SCHED_YIELD` = 0x02,
 `CU_CTX_SCHED_BLOCKING_SYNC` = 0x04,
 `CU_CTX_BLOCKING_SYNC` = 0x04 ,
 `CU_CTX_MAP_HOST` = 0x08,
 `CU_CTX_LMEM_RESIZE_TO_MAX` = 0x10,
 `CU_CTX_PRIMARY` = 0x20 }
- 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_PROFILER_DISABLED` = 5,
 `CUDA_ERROR_PROFILER_NOT_INITIALIZED` = 6,
 `CUDA_ERROR_PROFILER_ALREADY_STARTED` = 7,
 `CUDA_ERROR_PROFILER_ALREADY_STOPPED` = 8,
 `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_CONTEXT_ALREADY_IN_USE` = 216,

```
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_PEER_ACCESS_ALREADY_ENABLED = 704,  
CUDA_ERROR_PEER_ACCESS_NOT_ENABLED = 705,  
CUDA_ERROR_PEER_MEMORY_ALREADY_REGISTERED = 706,  
CUDA_ERROR_PEER_MEMORY_NOT_REGISTERED = 707,  
CUDA_ERROR_PRIMARY_CONTEXT_ACTIVE = 708,  
CUDA_ERROR_CONTEXT_IS_DESTROYED = 709,  
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_LAYERED_WIDTH = 27,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT = 28,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS = 29,
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,
CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE = 36,
CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH = 37,
CU_DEVICE_ATTRIBUTE_L2_CACHE_SIZE = 38,
CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR = 39,
CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT = 40,
CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING = 41,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH = 42,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS = 43 }

• 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,
 `CU_MEMORYTYPE_UNIFIED` = 0x04 }
- enum `CUOutputMode_st`

- enum CUpointer_attribute_enum {
 CU_POINTER_ATTRIBUTE_CONTEXT = 1,
 CU_POINTER_ATTRIBUTE_MEMORY_TYPE = 2,
 CU_POINTER_ATTRIBUTE_DEVICE_POINTER = 3,
 CU_POINTER_ATTRIBUTE_HOST_POINTER = 4 }

4.27.1 Define Documentation

4.27.1.1 #define CU_LAUNCH_PARAM_BUFFER_POINTER ((void*)0x01)

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

4.27.1.2 #define CU_LAUNCH_PARAM_BUFFER_SIZE ((void*)0x02)

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

4.27.1.3 #define CU_LAUNCH_PARAM_END ((void*)0x00)

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

4.27.1.4 #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()`

4.27.1.5 #define CU_MEMHOSTALLOC_PORTABLE 0x01

If set, host memory is portable between CUDA contexts. Flag for `cuMemHostAlloc()`

4.27.1.6 #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()`

4.27.1.7 #define CU_MEMHOSTREGISTER_DEVICEMAP 0x02

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

4.27.1.8 #define CU_MEMHOSTREGISTER_PORTABLE 0x01

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

4.27.1.9 #define CU_MEMPEERREGISTER_DEVICEMAP 0x02

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

4.27.1.10 #define CU_PARAM_TR_DEFAULT -1

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

4.27.1.11 #define CU_TRSA_OVERRIDE_FORMAT 0x01

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

4.27.1.12 #define CU_TRSF_NORMALIZED_COORDINATES 0x02

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

4.27.1.13 #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\(\)](#)

4.27.1.14 #define CU_TRSF_SRGB 0x10

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

4.27.1.15 #define CUDA_ARRAY3D_2DARRAY 0x01

Deprecated, use CUDA_ARRAY3D_LAYERED

4.27.1.16 #define CUDA_ARRAY3D_LAYERED 0x01

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

4.27.1.17 #define CUDA_ARRAY3D_SURFACE_LDST 0x02

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

4.27.1.18 #define CUDA_VERSION 4000

CUDA API version number

4.27.2 Typedef Documentation

4.27.2.1 **typedef enum CUaddress_mode_enum CUaddress_mode**

Texture reference addressing modes

4.27.2.2 **typedef struct Cuarray_st* Cuarray**

CUDA array

4.27.2.3 **typedef enum CUarray_cubemap_face_enum CUarray_cubemap_face**

Array indices for cube faces

4.27.2.4 **typedef enum CUarray_format_enum CUarray_format**

Array formats

4.27.2.5 **typedef enum CUcomputemode_enum CUcomputemode**

Compute Modes

4.27.2.6 **typedef struct CUctx_st* CUcontext**

CUDA context

4.27.2.7 **typedef enum CUctx_flags_enum CUctx_flags**

Context creation flags

4.27.2.8 **typedef struct CUDA_ARRAY3D_DESCRIPTOR_st CUDA_ARRAY3D_DESCRIPTOR**

3D array descriptor

4.27.2.9 **typedef struct CUDA_ARRAY_DESCRIPTOR_st CUDA_ARRAY_DESCRIPTOR**

Array descriptor

4.27.2.10 **typedef struct CUDA_MEMCPY2D_st CUDA_MEMCPY2D**

2D memory copy parameters

4.27.2.11 **typedef struct CUDA_MEMCPY3D_st CUDA_MEMCPY3D**

3D memory copy parameters

4.27.2.12 `typedef struct CUDA_MEMCPY3D_PEER_st CUDA_MEMCPY3D_PEER`

3D memory cross-context copy parameters

4.27.2.13 `typedef int CUdevice`

CUDA device

4.27.2.14 `typedef enum CUdevice_attribute_enum CUdevice_attribute`

Device properties

4.27.2.15 `typedef unsigned int CUdeviceptr`

CUDA device pointer

4.27.2.16 `typedef struct CUdevprop_st CUdevprop`

Legacy device properties

4.27.2.17 `typedef struct CUevent_st* CUevent`

CUDA event

4.27.2.18 `typedef enum CUevent_flags_enum CUevent_flags`

Event creation flags

4.27.2.19 `typedef enum CUfilter_mode_enum CUfilter_mode`

Texture reference filtering modes

4.27.2.20 `typedef enum CUfunc_cache_enum CUfunc_cache`

Function cache configurations

4.27.2.21 `typedef struct CUfunc_st* CUfunction`

CUDA function

4.27.2.22 `typedef enum CUfunction_attribute_enum CUfunction_attribute`

Function properties

4.27.2.23 `typedef enum CUgraphicsMapResourceFlags_enum CUgraphicsMapResourceFlags`

Flags for mapping and unmapping interop resources

4.27.2.24 `typedef enum CUgraphicsRegisterFlags_enum CUgraphicsRegisterFlags`

Flags to register a graphics resource

4.27.2.25 `typedef struct CUgraphicsResource_st* CUgraphicsResource`

CUDA graphics interop resource

4.27.2.26 `typedef enum CUjit_fallback_enum CUjit_fallback`

Cubin matching fallback strategies

4.27.2.27 `typedef enum CUjit_option_enum CUjit_option`

Online compiler options

4.27.2.28 `typedef enum CUjit_target_enum CUjit_target`

Online compilation targets

4.27.2.29 `typedef enum CULimit_enum CULimit`

Limits

4.27.2.30 `typedef enum CUMemorytype_enum CUMemorytype`

Memory types

4.27.2.31 `typedef struct CUMod_st* CUModule`

CUDA module

4.27.2.32 `typedef enum CUOutputMode_st CUOutputMode`

Profiler Output Modes

4.27.2.33 `typedef enum CUPointer_attribute_enum CUPointer_attribute`

Pointer information

4.27.2.34 `typedef enum cudaError_enum CUresult`

Error codes

4.27.2.35 `typedef struct CUstream_st* CUstream`

CUDA stream

4.27.2.36 `typedef struct CUsurfref_st* CUsurfref`

CUDA surface reference

4.27.2.37 `typedef struct CUtexref_st* CUtexref`

CUDA texture reference

4.27.3 Enumeration Type Documentation

4.27.3.1 `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

4.27.3.2 `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

4.27.3.3 `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

4.27.3.4 enum CUcomputemode_enum

Compute Modes

Enumerator:

- CU_COMPUTEMODE_DEFAULT*** Default compute mode (Multiple contexts allowed per device)
- CU_COMPUTEMODE_EXCLUSIVE*** Compute-exclusive-thread mode (Only one context used by a single thread 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)
- CU_COMPUTEMODE_EXCLUSIVE_PROCESS*** Compute-exclusive-process mode (Only one context used by a single process can be present on this device at a time)

4.27.3.5 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_SCHED_BLOCKING_SYNC*** Set blocking synchronization as default scheduling
 - CU_CTX_BLOCKING_SYNC*** Set blocking synchronization as default scheduling
- Deprecated**

- CU_CTX_MAP_HOST*** Support mapped pinned allocations
- CU_CTX_LMEM_RESIZE_TO_MAX*** Keep local memory allocation after launch
- CU_CTX_PRIMARY*** Initialize and return the primary context

4.27.3.6 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_PROFILER_DISABLED*** This indicates profiling APIs are called while application is running in visual profiler mode.

CUDA_ERROR_PROFILER_NOT_INITIALIZED This indicates profiling has not been initialized for this context. Call [cuProfilerInitialize\(\)](#) to resolve this.

CUDA_ERROR_PROFILER_ALREADY_STARTED This indicates profiler has already been started and probably [cuProfilerStart\(\)](#) is incorrectly called.

CUDA_ERROR_PROFILER_ALREADY_STOPPED This indicates profiler has already been stopped and probably [cuProfilerStop\(\)](#) is incorrectly called.

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

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

CUDA_ERROR_PEER_ACCESS_NOT_ENABLED This error indicates that a call to [cuMemPeerRegister](#) is trying to register memory from a context which has not had peer access enabled yet via [cuCtxEnablePeerAccess\(\)](#), or that [cuCtxDisablePeerAccess\(\)](#) is trying to disable peer access which has not been enabled yet.

CUDA_ERROR_PEER_MEMORY_ALREADY_REGISTERED This error indicates that a call to [cuMemPeerRegister](#) is trying to register already-registered memory.

CUDA_ERROR_PEER_MEMORY_NOT_REGISTERED This error indicates that a call to [cuMemPeerUnregister](#) is trying to unregister memory that has not been registered.

CUDA_ERROR_PRIMARY_CONTEXT_ACTIVE This error indicates that [cuCtxCreate](#) was called with the flag [CU_CTX_PRIMARY](#) on a device which already has initialized its primary context.

CUDA_ERROR_CONTEXT_IS_DESTROYED This error indicates that the context current to the calling thread has been destroyed using [cuCtxDestroy](#), or is a primary context which has not yet been initialized.

CUDA_ERROR_UNKNOWN This indicates that an unknown internal error has occurred.

4.27.3.7 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. Deprecated. Use instead `CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT`.

`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_LAYERED_WIDTH` Maximum 2D layered texture width

`CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT` Maximum 2D layered texture height

`CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS` Maximum layers in a 2D layered texture

`CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_WIDTH` Deprecated, use `CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH`

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_HEIGHT Deprecated, use **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT**

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_NUMSLICES Deprecated, use **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS**

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

CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE Peak memory clock frequency in kilohertz

CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH Global memory bus width in bits

CU_DEVICE_ATTRIBUTE_L2_CACHE_SIZE Size of L2 cache in bytes

CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR Maximum resident threads per multiprocessor

CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT Number of asynchronous engines

CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING Device uses shares a unified address space with the host

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH Maximum 1D layered texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS Maximum layers in a 1D layered texture

4.27.3.8 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

4.27.3.9 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

4.27.3.10 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

4.27.3.11 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.

4.27.3.12 enum CUGraphicsMapResourceFlags_enum

Flags for mapping and unmapping interop resources

4.27.3.13 enum CUGraphicsRegisterFlags_enum

Flags to register a graphics resource

4.27.3.14 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

4.27.3.15 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 fo 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 messsages from PTXAS that are informational in nature (the buffer size is specified via option [CU_JIT_INFO_LOG_BUFFER_SIZE_BYT](#)ES)

Option type: char*

***CU_JIT_INFO_LOG_BUFFER_SIZE_BYT*E**S 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_BYT](#)ES)

Option type: char*

***CU_JIT_ERROR_LOG_BUFFER_SIZE_BYT*E**S 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 [CUjitFallback_enum](#).

Option type: unsigned int for enumerated type [CUjitFallback_enum](#)

4.27.3.16 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

4.27.3.17 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

4.27.3.18 enum CUmemorytype_enum

Memory types

Enumerator:

- CU_MEMORYTYPE_HOST*** Host memory
- CU_MEMORYTYPE_DEVICE*** Device memory
- CU_MEMORYTYPE_ARRAY*** Array memory
- CU_MEMORYTYPE_UNIFIED*** Unified device or host memory

4.27.3.19 enum CUOutputMode_st

Profiler Output Modes

4.27.3.20 enum CUpointer_attribute_enum

Pointer information

Enumerator:

- CU_POINTER_ATTRIBUTE_CONTEXT*** The [CUcontext](#) on which a pointer was allocated or registered
- CU_POINTER_ATTRIBUTE_MEMORY_TYPE*** The [CUmemorytype](#) describing the physical location of a pointer
- CU_POINTER_ATTRIBUTE_DEVICE_POINTER*** The address at which a pointer's memory may be accessed on the device
- CU_POINTER_ATTRIBUTE_HOST_POINTER*** The address at which a pointer's memory may be accessed on the host

4.28 Initialization

Functions

- **CUresult cuInit (unsigned int Flags)**

Initialize the CUDA driver API.

4.28.1 Detailed Description

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

4.28.2 Function Documentation

4.28.2.1 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.

4.29 Version Management

Functions

- CUresult cuDriverGetVersion (int *driverVersion)

Returns the CUDA driver version.

4.29.1 Detailed Description

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

4.29.2 Function Documentation

4.29.2.1 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.

4.30 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.

4.30.1 Detailed Description

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

4.30.2 Function Documentation

4.30.2.1 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](#)

4.30.2.2 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](#)

4.30.2.3 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_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_TEXTURE1D_LAYERED_WIDTH**: Maximum 1D layered texture width;
- **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS**: Maximum layers in a 1D layered texture;
- **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH**: Maximum 2D layered texture width;
- **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT**: Maximum 2D layered texture height;
- **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS**: Maximum layers in a 2D layered texture;
- **CU_DEVICE_ATTRIBUTE_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;
- [CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE](#): Peak memory clock frequency in kilohertz;
- [CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH](#): Global memory bus width in bits;
- [CU_DEVICE_ATTRIBUTE_L2_CACHE_SIZE](#): Size of L2 cache in bytes. 0 if the device doesn't have L2 cache;
- [CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR](#): Maximum resident threads per multiprocessor;

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](#)

4.30.2.4 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](#)

4.30.2.5 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](#)

4.30.2.6 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](#)

4.30.2.7 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](#),

4.31 Context Management

Modules

- [Context Management \[DEPRECATED\]](#)

Functions

- **CUresult cuCtxCreate** (**CUcontext** *pctx, unsigned int flags, **CUdevice** dev)
Create a CUDA context.
- **CUresult cuCtxDestroy** (**CUcontext** ctx)
Destroy a CUDA context.
- **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 cuCtxGetCurrent** (**CUcontext** *pctx)
Returns the CUDA context bound to the calling CPU thread.
- **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 context on the current CPU thread.
- **CUresult cuCtxSetCacheConfig** (**CUfunc_cache** config)
Sets the preferred cache configuration for the current context.
- **CUresult cuCtxSetCurrent** (**CUcontext** ctx)
Binds the specified CUDA context to the calling CPU thread.
- **CUresult cuCtxSetLimit** (**CULimit** limit, size_t value)
Set resource limits.
- **CUresult cuCtxSynchronize** (void)
Block for a context's tasks to complete.

4.31.1 Detailed Description

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

4.31.2 Function Documentation

4.31.2.1 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 when done using the context. If a context is already current to the thread, it is supplanted by the newly created context and may be restored by a subsequent call to [cuCtxPopCurrent\(\)](#).

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

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

Deprecated

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

- [CU_CTX_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.
- [CU_CTX_PRIMARY](#): Instruct CUDA to initialize the primary context on the specified device. If the primary context is already initialized then this will return [CUDA_ERROR_PRIMARY_CONTEXT_ACTIVE](#).

Note to Linux users:

Context creation will fail with `CUDA_ERROR_UNKNOWN` if the compute mode of the device is `CU_COMPUTEMODE_PROHIBITED`. Similarly, context creation will also fail with `CUDA_ERROR_UNKNOWN` if the compute mode for the device is set to `CU_COMPUTEMODE_EXCLUSIVE` and there is already an active context on the device. The function `cuDeviceGetAttribute()` can be used with `CU_DEVICE_ATTRIBUTE_COMPUTE_MODE` to determine the compute mode of the device. The `nvidia-smi` tool can be used to set the compute mode for devices. Documentation for `nvidia-smi` can be obtained by passing a `-h` option to it.

Parameters:

`pctx` - Returned context handle of the new context

`flags` - Context creation flags

`dev` - Device to create context on

Returns:

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

Note:

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

See also:

`cuCtxDestroy`, `cuCtxGetApiVersion`, `cuCtxGetCacheConfig`, `cuCtxGetDevice`, `cuCtxGetLimit`, `cuCtxPopCurrent`, `cuCtxPushCurrent`, `cuCtxSetCacheConfig`, `cuCtxSetLimit`, `cuCtxSynchronize`

4.31.2.2 CUresult cuCtxDestroy (CUcontext *ctx*)

Destroys the CUDA context specified by `ctx`. The context `ctx` will be destroyed regardless of how many threads it is current to. It is the caller's responsibility to ensure that no API call is issued to `ctx` while `cuCtxDestroy()` is executing.

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

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:

`cuCtxCreate`, `cuCtxGetApiVersion`, `cuCtxGetCacheConfig`, `cuCtxGetDevice`, `cuCtxGetLimit`, `cuCtxPopCurrent`, `cuCtxPushCurrent`, `cuCtxSetCacheConfig`, `cuCtxSetLimit`, `cuCtxSynchronize`

4.31.2.3 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:

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

4.31.2.4 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:

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

4.31.2.5 CUresult cuCtxGetCurrent (CUcontext * *pctx*)

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

Parameters:

pctx - Returned context handle

Returns:

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

Note:

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

See also:

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

4.31.2.6 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:

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

4.31.2.7 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:

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

4.31.2.8 CUrresult cuCtxPopCurrent (CUcontext **pctx*)

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

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

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:

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

4.31.2.9 CUrresult 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\(\)](#).

Parameters:

ctx - Context to push

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

4.31.2.10 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:

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

4.31.2.11 CUresult cuCtxSetCurrent (CUcontext *ctx*)

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

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

Parameters:

ctx - Context to bind to the calling CPU thread

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:

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

4.31.2.12 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:

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

4.31.2.13 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_SCHED_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:

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

4.32 Context Management [DEPRECATED]

Functions

- CUresult [cuCtxAttach \(CUcontext *pctx, unsigned int flags\)](#)

Increment a context's usage-count.

- CUresult [cuCtxDetach \(CUcontext ctx\)](#)

Decrement a context's usage-count.

4.32.1 Detailed Description

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

4.32.2 Function Documentation

4.32.2.1 CUresult cuCtxAttach (CUcontext * *pctx*, unsigned int *flags*)

Deprecated

Note that this function is deprecated and should not be used.

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

Currently, the *flags* parameter must be 0.

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](#)

4.32.2.2 CUresult cuCtxDetach (CUcontext *ctx*)

Deprecated

Note that this function is deprecated and should not be used.

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

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:

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

4.33 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.

4.33.1 Detailed Description

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

4.33.2 Function Documentation

4.33.2.1 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](#)

4.33.2.2 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](#)

4.33.2.3 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`

4.33.2.4 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`

4.33.2.5 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, or a *fatbin* file as output by `nvcc` from toolchain 4.0 or later.

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](#)

4.33.2.6 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* or *fatbin* file, passing a *cubin* or *PTX* or *fatbin* file as a NULL-terminated text string, or incorporating a *cubin* or *fatbin* 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](#)

4.33.2.7 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* or *fatbin* file, passing a *cubin* or *PTX* or *fatbin* file as a NULL-terminated text string, or incorporating a *cubin* or *fatbin* 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](#)

4.33.2.8 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* and/or *PTX* files, all representing the same device code, but compiled and optimized for different architectures.

Prior to CUDA 4.0, there was no documented API for constructing and using fat binary objects by programmers. Starting with CUDA 4.0, fat binary objects can be constructed by providing the *-fatbin option* to **nvcc**. More information can be found in the **nvcc** document.

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](#)

4.33.2.9 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](#)

4.34 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 cuMemcpy (CUdeviceptr dst, CUdeviceptr src, size_t ByteCount)**
Copies 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 cuMemcpy3DPeer (const CUDA_MEMCPY3D_PEER *pCopy)**
Copies memory between contexts.

- **CUresult cuMemcpy3DPeerAsync** (const **CUDA_MEMCPY3D_PEER** *pCopy, **CUstream** hStream)
Copies memory between contexts asynchronously.
- **CUresult cuMemcpyAsync** (**CUdeviceptr** dst, **CUdeviceptr** src, **size_t** ByteCount, **CUstream** hStream)
Copies memory asynchronously.
- **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.

Copies memory from Host to Device.

- **CUresult cuMemcpyPeer** (**CUdeviceptr** dstDevice, **CUcontext** dstContext, **CUdeviceptr** srcDevice, **CUcontext** srcContext, **size_t** ByteCount)

Copies device memory between two contexts.

- **CUresult cuMemcpyPeerAsync** (**CUdeviceptr** dstDevice, **CUcontext** dstContext, **CUdeviceptr** srcDevice, **CUcontext** srcContext, **size_t** ByteCount, **CUstream** hStream)

Copies device memory between two contexts asynchronously.

- **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 cuMemHostRegister** (**void** *p, **size_t** bytesize, **unsigned int** Flags)

Registers an existing host memory range for use by CUDA.

- **CUresult cuMemHostUnregister** (**void** *p)

Unregisters a memory range that was registered with [cuMemHostRegister\(\)](#).

- **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.
- **CUresult cuProfilerInitialize** (**const char** *configFile, **const char** *outputFile, **CUOutputMode** outputMode)
Initialize the [profiling::cuProfilerInitialize\(\)](#).
- **CUresult cuProfilerStart** (**void**)
Start the [profiling::cuProfilerStart\(\)](#).
- **CUresult cuProfilerStop** (**void**)
Stop the [profiling::cuProfilerStop\(\)](#).

4.34.1 Detailed Description

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

4.34.2 Function Documentation

4.34.2.1 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; If the `CUDA_ARRAY3D_LAYERED` flag is set, then the CUDA array is a collection of layers, where `Depth` indicates the number of layers. Each layer is a 1D array if `Height` is 0, and a 2D array 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 = 0xa,
    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_LAYERED` to enable creation of layered CUDA arrays. If this flag is set, `Depth` specifies the number of layers, not the depth of a 3D array.
 - `CUDA_ARRAY3D_SURFACE_LDST` to enable surface references to be bound to the CUDA array. If this flag is not set, `cuSurfRefSetArray` will fail when attempting to bind the CUDA array to a surface reference.

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](#)

4.34.2.2 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](#)

4.34.2.3 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 = 0xa,
    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](#)

4.34.2.4 CUr esult 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](#)

4.34.2.5 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](#)

4.34.2.6 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](#), [cuMemsetD32](#), [cuMemsetD8](#), [cuMemsetD16](#)

4.34.2.7 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.

Note all host memory allocated using [cuMemHostAlloc\(\)](#) will automatically be immediately accessible to all contexts on all devices which support unified addressing (as may be queried using [CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING](#)). The device pointer that may be used to access this host memory from those contexts is always equal to the returned host pointer **pp*. See [Unified Addressing](#) for additional details.

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](#), [cuMemsetD32](#), [cuMemsetD8](#), [cuMemsetD16](#)

4.34.2.8 CUr esult 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](#)

4.34.2.9 CUr esult cuMemcpy (CUdeviceptr *dst*, CUdeviceptr *src*, size_t *ByteCount*)

Copies data between two pointers. *dst* and *src* are base pointers of the destination and source, respectively. *ByteCount* specifies the number of bytes to copy. Note that this function infers the type of the transfer (host to

host, host to device, device to device, or device to host) from the pointer values. This function is only allowed in contexts which support unified addressing. Note that this function is synchronous.

Parameters:

- dst* - Destination unified virtual address space pointer
- src* - Source unified virtual address space pointer
- ByteCount* - Size of memory copy in bytes

Returns:

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

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](#)

4.34.2.10 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](#) is defined as:

```
typedef enum CUmemorytype_enum {
    CU_MEMORYTYPE_HOST = 0x01,
    CU_MEMORYTYPE_DEVICE = 0x02,
    CU_MEMORYTYPE_ARRAY = 0x03,
    CU_MEMORYTYPE_UNIFIED = 0x04
} CUmemorytype;
```

If srcMemoryType is [CU_MEMORYTYPE_UNIFIED](#), srcDevice and srcPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. srcArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If srcMemoryType is [CU_MEMORYTYPE_HOST](#), srcHost and srcPitch specify the (host) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is [CU_MEMORYTYPE_DEVICE](#), srcDevice and srcPitch specify the (device) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is [CU_MEMORYTYPE_ARRAY](#), srcArray specifies the handle of the source data. srcHost, srcDevice and srcPitch are ignored.

If dstMemoryType is [CU_MEMORYTYPE_HOST](#), dstHost and dstPitch specify the (host) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is [CU_MEMORYTYPE_UNIFIED](#), dstDevice and dstPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. dstArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If dstMemoryType is [CU_MEMORYTYPE_DEVICE](#), dstDevice and dstPitch specify the (device) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is [CU_MEMORYTYPE_ARRAY](#), dstArray specifies the handle of the destination data. dstHost, dstDevice and dstPitch are ignored.

- srcXInBytes and srcY specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*) ((char*)srcHost+srcY*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
```

For CUDA arrays, srcXInBytes must be evenly divisible by the array element size.

- dstXInBytes and dstY specify the base address of the destination data for the copy.

For host pointers, the base address is

```
void* dstStart = (void*) ((char*)dstHost+dstY*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
```

For CUDA arrays, dstXInBytes must be evenly divisible by the array element size.

- WidthInBytes and Height specify the width (in bytes) and height of the 2D copy being performed.
- If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.

[cuMemcpy2D\(\)](#) returns an error if any pitch is greater than the maximum allowed ([CU_DEVICE_ATTRIBUTE_MAX_PITCH](#)). [cuMemAllocPitch\(\)](#) passes back pitches that always work with [cuMemcpy2D\(\)](#). On intra-device memory copies (device to device, CUDA array to device, CUDA array to CUDA array), [cuMemcpy2D\(\)](#) may fail for pitches not computed by [cuMemAllocPitch\(\)](#). [cuMemcpy2DUnaligned\(\)](#) does not have this restriction, but may run significantly slower in the cases where [cuMemcpy2D\(\)](#) would have returned an error code.

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](#)

4.34.2.11 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,
    CU_MEMORYTYPE_UNIFIED = 0x04
} CUmemorytype;

```

If srcMemoryType is [CU_MEMORYTYPE_HOST](#), srcHost and srcPitch specify the (host) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is [CU_MEMORYTYPE_UNIFIED](#), srcDevice and srcPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. srcArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If srcMemoryType is [CU_MEMORYTYPE_DEVICE](#), srcDevice and srcPitch specify the (device) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is [CU_MEMORYTYPE_ARRAY](#), srcArray specifies the handle of the source data. srcHost, srcDevice and srcPitch are ignored.

If dstMemoryType is [CU_MEMORYTYPE_UNIFIED](#), dstDevice and dstPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. dstArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If dstMemoryType is [CU_MEMORYTYPE_HOST](#), dstHost and dstPitch specify the (host) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is [CU_MEMORYTYPE_DEVICE](#), dstDevice and dstPitch specify the (device) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is [CU_MEMORYTYPE_ARRAY](#), dstArray specifies the handle of the destination data. dstHost, dstDevice and dstPitch are ignored.

- srcXInBytes and srcY specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*) ((char*)srcHost+srcY*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
```

For CUDA arrays, srcXInBytes must be evenly divisible by the array element size.

- dstXInBytes and dstY specify the base address of the destination data for the copy.

For host pointers, the base address is

```
void* dstStart = (void*) ((char*)dstHost+dstY*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
```

For CUDA arrays, dstXInBytes must be evenly divisible by the array element size.

- WidthInBytes and Height specify the width (in bytes) and height of the 2D copy being performed.
- If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.
- If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.
- If specified, srcHeight must be greater than or equal to Height + srcY, and dstHeight must be greater than or equal to Height + dstY.

[cuMemcpy2D\(\)](#) returns an error if any pitch is greater than the maximum allowed ([CU_DEVICE_ATTRIBUTE_MAX_PITCH](#)). [cuMemAllocPitch\(\)](#) passes back pitches that always work with [cuMemcpy2D\(\)](#). On intra-device memory copies (device to device, CUDA array to device, CUDA array to CUDA array), [cuMemcpy2D\(\)](#) may fail for pitches not computed by [cuMemAllocPitch\(\)](#). [cuMemcpy2DUnaligned\(\)](#) does not have this restriction, but may run significantly slower in the cases where [cuMemcpy2D\(\)](#) would have returned an error code.

[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

4.34.2.12 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,
    CU_MEMORYTYPE_UNIFIED = 0x04
} CUmemorytype;
```

If srcMemoryType is **CU_MEMORYTYPE_UNIFIED**, srcDevice and srcPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. srcArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If srcMemoryType is **CU_MEMORYTYPE_HOST**, srcHost and srcPitch specify the (host) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is **CU_MEMORYTYPE_DEVICE**, srcDevice and srcPitch specify the (device) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is **CU_MEMORYTYPE_ARRAY**, srcArray specifies the handle of the source data. srcHost, srcDevice and srcPitch are ignored.

If dstMemoryType is **CU_MEMORYTYPE_UNIFIED**, dstDevice and dstPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. dstArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If dstMemoryType is **CU_MEMORYTYPE_HOST**, dstHost and dstPitch specify the (host) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is **CU_MEMORYTYPE_DEVICE**, dstDevice and dstPitch specify the (device) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is **CU_MEMORYTYPE_ARRAY**, dstArray specifies the handle of the destination data. dstHost, dstDevice and dstPitch are ignored.

- srcXInBytes and srcY specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*) ((char*)srcHost+srcY*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
```

For CUDA arrays, srcXInBytes must be evenly divisible by the array element size.

- dstXInBytes and dstY specify the base address of the destination data for the copy.

For host pointers, the base address is

```
void* dstStart = (void*) ((char*)dstHost+dstY*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
```

For CUDA arrays, dstXInBytes must be evenly divisible by the array element size.

- WidthInBytes and Height specify the width (in bytes) and height of the 2D copy being performed.
- If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.

[cuMemcpy2D\(\)](#) returns an error if any pitch is greater than the maximum allowed ([CU_DEVICE_ATTRIBUTE_MAX_PITCH](#)). [cuMemAllocPitch\(\)](#) passes back pitches that always work with [cuMemcpy2D\(\)](#). On intra-device memory copies (device to device, CUDA array to device, CUDA array to CUDA array), [cuMemcpy2D\(\)](#) may fail for pitches not computed by [cuMemAllocPitch\(\)](#). [cuMemcpy2DUnaligned\(\)](#) does not have this restriction, but may run significantly slower in the cases where [cuMemcpy2D\(\)](#) would have returned an error code.

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](#)

4.34.2.13 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,
    CU_MEMORYTYPE_UNIFIED = 0x04
} CUmemorytype;

```

If srcMemoryType is [CU_MEMORYTYPE_UNIFIED](#), srcDevice and srcPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. srcArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If srcMemoryType is [CU_MEMORYTYPE_HOST](#), srcHost, srcPitch and srcHeight specify the (host) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. srcArray is ignored.

If srcMemoryType is [CU_MEMORYTYPE_DEVICE](#), srcDevice, srcPitch and srcHeight specify the (device) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. srcArray is ignored.

If srcMemoryType is [CU_MEMORYTYPE_ARRAY](#), srcArray specifies the handle of the source data. srcHost, srcDevice, srcPitch and srcHeight are ignored.

If dstMemoryType is [CU_MEMORYTYPE_UNIFIED](#), dstDevice and dstPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. dstArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If dstMemoryType is [CU_MEMORYTYPE_HOST](#), dstHost and dstPitch specify the (host) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. dstArray is ignored.

If dstMemoryType is [CU_MEMORYTYPE_DEVICE](#), dstDevice and dstPitch specify the (device) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. dstArray is ignored.

If dstMemoryType is [CU_MEMORYTYPE_ARRAY](#), dstArray specifies the handle of the destination data. dstHost, dstDevice, dstPitch and dstHeight are ignored.

- srcXInBytes, srcY and srcZ specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*) ((char*)srcHost + (srcZ*srcHeight+srcY)*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice + (srcZ*srcHeight+srcY)*srcPitch+srcXInBytes;
```

For CUDA arrays, srcXInBytes must be evenly divisible by the array element size.

- dstXInBytes, dstY and dstZ specify the base address of the destination data for the copy.

For host pointers, the base address is

```
void* dstStart = (void*) ((char*)dstHost + (dstZ*dstHeight+dstY)*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr dstStart = dstDevice + (dstZ*dstHeight+dstY)*dstPitch+dstXInBytes;
```

For CUDA arrays, dstXInBytes must be evenly divisible by the array element size.

- WidthInBytes, Height and Depth specify the width (in bytes), height and depth of the 3D copy being performed.
- If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.
- If specified, srcHeight must be greater than or equal to Height + srcY, and dstHeight must be greater than or equal to Height + dstY.

[cuMemcpy3D\(\)](#) returns an error if any pitch is greater than the maximum allowed ([CU_DEVICE_ATTRIBUTE_MAX_PITCH](#)).

The srcLOD and dstLOD members of the [CUDA_MEMCPY3D](#) structure must be set to 0.

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](#)

4.34.2.14 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,
    CU_MEMORYTYPE_UNIFIED = 0x04
} CUmemorytype;
```

If *srcMemoryType* is [CU_MEMORYTYPE_UNIFIED](#), *srcDevice* and *srcPitch* specify the (unified virtual address space) base address of the source data and the bytes per row to apply. *srcArray* is ignored. This value may be used only if unified addressing is supported in the calling context.

If srcMemoryType is **CU_MEMORYTYPE_HOST**, srcHost, srcPitch and srcHeight specify the (host) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. srcArray is ignored.

If srcMemoryType is **CU_MEMORYTYPE_DEVICE**, srcDevice, srcPitch and srcHeight specify the (device) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. srcArray is ignored.

If srcMemoryType is **CU_MEMORYTYPE_ARRAY**, srcArray specifies the handle of the source data. srcHost, srcDevice, srcPitch and srcHeight are ignored.

If dstMemoryType is **CU_MEMORYTYPE_UNIFIED**, dstDevice and dstPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. dstArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If dstMemoryType is **CU_MEMORYTYPE_HOST**, dstHost and dstPitch specify the (host) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. dstArray is ignored.

If dstMemoryType is **CU_MEMORYTYPE_DEVICE**, dstDevice and dstPitch specify the (device) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. dstArray is ignored.

If dstMemoryType is **CU_MEMORYTYPE_ARRAY**, dstArray specifies the handle of the destination data. dstHost, dstDevice, dstPitch and dstHeight are ignored.

- srcXInBytes, srcY and srcZ specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*) ((char*)srcHost + (srcZ*srcHeight+srcY)*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice + (srcZ*srcHeight+srcY)*srcPitch + srcXInBytes;
```

For CUDA arrays, srcXInBytes must be evenly divisible by the array element size.

- dstXInBytes, dstY and dstZ specify the base address of the destination data for the copy.

For host pointers, the base address is

```
void* dstStart = (void*) ((char*)dstHost + (dstZ*dstHeight+dstY)*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr dstStart = dstDevice + (dstZ*dstHeight+dstY)*dstPitch + dstXInBytes;
```

For CUDA arrays, dstXInBytes must be evenly divisible by the array element size.

- WidthInBytes, Height and Depth specify the width (in bytes), height and depth of the 3D copy being performed.
- If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.
- If specified, srcHeight must be greater than or equal to Height + srcY, and dstHeight must be greater than or equal to Height + dstY.

[cuMemcpy3D\(\)](#) returns an error if any pitch is greater than the maximum allowed ([CU_DEVICE_ATTRIBUTE_MAX_PITCH](#)).

[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](#)

4.34.2.15 CUresult cuMemcpy3DPeer (const CUDA_MEMCPY3D_PEER * *pCopy*)

Perform a 3D memory copy according to the parameters specified in *pCopy*. See the definition of the [CUDA_MEMCPY3D_PEER](#) structure for documentation of its parameters.

Note that this function is synchronous with respect to the host only if the source or destination memory is of type [CU_MEMORYTYPE_HOST](#). Note also that this copy is serialized with respect all pending and future asynchronous work in to the current context, the copy's source context, and the copy's destination context (use [cuMemcpy3DPeerAsync](#) to avoid this synchronization).

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:

[cuMemcpyDtoD](#), [cuMemcpyPeer](#), [cuMemcpyDtoDAsync](#), [cuMemcpyPeerAsync](#), [cuMemcpy3DPeerAsync](#)

4.34.2.16 CUresult cuMemcpy3DPeerAsync (const CUDA_MEMCPY3D_PEER * *pCopy*, CUstream *hStream*)

Perform a 3D memory copy according to the parameters specified in *pCopy*. See the definition of the [CUDA_MEMCPY3D_PEER](#) structure for documentation of its parameters.

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:

[cuMemcpyDtoD](#), [cuMemcpyPeer](#), [cuMemcpyDtoDAsync](#), [cuMemcpyPeerAsync](#), [cuMemcpy3DPeerAsync](#)

4.34.2.17 CUresult cuMemcpyAsync (CUdeviceptr *dst*, CUdeviceptr *src*, size_t *ByteCount*, CUstream *hStream*)

Copies data between two pointers. *dst* and *src* are base pointers of the destination and source, respectively. *ByteCount* specifies the number of bytes to copy. Note that this function infers the type of the transfer (host to host, host to device, device to device, or device to host) from the pointer values. This function is only allowed in contexts which support unified addressing. Note that this function is asynchronous and can optionally be associated to a stream by passing a non-zero *hStream* argument

Parameters:

dst - Destination unified virtual address space pointer

src - Source unified virtual address space pointer

ByteCount - Size of memory copy in bytes

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

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, cuMemcpyHtoAAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

4.34.2.18 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](#)

4.34.2.19 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](#)

4.34.2.20 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`

4.34.2.21 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`

4.34.2.22 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](#), [cuMemcpyHtoAAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

4.34.2.23 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](#), [cuMemcpyHtoAAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

4.34.2.24 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](#), [cuMemcpyHtoAAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD32](#), [cuMemsetD32Async](#)

4.34.2.25 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](#), [cuMemcpyHtoAAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D16](#), [cuMemsetD2D32](#), [cuMemsetD8](#), [cuMemsetD16](#), [cuMemsetD32](#)

4.34.2.26 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](#), [cuMemcpyHtoAAAsync](#), [cuMemcpyHtoD](#), [cuMemcpyHtoDAsync](#), [cuMemFree](#), [cuMemFreeHost](#), [cuMemGetAddressRange](#), [cuMemGetInfo](#), [cuMemHostAlloc](#), [cuMemHostGetDevicePointer](#), [cuMemsetD2D8](#), [cuMemsetD2D8Async](#), [cuMemsetD2D16](#), [cuMemsetD2D16Async](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#)

4.34.2.27 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](#)

4.34.2.28 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

4.34.2.29 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

4.34.2.30 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`

4.34.2.31 CUresult cuMemcpyPeer (CUdeviceptr *dstDevice*, CUcontext *dstContext*, CUdeviceptr *srcDevice*, CUcontext *srcContext*, size_t *ByteCount*)

Copies from device memory in one context to device memory in another context. `dstDevice` is the base device pointer of the destination memory and `dstContext` is the destination context. `srcDevice` is the base device pointer of the source memory and `srcContext` is the source pointer. `ByteCount` specifies the number of bytes to copy.

Note that this function is asynchronous with respect to the host, but serialized with respect all pending and future asynchronous work in to the current context, `srcContext`, and `dstContext` (use `cuMemcpyPeerAsync` to avoid this synchronization).

Parameters:

- `dstDevice` - Destination device pointer
- `dstContext` - Destination context
- `srcDevice` - Source device pointer
- `srcContext` - Source context
- `ByteCount` - Size of memory copy in bytes

Returns:

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

Note:

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

See also:

[cuMemcpyDtoD](#), [cuMemcpy3DPeer](#), [cuMemcpyDtoDAsync](#), [cuMemcpyPeerAsync](#), [cuMemcpy3DPeerAsync](#)

4.34.2.32 CUresult cuMemcpyPeerAsync (CUdeviceptr *dstDevice*, CUcontext *dstContext*, CUdeviceptr *srcDevice*, CUcontext *srcContext*, size_t *ByteCount*, CUstream *hStream*)

Copies from device memory in one context to device memory in another context. *dstDevice* is the base device pointer of the destination memory and *dstContext* is the destination context. *srcDevice* is the base device pointer of the source memory and *srcContext* is the source pointer. *ByteCount* specifies the number of bytes to copy. Note that this function is asynchronous with respect to the host and all work in other streams in other devices.

Parameters:

dstDevice - Destination device pointer
dstContext - Destination context
srcDevice - Source device pointer
srcContext - Source context
ByteCount - Size of memory copy in bytes
hStream - Stream identifier

Returns:

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

Note:

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

See also:

[cuMemcpyDtoD](#), [cuMemcpyPeer](#), [cuMemcpy3DPeer](#), [cuMemcpyDtoDAsync](#), [cuMemcpy3DPeerAsync](#)

4.34.2.33 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](#)

4.34.2.34 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](#)

4.34.2.35 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](#)

4.34.2.36 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](#)

4.34.2.37 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()`.

Note all host memory allocated using `cuMemHostAlloc()` will automatically be immediately accessible to all contexts on all devices which support unified addressing (as may be queried using `CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING`). Unless the flag `CU_MEMHOSTALLOC_WRITECOMBINED` is specified, the device pointer that may be used to access this host memory from those contexts is always equal to the returned host pointer `*pp`. If the flag `CU_MEMHOSTALLOC_WRITECOMBINED` is specified, then the function `cuMemHostGetDevicePointer()` must be used to query the device pointer, even if the context supports unified addressing. See [Unified Addressing](#) for additional details.

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`

4.34.2.38 CUresult cuMemHostGetDevicePointer (CUdeviceptr * *pdptra*, void * *p*, unsigned int *Flags*)

Passes back the device pointer *pdptra* 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:

pdptra - 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](#)

4.34.2.39 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](#)

4.34.2.40 CUresult cuMemHostRegister (void * *p*, size_t *bytesize*, unsigned int *Flags*)

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

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

- [CU_MEMHOSTREGISTER_PORTABLE](#): The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
- [CU_MEMHOSTREGISTER_DEVICEMAP](#): Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling [cuMemHostGetDevicePointer\(\)](#). This feature is available only on GPUs with compute capability greater than or equal to 1.1.

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

The CUDA context must have been created with the [CU_CTX_MAP_HOST](#) flag in order for the [CU_MEMHOSTREGISTER_DEVICEMAP](#) flag to have any effect.

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

The pointer *p* and size *bytesize* must be aligned to the host page size (4 KB).

The memory page-locked by this function must be unregistered with [cuMemHostUnregister\(\)](#).

Parameters:

- p* - Host pointer to memory to page-lock
bytesize - Size in bytes of the address range to page-lock
Flags - Flags for allocation request

Returns:

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

Note:

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

See also:

[cuMemHostUnregister](#), [cuMemHostGetFlags](#), [cuMemHostGetDevicePointer](#)

4.34.2.41 CUresult cuMemHostUnregister (void * *p*)

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

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

Parameters:

p - Host pointer to memory to unregister

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Note:

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

See also:

[cuMemHostRegister](#)

4.34.2.42 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](#)

4.34.2.43 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](#)

4.34.2.44 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](#)

4.34.2.45 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](#), [cuMemsetD2D32](#), [cuMemsetD2D32Async](#), [cuMemsetD8](#), [cuMemsetD8Async](#), [cuMemsetD16](#), [cuMemsetD16Async](#), [cuMemsetD32](#), [cuMemsetD32Async](#)

4.34.2.46 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](#)

4.34.2.47 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](#)

4.34.2.48 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](#)

4.34.2.49 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](#)

4.34.2.50 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](#), [cuMemsetD32](#), [cuMemsetD32Async](#)

4.34.2.51 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](#)

4.34.2.52 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](#)

4.34.2.53 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](#)

4.34.2.54 CUresult cuProfilerInitialize (const char * *configFile*, const char * *outputFile*, CUOutputMode *outputMode*)

[cuProfilerInitialize](#) is used to programmatically initialize the profiling. Using this API user can specify config file, output file and output file format. This API is generally used to profile different set of counters by looping the kernel launch. *configFile* parameter can be used to load new set of counters for profiling.

Configurations defined initially by environment variable settings are overwritten by [cuProfilerInitialize\(\)](#).

Limitation: Profiling APIs do not work when the application is running with any profiler tool. User must handle error CUDA_ERROR_PROFILER_DISABLED returned by profiler APIs if application is likely to be used with any profiler tool.

Parameters:

configFile - Name of the config file that lists the counters for profiling.

outputFile - Name of the outputFile where the profiling results will be stored.

outputMode - outputMode, can be CU_KEY_VALUE_PAIR or CU_CSV.

Returns:

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

Note:

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

See also:

[cuProfilerStart](#),
[cuProfilerStop](#)

4.34.2.55 CUresult cuProfilerStart (void)

cuProfilerStart/cuProfilerStop is used to programmatically control the profiling duration. APIs give added benefit of controlling the profiling granularity i.e. allows profiling to be done only on selective pieces of code.

[cuProfilerStart\(\)](#) can also be used to selectively start profiling on a particular context even when profiling is NOT ENABLED using environment variable. Profiling structures must be initialized using [cuProfilerInitialize\(\)](#) before making a call to [cuProfilerStart\(\)](#).

Returns:

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_PROFILER_DISABLED](#),
[CUDA_ERROR_PROFILER_ALREADY_STARTED](#)

Note:

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

See also:

[cuProfilerInitialize](#),
[cuProfilerStop](#)

4.34.2.56 CUresult cuProfilerStop (void)

This API can be used in conjunction with cuProfilerStart to selectively profile subsets of the CUDA program. [cuProfilerStop\(\)](#) can also be used to stop profiling for current context even when profiling is NOT ENABLED using environment variable. Profiling structures must be initialized using [cuProfilerInitialize\(\)](#) before making a call to [cuProfilerStop\(\)](#).

Returns:

`CUDA_SUCCESS`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_PROFILER_DISABLED`,
`CUDA_ERROR_PROFILER_ALREADY_STOPPED`

Note:

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

See also:

[cuProfilerInitialize](#),
[cuProfilerStart](#)

4.35 Unified Addressing

Functions

- [CUresult cuPointerGetAttribute](#) (void *data, [CUpointer_attribute](#) attribute, [CUdeviceptr](#) ptr)

Returns information about a pointer.

4.35.1 Detailed Description

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

4.35.2 Overview

CUDA devices can share a unified address space with the host. For these devices there is no distinction between a device pointer and a host pointer – the same pointer value may be used to access memory from the host program and from a kernel running on the device (with exceptions enumerated below).

4.35.3 Supported Platforms

Whether or not a device supports unified addressing may be queried by calling [cuDeviceGetAttribute\(\)](#) with the device attribute [CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING](#).

Unified addressing is automatically enabled in 64-bit processes on devices with compute capability greater than or equal to 2.0.

Unified addressing is not yet supported on Windows Vista or Windows 7 for devices that do not use the TCC driver model.

4.35.4 Looking Up Information from Pointer Values

It is possible to look up information about the memory which backs a pointer value. For instance, one may want to know if a pointer points to host or device memory. As another example, in the case of device memory, one may want to know on which CUDA device the memory resides. These properties may be queried using the function [cuPointerGetAttribute\(\)](#).

Because pointers are unique, it is not necessary to specify information about the pointers specified to the various copy functions in the CUDA API. The function [cuMemcpy\(\)](#) may be used to perform a copy between two pointers, ignoring whether they point to host or device memory (making [cuMemcpyHtoD\(\)](#), [cuMemcpyDtoD\(\)](#), and [cuMemcpyDtoH\(\)](#) unnecessary for devices supporting unified addressing). For multidimensional copies, the memory type [CU_MEMORYTYPE_UNIFIED](#) may be used to specify that the CUDA driver should infer the location of the pointer from its value.

4.35.5 Automatic Mapping of Host Allocated Host Memory

All host memory allocated in all contexts using [cuMemAllocHost\(\)](#) and [cuMemHostAlloc\(\)](#) is always directly accessible from all contexts on all devices that support unified addressing. This is the case regardless of whether or not the flags [CU_MEMHOSTALLOC_PORTABLE](#) and [CU_MEMHOSTALLOC_DEVICEMAP](#) are specified.

The pointer value through which allocated host memory may be accessed in kernels on all devices that support unified addressing is the same as the pointer value through which that memory is accessed on the host, so it is not necessary to call `cuMemHostGetDevicePointer()` to get the device pointer for these allocations.

Note that this is not the case for memory allocated using the flag `CU_MEMHOSTALLOC_WRITECOMBINED`, as discussed below.

4.35.6 Automatic Registration of Peer Memory

Upon enabling direct access from a context that supports unified addressing to another peer context that supports unified addressing using `cuCtxEnablePeerAccess()` all memory allocated in the peer context using `cuMemAlloc()` and `cuMemAllocPitch()` will immediately be accessible by the current context. In this case, explicitly registering allocations using `cuMemPeerRegister()` is not necessary. The device pointer value through which any peer memory may be accessed in the current context is the same pointer value through which that memory may be accessed in the peer context, so it is not necessary to call `cuMemPeerGetDevicePointer()` to get the device pointer for these allocations.

4.35.7 Exceptions, Disjoint Addressing

Not all memory may be accessed on devices through the same pointer value through which they are accessed on the host. These exceptions are host memory registered using `cuMemHostRegister()` and host memory allocated using the flag `CU_MEMHOSTALLOC_WRITECOMBINED`. For these exceptions, there exists a distinct host and device address for the memory. The device address is guaranteed to not overlap any valid host pointer range and is guaranteed to have the same value across all contexts that support unified addressing.

This device address may be queried using `cuMemHostGetDevicePointer()` when a context using unified addressing is current. Either the host or the unified device pointer value may be used to refer to this memory through `cuMemcpy()` and similar functions using the `CU_MEMORYTYPE_UNIFIED` memory type.

4.35.8 Function Documentation

4.35.8.1 CUrresult cuPointerGetAttribute (void * *data*, CUpointer_attribute *attribute*, CUdeviceptr *ptr*)

The supported attributes are:

- `CU_POINTER_ATTRIBUTE_CONTEXT`:

Returns in **data* the `CUcontext` in which *ptr* was allocated or registered. The type of *data* must be `CUcontext *`.

If *ptr* was not allocated by, mapped by, or registered with a `CUcontext` which uses unified virtual addressing then `CUDA_ERROR_INVALID_VALUE` is returned.

- `CU_POINTER_ATTRIBUTE_MEMORY_TYPE`:

Returns in **data* the physical memory type of the memory that *ptr* addresses as a `CUmemorytype` enumerated value. The type of *data* must be `unsigned int`.

If *ptr* addresses device memory then **data* is set to `CU_MEMORYTYPE_DEVICE`. The particular `CUdevice` on which the memory resides is the `CUdevice` of the `CUcontext` returned by the `CU_POINTER_ATTRIBUTE_CONTEXT` attribute of *ptr*.

If *ptr* addresses host memory then **data* is set to `CU_MEMORYTYPE_HOST`.

If *ptr* was not allocated by, mapped by, or registered with a `CUcontext` which uses unified virtual addressing then `CUDA_ERROR_INVALID_VALUE` is returned.

If the current [CUcontext](#) does not support unified virtual addressing then [CUDA_ERROR_INVALID_CONTEXT](#) is returned.

- [CU_POINTER_ATTRIBUTE_DEVICE_POINTER](#):

Returns in *data the device pointer value through which *ptr* may be accessed by kernels running in the current [CUcontext](#). The type of data must be [CUdeviceptr](#) *.

If there exists no device pointer value through which kernels running in the current [CUcontext](#) may access *ptr* then [CUDA_ERROR_INVALID_VALUE](#) is returned.

If there is no current [CUcontext](#) then [CUDA_ERROR_INVALID_CONTEXT](#) is returned.

Except in the exceptional disjoint addressing cases discussed below, the value returned in *data will equal the input value *ptr*.

- [CU_POINTER_ATTRIBUTE_HOST_POINTER](#):

Returns in *data the host pointer value through which *ptr* may be accessed by the host program. The type of data must be void **. If there exists no host pointer value through which the host program may directly access *ptr* then [CUDA_ERROR_INVALID_VALUE](#) is returned.

Except in the exceptional disjoint addressing cases discussed below, the value returned in *data will equal the input value *ptr*.

Note that for most allocations in the unified virtual address space the host and device pointer for accessing the allocation will be the same. The exceptions to this are

- user memory registered using [cuMemHostRegister](#)
- host memory allocated using [cuMemHostAlloc](#) with the [CU_MEMHOSTALLOC_WRITECOMBINED](#) flag
For these types of allocation there will exist separate, disjoint host and device addresses for accessing the allocation. In particular
 - The host address will correspond to an invalid unmapped device address (which will result in an exception if accessed from the device)
 - The device address will correspond to an invalid unmapped host address (which will result in an exception if accessed from the host). For these types of allocations, querying [CU_POINTER_ATTRIBUTE_HOST_POINTER](#) and [CU_POINTER_ATTRIBUTE_DEVICE_POINTER](#) may be used to retrieve the host and device addresses from either address.

Parameters:

data - Returned pointer attribute value

attribute - Pointer attribute to query

ptr - Pointer

Returns:

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

Note:

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

See also:

[cuMemAlloc](#), [cuMemFree](#), [cuMemAllocHost](#), [cuMemFreeHost](#), [cuMemHostAlloc](#), [cuMemHostRegister](#), [cuMemHostUnregister](#)

4.36 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.

4.36.1 Detailed Description

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

4.36.2 Function Documentation

4.36.2.1 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](#)

4.36.2.2 CUresult cuStreamDestroy (CUstream *hStream*)

Destroys the stream specified by *hStream*.

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

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](#)

4.36.2.3 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](#)

4.36.2.4 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_SCHED_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](#)

4.36.2.5 CUresult cuStreamWaitEvent (CUsream *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 event *hEvent* may be from a different context than *hStream*, in which case this function will perform cross-device synchronization.

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](#)

4.37 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.

4.37.1 Detailed Description

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

4.37.2 Function Documentation

4.37.2.1 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](#)

4.37.2.2 CUREsult cuEventDestroy (CUEvent hEvent)

Destroys the event specified by hEvent.

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

4.37.2.3 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](#)

4.37.2.4 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_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](#)

4.37.2.5 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\(\)](#).

It is necessary that *hEvent* and *hStream* be created on the same context.

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](#)

4.37.2.6 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](#)

4.38 Execution Control

Modules

- Execution Control [DEPRECATED]

Functions

- CUresult cuFuncGetAttribute (int *pi, CUfunction_attribute attrib, CUfunction hfunc)**
Returns information about a function.
- CUresult cuFuncSetCacheConfig (CUfunction hfunc, CUfunc_cache config)**
Sets the preferred cache configuration for a device function.
- CUresult cuLaunchKernel (CUfunction f, unsigned int gridDimX, unsigned int gridDimY, unsigned int gridDimZ, unsigned int blockDimX, unsigned int blockDimY, unsigned int blockDimZ, unsigned int sharedMemBytes, CUstream hStream, void **kernelParams, void **extra)**
Launches a CUDA function.

4.38.1 Detailed Description

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

4.38.2 Function Documentation

4.38.2.1 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:

[cuCtxGetCacheConfig](#), [cuCtxSetCacheConfig](#), [cuFuncSetCacheConfig](#), [cuLaunchKernel](#)

4.38.2.2 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](#), [cuFuncGetAttribute](#), [cuLaunchKernel](#)

4.38.2.3 CUresult cuLaunchKernel (CUfunction *f*, unsigned int *gridDimX*, unsigned int *gridDimY*, unsigned int *gridDimZ*, unsigned int *blockDimX*, unsigned int *blockDimY*, unsigned int *blockDimZ*, unsigned int *sharedMemBytes*, CUstream *hStream*, void ** *kernelParams*, void ** *extra*)

Invokes the kernel *f* on a *gridDimX* x *gridDimY* x *gridDimZ* grid of blocks. Each block contains *blockDimX* x *blockDimY* x *blockDimZ* threads.

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

[cuLaunchKernel\(\)](#) can optionally be associated to a stream by passing a non-zero *hStream* argument.

Kernel parameters to *f* can be specified in one of two ways:

1) Kernel parameters can be specified via *kernelParams*. If *f* has *N* parameters, then *kernelParams* needs to be an array of *N* pointers. Each of *kernelParams[0]* through *kernelParams[N-1]* must point to a region of memory from which the actual kernel parameter will be copied. The number of kernel parameters and their offsets and sizes do not need to be specified as that information is retrieved directly from the kernel's image.

2) Kernel parameters can also be packaged by the application into a single buffer that is passed in via the *extra* parameter. This places the burden on the application of knowing each kernel parameter's size and alignment/padding within the buffer. Here is an example of using the *extra* parameter in this manner:

```
size_t argBufferSize;
char argBuffer[256];

// populate argBuffer and argBufferSize

void *config[] = {
    CU_LAUNCH_PARAM_BUFFER_POINTER, argBuffer,
    CU_LAUNCH_PARAM_BUFFER_SIZE,    &argBufferSize,
    CU_LAUNCH_PARAM_END
};
status = cuLaunchKernel(f, gx, gy, gz, bx, by, bz, sh, s, NULL, config);
```

The *extra* parameter exists to allow [cuLaunchKernel](#) to take additional less commonly used arguments. *extra* specifies a list of names of extra settings and their corresponding values. Each extra setting name is immediately followed by the corresponding value. The list must be terminated with either NULL or [CU_LAUNCH_PARAM_END](#).

- [CU_LAUNCH_PARAM_END](#), which indicates the end of the *extra* array;
- [CU_LAUNCH_PARAM_BUFFER_POINTER](#), which specifies that the next value in *extra* will be a pointer to a buffer containing all the kernel parameters for launching kernel *f*;
- [CU_LAUNCH_PARAM_BUFFER_SIZE](#), which specifies that the next value in *extra* will be a pointer to a *size_t* containing the size of the buffer specified with [CU_LAUNCH_PARAM_BUFFER_POINTER](#);

The error [CUDA_ERROR_INVALID_VALUE](#) will be returned if kernel parameters are specified with both *kernelParams* and *extra* (i.e. both *kernelParams* and *extra* are non-NULL).

Calling [cuLaunchKernel\(\)](#) sets persistent function state that is the same as function state set through the following deprecated APIs:

[cuFuncSetBlockShape\(\)](#) [cuFuncSetSharedSize\(\)](#) [cuParamSetSize\(\)](#) [cuParamSeti\(\)](#) [cuParamSetf\(\)](#) [cuParamSetv\(\)](#)

When the kernel *f* is launched via [cuLaunchKernel\(\)](#), the previous block shape, shared size and parameter info associated with *f* is overwritten.

Note that to use [cuLaunchKernel\(\)](#), the kernel *f* must either have been compiled with toolchain version 3.2 or later so that it will contain kernel parameter information, or have no kernel parameters. If either of these conditions is not met, then [cuLaunchKernel\(\)](#) will return [CUDA_ERROR_INVALID_IMAGE](#).

Parameters:

f - Kernel to launch
gridDimX - Width of grid in blocks
gridDimY - Height of grid in blocks
gridDimZ - Depth of grid in blocks
blockDimX - X dimension of each thread block
blockDimY - Y dimension of each thread block
blockDimZ - Z dimension of each thread block
sharedMemBytes - Dynamic shared-memory size per thread block in bytes
hStream - Stream identifier
kernelParams - Array of pointers to kernel parameters
extra - Extra options

Returns:

[CUDA_SUCCESS](#), [CUDA_ERROR_DEINITIALIZED](#), [CUDA_ERROR_NOT_INITIALIZED](#), [CUDA_ERROR_INVALID_CONTEXT](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_INVALID_IMAGE](#), [CUDA_ERROR_INVALID_VALUE](#), [CUDA_ERROR_LAUNCH_FAILED](#), [CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES](#), [CUDA_ERROR_LAUNCH_TIMEOUT](#), [CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING](#), [CUDA_ERROR_SHARED_OBJECT_INIT_FAILED](#)

Note:

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

See also:

[cuCtxGetCacheConfig](#), [cuCtxSetCacheConfig](#), [cuFuncSetCacheConfig](#), [cuFuncGetAttribute](#),

4.39 Execution Control [DEPRECATED]

Functions

- **CUresult cuFuncSetBlockShape (CUfunction hfunc, int x, int y, int z)**
Sets the block-dimensions for the 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 cuParamSetTexRef (CUfunction hfunc, int texunit, CUtexref hTexRef)**
Adds a texture-reference to the function's argument list.
- **CUresult cuParamSetv (CUfunction hfunc, int offset, void *ptr, unsigned int numbytes)**
Adds arbitrary data to the function's argument list.

4.39.1 Detailed Description

This section describes the deprecated execution control functions of the low-level CUDA driver application programming interface.

4.39.2 Function Documentation

4.39.2.1 CUresult cuFuncSetBlockShape (CUfunction *hfunc*, int *x*, int *y*, int *z*)

Deprecated

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](#), [cuLaunchKernel](#)

4.39.2.2 CUresult cuFuncSetSharedSize (CUfunction *hfunc*, unsigned int *bytes*)

Deprecated

Sets through *bytes* the amount of dynamic shared memory that will be available to each thread block when the kernel given by *hfunc* is launched.

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](#), [cuLaunchKernel](#)

4.39.2.3 CUresult cuLaunch (CUfunction *f*)

Deprecated

Invokes the kernel f on a $1 \times 1 \times 1$ 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](#), [cuLaunchKernel](#)

4.39.2.4 CUresult cuLaunchGrid (CUfunction f , int $grid_width$, int $grid_height$)

Deprecated

Invokes the kernel f on a $grid_width \times grid_height$ 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](#), [cuLaunchKernel](#)

4.39.2.5 CUresult cuLaunchGridAsync (CUfunction *f*, int *grid_width*, int *grid_height*, CUstream *hStream*)

Deprecated

Invokes the kernel *f* on a *grid_width* x *grid_height* grid of blocks. Each block contains the number of threads specified by a previous call to [cuFuncSetBlockShape\(\)](#).

[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_HANDLE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_LAUNCH_FAILED, CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES, CUDA_ERROR_LAUNCH_TIMEOUT, CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

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](#), [cuLaunchKernel](#)

4.39.2.6 CUresult cuParamSetf (CUfunction *hfunc*, int *offset*, float *value*)

Deprecated

Sets a floating-point parameter that will be specified the next time the kernel corresponding to *hfunc* will be invoked. *offset* is a byte offset.

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](#), [cuParamGetSize](#), [cuParamSeti](#), [cuParamSetv](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#), [cuLaunchKernel](#)

4.39.2.7 CUresult cuParamSeti (CUfunction *hfunc*, int *offset*, unsigned int *value*)

Deprecated

Sets an integer parameter that will be specified the next time the kernel corresponding to *hfunc* will be invoked. *offset* is a byte offset.

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](#), [cuParamGetSize](#), [cuParamSetf](#), [cuParamSetv](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#), [cuLaunchKernel](#)

4.39.2.8 CUresult cuParamGetSize (CUfunction *hfunc*, unsigned int *numbytes*)

Deprecated

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](#), [cuLaunchKernel](#)

4.39.2.9 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.

4.39.2.10 CUresult cuParamSetv (CUfunction *hfunc*, int *offset*, void **ptr*, unsigned int *numbytes*)**Deprecated**

Copies an arbitrary amount of data (specified in *numbytes*) from *ptr* into the parameter space of the kernel corresponding to *hfunc*. *offset* is a byte offset.

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](#), [cuParamGetSize](#), [cuParamSetf](#), [cuParamSeti](#), [cuLaunch](#), [cuLaunchGrid](#), [cuLaunchGridAsync](#), [cuLaunchKernel](#)

4.40 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.

4.40.1 Detailed Description

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

4.40.2 Function Documentation

4.40.2.1 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](#)

4.40.2.2 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](#)

4.40.2.3 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](#)

4.40.2.4 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](#)

4.40.2.5 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](#)

4.40.2.6 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](#)

4.40.2.7 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](#)

4.40.2.8 CUrresult 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](#)

4.40.2.9 CUrresult 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. Also, if the flag, [CU_TRSF_NORMALIZED_COORDINATES](#), is not set, the only supported address mode is [CU_TR_ADDRESS_MODE_CLAMP](#).

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](#)

4.40.2.10 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](#)

4.40.2.11 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

4.40.2.12 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

4.40.2.13 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`

4.41 Texture Reference Management [DEPRECATED]

Functions

- **CUresult cuTexRefCreate (CUtexref *pTexRef)**
Creates a texture reference.
- **CUresult cuTexRefDestroy (CUtexref hTexRef)**
Destroys a texture reference.

4.41.1 Detailed Description

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

4.41.2 Function Documentation

4.41.2.1 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](#)

4.41.2.2 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](#)

4.42 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.

4.42.1 Detailed Description

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

4.42.2 Function Documentation

4.42.2.1 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](#)

4.42.2.2 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](#)

4.43 Peer Context Memory Access

Functions

- **CUresult cuCtxDisablePeerAccess (CUcontext peerContext)**
Disables direct access to memory allocations in a peer context and unregisters any registered allocations.
- **CUresult cuCtxEnablePeerAccess (CUcontext peerContext, unsigned int Flags)**
Enables direct access to memory allocations in a peer context.
- **CUresult cuDeviceCanAccessPeer (int *canAccessPeer, CUdevice dev, CUdevice peerDev)**
Queries if a device may directly access a peer device's memory.
- **CUresult cuMemPeerGetDevicePointer (CUdeviceptr *pdptr, CUdeviceptr peerPointer, CUcontext peerContext, unsigned int Flags)**
Retrieves a device pointer through which a peer context's registered memory may be directly accessed.
- **CUresult cuMemPeerRegister (CUdeviceptr peerPointer, CUcontext peerContext, unsigned int Flags)**
Registers an existing memory allocation in a peer context for direct access from the current context.
- **CUresult cuMemPeerUnregister (CUdeviceptr peerPointer, CUcontext peerContext)**
Unregisters a peer context's memory allocation, disabling direct access.

4.43.1 Detailed Description

This section describes the direct peer context memory access functions of the low-level CUDA driver application programming interface.

4.43.2 Function Documentation

4.43.2.1 CUresult cuCtxDisablePeerAccess (CUcontext *peerContext*)

Disables registering memory from *peerContext* for direct access in the current context. If there are any allocations from *peerContext* which were registered in the current context using [cuMemPeerRegister\(\)](#) then these allocations will be automatically unregistered.

Returns [CUDA_ERROR_PEER_ACCESS_NOT_ENABLED](#) if direct peer access has not yet been enabled from *peerContext* to the current context.

Returns [CUDA_ERROR_INVALID_CONTEXT](#) if there is no current context, or if *peerContext* is not a valid context.

Parameters:

peerContext - Peer context to disable direct access to

Returns:

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

Note:

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

See also:

[cuDeviceCanAccessPeer](#), [cuCtxEnablePeerAccess](#), [cuMemPeerRegister](#), [cuMemPeerUnregister](#), [cuMemPeerGetDevicePointer](#)

4.43.2.2 CUresult cuCtxEnablePeerAccess (CUcontext *peerContext*, unsigned int *Flags*)

Enables registering memory from *peerContext* for direct access in the current context. On success, allocations from *peerContext* may be registered for access in the current context using [cuMemPeerRegister\(\)](#). Registering peer memory will be possible until it is explicitly disabled using [cuCtxDisablePeerAccess\(\)](#), or either the current context or *peerContext* is destroyed.

If both the current context and *peerContext* are on devices which support unified addressing (as may be queried using [CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING](#)), then on success all allocations from *peerContext* will immediately be accessible by the current context. In this case, explicitly sharing allocations using [cuMemPeerRegister\(\)](#) is not necessary. See [Unified Addressing](#) for additional details.

Note that access granted by this call is unidirectional and that in order to access memory from the current context in *peerContext*, a separate symmetric call to [cuCtxEnablePeerAccess\(\)](#) is required.

Returns [CUDA_ERROR_INVALID_DEVICE](#) if [cuDeviceCanAccessPeer\(\)](#) indicates that the [CUdevice](#) of the current context cannot directly access memory from the [CUdevice](#) of *peerContext*.

Returns [CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED](#) if direct access of *peerContext* from the current context has already been enabled.

Returns [CUDA_ERROR_INVALID_CONTEXT](#) if there is no current context, *peerContext* is not a valid context, or if the current context is *peerContext*.

Returns [CUDA_ERROR_INVALID_VALUE](#) if *Flags* is not 0.

Parameters:

peerContext - Peer context to enable direct access to from the current context

Flags - Reserved for future use and must be set to 0

Returns:

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

Note:

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

See also:

[cuDeviceCanAccessPeer](#), [cuCtxDisablePeerAccess](#), [cuMemPeerRegister](#), [cuMemPeerUnregister](#), [cuMemPeerGetDevicePointer](#)

4.43.2.3 CUresult cuDeviceCanAccessPeer (int * *canAccessPeer*, CUdevice *dev*, CUdevice *peerDev*)

Returns in **canAccessPeer* a value of 1 if contexts on *dev* are capable of directly accessing memory from contexts on *peerDev* and 0 otherwise. If direct access of *peerDev* from *dev* is possible, then access may be enabled on two specific contexts by calling [cuCtxEnablePeerAccess\(\)](#).

Parameters:

canAccessPeer - Returned access capability

dev - Device from which allocations on *peerDev* are to be directly accessed.

peerDev - Device on which the allocations to be directly accessed by *dev* reside.

Returns:

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

Note:

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

See also:

[cuCtxEnablePeerAccess](#), [cuCtxDisablePeerAccess](#), [cuMemPeerRegister](#), [cuMemPeerUnregister](#), [cuMemPeerGetDevicePointer](#)

4.43.2.4 CUresult cuMemPeerGetDevicePointer (CUdeviceptr * *pdptra*, CUdeviceptr *peerPointer*, CUcontext *peerContext*, unsigned int *Flags*)

Retrieves a device pointer through which memory registered using [cuMemPeerRegister\(\)](#) may be directly accessed in the current context. On success, returns in **pdptra* a device pointer through which the memory at *peerPointer* in *peerContext* may be accessed through kernels and copies in the current context.

If both the current context and *peerContext* are on devices which support unified addressing (as may be queried using [CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING](#)), then the pointer to access *peerPointer* from *peerContext* in the current context will be *peerPointer* itself, and querying this pointer value is not necessary. See [Unified Addressing](#) for additional details.

Returns [CUDA_ERROR_NOT_MAPPED_AS_POINTER](#) if *peerPointer* in *peerContext* was not registered with the flag [CU_MEMPEERREGISTER_DEVICEMAP](#).

Returns [CUDA_ERROR_PEER_MEMORY_NOT_REGISTERED](#) if *peerPointer* in *peerContext* has not been registered yet.

Returns [CUDA_ERROR_INVALID_VALUE](#) if *peerPointer* is not a pointer value return by [cuMemAlloc\(\)](#) or [cuMemAllocPitch\(\)](#) in *peerContext*, or if *Flags* is not 0.

Returns [CUDA_ERROR_INVALID_CONTEXT](#) if there is no current context, or if *peerContext* is not a valid context.

Parameters:

pdptra - Pointer in peer context to unregister.

peerPointer - Pointer in peer context to unregister.

peerContext - Peer context in which *peerPointer* was allocated.

Flags - Reserved for future use and must be set to 0

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_PEER_MEMORY_NOT_REGISTERED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_CONTEXT,

Note:

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

See also:

[cuDeviceCanAccessPeer](#), [cuCtxEnablePeerAccess](#), [cuCtxDisablePeerAccess](#), [cuMemPeerRegister](#), [cuMemPeerUnregister](#)

4.43.2.5 CUresult cuMemPeerRegister (CUdeviceptr *peerPointer*, CUcontext *peerContext*, unsigned int *Flags*)

Registers the memory associated with *peerPointer* in *peerContext* for direct access in the current context. Memory that has been registered will be read and written directly when accessed through [cuMemcpyPeer\(\)](#) and its variants (as opposed to being staged through host memory). Memory which has been registered with the flag [CU_MEMPEERREGISTER_DEVICEMAP](#) may be accessed directly by pointers in kernels.

This memory will remain accessible until explicitly unregistered via [cuMemPeerUnregister\(\)](#) or implicitly unregistered when peer access to *peerContext* is disabled by [cuCtxDisablePeerAccess\(\)](#) or context destruction.

Note that explicitly registering memory is not required for direct access between contexts on devices which support unified addressing (as may be queried using [CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING](#)). In such cases all memory allocated in *peerContext* is automatically made accessible to the current context by [cuCtxEnablePeerAccess\(\)](#). See [Unified Addressing](#) for additional details.

The *Flags* parameter enables different options to be specified that affect the registration, as follows.

- [CU_MEMPEERREGISTER_DEVICEMAP](#): Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling [cuMemPeerGetDevicePointer\(\)](#).

Returns [CUDA_ERROR_PEER_ACCESS_NOT_ENABLED](#) if direct access has not yet been enabled from *peerContext* to the current context.

Returns [CUDA_ERROR_PEER_MEMORY_ALREADY_REGISTERED](#) if *peerPointer* from *peerContext* has already been registered with the current context.

Returns [CUDA_ERROR_INVALID_VALUE](#) if *peerPointer* is not a pointer value return by [cuMemAlloc\(\)](#) or [cuMemAllocPitch\(\)](#) in *peerContext*.

Returns [CUDA_ERROR_INVALID_CONTEXT](#) if there is no current context, or if *peerContext* is not a valid context.

Parameters:

peerPointer - Pointer in peer context to register for direct access in the current context

peerContext - Peer context in which *peerPointer* was allocated

Flags - Flags affecting allocation registration.

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_PEER_ACCESS_NOT_ENABLED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_CONTEXT,

Note:

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

See also:

[cuDeviceCanAccessPeer](#), [cuCtxEnablePeerAccess](#), [cuCtxDisablePeerAccess](#), [cuMemPeerUnregister](#), [cuMemPeerGetDevicePointer](#)

4.43.2.6 CURESULT cuMemPeerUnregister (CUdeviceptr *peerPointer*, CUcontext *peerContext*)

Unregisters peer memory which was registered with the current context using [cuMemPeerRegister\(\)](#).

Returns [CUDA_ERROR_PEER_MEMORY_NOT_REGISTERED](#) if *peerPointer* in *peerContext* has not been registered yet.

Returns [CUDA_ERROR_INVALID_VALUE](#) if *peerPointer* is not a pointer value return by [cuMemAlloc\(\)](#) or [cuMemAllocPitch\(\)](#) in *peerContext*.

Returns [CUDA_ERROR_INVALID_CONTEXT](#) if there is no current context, or if *peerContext* is not a valid context.

Parameters:

peerPointer - Pointer in *peerContext* to unregister.

peerContext - Peer context in which *peerPointer* was allocated.

Returns:

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

Note:

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

See also:

[cuDeviceCanAccessPeer](#), [cuCtxEnablePeerAccess](#), [cuCtxDisablePeerAccess](#), [cuMemPeerRegister](#), [cuMemPeerGetDevicePointer](#)

4.44 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.

4.44.1 Detailed Description

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

4.44.2 Function Documentation

4.44.2.1 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](#)

4.44.2.2 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](#)

4.44.2.3 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](#)

4.44.2.4 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](#)

4.44.2.5 CUrresult 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](#)

4.44.2.6 CUrresult cuGraphicsUnregisterResource (*CUgraphicsResource resource*)

Registers 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](#)

4.45 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.

4.45.1 Detailed Description

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

4.45.2 Function Documentation

4.45.2.1 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](#)

4.45.2.2 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 register flags *Flags* specify the intended usage, as follows:

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

Parameters:

pCudaResource - Pointer to the returned object handle

buffer - name of buffer object to be registered

Flags - Register flags

Returns:

[CUDA_SUCCESS](#), [CUDA_ERROR_INVALID_HANDLE](#), [CUDA_ERROR_ALREADY_MAPPED](#), [CUDA_ERROR_INVALID_CONTEXT](#),

Note:

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

See also:

[cuGLCtxCreate](#), [cuGraphicsUnregisterResource](#), [cuGraphicsMapResources](#), [cuGraphicsResourceGetMappedPointer](#)

4.45.2.3 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 register flags *Flags* specify the intended usage, as follows:

- CU_GRAPHICS_REGISTER_FLAGS_NONE: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- CU_GRAPHICS_REGISTER_FLAGS_READ_ONLY: Specifies that CUDA will not write to this resource.

- CU_GRAPHICS_REGISTER_FLAGS_WRITE_DISCARD: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.
- CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST: Specifies that CUDA will bind this resource to a surface reference.

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 - Register 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](#)

4.45.2.4 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](#)

4.46 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.

4.46.1 Detailed Description

This section describes deprecated OpenGL interoperability functionality.

4.46.2 Typedef Documentation

4.46.2.1 `typedef enum CUGLmap_flags_enum CUGLmap_flags`

Flags to map or unmap a resource

4.46.3 Enumeration Type Documentation

4.46.3.1 enum CUGLmap_flags_enum

Flags to map or unmap a resource

4.46.4 Function Documentation

4.46.4.1 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`

4.46.4.2 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](#)

4.46.4.3 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](#)

4.46.4.4 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](#)

4.46.4.5 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](#)

4.46.4.6 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](#)

4.46.4.7 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](#)

4.46.4.8 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](#)

4.47 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 cud-aDeviceCount, 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.

4.47.1 Detailed Description

This section describes the Direct3D 9 interoperability functions of the low-level CUDA driver application programming interface.

4.47.2 Typedef Documentation

4.47.2.1 **typedef enum CUd3d9DeviceList_enum CUd3d9DeviceList**

CUDA devices corresponding to a D3D9 device

4.47.3 Enumeration Type Documentation

4.47.3.1 **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

4.47.4 Function Documentation

4.47.4.1 **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](#)

4.47.4.2 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](#)

4.47.4.3 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](#)

4.47.4.4 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](#)

4.47.4.5 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](#)

4.47.4.6 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 valid values for this parameter are

- CU_GRAPHICS_REGISTER_FLAGS_NONE: Specifies no hints about how this resource will be used.
- CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST: Specifies that CUDA will bind this resource to a surface reference.

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](#)

4.48 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.

4.48.1 Detailed Description

This section describes deprecated Direct3D 9 interoperability functionality.

4.48.2 Typedef Documentation

4.48.2.1 `typedef enum CUd3d9map_flags_enum CUd3d9map_flags`

Flags to map or unmap a resource

4.48.2.2 `typedef enum CUd3d9register_flags_enum CUd3d9register_flags`

Flags to register a resource

4.48.3 Enumeration Type Documentation

4.48.3.1 `enum CUd3d9map_flags_enum`

Flags to map or unmap a resource

4.48.3.2 `enum CUd3d9register_flags_enum`

Flags to register a resource

4.48.4 Function Documentation

4.48.4.1 `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`

4.48.4.2 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](#)

4.48.4.3 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](#)

4.48.4.4 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](#)

4.48.4.5 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](#)

4.48.4.6 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](#)

4.48.4.7 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](#)

4.48.4.8 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](#)

4.48.4.9 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](#)

4.48.4.10 CUresult cuD3D9UnregisterResource (*IDirect3DResource9 * pResource*)

Deprecated

This function is deprecated as of Cuda 3.0.

Registers 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](#)

4.49 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 cu-
daDeviceCount, 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.

4.49.1 Detailed Description

This section describes the Direct3D 10 interoperability functions of the low-level CUDA driver application programming interface.

4.49.2 Typedef Documentation

4.49.2.1 **typedef enum CUd3d10DeviceList_enum CUd3d10DeviceList**

CUDA devices corresponding to a D3D10 device

4.49.3 Enumeration Type Documentation

4.49.3.1 **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

4.49.4 Function Documentation

4.49.4.1 **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](#)

4.49.4.2 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](#)

4.49.4.3 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](#)

4.49.4.4 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](#)

4.49.4.5 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](#)

4.49.4.6 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 valid values for this parameter are

- CU_GRAPHICS_REGISTER_FLAGS_NONE: Specifies no hints about how this resource will be used.
- CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST: Specifies that CUDA will bind this resource to a surface reference.

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](#)

4.50 Direct3D 10 Interoperability [DEPRECATED]

Typedefs

- `typedef enum CUD3D10map_flags_enum CUD3D10map_flags`
- `typedef enum CUD3D10register_flags_enum CUD3D10register_flags`

Enumerations

- `enum CUD3D10map_flags_enum`
- `enum CUD3D10register_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.

4.50.1 Detailed Description

This section describes deprecated Direct3D 10 interoperability functionality.

4.50.2 Typedef Documentation

4.50.2.1 `typedef enum CUD3D10map_flags_enum CUD3D10map_flags`

Flags to map or unmap a resource

4.50.2.2 `typedef enum CUD3D10register_flags_enum CUD3D10register_flags`

Flags to register a resource

4.50.3 Enumeration Type Documentation

4.50.3.1 `enum CUD3D10map_flags_enum`

Flags to map or unmap a resource

4.50.3.2 `enum CUD3D10register_flags_enum`

Flags to register a resource

4.50.4 Function Documentation

4.50.4.1 `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](#)

4.50.4.2 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](#)

4.50.4.3 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](#)

4.50.4.4 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 * *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 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](#)

4.50.4.5 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](#)

4.50.4.6 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](#)

4.50.4.7 CUresult cuD3D10ResourceGetSurfaceDimensions (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 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](#)

4.50.4.8 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](#)

4.50.4.9 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](#)

4.50.4.10 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](#)

4.51 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 cu-
daDeviceCount, 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.

4.51.1 Detailed Description

This section describes the Direct3D 11 interoperability functions of the low-level CUDA driver application programming interface.

4.51.2 Typedef Documentation

4.51.2.1 `typedef enum CUd3d11DeviceList_enum CUd3d11DeviceList`

CUDA devices corresponding to a D3D11 device

4.51.3 Enumeration Type Documentation

4.51.3.1 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

4.51.4 Function Documentation

4.51.4.1 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](#)

4.51.4.2 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`

4.51.4.3 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`

4.51.4.4 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](#)

4.51.4.5 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](#)

4.51.4.6 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 valid values for this parameter are

- CU_GRAPHICS_REGISTER_FLAGS_NONE: Specifies no hints about how this resource will be used.
- CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST: Specifies that CUDA will bind this resource to a surface reference.

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](#)

4.52 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 cuVDPACtxCreate** (**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.

4.52.1 Detailed Description

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

4.52.2 Function Documentation

4.52.2.1 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](#), [cuVDPAUCtxCreate](#), [cuGraphicsVDPAURegisterVideoSurface](#), [cuGraphicsUnregisterResource](#), [cuGraphicsResourceSetMapFlags](#), [cuGraphicsMapResources](#), [cuGraphicsUnmapResources](#), [cuGraphicsSubResourceGetMappedArray](#), [cuVDPAUGetDevice](#)

4.52.2.2 CURESULT cuGraphicsVDPAURegisterVideoSurface (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 VDPAU 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](#)

4.52.2.3 CURESTU 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](#)

4.52.2.4 CURESTU 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_PEER_st Struct Reference

Data Fields

- size_t **Depth**
- CUarray **dstArray**
- CUcontext **dstContext**
- 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**
- CUarray **srcArray**
- CUcontext **srcContext**
- 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 cross-context copy parameters

5.4.2 Field Documentation

5.4.2.1 size_t CUDA_MEMCPY3D_PEER_st::Depth

Depth of 3D memory copy

5.4.2.2 CUarray CUDA_MEMCPY3D_PEER_st::dstArray

Destination array reference

5.4.2.3 CUcontext CUDA_MEMCPY3D_PEER_st::dstContext

Destination context (ignored with dstMemoryType is [CU_MEMORYTYPE_ARRAY](#))

5.4.2.4 CUdeviceptr CUDA_MEMCPY3D_PEER_st::dstDevice

Destination device pointer

5.4.2.5 size_t CUDA_MEMCPY3D_PEER_st::dstHeight

Destination height (ignored when dst is array; may be 0 if Depth==1)

5.4.2.6 void* CUDA_MEMCPY3D_PEER_st::dstHost

Destination host pointer

5.4.2.7 size_t CUDA_MEMCPY3D_PEER_st::dstLOD

Destination LOD

5.4.2.8 CUmemorytype CUDA_MEMCPY3D_PEER_st::dstMemoryType

Destination memory type (host, device, array)

5.4.2.9 size_t CUDA_MEMCPY3D_PEER_st::dstPitch

Destination pitch (ignored when dst is array)

5.4.2.10 size_t CUDA_MEMCPY3D_PEER_st::dstXInBytes

Destination X in bytes

5.4.2.11 size_t CUDA_MEMCPY3D_PEER_st::dstY

Destination Y

5.4.2.12 size_t CUDA_MEMCPY3D_PEER_st::dstZ

Destination Z

5.4.2.13 size_t CUDA_MEMCPY3D_PEER_st::Height

Height of 3D memory copy

5.4.2.14 CUarray CUDA_MEMCPY3D_PEER_st::srcArray

Source array reference

5.4.2.15 CUcontext CUDA_MEMCPY3D_PEER_st::srcContext

Source context (ignored with srcMemoryType is [CU_MEMORYTYPE_ARRAY](#))

5.4.2.16 CUdeviceptr CUDA_MEMCPY3D_PEER_st::srcDevice

Source device pointer

5.4.2.17 size_t CUDA_MEMCPY3D_PEER_st::srcHeight

Source height (ignored when src is array; may be 0 if Depth==1)

5.4.2.18 const void* CUDA_MEMCPY3D_PEER_st::srcHost

Source host pointer

5.4.2.19 size_t CUDA_MEMCPY3D_PEER_st::srcLOD

Source LOD

5.4.2.20 CUmemorytype CUDA_MEMCPY3D_PEER_st::srcMemoryType

Source memory type (host, device, array)

5.4.2.21 size_t CUDA_MEMCPY3D_PEER_st::srcPitch

Source pitch (ignored when src is array)

5.4.2.22 size_t CUDA_MEMCPY3D_PEER_st::srcXInBytes

Source X in bytes

5.4.2.23 size_t CUDA_MEMCPY3D_PEER_st::srcY

Source Y

5.4.2.24 size_t CUDA_MEMCPY3D_PEER_st::srcZ

Source Z

5.4.2.25 size_t CUDA_MEMCPY3D_PEER_st::WidthInBytes

Width of 3D memory copy in bytes

5.5 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`

5.5.1 Detailed Description

3D memory copy parameters

5.5.2 Field Documentation

5.5.2.1 `size_t CUDA_MEMCPY3D_st::Depth`

Depth of 3D memory copy

5.5.2.2 `CUarray CUDA_MEMCPY3D_st::dstArray`

Destination array reference

5.5.2.3 `CUdeviceptr CUDA_MEMCPY3D_st::dstDevice`

Destination device pointer

5.5.2.4 size_t CUDA_MEMCPY3D_st::dstHeight

Destination height (ignored when dst is array; may be 0 if Depth==1)

5.5.2.5 void* CUDA_MEMCPY3D_st::dstHost

Destination host pointer

5.5.2.6 size_t CUDA_MEMCPY3D_st::dstLOD

Destination LOD

5.5.2.7 CUmemorytype CUDA_MEMCPY3D_st::dstMemoryType

Destination memory type (host, device, array)

5.5.2.8 size_t CUDA_MEMCPY3D_st::dstPitch

Destination pitch (ignored when dst is array)

5.5.2.9 size_t CUDA_MEMCPY3D_st::dstXInBytes

Destination X in bytes

5.5.2.10 size_t CUDA_MEMCPY3D_st::dstY

Destination Y

5.5.2.11 size_t CUDA_MEMCPY3D_st::dstZ

Destination Z

5.5.2.12 size_t CUDA_MEMCPY3D_st::Height

Height of 3D memory copy

5.5.2.13 void* CUDA_MEMCPY3D_st::reserved0

Must be NULL

5.5.2.14 void* CUDA_MEMCPY3D_st::reserved1

Must be NULL

5.5.2.15 CUarray CUDA_MEMCPY3D_st::srcArray

Source array reference

5.5.2.16 CUdeviceptr CUDA_MEMCPY3D_st::srcDevice

Source device pointer

5.5.2.17 size_t CUDA_MEMCPY3D_st::srcHeight

Source height (ignored when src is array; may be 0 if Depth==1)

5.5.2.18 const void* CUDA_MEMCPY3D_st::srcHost

Source host pointer

5.5.2.19 size_t CUDA_MEMCPY3D_st::srcLOD

Source LOD

5.5.2.20 CUmemorytype CUDA_MEMCPY3D_st::srcMemoryType

Source memory type (host, device, array)

5.5.2.21 size_t CUDA_MEMCPY3D_st::srcPitch

Source pitch (ignored when src is array)

5.5.2.22 size_t CUDA_MEMCPY3D_st::srcXInBytes

Source X in bytes

5.5.2.23 size_t CUDA_MEMCPY3D_st::srcY

Source Y

5.5.2.24 size_t CUDA_MEMCPY3D_st::srcZ

Source Z

5.5.2.25 size_t CUDA_MEMCPY3D_st::WidthInBytes

Width of 3D memory copy in bytes

5.6 cudaChannelFormatDesc Struct Reference

Data Fields

- enum `cudaChannelFormatKind` `f`
- int `w`
- int `x`
- int `y`
- int `z`

5.6.1 Detailed Description

CUDA Channel format descriptor

5.6.2 Field Documentation

5.6.2.1 enum cudaChannelFormatKind cudaChannelFormatDesc::f

Channel format kind

5.6.2.2 int cudaChannelFormatDesc::w

`w`

5.6.2.3 int cudaChannelFormatDesc::x

`x`

5.6.2.4 int cudaChannelFormatDesc::y

`y`

5.6.2.5 int cudaChannelFormatDesc::z

`z`

5.7 cudaDeviceProp Struct Reference

Data Fields

- int `asyncEngineCount`
- int `canMapHostMemory`
- int `clockRate`
- int `computeMode`
- int `concurrentKernels`
- int `deviceOverlap`
- int `ECCEnabled`
- int `integrated`
- int `kernelExecTimeoutEnabled`
- int `major`
- int `maxGridSize` [3]
- int `maxTexture1D`
- int `maxTexture1DLayered` [2]
- int `maxTexture2D` [2]
- int `maxTexture2DLayered` [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 `unifiedAddressing`
- int `warpSize`

5.7.1 Detailed Description

CUDA device properties

5.7.2 Field Documentation

5.7.2.1 int cudaDeviceProp::asyncEngineCount

Number of asynchronous engines

5.7.2.2 int cudaDeviceProp::canMapHostMemory

Device can map host memory with cudaHostAlloc/cudaHostGetDevicePointer

5.7.2.3 int cudaDeviceProp::clockRate

Clock frequency in kilohertz

5.7.2.4 int cudaDeviceProp::computeMode

Compute mode (See [cudaComputeMode](#))

5.7.2.5 int cudaDeviceProp::concurrentKernels

Device can possibly execute multiple kernels concurrently

5.7.2.6 int cudaDeviceProp::deviceOverlap

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

5.7.2.7 int cudaDeviceProp::ECCEnabled

Device has ECC support enabled

5.7.2.8 int cudaDeviceProp::integrated

Device is integrated as opposed to discrete

5.7.2.9 int cudaDeviceProp::kernelExecTimeoutEnabled

Specified whether there is a run time limit on kernels

5.7.2.10 int cudaDeviceProp::major

Major compute capability

5.7.2.11 int cudaDeviceProp::maxGridSize[3]

Maximum size of each dimension of a grid

5.7.2.12 int cudaDeviceProp::maxTexture1D

Maximum 1D texture size

5.7.2.13 int cudaDeviceProp::maxTexture1DLayered[2]

Maximum 1D layered texture dimensions

5.7.2.14 int cudaDeviceProp::maxTexture2D[2]

Maximum 2D texture dimensions

5.7.2.15 int cudaDeviceProp::maxTexture2DLayered[3]

Maximum 2D layered texture dimensions

5.7.2.16 int cudaDeviceProp::maxTexture3D[3]

Maximum 3D texture dimensions

5.7.2.17 int cudaDeviceProp::maxThreadsDim[3]

Maximum size of each dimension of a block

5.7.2.18 int cudaDeviceProp::maxThreadsPerBlock

Maximum number of threads per block

5.7.2.19 size_t cudaDeviceProp::memPitch

Maximum pitch in bytes allowed by memory copies

5.7.2.20 int cudaDeviceProp::minor

Minor compute capability

5.7.2.21 int cudaDeviceProp::multiProcessorCount

Number of multiprocessors on device

5.7.2.22 char cudaDeviceProp::name[256]

ASCII string identifying device

5.7.2.23 int cudaDeviceProp::pciBusID

PCI bus ID of the device

5.7.2.24 int cudaDeviceProp::pciDeviceID

PCI device ID of the device

5.7.2.25 int cudaDeviceProp::regsPerBlock

32-bit registers available per block

5.7.2.26 size_t cudaDeviceProp::sharedMemPerBlock

Shared memory available per block in bytes

5.7.2.27 size_t cudaDeviceProp::surfaceAlignment

Alignment requirements for surfaces

5.7.2.28 int cudaDeviceProp::tccDriver

1 if device is a Tesla device using TCC driver, 0 otherwise

5.7.2.29 size_t cudaDeviceProp::textureAlignment

Alignment requirement for textures

5.7.2.30 size_t cudaDeviceProp::totalConstMem

Constant memory available on device in bytes

5.7.2.31 size_t cudaDeviceProp::totalGlobalMem

Global memory available on device in bytes

5.7.2.32 int cudaDeviceProp::unifiedAddressing

Device shares a unified address space with the host

5.7.2.33 int cudaDeviceProp::warpSize

Warp size in threads

5.8 cudaExtent Struct Reference

Data Fields

- size_t [depth](#)
- size_t [height](#)
- size_t [width](#)

5.8.1 Detailed Description

CUDA extent

See also:

[make_cudaExtent](#)

5.8.2 Field Documentation

5.8.2.1 size_t cudaExtent::depth

Depth in elements

5.8.2.2 size_t cudaExtent::height

Height in elements

5.8.2.3 size_t cudaExtent::width

Width in elements when referring to array memory, in bytes when referring to linear memory

5.9 cudaFuncAttributes Struct Reference

Data Fields

- int `binaryVersion`
- size_t `constSizeBytes`
- size_t `localSizeBytes`
- int `maxThreadsPerBlock`
- int `numRegs`
- int `ptxVersion`
- size_t `sharedSizeBytes`

5.9.1 Detailed Description

CUDA function attributes

5.9.2 Field Documentation

5.9.2.1 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.

5.9.2.2 size_t cudaFuncAttributes::constSizeBytes

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

5.9.2.3 size_t cudaFuncAttributes::localSizeBytes

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

5.9.2.4 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.

5.9.2.5 int cudaFuncAttributes::numRegs

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

5.9.2.6 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.

5.9.2.7 `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.

5.10 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`

5.10.1 Detailed Description

CUDA 3D memory copying parameters

5.10.2 Field Documentation

5.10.2.1 struct `cudaArray`* `cudaMemcpy3DParms::dstArray` [read]

Destination memory address

5.10.2.2 struct `cudaPos` `cudaMemcpy3DParms::dstPos` [read]

Destination position offset

5.10.2.3 struct `cudaPitchedPtr` `cudaMemcpy3DParms::dstPtr` [read]

Pitched destination memory address

5.10.2.4 struct `cudaExtent` `cudaMemcpy3DParms::extent` [read]

Requested memory copy size

5.10.2.5 enum `cudaMemcpyKind` `cudaMemcpy3DParms::kind`

Type of transfer

5.10.2.6 struct `cudaArray`* `cudaMemcpy3DParms::srcArray` [read]

Source memory address

5.10.2.7 struct `cudaPos` `cudaMemcpy3DParms::srcPos` [read]

Source position offset

5.10.2.8 struct cudaPitchedPtr cudaMemcpy3DParms::srcPtr [read]

Pitched source memory address

5.11 cudaMemcpy3DPeerParms Struct Reference

Data Fields

- struct `cudaArray * dstArray`
- int `dstDevice`
- struct `cudaPos dstPos`
- struct `cudaPitchedPtr dstPtr`
- struct `cudaExtent extent`
- struct `cudaArray * srcArray`
- int `srcDevice`
- struct `cudaPos srcPos`
- struct `cudaPitchedPtr srcPtr`

5.11.1 Detailed Description

CUDA 3D cross-device memory copying parameters

5.11.2 Field Documentation

5.11.2.1 struct `cudaArray* cudaMemcpy3DPeerParms::dstArray` [read]

Destination memory address

5.11.2.2 int `cudaMemcpy3DPeerParms::dstDevice`

Destination device

5.11.2.3 struct `cudaPos cudaMemcpy3DPeerParms::dstPos` [read]

Destination position offset

5.11.2.4 struct `cudaPitchedPtr cudaMemcpy3DPeerParms::dstPtr` [read]

Pitched destination memory address

5.11.2.5 struct `cudaExtent cudaMemcpy3DPeerParms::extent` [read]

Requested memory copy size

5.11.2.6 struct `cudaArray* cudaMemcpy3DPeerParms::srcArray` [read]

Source memory address

5.11.2.7 int `cudaMemcpy3DPeerParms::srcDevice`

Source device

5.11.2.8 struct cudaPos cudaMemcpy3DPeerParms::srcPos [read]

Source position offset

5.11.2.9 struct cudaPitchedPtr cudaMemcpy3DPeerParms::srcPtr [read]

Pitched source memory address

5.12 cudaPitchedPtr Struct Reference

Data Fields

- size_t `pitch`
- void * `ptr`
- size_t `xsize`
- size_t `ysize`

5.12.1 Detailed Description

CUDA Pitched memory pointer

See also:

[make_cudaPitchedPtr](#)

5.12.2 Field Documentation

5.12.2.1 size_t cudaPitchedPtr::pitch

Pitch of allocated memory in bytes

5.12.2.2 void* cudaPitchedPtr::ptr

Pointer to allocated memory

5.12.2.3 size_t cudaPitchedPtr::xsize

Logical width of allocation in elements

5.12.2.4 size_t cudaPitchedPtr::ysize

Logical height of allocation in elements

5.13 cudaPointerAttributes Struct Reference

Data Fields

- int `device`
- void * `devicePointer`
- void * `hostPointer`
- enum `cudaMemoryType` `memoryType`

5.13.1 Detailed Description

CUDA pointer attributes

5.13.2 Field Documentation

5.13.2.1 int cudaPointerAttributes::device

The device against which the memory was allocated or registered. If the memory type is `cudaMemoryTypeDevice` then this identifies the device on which the memory referred physically resides. If the memory type is `cudaMemoryTypeHost` then this identifies the device which was current when the memory was allocated or registered (and if that device is deinitialized then this allocation will vanish with that device's state).

5.13.2.2 void* cudaPointerAttributes::devicePointer

The address which may be dereferenced on the current device to access the memory or NULL if no such address exists.

5.13.2.3 void* cudaPointerAttributes::hostPointer

The address which may be dereferenced on the host to access the memory or NULL if no such address exists.

5.13.2.4 enum cudaMemoryType cudaPointerAttributes::memoryType

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

5.14 cudaPos Struct Reference

Data Fields

- size_t [x](#)
- size_t [y](#)
- size_t [z](#)

5.14.1 Detailed Description

CUDA 3D position

See also:

[make_cudaPos](#)

5.14.2 Field Documentation

5.14.2.1 size_t cudaPos::x

x

5.14.2.2 size_t cudaPos::y

y

5.14.2.3 size_t cudaPos::z

z

5.15 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`

5.15.1 Detailed Description

Legacy device properties

5.15.2 Field Documentation

5.15.2.1 int CUdevprop_st::clockRate

Clock frequency in kilohertz

5.15.2.2 int CUdevprop_st::maxGridSize[3]

Maximum size of each dimension of a grid

5.15.2.3 int CUdevprop_st::maxThreadsDim[3]

Maximum size of each dimension of a block

5.15.2.4 int CUdevprop_st::maxThreadsPerBlock

Maximum number of threads per block

5.15.2.5 int CUdevprop_st::memPitch

Maximum pitch in bytes allowed by memory copies

5.15.2.6 int CUdevprop_st::regsPerBlock

32-bit registers available per block

5.15.2.7 int CUdevprop_st::sharedMemPerBlock

Shared memory available per block in bytes

5.15.2.8 int CUdevprop_st::SIMDWidth

Warp size in threads

5.15.2.9 int CUdevprop_st::textureAlign

Alignment requirement for textures

5.15.2.10 int CUdevprop_st::totalConstantMemory

Constant memory available on device in bytes

5.16 surfaceReference Struct Reference

Data Fields

- struct [cudaChannelFormatDesc](#) channelDesc

5.16.1 Detailed Description

CUDA Surface reference

5.16.2 Field Documentation

5.16.2.1 struct cudaChannelFormatDesc surfaceReference::channelDesc [read]

Channel descriptor for surface reference

5.17 textureReference Struct Reference

Data Fields

- enum `cudaTextureAddressMode` `addressMode` [3]
- struct `cudaChannelFormatDesc` `channelDesc`
- enum `cudaTextureFilterMode` `filterMode`
- int `normalized`
- int `sRGB`

5.17.1 Detailed Description

CUDA texture reference

5.17.2 Field Documentation

5.17.2.1 enum `cudaTextureAddressMode` `textureReference::addressMode[3]`

Texture address mode for up to 3 dimensions

5.17.2.2 struct `cudaChannelFormatDesc` `textureReference::channelDesc` [read]

Channel descriptor for the texture reference

5.17.2.3 enum `cudaTextureFilterMode` `textureReference::filterMode`

Texture filter mode

5.17.2.4 int `textureReference::normalized`

Indicates whether texture reads are normalized or not

5.17.2.5 int `textureReference::sRGB`

Perform sRGB->linear conversion during texture read

Index

addressMode
 textureReference, [390](#)
asyncEngineCount
 cudaDeviceProp, [373](#)

binaryVersion
 cudaFuncAttributes, [378](#)

C++ API Routines, [118](#)
canMapHostMemory
 cudaDeviceProp, [373](#)
channelDesc
 surfaceReference, [389](#)
 textureReference, [390](#)
clockRate
 cudaDeviceProp, [374](#)
 CUdevprop_st, [387](#)
computeMode
 cudaDeviceProp, [374](#)
concurrentKernels
 cudaDeviceProp, [374](#)
constSizeBytes
 cudaFuncAttributes, [378](#)
Context Management, [199](#)
CU_AD_FORMAT_FLOAT
 CUDA_TYPES, [182](#)
CU_AD_FORMAT_HALF
 CUDA_TYPES, [182](#)
CU_AD_FORMAT_SIGNED_INT16
 CUDA_TYPES, [182](#)
CU_AD_FORMAT_SIGNED_INT32
 CUDA_TYPES, [182](#)
CU_AD_FORMAT_SIGNED_INT8
 CUDA_TYPES, [182](#)
CU_AD_FORMAT_UNSIGNED_INT16
 CUDA_TYPES, [182](#)
CU_AD_FORMAT_UNSIGNED_INT32
 CUDA_TYPES, [182](#)
CU_AD_FORMAT_UNSIGNED_INT8
 CUDA_TYPES, [182](#)
CU_COMPUTEMODE_DEFAULT
 CUDA_TYPES, [183](#)
CU_COMPUTEMODE_EXCLUSIVE
 CUDA_TYPES, [183](#)
CU_COMPUTEMODE_EXCLUSIVE_PROCESS
 CUDA_TYPES, [183](#)
CU_COMPUTEMODE_PROHIBITED
 CUDA_TYPES, [183](#)
CU_CTX_BLOCKING_SYNC
 CUDA_TYPES, [183](#)
CU_CTX_LMEM_RESIZE_TO_MAX
 CUDA_TYPES, [183](#)
CU_CTX_MAP_HOST
 CUDA_TYPES, [183](#)
CU_CTX_PRIMARY
 CUDA_TYPES, [183](#)
CU_CTX_SCHED_AUTO
 CUDA_TYPES, [183](#)
CU_CTX_SCHED_BLOCKING_SYNC
 CUDA_TYPES, [183](#)
CU_CTX_SCHED_SPIN
 CUDA_TYPES, [183](#)
CU_CTX_SCHED_YIELD
 CUDA_TYPES, [183](#)
CU_CUBEMAP_FACE_NEGATIVE_X
 CUDA_TYPES, [182](#)
CU_CUBEMAP_FACE_NEGATIVE_Y
 CUDA_TYPES, [182](#)
CU_CUBEMAP_FACE_NEGATIVE_Z
 CUDA_TYPES, [182](#)
CU_CUBEMAP_FACE_POSITIVE_X
 CUDA_TYPES, [182](#)
CU_CUBEMAP_FACE_POSITIVE_Y
 CUDA_TYPES, [182](#)
CU_CUBEMAP_FACE_POSITIVE_Z
 CUDA_TYPES, [182](#)
CU_D3D10_DEVICE_LIST_ALL
 CUDA_D3D10, [336](#)
CU_D3D10_DEVICE_LIST_CURRENT_FRAME
 CUDA_D3D10, [336](#)
CU_D3D10_DEVICE_LIST_NEXT_FRAME
 CUDA_D3D10, [336](#)
CU_D3D11_DEVICE_LIST_ALL
 CUDA_D3D11, [351](#)
CU_D3D11_DEVICE_LIST_CURRENT_FRAME
 CUDA_D3D11, [351](#)
CU_D3D11_DEVICE_LIST_NEXT_FRAME
 CUDA_D3D11, [351](#)
CU_D3D9_DEVICE_LIST_ALL
 CUDA_D3D9, [321](#)

CU_D3D9_DEVICE_LIST_CURRENT_FRAME
 CUDA_D3D9, 321
 CU_D3D9_DEVICE_LIST_NEXT_FRAME
 CUDA_D3D9, 321
 CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT
 CUDA_TYPES, 187
 CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_CLOCK_RATE
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_COMPUTE_MODE
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS
 CUDA_TYPES, 187
 CU_DEVICE_ATTRIBUTE_ECC_ENABLED
 CUDA_TYPES, 187
 CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH
 CUDA_TYPES, 187
 CU_DEVICE_ATTRIBUTE_GPU_OVERLAP
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_INTEGRATED
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_L2_CACHE_SIZE
 CUDA_TYPES, 187
 CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAX_PITCH
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK
 CUDA_TYPES, 186

CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR
 CUDA_TYPES, 187
 CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS
 CUDA_TYPES, 187
 CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH
 CUDA_TYPES, 187
 CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_HEIGHT
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_NUMSLICES
 CUDA_TYPES, 187
 CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_WIDTH
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_HEIGHT
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_WIDTH
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE
 CUDA_TYPES, 187
 CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT
 CUDA_TYPES, 186
 CU_DEVICE_ATTRIBUTE_PCI_BUS_ID
 CUDA_TYPES, 187
 CU_DEVICE_ATTRIBUTE_PCI_DEVICE_ID

CUDA_TYPES, 187
CU_DEVICE_ATTRIBUTE_REGISTERS_PER_BLOCK
 CUDA_TYPES, 186
CU_DEVICE_ATTRIBUTE_SHARED_MEMORY_PER_BLOCK
 CUDA_TYPES, 186
CU_DEVICE_ATTRIBUTE_SURFACE_ALIGNMENT
 CUDA_TYPES, 187
CU_DEVICE_ATTRIBUTE_TCC_DRIVER
 CUDA_TYPES, 187
CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT
 CUDA_TYPES, 186
CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY
 CUDA_TYPES, 186
CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING
 CUDA_TYPES, 187
CU_DEVICE_ATTRIBUTE_WARP_SIZE
 CUDA_TYPES, 186
CU_EVENT_BLOCKING_SYNC
 CUDA_TYPES, 187
CU_EVENT_DEFAULT
 CUDA_TYPES, 187
CU_EVENT_DISABLE_TIMING
 CUDA_TYPES, 187
CU_FUNC_ATTRIBUTE_BINARY_VERSION
 CUDA_TYPES, 188
CU_FUNC_ATTRIBUTE_CONST_SIZE_BYTES
 CUDA_TYPES, 188
CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES
 CUDA_TYPES, 188
CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK
 CUDA_TYPES, 188
CU_FUNC_ATTRIBUTE_NUM_REGS
 CUDA_TYPES, 188
CU_FUNC_ATTRIBUTE_PTX_VERSION
 CUDA_TYPES, 188
CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES
 CUDA_TYPES, 188
CU_FUNC_CACHE_PREFER_L1
 CUDA_TYPES, 188
CU_FUNC_CACHE_PREFER_NONE
 CUDA_TYPES, 188
CU_FUNC_CACHE_PREFER_SHARED
 CUDA_TYPES, 188
CU_JIT_ERROR_LOG_BUFFER
 CUDA_TYPES, 189
CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES
 CUDA_TYPES, 189
CU_JIT_FALLBACK_STRATEGY
 CUDA_TYPES, 189
CU_JIT_INFO_LOG_BUFFER
 CUDA_TYPES, 189
CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES
 CUDA_TYPES, 189
CU_JIT_MAX_REGISTERS
 CUDA_TYPES, 189
CU_JIT_OPTIMIZATION_LEVEL
 CUDA_TYPES, 189
CU_JIT_TARGET
 CUDA_TYPES, 189
CU_JIT_TARGET_FROM_CUCONTEXT
 CUDA_TYPES, 189
CU_JIT_THREADS_PER_BLOCK
 CUDA_TYPES, 189
CU_JIT_WALL_TIME
 CUDA_TYPES, 189
CU_LIMIT_MALLOC_HEAP_SIZE
 CUDA_TYPES, 190
CU_LIMIT_PRINTF_FIFO_SIZE
 CUDA_TYPES, 190
CU_LIMIT_STACK_SIZE
 CUDA_TYPES, 190
CU_MEMORYTYPE_ARRAY
 CUDA_TYPES, 190
CU_MEMORYTYPE_DEVICE
 CUDA_TYPES, 190
CU_MEMORYTYPE_HOST
 CUDA_TYPES, 190
CU_MEMORYTYPE_UNIFIED
 CUDA_TYPES, 190
CU_POINTER_ATTRIBUTE_CONTEXT
 CUDA_TYPES, 190
CU_POINTER_ATTRIBUTE_DEVICE_POINTER
 CUDA_TYPES, 190
CU_POINTER_ATTRIBUTE_HOST_POINTER
 CUDA_TYPES, 190
CU_POINTER_ATTRIBUTE_MEMORY_TYPE
 CUDA_TYPES, 190
CU_PREFER_BINARY
 CUDA_TYPES, 189
CU_PREFER_PTX
 CUDA_TYPES, 189
CU_TARGET_COMPUTE_10
 CUDA_TYPES, 190
CU_TARGET_COMPUTE_11
 CUDA_TYPES, 190
CU_TARGET_COMPUTE_12
 CUDA_TYPES, 190
CU_TARGET_COMPUTE_13
 CUDA_TYPES, 190
CU_TARGET_COMPUTE_20
 CUDA_TYPES, 190
CU_TARGET_COMPUTE_21
 CUDA_TYPES, 190
CU_TR_ADDRESS_MODE_BORDER

CUDA_TYPES, 182
 CU_TR_ADDRESS_MODE_CLAMP
 CUDA_TYPES, 182
 CU_TR_ADDRESS_MODE_MIRROR
 CUDA_TYPES, 182
 CU_TR_ADDRESS_MODE_WRAP
 CUDA_TYPES, 182
 CU_TR_FILTER_MODE_LINEAR
 CUDA_TYPES, 187
 CU_TR_FILTER_MODE_POINT
 CUDA_TYPES, 187
 CU_LAUNCH_PARAM_BUFFER_POINTER
 CUDA_TYPES, 177
 CU_LAUNCH_PARAM_BUFFER_SIZE
 CUDA_TYPES, 177
 CU_LAUNCH_PARAM_END
 CUDA_TYPES, 177
 CU_MEMHOSTALLOC_DEVICEMAP
 CUDA_TYPES, 177
 CU_MEMHOSTALLOC_PORTABLE
 CUDA_TYPES, 177
 CU_MEMHOSTALLOC_WRITECOMBINED
 CUDA_TYPES, 177
 CU_MEMHOSTREGISTER_DEVICEMAP
 CUDA_TYPES, 177
 CU_MEMHOSTREGISTER_PORTABLE
 CUDA_TYPES, 177
 CU_MEMPEERREGISTER_DEVICEMAP
 CUDA_TYPES, 178
 CU_PARAM_TR_DEFAULT
 CUDA_TYPES, 178
 CU_TRA_OVERRIDE_FORMAT
 CUDA_TYPES, 178
 CU_TRSF_NORMALIZED_COORDINATES
 CUDA_TYPES, 178
 CU_TRSF_READ_AS_INTEGER
 CUDA_TYPES, 178
 CU_TRSF_SRGB
 CUDA_TYPES, 178
 CUaddress_mode
 CUDA_TYPES, 179
 CUaddress_mode_enum
 CUDA_TYPES, 182
 CUarray
 CUDA_TYPES, 179
 cuArray3DCreate
 CUDA_MEM, 220
 cuArray3DGetDescriptor
 CUDA_MEM, 222
 CUarray_cubemap_face
 CUDA_TYPES, 179
 CUarray_cubemap_face_enum
 CUDA_TYPES, 182
 CUarray_format
 CUDA_TYPES, 179
 CUarray_format_enum
 CUDA_TYPES, 182
 cuArrayCreate
 CUDA_MEM, 223
 cuArrayDestroy
 CUDA_MEM, 224
 cuArrayGetDescriptor
 CUDA_MEM, 225
 CUcomputemode
 CUDA_TYPES, 179
 CUcomputemode_enum
 CUDA_TYPES, 182
 CUcontext
 CUDA_TYPES, 179
 CUctx_flags
 CUDA_TYPES, 179
 CUctx_flags_enum
 CUDA_TYPES, 183
 cuCtxAttach
 CUDA_CTX_DEPRECATED, 208
 cuCtxCreate
 CUDA_CTX, 200
 cuCtxDestroy
 CUDA_CTX, 201
 cuCtxDetach
 CUDA_CTX_DEPRECATED, 208
 cuCtxDisablePeerAccess
 CUDA_PEER_ACCESS, 301
 cuCtxEnablePeerAccess
 CUDA_PEER_ACCESS, 302
 cuCtxGetApiVersion
 CUDA_CTX, 201
 cuCtxGetCacheConfig
 CUDA_CTX, 202
 cuCtxGetCurrent
 CUDA_CTX, 203
 cuCtxGetDevice
 CUDA_CTX, 203
 cuCtxGetLimit
 CUDA_CTX, 203
 cuCtxPopCurrent
 CUDA_CTX, 204
 cuCtxPushCurrent
 CUDA_CTX, 204
 cuCtxSetCacheConfig
 CUDA_CTX, 205
 cuCtxSetCurrent
 CUDA_CTX, 205
 cuCtxSetLimit
 CUDA_CTX, 206
 cuCtxSynchronize
 CUDA_CTX, 207
 cuD3D10CtxCreate

CUDA_D3D10, 336
cuD3D10CtxCreateOnDevice
 CUDA_D3D10, 336
CUD3d10DeviceList
 CUDA_D3D10, 336
CUD3d10DeviceList_enum
 CUDA_D3D10, 336
cuD3D10GetDevice
 CUDA_D3D10, 337
cuD3D10GetDevices
 CUDA_D3D10, 337
cuD3D10GetDirect3DDevice
 CUDA_D3D10, 338
CUD3D10map_flags
 CUDA_D3D10_DEPRECATED, 342
CUD3D10map_flags_enum
 CUDA_D3D10_DEPRECATED, 342
cuD3D10MapResources
 CUDA_D3D10_DEPRECATED, 342
CUD3D10register_flags
 CUDA_D3D10_DEPRECATED, 342
CUD3D10register_flags_enum
 CUDA_D3D10_DEPRECATED, 342
cuD3D10RegisterResource
 CUDA_D3D10_DEPRECATED, 343
cuD3D10ResourceGetMappedArray
 CUDA_D3D10_DEPRECATED, 344
cuD3D10ResourceGetMappedPitch
 CUDA_D3D10_DEPRECATED, 345
cuD3D10ResourceGetMappedPointer
 CUDA_D3D10_DEPRECATED, 345
cuD3D10ResourceGetMappedSize
 CUDA_D3D10_DEPRECATED, 346
cuD3D10ResourceGetSurfaceDimensions
 CUDA_D3D10_DEPRECATED, 347
cuD3D10ResourceSetMapFlags
 CUDA_D3D10_DEPRECATED, 347
cuD3D10UnmapResources
 CUDA_D3D10_DEPRECATED, 348
cuD3D10UnregisterResource
 CUDA_D3D10_DEPRECATED, 349
cuD3D11CtxCreate
 CUDA_D3D11, 351
cuD3D11CtxCreateOnDevice
 CUDA_D3D11, 351
CUD3d11DeviceList
 CUDA_D3D11, 350
CUD3d11DeviceList_enum
 CUDA_D3D11, 351
cuD3D11GetDevice
 CUDA_D3D11, 352
cuD3D11GetDevices
 CUDA_D3D11, 352
cuD3D11GetDirect3DDevice
 CUDA_D3D11, 353
cuD3D9CtxCreate
 CUDA_D3D9, 321
cuD3D9CtxCreateOnDevice
 CUDA_D3D9, 321
CUD3d9DeviceList
 CUDA_D3D9, 321
CUD3d9DeviceList_enum
 CUDA_D3D9, 321
cuD3D9GetDevice
 CUDA_D3D9, 322
cuD3D9GetDevices
 CUDA_D3D9, 322
cuD3D9GetDirect3DDevice
 CUDA_D3D9, 323
CUD3d9map_flags
 CUDA_D3D9_DEPRECATED, 327
CUD3d9map_flags_enum
 CUDA_D3D9_DEPRECATED, 327
cuD3D9MapResources
 CUDA_D3D9_DEPRECATED, 327
CUD3d9register_flags
 CUDA_D3D9_DEPRECATED, 327
CUD3d9register_flags_enum
 CUDA_D3D9_DEPRECATED, 327
cuD3D9RegisterResource
 CUDA_D3D9_DEPRECATED, 328
cuD3D9ResourceGetMappedArray
 CUDA_D3D9_DEPRECATED, 329
cuD3D9ResourceGetMappedPitch
 CUDA_D3D9_DEPRECATED, 330
cuD3D9ResourceGetMappedPointer
 CUDA_D3D9_DEPRECATED, 331
cuD3D9ResourceGetMappedSize
 CUDA_D3D9_DEPRECATED, 331
cuD3D9ResourceGetSurfaceDimensions
 CUDA_D3D9_DEPRECATED, 332
cuD3D9ResourceSetMapFlags
 CUDA_D3D9_DEPRECATED, 333
cuD3D9UnmapResources
 CUDA_D3D9_DEPRECATED, 333
cuD3D9UnregisterResource
 CUDA_D3D9_DEPRECATED, 334
CUDA Driver API, 170
CUDA Runtime API, 11
CUDA_D3D10
 CU_D3D10_DEVICE_LIST_ALL, 336
 CU_D3D10_DEVICE_LIST_CURRENT_FRAME,
 336
 CU_D3D10_DEVICE_LIST_NEXT_FRAME, 336
CUDA_D3D11
 CU_D3D11_DEVICE_LIST_ALL, 351
 CU_D3D11_DEVICE_LIST_CURRENT_FRAME,
 351

CU_D3D11_DEVICE_LIST_NEXT_FRAME, [351](#)
CUDA_D3D9
 CU_D3D9_DEVICE_LIST_ALL, [321](#)
 CU_D3D9_DEVICE_LIST_CURRENT_FRAME, [321](#)
 CU_D3D9_DEVICE_LIST_NEXT_FRAME, [321](#)
CUDA_ERROR_ALREADY_ACQUIRED
 CUDA_TYPES, [184](#)
CUDA_ERROR_ALREADY_MAPPED
 CUDA_TYPES, [184](#)
CUDA_ERROR_ARRAY_IS_MAPPED
 CUDA_TYPES, [184](#)
CUDA_ERROR_CONTEXT_ALREADY_CURRENT
 CUDA_TYPES, [184](#)
CUDA_ERROR_CONTEXT_ALREADY_IN_USE
 CUDA_TYPES, [184](#)
CUDA_ERROR_CONTEXT_IS_DESTROYED
 CUDA_TYPES, [185](#)
CUDA_ERROR_DEINITIALIZED
 CUDA_TYPES, [183](#)
CUDA_ERROR_ECC_UNCORRECTABLE
 CUDA_TYPES, [184](#)
CUDA_ERROR_FILE_NOT_FOUND
 CUDA_TYPES, [184](#)
CUDA_ERROR_INVALID_CONTEXT
 CUDA_TYPES, [184](#)
CUDA_ERROR_INVALID_DEVICE
 CUDA_TYPES, [184](#)
CUDA_ERROR_INVALID_HANDLE
 CUDA_TYPES, [185](#)
CUDA_ERROR_INVALID_IMAGE
 CUDA_TYPES, [184](#)
CUDA_ERROR_INVALID_SOURCE
 CUDA_TYPES, [184](#)
CUDA_ERROR_INVALID_VALUE
 CUDA_TYPES, [183](#)
CUDA_ERROR_LAUNCH_FAILED
 CUDA_TYPES, [185](#)
CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING
 CUDA_TYPES, [185](#)
CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES
 CUDA_TYPES, [185](#)
CUDA_ERROR_LAUNCH_TIMEOUT
 CUDA_TYPES, [185](#)
CUDA_ERROR_MAP_FAILED
 CUDA_TYPES, [184](#)
CUDA_ERROR_NO_BINARY_FOR_GPU
 CUDA_TYPES, [184](#)
CUDA_ERROR_NO_DEVICE
 CUDA_TYPES, [184](#)
CUDA_ERROR_NOT_FOUND
 CUDA_TYPES, [185](#)
CUDA_ERROR_NOT_INITIALIZED
 CUDA_TYPES, [183](#)
CUDA_ERROR_NOT_MAPPED
 CUDA_TYPES, [184](#)
CUDA_ERROR_NOT_MAPPED_AS_ARRAY
 CUDA_TYPES, [184](#)
CUDA_ERROR_NOT_MAPPED_AS_POINTER
 CUDA_TYPES, [184](#)
CUDA_ERROR_NOT_READY
 CUDA_TYPES, [185](#)
CUDA_ERROR_OPERATING_SYSTEM
 CUDA_TYPES, [185](#)
CUDA_ERROR_OUT_OF_MEMORY
 CUDA_TYPES, [183](#)
CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED
 CUDA_TYPES, [185](#)
CUDA_ERROR_PEER_ACCESS_NOT_ENABLED
 CUDA_TYPES, [185](#)
CUDA_ERROR_PEER_MEMORY_ALREADY_REGISTERED
 CUDA_TYPES, [185](#)
CUDA_ERROR_PEER_MEMORY_NOT_REGISTERED
 CUDA_TYPES, [185](#)
CUDA_ERROR_PRIMARY_CONTEXT_ACTIVE
 CUDA_TYPES, [185](#)
CUDA_ERROR_PROFILER_ALREADY_STARTED
 CUDA_TYPES, [184](#)
CUDA_ERROR_PROFILER_ALREADY_STOPPED
 CUDA_TYPES, [184](#)
CUDA_ERROR_PROFILER_DISABLED
 CUDA_TYPES, [183](#)
CUDA_ERROR_PROFILER_NOT_INITIALIZED
 CUDA_TYPES, [183](#)
CUDA_ERROR_SHARED_OBJECT_INIT_FAILED
 CUDA_TYPES, [184](#)
CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND
 CUDA_TYPES, [184](#)
CUDA_ERROR_UNKNOWN
 CUDA_TYPES, [185](#)
CUDA_ERROR_UNMAP_FAILED
 CUDA_TYPES, [184](#)
CUDA_ERROR_UNSUPPORTED_LIMIT
 CUDA_TYPES, [184](#)
CUDA_SUCCESS
 CUDA_TYPES, [183](#)
CUDA_TYPES
 CU_AD_FORMAT_FLOAT, [182](#)
 CU_AD_FORMAT_HALF, [182](#)
 CU_AD_FORMAT_SIGNED_INT16, [182](#)
 CU_AD_FORMAT_SIGNED_INT32, [182](#)
 CU_AD_FORMAT_SIGNED_INT8, [182](#)
 CU_AD_FORMAT_UNSIGNED_INT16, [182](#)

CU_AD_FORMAT_UNSIGNED_INT32, 182
CU_AD_FORMAT_UNSIGNED_INT8, 182
CU_COMPUTEMODE_DEFAULT, 183
CU_COMPUTEMODE_EXCLUSIVE, 183
CU_COMPUTEMODE_EXCLUSIVE_PROCESS, 183
CU_COMPUTEMODE_PROHIBITED, 183
CU_CTX_BLOCKING_SYNC, 183
CU_CTX_LMEM_RESIZE_TO_MAX, 183
CU_CTX_MAP_HOST, 183
CU_CTX_PRIMARY, 183
CU_CTX_SCHED_AUTO, 183
CU_CTX_SCHED_BLOCKING_SYNC, 183
CU_CTX_SCHED_SPIN, 183
CU_CTX_SCHED_YIELD, 183
CU_CUBEMAP_FACE_NEGATIVE_X, 182
CU_CUBEMAP_FACE_NEGATIVE_Y, 182
CU_CUBEMAP_FACE_NEGATIVE_Z, 182
CU_CUBEMAP_FACE_POSITIVE_X, 182
CU_CUBEMAP_FACE_POSITIVE_Y, 182
CU_CUBEMAP_FACE_POSITIVE_Z, 182
CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT, 187
CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY, 186
CU_DEVICE_ATTRIBUTE_CLOCK_RATE, 186
CU_DEVICE_ATTRIBUTE_COMPUTE_MODE, 186
CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS, 187
CU_DEVICE_ATTRIBUTE_ECC_ENABLED, 187
CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH, 187
CU_DEVICE_ATTRIBUTE_GPU_OVERLAP, 186
CU_DEVICE_ATTRIBUTE_INTEGRATED, 186
CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT, 186
CU_DEVICE_ATTRIBUTE_L2_CACHE_SIZE, 187
CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X, 186
CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y, 186
CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z, 186
CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X, 186
CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y, 186
CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z, 186
CU_DEVICE_ATTRIBUTE_MAX_PITCH, 186
CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK, 186
CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR, 187
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS, 187
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH, 186
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_HEIGHT, 186
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_NUMSLICES, 187
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_WIDTH, 186
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_HEIGHT, 186
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT, 186
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS, 186
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH, 186
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_WIDTH, 186
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH, 186
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT, 186
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH, 186
CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE, 187
CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT, 186
CU_DEVICE_ATTRIBUTE_PCI_BUS_ID, 187
CU_DEVICE_ATTRIBUTE_PCI_DEVICE_ID, 187
CU_DEVICE_ATTRIBUTE_REGISTERS_PER_BLOCK, 186
CU_DEVICE_ATTRIBUTE_SHARED_MEMORY_PER_BLOCK, 186
CU_DEVICE_ATTRIBUTE_SURFACE_ALIGNMENT, 187
CU_DEVICE_ATTRIBUTE_TCC_DRIVER, 187
CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT, 186
CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY, 186

CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING, 187
 CU_DEVICE_ATTRIBUTE_WARP_SIZE, 186
 CU_EVENT_BLOCKING_SYNC, 187
 CU_EVENT_DEFAULT, 187
 CU_EVENT_DISABLE_TIMING, 187
 CU_FUNC_ATTRIBUTE_BINARY_VERSION, 188
 CU_FUNC_ATTRIBUTE_CONST_SIZE_BYTES, 188
 CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES, 188
 CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK, 188
 CU_FUNC_ATTRIBUTE_NUM_REGS, 188
 CU_FUNC_ATTRIBUTE_PTX_VERSION, 188
 CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYT, 188
 CU_FUNC_CACHE_PREFER_L1, 188
 CU_FUNC_CACHE_PREFER_NONE, 188
 CU_FUNC_CACHE_PREFER_SHARED, 188
 CU_JIT_ERROR_LOG_BUFFER, 189
 CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES, 189
 CU_JIT_FALLBACK_STRATEGY, 189
 CU_JIT_INFO_LOG_BUFFER, 189
 CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES, 189
 CU_JIT_MAX_REGISTERS, 189
 CU_JIT_OPTIMIZATION_LEVEL, 189
 CU_JIT_TARGET, 189
 CU_JIT_TARGET_FROM_CUCONTEXT, 189
 CU_JIT_THREADS_PER_BLOCK, 189
 CU_JIT_WALL_TIME, 189
 CU_LIMIT_MALLOC_HEAP_SIZE, 190
 CU_LIMIT_PRINTF_FIFO_SIZE, 190
 CU_LIMIT_STACK_SIZE, 190
 CU_MEMORYTYPE_ARRAY, 190
 CU_MEMORYTYPE_DEVICE, 190
 CU_MEMORYTYPE_HOST, 190
 CU_MEMORYTYPE_UNIFIED, 190
 CU_POINTER_ATTRIBUTE_CONTEXT, 190
 CU_POINTER_ATTRIBUTE_DEVICE_POINTER, 190
 CU_POINTER_ATTRIBUTE_HOST_POINTER, 190
 CU_POINTER_ATTRIBUTE_MEMORY_TYPE, 190
 CU_PREFER_BINARY, 189
 CU_PREFER_PTX, 189
 CU_TARGET_COMPUTE_10, 190
 CU_TARGET_COMPUTE_11, 190
 CU_TARGET_COMPUTE_12, 190
 CU_TARGET_COMPUTE_13, 190

CU_TARGET_COMPUTE_20, 190
 CU_TARGET_COMPUTE_21, 190
 CU_TR_ADDRESS_MODE_BORDER, 182
 CU_TR_ADDRESS_MODE_CLAMP, 182
 CU_TR_ADDRESS_MODE_MIRROR, 182
 CU_TR_ADDRESS_MODE_WRAP, 182
 CU_TR_FILTER_MODE_LINEAR, 187
 CU_TR_FILTER_MODE_POINT, 187
 CUDA_ERROR_ALREADY_ACQUIRED, 184
 CUDA_ERROR_ALREADY_MAPPED, 184
 CUDA_ERROR_ARRAY_IS_MAPPED, 184
 CUDA_ERROR_CONTEXT_ALREADY_CURRENT, 184
 CUDA_ERROR_CONTEXT_ALREADY_IN_USE, 184
 CUDA_ERROR_CONTEXT_IS_DESTROYED, 185
 CUDA_ERROR_DEINITIALIZED, 183
 CUDA_ERROR_ECC_UNCORRECTABLE, 184
 CUDA_ERROR_FILE_NOT_FOUND, 184
 CUDA_ERROR_INVALID_CONTEXT, 184
 CUDA_ERROR_INVALID_DEVICE, 184
 CUDA_ERROR_INVALID_HANDLE, 185
 CUDA_ERROR_INVALID_IMAGE, 184
 CUDA_ERROR_INVALID_SOURCE, 184
 CUDA_ERROR_INVALID_VALUE, 183
 CUDA_ERROR_LAUNCH_FAILED, 185
 CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING, 185
 CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES, 185
 CUDA_ERROR_LAUNCH_TIMEOUT, 185
 CUDA_ERROR_MAP_FAILED, 184
 CUDA_ERROR_NO_BINARY_FOR_GPU, 184
 CUDA_ERROR_NO_DEVICE, 184
 CUDA_ERROR_NOT_FOUND, 185
 CUDA_ERROR_NOT_INITIALIZED, 183
 CUDA_ERROR_NOT_MAPPED, 184
 CUDA_ERROR_NOT_MAPPED_AS_ARRAY, 184
 CUDA_ERROR_NOT_MAPPED_AS_POINTER, 184
 CUDA_ERROR_NOT_READY, 185
 CUDA_ERROR_OPERATING_SYSTEM, 185
 CUDA_ERROR_OUT_OF_MEMORY, 183
 CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED, 185
 CUDA_ERROR_PEER_ACCESS_NOT_ENABLED, 185
 CUDA_ERROR_PEER_MEMORY_ALREADY_REGISTERED, 185
 CUDA_ERROR_PEER_MEMORY_NOT_REGISTERED, 185

CUDA_ERROR_PRIMARY_CONTEXT_ACTIVE, 185
CUDA_ERROR_PROFILER_ALREADY_STARTED, 184
CUDA_ERROR_PROFILER_ALREADY_STOPPED, 184
CUDA_ERROR_PROFILER_DISABLED, 183
CUDA_ERROR_PROFILER_NOT_INITIALIZED, 183
CUDA_ERROR_SHARED_OBJECT_INIT_FAILED, 184
CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND, 184
CUDA_ERROR_UNKNOWN, 185
CUDA_ERROR_UNMAP_FAILED, 184
CUDA_ERROR_UNSUPPORTED_LIMIT, 184
CUDA_SUCCESS, 183
CUDA_ARRAY3D_2DARRAY
 CUDA_TYPES, 178
CUDA_ARRAY3D_DESCRIPTOR
 CUDA_TYPES, 179
CUDA_ARRAY3D_DESCRIPTOR_st, 361
 Depth, 361
 Flags, 361
 Format, 361
 Height, 361
 NumChannels, 361
 Width, 362
CUDA_ARRAY3D_LAYERED
 CUDA_TYPES, 178
CUDA_ARRAY3D_SURFACE_LDST
 CUDA_TYPES, 178
CUDA_ARRAY_DESCRIPTOR
 CUDA_TYPES, 179
CUDA_ARRAY_DESCRIPTOR_st, 363
 Format, 363
 Height, 363
 NumChannels, 363
 Width, 363
CUDA_CTX
 cuCtxCreate, 200
 cuCtxDestroy, 201
 cuCtxGetApiVersion, 201
 cuCtxGetCacheConfig, 202
 cuCtxGetCurrent, 203
 cuCtxGetDevice, 203
 cuCtxGetLimit, 203
 cuCtxPopCurrent, 204
 cuCtxPushCurrent, 204
 cuCtxSetCacheConfig, 205
 cuCtxSetCurrent, 205
 cuCtxSetLimit, 206
 cuCtxSynchronize, 207
CUDA_CTX_DEPRECATED

cuCtxAttach, 208
cuCtxDetach, 208
CUDA_D3D10
 cuD3D10CtxCreate, 336
 cuD3D10CtxCreateOnDevice, 336
 CUd3d10DeviceList, 336
 CUd3d10DeviceList_enum, 336
 cuD3D10GetDevice, 337
 cuD3D10GetDevices, 337
 cuD3D10GetDirect3DDevice, 338
 cuGraphicsD3D10RegisterResource, 338
CUDA_D3D10_DEPRECATED
 CUD3D10map_flags, 342
 CUD3D10map_flags_enum, 342
 cuD3D10MapResources, 342
 CUD3D10register_flags, 342
 CUD3D10register_flags_enum, 342
 cuD3D10RegisterResource, 343
 cuD3D10ResourceGetMappedArray, 344
 cuD3D10ResourceGetMappedPitch, 345
 cuD3D10ResourceGetMappedPointer, 345
 cuD3D10ResourceGetMappedSize, 346
 cuD3D10ResourceGetSurfaceDimensions, 347
 cuD3D10ResourceSetMapFlags, 347
 cuD3D10UnmapResources, 348
 cuD3D10UnregisterResource, 349
CUDA_D3D11
 cuD3D11CtxCreate, 351
 cuD3D11CtxCreateOnDevice, 351
 CUd3d11DeviceList, 350
 CUd3d11DeviceList_enum, 351
 cuD3D11GetDevice, 352
 cuD3D11GetDevices, 352
 cuD3D11GetDirect3DDevice, 353
 cuGraphicsD3D11RegisterResource, 353
CUDA_D3D9
 cuD3D9CtxCreate, 321
 cuD3D9CtxCreateOnDevice, 321
 CUd3d9DeviceList, 321
 CUd3d9DeviceList_enum, 321
 cuD3D9GetDevice, 322
 cuD3D9GetDevices, 322
 cuD3D9GetDirect3DDevice, 323
 cuGraphicsD3D9RegisterResource, 323
CUDA_D3D9_DEPRECATED
 CUD3d9map_flags, 327
 CUD3d9map_flags_enum, 327
 cuD3D9MapResources, 327
 CUd3d9register_flags, 327
 CUD3d9register_flags_enum, 327
 cuD3D9RegisterResource, 328
 cuD3D9ResourceGetMappedArray, 329
 cuD3D9ResourceGetMappedPitch, 330
 cuD3D9ResourceGetMappedPointer, 331

cuD3D9ResourceGetMappedSize, 331
 cuD3D9ResourceGetSurfaceDimensions, 332
 cuD3D9ResourceSetMapFlags, 333
 cuD3D9UnmapResources, 333
 cuD3D9UnregisterResource, 334
CUDA_DEVICE
 cuDeviceComputeCapability, 193
 cuDeviceGet, 194
 cuDeviceGetAttribute, 194
 cuDeviceGetCount, 196
 cuDeviceGetName, 196
 cuDeviceGetProperties, 197
 cuDeviceTotalMem, 198
CUDA_EVENT
 cuEventCreate, 274
 cuEventDestroy, 275
 cuEventElapsedTime, 275
 cuEventQuery, 276
 cuEventRecord, 276
 cuEventSynchronize, 277
CUDA_EXEC
 cuFuncGetAttribute, 278
 cuFuncSetCacheConfig, 279
 cuLaunchKernel, 280
CUDA_EXEC_DEPRECATED
 cuFuncSetBlockShape, 282
 cuFuncSetSharedSize, 283
 cuLaunch, 283
 cuLaunchGrid, 284
 cuLaunchGridAsync, 284
 cuParamSetf, 285
 cuParamSeti, 286
 cuParamSetSize, 286
 cuParamSetTexRef, 287
 cuParamSetv, 287
CUDA_GL
 cuGLCtxCreate, 311
 cuGraphicsGLRegisterBuffer, 312
 cuGraphicsGLRegisterImage, 312
 cuWGLGetDevice, 313
CUDA_GL_DEPRECATED
 cuGLInit, 315
 CUGLmap_flags, 314
 CUGLmap_flags_enum, 315
 cuGLMapBufferObject, 315
 cuGLMapBufferObjectAsync, 316
 cuGLRegisterBufferObject, 316
 cuGLSetBufferObjectMapFlags, 317
 cuGLUnmapBufferObject, 317
 cuGLUnmapBufferObjectAsync, 318
 cuGLUnregisterBufferObject, 318
CUDA_GRAPHICS
 cuGraphicsMapResources, 306
 cuGraphicsResourceGetMappedPointer, 307
 cuGraphicsResourceSetMapFlags, 307
 cuGraphicsSubResourceGetMappedArray, 308
 cuGraphicsUnmapResources, 309
 cuGraphicsUnregisterResource, 309
CUDA_INITIALIZE
 cuInit, 191
CUDA_MEM
 cuArray3DCreate, 220
 cuArray3DGetDescriptor, 222
 cuArrayCreate, 223
 cuArrayDestroy, 224
 cuArrayGetDescriptor, 225
 cuMemAlloc, 225
 cuMemAllocHost, 226
 cuMemAllocPitch, 226
 cuMemcpy, 227
 cuMemcpy2D, 228
 cuMemcpy2DAsync, 230
 cuMemcpy2DUnaligned, 233
 cuMemcpy3D, 235
 cuMemcpy3DAsync, 238
 cuMemcpy3DPeer, 240
 cuMemcpy3DPeerAsync, 241
 cuMemcpyAsync, 241
 cuMemcpyAtoA, 242
 cuMemcpyAtoD, 243
 cuMemcpyAtoH, 243
 cuMemcpyAtoHAsync, 244
 cuMemcpyDtoA, 244
 cuMemcpyDtoD, 245
 cuMemcpyDtoDAsync, 246
 cuMemcpyDtoH, 246
 cuMemcpyDtoHAsync, 247
 cuMemcpyHtoA, 247
 cuMemcpyHtoAAsync, 248
 cuMemcpyHtoD, 249
 cuMemcpyHtoDAsync, 249
 cuMemcpyPeer, 250
 cuMemcpyPeerAsync, 251
 cuMemFree, 251
 cuMemFreeHost, 252
 cuMemGetAddressRange, 252
 cuMemGetInfo, 253
 cuMemHostAlloc, 253
 cuMemHostGetDevicePointer, 254
 cuMemHostGetFlags, 255
 cuMemHostRegister, 255
 cuMemHostUnregister, 256
 cuMemsetD16, 257
 cuMemsetD16Async, 257
 cuMemsetD2D16, 258
 cuMemsetD2D16Async, 259
 cuMemsetD2D32, 259
 cuMemsetD2D32Async, 260

cuMemsetD2D8, 261
cuMemsetD2D8Async, 261
cuMemsetD32, 262
cuMemsetD32Async, 262
cuMemsetD8, 263
cuMemsetD8Async, 264
cuProfilerInitialize, 264
cuProfilerStart, 265
cuProfilerStop, 265
CUDA_MEMCPY2D
 CUDA_TYPES, 179
CUDA_MEMCPY2D_st, 364
 dstArray, 364
 dstDevice, 364
 dstHost, 364
 dstMemoryType, 364
 dstPitch, 364
 dstXInBytes, 364
 dstY, 365
 Height, 365
 srcArray, 365
 srcDevice, 365
 srcHost, 365
 srcMemoryType, 365
 srcPitch, 365
 srcXInBytes, 365
 srcY, 365
 WidthInBytes, 365
CUDA_MEMCPY3D
 CUDA_TYPES, 179
CUDA_MEMCPY3D_PEER
 CUDA_TYPES, 179
CUDA_MEMCPY3D_PEER_st, 366
 Depth, 366
 dstArray, 366
 dstContext, 366
 dstDevice, 366
 dstHeight, 367
 dstHost, 367
 dstLOD, 367
 dstMemoryType, 367
 dstPitch, 367
 dstXInBytes, 367
 dstY, 367
 dstZ, 367
 Height, 367
 srcArray, 367
 srcContext, 367
 srcDevice, 367
 srcHeight, 368
 srcHost, 368
 srcLOD, 368
 srcMemoryType, 368
 srcPitch, 368
 srcXInBytes, 368
 srcY, 368
 srcZ, 368
 WidthInBytes, 368
CUDA_MEMCPY3D_st, 369
 Depth, 369
 dstArray, 369
 dstDevice, 369
 dstHeight, 369
 dstHost, 370
 dstLOD, 370
 dstMemoryType, 370
 dstPitch, 370
 dstXInBytes, 370
 dstY, 370
 dstZ, 370
 Height, 370
 reserved0, 370
 reserved1, 370
 srcArray, 370
 srcDevice, 370
 srcHeight, 371
 srcHost, 371
 srcLOD, 371
 srcMemoryType, 371
 srcPitch, 371
 srcXInBytes, 371
 srcY, 371
 srcZ, 371
 WidthInBytes, 371
CUDA_MODULE
 cuModuleGetFunction, 210
 cuModuleGetGlobal, 211
 cuModuleGetSurfRef, 211
 cuModuleGetTexRef, 212
 cuModuleLoad, 212
 cuModule loadData, 213
 cuModule loadDataEx, 213
 cuModuleLoadFatBinary, 215
 cuModuleUnload, 215
CUDA_PEER_ACCESS
 cuCtxDisablePeerAccess, 301
 cuCtxEnablePeerAccess, 302
 cuDeviceCanAccessPeer, 302
 cuMemPeerGetDevicePointer, 303
 cuMemPeerRegister, 304
 cuMemPeerUnregister, 305
CUDA_STREAM
 cuStreamCreate, 271
 cuStreamDestroy, 271
 cuStreamQuery, 272
 cuStreamSynchronize, 272
 cuStreamWaitEvent, 273
CUDA_SURFREF

cuSurfRefGetArray, 299
cuSurfRefSetArray, 299

CUDA_TEXREF
cuTexRefGetAddress, 290
cuTexRefGetAddressMode, 290
cuTexRefGetArray, 290
cuTexRefGetFilterMode, 291
cuTexRefGetFlags, 291
cuTexRefGetFormat, 292
cuTexRefSetAddress, 292
cuTexRefSetAddress2D, 293
cuTexRefSetAddressMode, 293
cuTexRefSetArray, 294
cuTexRefSetFilterMode, 294
cuTexRefSetFlags, 295
cuTexRefSetFormat, 295

CUDA_TEXREF_DEPRECATED
cuTexRefCreate, 297
cuTexRefDestroy, 297

CUDA_TYPES
CU_LAUNCH_PARAM_BUFFER_POINTER, 177
CU_LAUNCH_PARAM_BUFFER_SIZE, 177
CU_LAUNCH_PARAM_END, 177
CU_MEMHOSTALLOC_DEVICEMAP, 177
CU_MEMHOSTALLOC_PORTABLE, 177
CU_MEMHOSTALLOC_WRITECOMBINED,
 177
CU_MEMHOSTREGISTER_DEVICEMAP, 177
CU_MEMHOSTREGISTER_PORTABLE, 177
CU_MEMPEERREGISTER_DEVICEMAP, 178
CU_PARAM_TR_DEFAULT, 178
CU_TRAVA_OVERRIDE_FORMAT, 178
CU_TRSF_NORMALIZED_COORDINATES, 178
CU_TRSF_READ_AS_INTEGER, 178
CU_TRSF_SRGB, 178
CUaddress_mode, 179
CUaddress_mode_enum, 182
CUarray, 179
CUarray_cubemap_face, 179
CUarray_cubemap_face_enum, 182
CUarray_format, 179
CUarray_format_enum, 182
CUcomputemode, 179
CUcomputemode_enum, 182
CUcontext, 179
CUctx_flags, 179
CUctx_flags_enum, 183
CUDA_ARRAY3D_2DARRAY, 178
CUDA_ARRAY3D_DESCRIPTOR, 179
CUDA_ARRAY3D_LAYERED, 178
CUDA_ARRAY3D_SURFACE_LDST, 178
CUDA_ARRAY_DESCRIPTOR, 179
CUDA_MEMCPY2D, 179
CUDA_MEMCPY3D, 179

CUDA_MEMORYTYPE
CUDevice, 180
CUDevice_attribute, 180
CUDevice_attribute_enum, 185
CUDeviceptr, 180
CUDevprop, 180
CUevent, 180
CUevent_flags, 180
CUevent_flags_enum, 187
CUfilter_mode, 180
CUfilter_mode_enum, 187
CUfunc_cache, 180
CUfunc_cache_enum, 187
CUfunction, 180
CUfunction_attribute, 180
CUfunction_attribute_enum, 188
CUgraphicsMapResourceFlags, 180
CUgraphicsMapResourceFlags_enum, 188
CUgraphicsRegisterFlags, 180
CUgraphicsRegisterFlags_enum, 188
CUgraphicsResource, 181
CUjit_fallback, 181
CUjit_fallback_enum, 188
CUjit_option, 181
CUjit_option_enum, 189
CUjit_target, 181
CUjit_target_enum, 189
CULimit, 181
CULimit_enum, 190
CUMemorytype, 181
CUMemorytype_enum, 190
CUmodule, 181
CUOutputMode, 181
CUOutputMode_st, 190
CUpointer_attribute, 181
CUpointer_attribute_enum, 190
CUresult, 181
CUstream, 181
CUSurref, 181
CUTexref, 182

CUDA_UNIFIED
cuPointerGetAttribute, 268

CUDA_VDPAU
cuGraphicsVDPAURegisterOutputSurface, 356
cuGraphicsVDPAURegisterVideoSurface, 357
cuVDPACtxCreate, 358
cuVDPAGetDevice, 358

CUDA_VERSION
CUDA_TYPES, 178
cuDriverGetVersion, 192
cudaAddressModeBorder
CUDART_TYPES, 169

cudaAddressModeClamp
 CUDART_TYPES, 169
cudaAddressModeMirror
 CUDART_TYPES, 169
cudaAddressModeWrap
 CUDART_TYPES, 169
cudaArrayDefault
 CUDART_TYPES, 160
cudaArrayLayered
 CUDART_TYPES, 160
cudaArraySurfaceLoadStore
 CUDART_TYPES, 160
cudaBindSurfaceToArray
 CUDART_HIGHLEVEL, 119, 120
 CUDART_SURFACE, 115
cudaBindTexture
 CUDART_HIGHLEVEL, 120, 121
 CUDART_TEXTURE, 110
cudaBindTexture2D
 CUDART_HIGHLEVEL, 121, 122
 CUDART_TEXTURE, 111
cudaBindTextureToArray
 CUDART_HIGHLEVEL, 123
 CUDART_TEXTURE, 112
cudaBoundaryModeClamp
 CUDART_TYPES, 169
cudaBoundaryModeTrap
 CUDART_TYPES, 169
cudaBoundaryModeZero
 CUDART_TYPES, 169
cudaChannelFormatDesc, 372
 f, 372
 w, 372
 x, 372
 y, 372
 z, 372
cudaChannelFormatKind
 CUDART_TYPES, 163
cudaChannelFormatKindFloat
 CUDART_TYPES, 163
cudaChannelFormatKindNone
 CUDART_TYPES, 163
cudaChannelFormatKindSigned
 CUDART_TYPES, 163
cudaChannelFormatKindUnsigned
 CUDART_TYPES, 163
cudaChooseDevice
 CUDART_DEVICE, 14
cudaComputeMode
 CUDART_TYPES, 163
cudaComputeModeDefault
 CUDART_TYPES, 163
cudaComputeModeExclusive
 CUDART_TYPES, 163
cudaComputeModeExclusiveProcess
 CUDART_TYPES, 163
cudaComputeModeProhibited
 CUDART_TYPES, 163
cudaConfigureCall
 CUDART_EXECUTION, 37
cudaCreateChannelDesc
 CUDART_HIGHLEVEL, 124
 CUDART_TEXTURE, 112
cudaD3D10DeviceList
 CUDART_D3D10, 93
cudaD3D10DeviceListAll
 CUDART_D3D10, 93
cudaD3D10DeviceListCurrentFrame
 CUDART_D3D10, 93
cudaD3D10DeviceListNextFrame
 CUDART_D3D10, 93
cudaD3D10GetDevice
 CUDART_D3D10, 94
cudaD3D10GetDevices
 CUDART_D3D10, 94
cudaD3D10GetDirect3DDevice
 CUDART_D3D10, 95
cudaD3D10MapFlags
 CUDART_D3D10_DEPRECATED, 143
cudaD3D10MapFlagsNone
 CUDART_D3D10_DEPRECATED, 143
cudaD3D10MapFlagsReadOnly
 CUDART_D3D10_DEPRECATED, 143
cudaD3D10MapFlagsWriteDiscard
 CUDART_D3D10_DEPRECATED, 143
cudaD3D10MapResources
 CUDART_D3D10_DEPRECATED, 143
cudaD3D10RegisterFlags
 CUDART_D3D10_DEPRECATED, 143
cudaD3D10RegisterFlagsArray
 CUDART_D3D10_DEPRECATED, 143
cudaD3D10RegisterFlagsNone
 CUDART_D3D10_DEPRECATED, 143
cudaD3D10RegisterResource
 CUDART_D3D10_DEPRECATED, 144
cudaD3D10ResourceGetMappedArray
 CUDART_D3D10_DEPRECATED, 145
cudaD3D10ResourceGetMappedPitch
 CUDART_D3D10_DEPRECATED, 145
cudaD3D10ResourceGetMappedPointer
 CUDART_D3D10_DEPRECATED, 146
cudaD3D10ResourceGetMappedSize
 CUDART_D3D10_DEPRECATED, 147
cudaD3D10ResourceGetSurfaceDimensions
 CUDART_D3D10_DEPRECATED, 147
cudaD3D10ResourceSetMapFlags
 CUDART_D3D10_DEPRECATED, 148
cudaD3D10SetDirect3DDevice

CUDART_D3D10, 95
 cudaD3D10UnmapResources
 CUDART_D3D10_DEPRECATED, 149
 cudaD3D10UnregisterResource
 CUDART_D3D10_DEPRECATED, 149
 cudaD3D11DeviceList
 CUDART_D3D11, 98
 cudaD3D11DeviceListAll
 CUDART_D3D11, 98
 cudaD3D11DeviceListCurrentFrame
 CUDART_D3D11, 98
 cudaD3D11DeviceListNextFrame
 CUDART_D3D11, 98
 cudaD3D11GetDevice
 CUDART_D3D11, 99
 cudaD3D11GetDevices
 CUDART_D3D11, 99
 cudaD3D11GetDirect3DDevice
 CUDART_D3D11, 99
 cudaD3D11SetDirect3DDevice
 CUDART_D3D11, 100
 cudaD3D9DeviceList
 CUDART_D3D9, 88
 cudaD3D9DeviceListAll
 CUDART_D3D9, 88
 cudaD3D9DeviceListCurrentFrame
 CUDART_D3D9, 88
 cudaD3D9DeviceListNextFrame
 CUDART_D3D9, 88
 cudaD3D9GetDevice
 CUDART_D3D9, 89
 cudaD3D9GetDevices
 CUDART_D3D9, 89
 cudaD3D9GetDirect3DDevice
 CUDART_D3D9, 90
 cudaD3D9MapFlags
 CUDART_D3D9_DEPRECATED, 134
 cudaD3D9MapFlagsNone
 CUDART_D3D9_DEPRECATED, 134
 cudaD3D9MapFlagsReadOnly
 CUDART_D3D9_DEPRECATED, 134
 cudaD3D9MapFlagsWriteDiscard
 CUDART_D3D9_DEPRECATED, 134
 cudaD3D9MapResources
 CUDART_D3D9_DEPRECATED, 134
 cudaD3D9RegisterFlags
 CUDART_D3D9_DEPRECATED, 134
 cudaD3D9RegisterFlagsArray
 CUDART_D3D9_DEPRECATED, 134
 cudaD3D9RegisterFlagsNone
 CUDART_D3D9_DEPRECATED, 134
 cudaD3D9RegisterResource
 CUDART_D3D9_DEPRECATED, 135
 cudaD3D9ResourceGetMappedArray
 CUDART_D3D9_DEPRECATED, 136
 cudaD3D9ResourceGetMappedPitch
 CUDART_D3D9_DEPRECATED, 136
 cudaD3D9ResourceGetMappedPointer
 CUDART_D3D9_DEPRECATED, 137
 cudaD3D9ResourceGetMappedSize
 CUDART_D3D9_DEPRECATED, 138
 cudaD3D9ResourceGetSurfaceDimensions
 CUDART_D3D9_DEPRECATED, 139
 cudaD3D9ResourceSetMapFlags
 CUDART_D3D9_DEPRECATED, 139
 cudaD3D9SetDirect3DDevice
 CUDART_D3D9, 90
 cudaD3D9UnmapResources
 CUDART_D3D9_DEPRECATED, 140
 cudaD3D9UnregisterResource
 CUDART_D3D9_DEPRECATED, 141
 cudaDeviceBlockingSync
 CUDART_TYPES, 160
 cudaDeviceCanAccessPeer
 CUDART_PEER, 80
 cudaDeviceDisablePeerAccess
 CUDART_PEER, 81
 cudaDeviceEnablePeerAccess
 CUDART_PEER, 81
 cudaDeviceGetCacheConfig
 CUDART_DEVICE, 14
 cudaDeviceGetLimit
 CUDART_DEVICE, 14
 cudaDeviceLmemResizeToMax
 CUDART_TYPES, 160
 cudaDeviceMapHost
 CUDART_TYPES, 160
 cudaDeviceMask
 CUDART_TYPES, 160
 cudaDeviceProp, 373
 asyncEngineCount, 373
 canMapHostMemory, 373
 clockRate, 374
 computeMode, 374
 concurrentKernels, 374
 deviceOverlap, 374
 ECCEnabled, 374
 integrated, 374
 kernelExecTimeoutEnabled, 374
 major, 374
 maxGridSize, 374
 maxTexture1D, 374
 maxTexture1DLayered, 374
 maxTexture2D, 374
 maxTexture2DLayered, 375
 maxTexture3D, 375
 maxThreadsDim, 375
 maxThreadsPerBlock, 375

memPitch, 375
minor, 375
multiProcessorCount, 375
name, 375
pciBusID, 375
pciDeviceID, 375
regsPerBlock, 375
sharedMemPerBlock, 375
surfaceAlignment, 376
tccDriver, 376
textureAlignment, 376
totalConstMem, 376
totalGlobalMem, 376
unifiedAddressing, 376
warpSize, 376
cudaDevicePropDontCare
 CUDART_TYPES, 160
cudaDeviceReset
 CUDART_DEVICE, 15
cudaDeviceScheduleAuto
 CUDART_TYPES, 160
cudaDeviceScheduleBlockingSync
 CUDART_TYPES, 161
cudaDeviceScheduleSpin
 CUDART_TYPES, 161
cudaDeviceScheduleYield
 CUDART_TYPES, 161
cudaDeviceSetCacheConfig
 CUDART_DEVICE, 15
cudaDeviceSetLimit
 CUDART_DEVICE, 16
cudaDeviceSynchronize
 CUDART_DEVICE, 17
cudaDriverGetVersion
 CUDART__VERSION, 117
cudaError
 CUDART_TYPES, 163
cudaError_enum
 CUDA_TYPES, 183
cudaError_t
 CUDART_TYPES, 162
cudaErrorAddressOfConstant
 CUDART_TYPES, 164
cudaErrorApiFailureBase
 CUDART_TYPES, 167
cudaErrorCudartUnloading
 CUDART_TYPES, 165
cudaErrorDeviceAlreadyInUse
 CUDART_TYPES, 166
cudaErrorDevicesUnavailable
 CUDART_TYPES, 166
cudaErrorDuplicateSurfaceName
 CUDART_TYPES, 166
cudaErrorDuplicateTextureName
 CUDART_TYPES, 166
cudaErrorDuplicateVariableName
 CUDART_TYPES, 166
cudaErrorECCUncorrectable
 CUDART_TYPES, 166
cudaErrorIncompatibleDriverContext
 CUDART_TYPES, 166
cudaErrorInitializationError
 CUDART_TYPES, 163
cudaErrorInsufficientDriver
 CUDART_TYPES, 165
cudaErrorInvalidChannelDescriptor
 CUDART_TYPES, 164
cudaErrorInvalidConfiguration
 CUDART_TYPES, 164
cudaErrorInvalidDevice
 CUDART_TYPES, 164
cudaErrorInvalidDeviceFunction
 CUDART_TYPES, 164
cudaErrorInvalidDevicePointer
 CUDART_TYPES, 164
cudaErrorInvalidFilterSetting
 CUDART_TYPES, 165
cudaErrorInvalidHostPointer
 CUDART_TYPES, 164
cudaErrorInvalidKernelImage
 CUDART_TYPES, 166
cudaErrorInvalidMemcpyDirection
 CUDART_TYPES, 164
cudaErrorInvalidNormSetting
 CUDART_TYPES, 165
cudaErrorInvalidPitchValue
 CUDART_TYPES, 164
cudaErrorInvalidResourceHandle
 CUDART_TYPES, 165
cudaErrorInvalidSurface
 CUDART_TYPES, 166
cudaErrorInvalidSymbol
 CUDART_TYPES, 164
cudaErrorInvalidTexture
 CUDART_TYPES, 164
cudaErrorInvalidTextureBinding
 CUDART_TYPES, 164
cudaErrorInvalidValue
 CUDART_TYPES, 164
cudaErrorLaunchFailure
 CUDART_TYPES, 163
cudaErrorLaunchOutOfResources
 CUDART_TYPES, 164
cudaErrorLaunchTimeout
 CUDART_TYPES, 164
cudaErrorMapBufferObjectFailed
 CUDART_TYPES, 164
cudaErrorMemoryAllocation

CUDART_TYPES, 163
 cudaErrorMemoryValueTooLarge
 CUDART_TYPES, 165
 cudaErrorMissingConfiguration
 CUDART_TYPES, 163
 cudaErrorMixedDeviceExecution
 CUDART_TYPES, 165
 cudaErrorNoDevice
 CUDART_TYPES, 166
 cudaErrorNoKernelImageForDevice
 CUDART_TYPES, 166
 cudaErrorNotReady
 CUDART_TYPES, 165
 cudaErrorNotYetImplemented
 CUDART_TYPES, 165
 cudaErrorPeerAccessAlreadyEnabled
 CUDART_TYPES, 166
 cudaErrorPeerAccessNotEnabled
 CUDART_TYPES, 166
 cudaErrorPeerMemoryAlreadyRegistered
 CUDART_TYPES, 166
 cudaErrorPeerMemoryNotRegistered
 CUDART_TYPES, 166
 cudaErrorPriorLaunchFailure
 CUDART_TYPES, 163
 cudaErrorProfilerAlreadyStarted
 CUDART_TYPES, 167
 cudaErrorProfilerAlreadyStopped
 CUDART_TYPES, 167
 cudaErrorProfilerDisabled
 CUDART_TYPES, 166
 cudaErrorProfilerNotInitialized
 CUDART_TYPES, 166
 cudaErrorSetActiveProcess
 CUDART_TYPES, 165
 cudaErrorSharedObjectInitFailed
 CUDART_TYPES, 166
 cudaErrorSharedObjectSymbolNotFound
 CUDART_TYPES, 166
 cudaErrorStartupFailure
 CUDART_TYPES, 167
 cudaErrorSynchronizationError
 CUDART_TYPES, 165
 cudaErrorTextureFetchFailed
 CUDART_TYPES, 164
 cudaErrorTextureNotBound
 CUDART_TYPES, 165
 cudaErrorUnknown
 CUDART_TYPES, 165
 cudaErrorUnmapBufferObjectFailed
 CUDART_TYPES, 164
 cudaErrorUnsupportedLimit
 CUDART_TYPES, 166
 cudaEvent_t

CUDART_TYPES, 162
 cudaEventBlockingSync
 CUDART_TYPES, 161
 cudaEventCreate
 CUDART_EVENT, 33
 CUDART_HIGLEVEL, 124
 cudaEventCreateWithFlags
 CUDART_EVENT, 33
 cudaEventDefault
 CUDART_TYPES, 161
 cudaEventDestroy
 CUDART_EVENT, 34
 cudaEventDisableTiming
 CUDART_TYPES, 161
 cudaEventElapsedTime
 CUDART_EVENT, 34
 cudaEventQuery
 CUDART_EVENT, 35
 cudaEventRecord
 CUDART_EVENT, 35
 cudaEventSynchronize
 CUDART_EVENT, 36
 cudaExtent, 377
 depth, 377
 height, 377
 width, 377
 cudaFilterModeLinear
 CUDART_TYPES, 169
 cudaFilterModePoint
 CUDART_TYPES, 169
 cudaFormatModeAuto
 CUDART_TYPES, 169
 cudaFormatModeForced
 CUDART_TYPES, 169
 cudaFree
 CUDART_MEMORY, 44
 cudaFreeArray
 CUDART_MEMORY, 45
 cudaFreeHost
 CUDART_MEMORY, 45
 cudaFuncAttributes, 378
 binaryVersion, 378
 constSizeBytes, 378
 localSizeBytes, 378
 maxThreadsPerBlock, 378
 numRegs, 378
 ptxVersion, 378
 sharedSizeBytes, 378
 cudaFuncCache
 CUDART_TYPES, 167
 cudaFuncCachePreferL1
 CUDART_TYPES, 167
 cudaFuncCachePreferNone
 CUDART_TYPES, 167

cudaFuncCachePreferShared
 CUDART_TYPES, 167

cudaFuncGetAttributes
 CUDART_EXECUTION, 38

cudaFuncSetCacheConfig
 CUDART_EXECUTION, 38

cudaGetChannelDesc
 CUDART_TEXTURE, 113

cudaGetDevice
 CUDART_DEVICE, 17

cudaGetDeviceCount
 CUDART_DEVICE, 17

cudaGetDeviceProperties
 CUDART_DEVICE, 18

cudaGetErrorString
 CUDART_ERROR, 28

cudaGetLastError
 CUDART_ERROR, 28

cudaGetSurfaceReference
 CUDART_SURFACE, 115

cudaGetSymbolAddress
 CUDART_HIGHLEVEL, 126

 CUDART_MEMORY, 45

cudaGetSymbolSize
 CUDART_HIGHLEVEL, 127

 CUDART_MEMORY, 46

cudaGetTextureAlignmentOffset
 CUDART_HIGHLEVEL, 127

 CUDART_TEXTURE, 113

cudaGetTextureReference
 CUDART_TEXTURE, 114

cudaGLMapBufferObject
 CUDART_OPENGL_DEPRECATED, 152

cudaGLMapBufferObjectAsync
 CUDART_OPENGL_DEPRECATED, 152

cudaGLMapFlags
 CUDART_OPENGL_DEPRECATED, 151

cudaGLMapFlagsNone
 CUDART_OPENGL_DEPRECATED, 151

cudaGLMapFlagsReadOnly
 CUDART_OPENGL_DEPRECATED, 151

cudaGLMapFlagsWriteDiscard
 CUDART_OPENGL_DEPRECATED, 151

cudaGLRegisterBufferObject
 CUDART_OPENGL_DEPRECATED, 153

cudaGLSetBufferObjectMapFlags
 CUDART_OPENGL_DEPRECATED, 153

cudaGLUnmapBufferObject
 CUDART_OPENGL_DEPRECATED, 154

cudaGLUnmapBufferObjectAsync
 CUDART_OPENGL_DEPRECATED, 154

cudaGLUnregisterBufferObject
 CUDART_OPENGL_DEPRECATED, 154

cudaGraphicsCubeFace
 CUDART_TYPES, 167

cudaGraphicsCubeFaceNegativeX
 CUDART_TYPES, 167

cudaGraphicsCubeFaceNegativeY
 CUDART_TYPES, 167

cudaGraphicsCubeFaceNegativeZ
 CUDART_TYPES, 167

cudaGraphicsCubeFacePositiveX
 CUDART_TYPES, 167

cudaGraphicsCubeFacePositiveY
 CUDART_TYPES, 167

cudaGraphicsCubeFacePositiveZ
 CUDART_TYPES, 167

cudaGraphicsD3D10RegisterResource
 CUDART_D3D10, 95

cudaGraphicsD3D11RegisterResource
 CUDART_D3D11, 100

cudaGraphicsD3D9RegisterResource
 CUDART_D3D9, 90

cudaGraphicsGLRegisterBuffer
 CUDART_OPENGL, 85

cudaGraphicsGLRegisterImage
 CUDART_OPENGL, 86

cudaGraphicsMapFlags
 CUDART_TYPES, 167

cudaGraphicsMapFlagsNone
 CUDART_TYPES, 167

cudaGraphicsMapFlagsReadOnly
 CUDART_TYPES, 167

cudaGraphicsMapFlagsWriteDiscard
 CUDART_TYPES, 167

cudaGraphicsMapResources
 CUDART_INTEROP, 106

cudaGraphicsRegisterFlags
 CUDART_TYPES, 167

cudaGraphicsRegisterFlagsNone
 CUDART_TYPES, 168

cudaGraphicsRegisterFlagsReadOnly
 CUDART_TYPES, 168

cudaGraphicsRegisterFlagsSurfaceLoadStore
 CUDART_TYPES, 168

cudaGraphicsRegisterFlagsWriteDiscard
 CUDART_TYPES, 168

cudaGraphicsResource_t
 CUDART_TYPES, 162

cudaGraphicsResourceGetMappedPointer
 CUDART_INTEROP, 107

cudaGraphicsResourceSetMapFlags
 CUDART_INTEROP, 107

cudaGraphicsSubResourceGetMappedArray
 CUDART_INTEROP, 108

cudaGraphicsUnmapResources

CUDART_INTEROP, 108
 cudaGraphicsUnregisterResource
 CUDART_INTEROP, 109
 cudaGraphicsVDPAURegisterOutputSurface
 CUDART_VDPAU, 103
 cudaGraphicsVDPAURegisterVideoSurface
 CUDART_VDPAU, 104
 cudaHostAlloc
 CUDART_MEMORY, 46
 cudaHostAllocDefault
 CUDART_TYPES, 161
 cudaHostAllocMapped
 CUDART_TYPES, 161
 cudaHostAllocPortable
 CUDART_TYPES, 161
 cudaHostAllocWriteCombined
 CUDART_TYPES, 161
 cudaHostGetDevicePointer
 CUDART_MEMORY, 47
 cudaHostGetFlags
 CUDART_MEMORY, 48
 cudaHostRegister
 CUDART_MEMORY, 48
 cudaHostRegisterDefault
 CUDART_TYPES, 161
 cudaHostRegisterMapped
 CUDART_TYPES, 161
 cudaHostRegisterPortable
 CUDART_TYPES, 162
 cudaHostUnregister
 CUDART_MEMORY, 49
 cudaLaunch
 CUDART_EXECUTION, 39
 CUDART_HIGHLEVEL, 128
 cudaLimit
 CUDART_TYPES, 168
 cudaLimitMallocHeapSize
 CUDART_TYPES, 168
 cudaLimitPrintfFifoSize
 CUDART_TYPES, 168
 cudaLimitStackSize
 CUDART_TYPES, 168
 cudaMalloc
 CUDART_MEMORY, 49
 cudaMalloc3D
 CUDART_MEMORY, 50
 cudaMalloc3DArray
 CUDART_MEMORY, 50
 cudaMallocArray
 CUDART_MEMORY, 52
 cudaMallocHost
 CUDART_HIGHLEVEL, 128
 CUDART_MEMORY, 52
 cudaMallocPitch

CUDART_MEMORY, 53
 cudaMemcpy
 CUDART_MEMORY, 54
 cudaMemcpy2D
 CUDART_MEMORY, 54
 cudaMemcpy2DArrayToArray
 CUDART_MEMORY, 55
 cudaMemcpy2DAsync
 CUDART_MEMORY, 56
 cudaMemcpy2DFromArray
 CUDART_MEMORY, 56
 cudaMemcpy2DFromArrayAsync
 CUDART_MEMORY, 57
 cudaMemcpy2DToArray
 CUDART_MEMORY, 58
 cudaMemcpy2DToArrayAsync
 CUDART_MEMORY, 59
 cudaMemcpy3D
 CUDART_MEMORY, 60
 cudaMemcpy3DAsync
 CUDART_MEMORY, 61
 cudaMemcpy3DParms, 380
 dstArray, 380
 dstPos, 380
 dstPtr, 380
 extent, 380
 kind, 380
 srcArray, 380
 srcPos, 380
 srcPtr, 380
 cudaMemcpy3DPeer
 CUDART_MEMORY, 62
 cudaMemcpy3DPeerAsync
 CUDART_MEMORY, 63
 cudaMemcpy3DPeerParms, 382
 dstArray, 382
 dstDevice, 382
 dstPos, 382
 dstPtr, 382
 extent, 382
 srcArray, 382
 srcDevice, 382
 srcPos, 382
 srcPtr, 383
 cudaMemcpyArrayToArray
 CUDART_MEMORY, 63
 cudaMemcpyAsync
 CUDART_MEMORY, 64
 cudaMemcpyDefault
 CUDART_TYPES, 168
 cudaMemcpyDeviceToDevice
 CUDART_TYPES, 168
 cudaMemcpyDeviceToHost
 CUDART_TYPES, 168

cudaMemcpyFromArray
 CUDART_MEMORY, 65

cudaMemcpyFromArrayAsync
 CUDART_MEMORY, 65

cudaMemcpyFromSymbol
 CUDART_MEMORY, 66

cudaMemcpyFromSymbolAsync
 CUDART_MEMORY, 67

cudaMemcpyHostToDevice
 CUDART_TYPES, 168

cudaMemcpyHostToHost
 CUDART_TYPES, 168

cudaMemcpyKind
 CUDART_TYPES, 168

cudaMemcpyPeer
 CUDART_MEMORY, 67

cudaMemcpyPeerAsync
 CUDART_MEMORY, 68

cudaMemcpyToArray
 CUDART_MEMORY, 68

cudaMemcpyToArrayAsync
 CUDART_MEMORY, 69

cudaMemcpyToSymbol
 CUDART_MEMORY, 70

cudaMemcpyToSymbolAsync
 CUDART_MEMORY, 70

cudaMemGetInfo
 CUDART_MEMORY, 71

cudaMemoryType
 CUDART_TYPES, 168

cudaMemoryTypeDevice
 CUDART_TYPES, 168

cudaMemoryTypeHost
 CUDART_TYPES, 168

cudaMemset
 CUDART_MEMORY, 71

cudaMemset2D
 CUDART_MEMORY, 72

cudaMemset2DAsync
 CUDART_MEMORY, 72

cudaMemset3D
 CUDART_MEMORY, 73

cudaMemset3DAsync
 CUDART_MEMORY, 74

cudaMemsetAsync
 CUDART_MEMORY, 74

cudaOutputMode
 CUDART_TYPES, 168

cudaOutputMode_t
 CUDART_TYPES, 162

cudaPeekAtLastError
 CUDART_ERROR, 29

cudaPeerAccessDefault
 CUDART_TYPES, 162

cudaPeerDevicePointerDefault
 CUDART_TYPES, 162

cudaPeerGetDevicePointer
 CUDART_PEER, 82

cudaPeerRegister
 CUDART_PEER, 82

cudaPeerRegisterDefault
 CUDART_TYPES, 162

cudaPeerRegisterMapped
 CUDART_TYPES, 162

cudaPeerUnregister
 CUDART_PEER, 83

cudaPitchedPtr, 384

- pitch, 384
- ptr, 384
- xsize, 384
- ysize, 384

cudaPointerAttributes, 385

- device, 385
- devicePointer, 385
- hostPointer, 385
- memoryType, 385

cudaPointerGetAttributes
 CUDART_UNIFIED, 78

cudaPos, 386

- x, 386
- y, 386
- z, 386

cudaReadModeElementType
 CUDART_TYPES, 169

cudaReadModeNormalizedFloat
 CUDART_TYPES, 169

CUDART
 CUDART_VERSION, 12

CUDART_D3D10

- cudaD3D10DeviceListAll, 93
- cudaD3D10DeviceListCurrentFrame, 93
- cudaD3D10DeviceListNextFrame, 93

CUDART_D3D10_DEPRECATED

- cudaD3D10MapFlagsNone, 143
- cudaD3D10MapFlagsReadOnly, 143
- cudaD3D10MapFlagsWriteDiscard, 143
- cudaD3D10RegisterFlagsArray, 143
- cudaD3D10RegisterFlagsNone, 143

CUDART_D3D11

- cudaD3D11DeviceListAll, 98
- cudaD3D11DeviceListCurrentFrame, 98
- cudaD3D11DeviceListNextFrame, 98

CUDART_D3D9

- cudaD3D9DeviceListAll, 88
- cudaD3D9DeviceListCurrentFrame, 88
- cudaD3D9DeviceListNextFrame, 88

CUDART_D3D9_DEPRECATED

- cudaD3D9MapFlagsNone, 134

cudaD3D9MapFlagsReadOnly, 134
 cudaD3D9MapFlagsWriteDiscard, 134
 cudaD3D9RegisterFlagsArray, 134
 cudaD3D9RegisterFlagsNone, 134
CUDART_OPENGL_DEPRECATED
 cudaGLMapFlagsNone, 151
 cudaGLMapFlagsReadOnly, 151
 cudaGLMapFlagsWriteDiscard, 151
CUDART_TYPES
 cudaAddressModeBorder, 169
 cudaAddressModeClamp, 169
 cudaAddressModeMirror, 169
 cudaAddressModeWrap, 169
 cudaBoundaryModeClamp, 169
 cudaBoundaryModeTrap, 169
 cudaBoundaryModeZero, 169
 cudaChannelFormatKindFloat, 163
 cudaChannelFormatKindNone, 163
 cudaChannelFormatKindSigned, 163
 cudaChannelFormatKindUnsigned, 163
 cudaComputeModeDefault, 163
 cudaComputeModeExclusive, 163
 cudaComputeModeExclusiveProcess, 163
 cudaComputeModeProhibited, 163
 cudaErrorAddressOfConstant, 164
 cudaErrorApiFailureBase, 167
 cudaErrorCudartUnloading, 165
 cudaErrorDeviceAlreadyInUse, 166
 cudaErrorDevicesUnavailable, 166
 cudaErrorDuplicateSurfaceName, 166
 cudaErrorDuplicateTextureName, 166
 cudaErrorDuplicateVariableName, 166
 cudaErrorECCUncorrectable, 166
 cudaErrorIncompatibleDriverContext, 166
 cudaErrorInitializationError, 163
 cudaErrorInsufficientDriver, 165
 cudaErrorInvalidChannelDescriptor, 164
 cudaErrorInvalidConfiguration, 164
 cudaErrorInvalidDevice, 164
 cudaErrorInvalidDeviceFunction, 164
 cudaErrorInvalidDevicePointer, 164
 cudaErrorInvalidFilterSetting, 165
 cudaErrorInvalidHostPointer, 164
 cudaErrorInvalidKernelImage, 166
 cudaErrorInvalidMemcpyDirection, 164
 cudaErrorInvalidNormSetting, 165
 cudaErrorInvalidPitchValue, 164
 cudaErrorInvalidResourceHandle, 165
 cudaErrorInvalidSurface, 166
 cudaErrorInvalidSymbol, 164
 cudaErrorInvalidTexture, 164
 cudaErrorInvalidTextureBinding, 164
 cudaErrorInvalidValue, 164
 cudaErrorLaunchFailure, 163
 cudaErrorLaunchOutOfResources, 164
 cudaErrorLaunchTimeout, 164
 cudaErrorMapBufferObjectFailed, 164
 cudaErrorMemoryAllocation, 163
 cudaErrorMemoryValueTooLarge, 165
 cudaErrorMissingConfiguration, 163
 cudaErrorMixedDeviceExecution, 165
 cudaErrorNoDevice, 166
 cudaErrorNoKernelImageForDevice, 166
 cudaErrorNotReady, 165
 cudaErrorNotYetImplemented, 165
 cudaErrorPeerAccessAlreadyEnabled, 166
 cudaErrorPeerAccessNotEnabled, 166
 cudaErrorPeerMemoryAlreadyRegistered, 166
 cudaErrorPeerMemoryNotRegistered, 166
 cudaErrorPriorLaunchFailure, 163
 cudaErrorProfilerAlreadyStarted, 167
 cudaErrorProfilerAlreadyStopped, 167
 cudaErrorProfilerDisabled, 166
 cudaErrorProfilerNotInitialized, 166
 cudaErrorSetOnActiveProcess, 165
 cudaErrorSharedObjectInitFailed, 166
 cudaErrorSharedObjectSymbolNotFound, 166
 cudaErrorStartupFailure, 167
 cudaErrorSynchronizationError, 165
 cudaErrorTextureFetchFailed, 164
 cudaErrorTextureNotBound, 165
 cudaErrorUnknown, 165
 cudaErrorUnmapBufferObjectFailed, 164
 cudaErrorUnsupportedLimit, 166
 cudaFilterModeLinear, 169
 cudaFilterModePoint, 169
 cudaFormatModeAuto, 169
 cudaFormatModeForced, 169
 cudaFuncCachePreferL1, 167
 cudaFuncCachePreferNone, 167
 cudaFuncCachePreferShared, 167
 cudaGraphicsCubeFaceNegativeX, 167
 cudaGraphicsCubeFaceNegativeY, 167
 cudaGraphicsCubeFaceNegativeZ, 167
 cudaGraphicsCubeFacePositiveX, 167
 cudaGraphicsCubeFacePositiveY, 167
 cudaGraphicsCubeFacePositiveZ, 167
 cudaGraphicsMapFlagsNone, 167
 cudaGraphicsMapFlagsReadOnly, 167
 cudaGraphicsMapFlagsWriteDiscard, 167
 cudaGraphicsRegisterFlagsNone, 168
 cudaGraphicsRegisterFlagsReadOnly, 168
 cudaGraphicsRegisterFlagsSurfaceLoadStore, 168
 cudaGraphicsRegisterFlagsWriteDiscard, 168
 cudaLimitMallocHeapSize, 168
 cudaLimitPrintfFifoSize, 168
 cudaLimitStackSize, 168
 cudaMemcpyDefault, 168

cudaMemcpyDeviceToDevice, 168
cudaMemcpyDeviceToHost, 168
cudaMemcpyHostToDevice, 168
cudaMemcpyHostToHost, 168
cudaMemoryTypeDevice, 168
cudaMemoryTypeHost, 168
cudaReadModeElementType, 169
cudaReadModeNormalizedFloat, 169
cudaSuccess, 163

CUDART__VERSION
 cudaDriverGetVersion, 117
 cudaRuntimeGetVersion, 117

CUDART_D3D10
 cudaD3D10DeviceList, 93
 cudaD3D10GetDevice, 94
 cudaD3D10GetDevices, 94
 cudaD3D10GetDirect3DDevice, 95
 cudaD3D10SetDirect3DDevice, 95
 cudaGraphicsD3D10RegisterResource, 95

CUDART_D3D10_DEPRECATED
 cudaD3D10MapFlags, 143
 cudaD3D10MapResources, 143
 cudaD3D10RegisterFlags, 143
 cudaD3D10RegisterResource, 144
 cudaD3D10ResourceGetMappedArray, 145
 cudaD3D10ResourceGetMappedPitch, 145
 cudaD3D10ResourceGetMappedPointer, 146
 cudaD3D10ResourceGetMappedSize, 147
 cudaD3D10ResourceGetSurfaceDimensions, 147
 cudaD3D10ResourceSetMapFlags, 148
 cudaD3D10UnmapResources, 149
 cudaD3D10UnregisterResource, 149

CUDART_D3D11
 cudaD3D11DeviceList, 98
 cudaD3D11GetDevice, 99
 cudaD3D11GetDevices, 99
 cudaD3D11GetDirect3DDevice, 99
 cudaD3D11SetDirect3DDevice, 100
 cudaGraphicsD3D11RegisterResource, 100

CUDART_D3D9
 cudaD3D9DeviceList, 88
 cudaD3D9GetDevice, 89
 cudaD3D9GetDevices, 89
 cudaD3D9GetDirect3DDevice, 90
 cudaD3D9SetDirect3DDevice, 90
 cudaGraphicsD3D9RegisterResource, 90

CUDART_D3D9_DEPRECATED
 cudaD3D9MapFlags, 134
 cudaD3D9MapResources, 134
 cudaD3D9RegisterFlags, 134
 cudaD3D9RegisterResource, 135
 cudaD3D9ResourceGetMappedArray, 136
 cudaD3D9ResourceGetMappedPitch, 136
 cudaD3D9ResourceGetMappedPointer, 137

cudaD3D9ResourceGetMappedSize, 138
cudaD3D9ResourceGetSurfaceDimensions, 139
cudaD3D9ResourceSetMapFlags, 139
cudaD3D9UnmapResources, 140
cudaD3D9UnregisterResource, 141

CUDART_DEVICE
 cudaChooseDevice, 14
 cudaDeviceGetCacheConfig, 14
 cudaDeviceGetLimit, 14
 cudaDeviceReset, 15
 cudaDeviceSetCacheConfig, 15
 cudaDeviceSetLimit, 16
 cudaDeviceSynchronize, 17
 cudaGetDevice, 17
 cudaGetDeviceCount, 17
 cudaGetDeviceProperties, 18
 cudaSetDevice, 20
 cudaSetDeviceFlags, 20
 cudaSetValidDevices, 21

CUDART_ERROR
 cudaGetErrorString, 28
 cudaGetLastError, 28
 cudaPeekAtLastError, 29

CUDART_EVENT
 cudaEventCreate, 33
 cudaEventCreateWithFlags, 33
 cudaEventDestroy, 34
 cudaEventElapsedTime, 34
 cudaEventQuery, 35
 cudaEventRecord, 35
 cudaEventSynchronize, 36

CUDART_EXECUTION
 cudaConfigureCall, 37
 cudaFuncGetAttributes, 38
 cudaFuncSetCacheConfig, 38
 cudaLaunch, 39
 cudaSetDoubleForDevice, 39
 cudaSetDoubleForHost, 40
 cudaSetupArgument, 40

CUDART_HIGLEVEL
 cudaBindSurfaceToArray, 119, 120
 cudaBindTexture, 120, 121
 cudaBindTexture2D, 121, 122
 cudaBindTextureToArray, 123
 cudaCreateChannelDesc, 124
 cudaEventCreate, 124
 cudaFuncGetAttributes, 125
 cudaFuncSetCacheConfig, 126
 cudaGetSymbolAddress, 126
 cudaGetSymbolSize, 127
 cudaGetTextureAlignmentOffset, 127
 cudaLaunch, 128
 cudaMallocHost, 128
 cudaSetupArgument, 129

cudaUnbindTexture, 130
CUDART_INTEROP
 cudaGraphicsMapResources, 106
 cudaGraphicsResourceGetMappedPointer, 107
 cudaGraphicsResourceSetMapFlags, 107
 cudaGraphicsSubResourceGetMappedArray, 108
 cudaGraphicsUnmapResources, 108
 cudaGraphicsUnregisterResource, 109
CUDART_MEMORY
 cudaFree, 44
 cudaFreeArray, 45
 cudaFreeHost, 45
 cudaGetSymbolAddress, 45
 cudaGetSymbolSize, 46
 cudaHostAlloc, 46
 cudaHostGetDevicePointer, 47
 cudaHostGetFlags, 48
 cudaHostRegister, 48
 cudaHostUnregister, 49
 cudaMalloc, 49
 cudaMalloc3D, 50
 cudaMalloc3DArray, 50
 cudaMallocArray, 52
 cudaMallocHost, 52
 cudaMallocPitch, 53
 cudaMemcpy, 54
 cudaMemcpy2D, 54
 cudaMemcpy2DArrayToArray, 55
 cudaMemcpy2DAsync, 56
 cudaMemcpy2DFromArray, 56
 cudaMemcpy2DFromArrayAsync, 57
 cudaMemcpy2DToArray, 58
 cudaMemcpy2DToArrayAsync, 59
 cudaMemcpy3D, 60
 cudaMemcpy3DAsync, 61
 cudaMemcpy3DPeer, 62
 cudaMemcpy3DPeerAsync, 63
 cudaMemcpyArrayToArray, 63
 cudaMemcpyAsync, 64
 cudaMemcpyFromArray, 65
 cudaMemcpyFromArrayAsync, 65
 cudaMemcpyFromSymbol, 66
 cudaMemcpyFromSymbolAsync, 67
 cudaMemcpyPeer, 67
 cudaMemcpyPeerAsync, 68
 cudaMemcpyToArray, 68
 cudaMemcpyToArrayAsync, 69
 cudaMemcpyToSymbol, 70
 cudaMemcpyToSymbolAsync, 70
 cudaMemGetInfo, 71
 cudaMemset, 71
 cudaMemset2D, 72
 cudaMemset2DAsync, 72
 cudaMemset3D, 73
 cudaMemset3DAsync, 74
 cudaMemsetAsync, 74
 make_cudaExtent, 75
 make_cudaPitchedPtr, 75
 make_cudaPos, 76
CUDART_OPENGL
 cudaGraphicsGLRegisterBuffer, 85
 cudaGraphicsGLRegisterImage, 86
 cudaWGLGetDevice, 86
CUDART_OPENGL_DEPRECATED
 cudaGLMapBufferObject, 152
 cudaGLMapBufferObjectAsync, 152
 cudaGLMapFlags, 151
 cudaGLRegisterBufferObject, 153
 cudaGLSetBufferObjectMapFlags, 153
 cudaGLUnmapBufferObject, 154
 cudaGLUnmapBufferObjectAsync, 154
 cudaGLUnregisterBufferObject, 155
CUDART_PEER
 cudaDeviceCanAccessPeer, 80
 cudaDeviceDisablePeerAccess, 81
 cudaDeviceEnablePeerAccess, 81
 cudaPeerGetDevicePointer, 82
 cudaPeerRegister, 82
 cudaPeerUnregister, 83
CUDART_STREAM
 cudaStreamCreate, 30
 cudaStreamDestroy, 30
 cudaStreamQuery, 31
 cudaStreamSynchronize, 31
 cudaStreamWaitEvent, 31
CUDART_SURFACE
 cudaBindSurfaceToArray, 115
 cudaGetSurfaceReference, 115
CUDART_TEXTURE
 cudaBindTexture, 110
 cudaBindTexture2D, 111
 cudaBindTextureToArray, 112
 cudaCreateChannelDesc, 112
 cudaGetChannelDesc, 113
 cudaGetTextureAlignmentOffset, 113
 cudaGetTextureReference, 114
 cudaUnbindTexture, 114
CUDART_THREAD_DEPRECATED
 cudaThreadExit, 23
 cudaThreadGetCacheConfig, 24
 cudaThreadGetLimit, 24
 cudaThreadSetCacheConfig, 25
 cudaThreadSetLimit, 26
 cudaThreadSynchronize, 26
CUDART_TYPES
 cudaArrayDefault, 160
 cudaArrayLayered, 160
 cudaArraySurfaceLoadStore, 160

cudaChannelFormatKind, 163
cudaComputeMode, 163
cudaDeviceBlockingSync, 160
cudaDeviceLmemResizeToMax, 160
cudaDeviceMapHost, 160
cudaDeviceMask, 160
cudaDevicePropDontCare, 160
cudaDeviceScheduleAuto, 160
cudaDeviceScheduleBlockingSync, 161
cudaDeviceScheduleSpin, 161
cudaDeviceScheduleYield, 161
cudaError, 163
cudaError_t, 162
cudaEvent_t, 162
cudaEventBlockingSync, 161
cudaEventDefault, 161
cudaEventDisableTiming, 161
cudaFuncCache, 167
cudaGraphicsCubeFace, 167
cudaGraphicsMapFlags, 167
cudaGraphicsRegisterFlags, 167
cudaGraphicsResource_t, 162
cudaHostAllocDefault, 161
cudaHostAllocMapped, 161
cudaHostAllocPortable, 161
cudaHostAllocWriteCombined, 161
cudaHostRegisterDefault, 161
cudaHostRegisterMapped, 161
cudaHostRegisterPortable, 162
cudaLimit, 168
cudaMemcpyKind, 168
cudaMemoryType, 168
cudaOutputMode, 168
cudaOutputMode_t, 162
cudaPeerAccessDefault, 162
cudaPeerDevicePointerDefault, 162
cudaPeerRegisterDefault, 162
cudaPeerRegisterMapped, 162
cudaStream_t, 162
cudaSurfaceBoundaryMode, 168
cudaSurfaceFormatMode, 169
cudaTextureAddressMode, 169
cudaTextureFilterMode, 169
cudaTextureReadMode, 169
cudaUUID_t, 162
CUDART_UNIFIED
 cudaPointerGetAttributes, 78
CUDART_VDPAU
 cudaGraphicsVDPAURegisterOutputSurface, 103
 cudaGraphicsVDPAURegisterVideoSurface, 104
 cudaVDPAUGetDevice, 104
 cudaVDPAUSetVDPAUDevice, 105
CUDART_VERSION
 CUDART, 12
 cudaRuntimeGetVersion
 CUDART__VERSION, 117
 cudaSetDevice
 CUDART_DEVICE, 20
 cudaSetDeviceFlags
 CUDART_DEVICE, 20
 cudaSetDoubleForDevice
 CUDART_EXECUTION, 39
 cudaSetDoubleForHost
 CUDART_EXECUTION, 40
 cudaSetupArgument
 CUDART_EXECUTION, 40
 CUDART_HIGHLVEL, 129
 cudaSetValidDevices
 CUDART_DEVICE, 21
 cudaStream_t
 CUDART_TYPES, 162
 cudaStreamCreate
 CUDART_STREAM, 30
 cudaStreamDestroy
 CUDART_STREAM, 30
 cudaStreamQuery
 CUDART_STREAM, 31
 cudaStreamSynchronize
 CUDART_STREAM, 31
 cudaStreamWaitEvent
 CUDART_STREAM, 31
 cudaSuccess
 CUDART_TYPES, 163
 cudaSurfaceBoundaryMode
 CUDART_TYPES, 168
 cudaSurfaceFormatMode
 CUDART_TYPES, 169
 cudaTextureAddressMode
 CUDART_TYPES, 169
 cudaTextureFilterMode
 CUDART_TYPES, 169
 cudaTextureReadMode
 CUDART_TYPES, 169
 cudaThreadExit
 CUDART_THREAD_DEPRECATED, 23
 cudaThreadGetCacheConfig
 CUDART_THREAD_DEPRECATED, 24
 cudaThreadGetLimit
 CUDART_THREAD_DEPRECATED, 24
 cudaThreadSetCacheConfig
 CUDART_THREAD_DEPRECATED, 25
 cudaThreadSetLimit
 CUDART_THREAD_DEPRECATED, 26
 cudaThreadSynchronize
 CUDART_THREAD_DEPRECATED, 26
 cudaUnbindTexture
 CUDART_HIGHLVEL, 130
 CUDART_TEXTURE, 114

cudaUUID_t
 CUDART_TYPES, 162
 cudaVDPAUGetDevice
 CUDART_VDPAU, 104
 cudaVDPAUSetVDPAUDevice
 CUDART_VDPAU, 105
 cudaWGLGetDevice
 CUDART_OPENGL, 86
 CUdevice
 CUDA_TYPES, 180
 CUdevice_attribute
 CUDA_TYPES, 180
 CUdevice_attribute_enum
 CUDA_TYPES, 185
 cuDeviceCanAccessPeer
 CUDA_PEER_ACCESS, 302
 cuDeviceComputeCapability
 CUDA_DEVICE, 193
 cuDeviceGet
 CUDA_DEVICE, 194
 cuDeviceGetAttribute
 CUDA_DEVICE, 194
 cuDeviceGetCount
 CUDA_DEVICE, 196
 cuDeviceGetName
 CUDA_DEVICE, 196
 cuDeviceGetProperties
 CUDA_DEVICE, 197
 CUdeviceptr
 CUDA_TYPES, 180
 cuDeviceTotalMem
 CUDA_DEVICE, 198
 CUdevprop
 CUDA_TYPES, 180
 CUdevprop_st, 387
 clockRate, 387
 maxGridSize, 387
 maxThreadsDim, 387
 maxThreadsPerBlock, 387
 memPitch, 387
 regsPerBlock, 387
 sharedMemPerBlock, 387
 SIMDWidth, 387
 textureAlign, 388
 totalConstantMemory, 388
 cuDriverGetVersion
 CUDA_VERSION, 192
 CUevent
 CUDA_TYPES, 180
 CUevent_flags
 CUDA_TYPES, 180
 CUevent_flags_enum
 CUDA_TYPES, 187
 cuEventCreate
 CUDA_EVENT, 274
 cuEventDestroy
 CUDA_EVENT, 275
 cuEventElapsedTime
 CUDA_EVENT, 275
 cuEventQuery
 CUDA_EVENT, 276
 cuEventRecord
 CUDA_EVENT, 276
 cuEventSynchronize
 CUDA_EVENT, 277
 CUfilter_mode
 CUDA_TYPES, 180
 CUfilter_mode_enum
 CUDA_TYPES, 187
 CUfunc_cache
 CUDA_TYPES, 180
 CUfunc_cache_enum
 CUDA_TYPES, 187
 cuFuncGetAttribute
 CUDA_EXEC, 278
 cuFuncSetBlockShape
 CUDA_EXEC_DEPRECATED, 282
 cuFuncSetCacheConfig
 CUDA_EXEC, 279
 cuFuncSetSharedSize
 CUDA_EXEC_DEPRECATED, 283
 CUfunction
 CUDA_TYPES, 180
 CUfunction_attribute
 CUDA_TYPES, 180
 CUfunction_attribute_enum
 CUDA_TYPES, 188
 cuGLCtxCreate
 CUDA_GL, 311
 cuGLInit
 CUDA_GL_DEPRECATED, 315
 CUGLmap_flags
 CUDA_GL_DEPRECATED, 314
 CUGLmap_flags_enum
 CUDA_GL_DEPRECATED, 315
 cuGLMapBufferObject
 CUDA_GL_DEPRECATED, 315
 cuGLMapBufferObjectAsync
 CUDA_GL_DEPRECATED, 316
 cuGLRegisterBufferObject
 CUDA_GL_DEPRECATED, 316
 cuGLSetBufferObjectMapFlags
 CUDA_GL_DEPRECATED, 317
 cuGLUnmapBufferObject
 CUDA_GL_DEPRECATED, 317
 cuGLUnmapBufferObjectAsync
 CUDA_GL_DEPRECATED, 318
 cuGLUnregisterBufferObject

CUDA_GL_DEPRECATED, 318
cuGraphicsD3D10RegisterResource
 CUDA_D3D10, 338
cuGraphicsD3D11RegisterResource
 CUDA_D3D11, 353
cuGraphicsD3D9RegisterResource
 CUDA_D3D9, 323
cuGraphicsGLRegisterBuffer
 CUDA_GL, 312
cuGraphicsGLRegisterImage
 CUDA_GL, 312
CUgraphicsMapResourceFlags
 CUDA_TYPES, 180
CUgraphicsMapResourceFlags_enum
 CUDA_TYPES, 188
cuGraphicsMapResources
 CUDA_GRAPHICS, 306
CUgraphicsRegisterFlags
 CUDA_TYPES, 180
CUgraphicsRegisterFlags_enum
 CUDA_TYPES, 188
CUgraphicsResource
 CUDA_TYPES, 181
cuGraphicsResourceGetMappedPointer
 CUDA_GRAPHICS, 307
cuGraphicsResourceSetMapFlags
 CUDA_GRAPHICS, 307
cuGraphicsSubResourceGetMappedArray
 CUDA_GRAPHICS, 308
cuGraphicsUnmapResources
 CUDA_GRAPHICS, 309
cuGraphicsUnregisterResource
 CUDA_GRAPHICS, 309
cuGraphicsVDPAURegisterOutputSurface
 CUDA_VDPAU, 356
cuGraphicsVDPAURegisterVideoSurface
 CUDA_VDPAU, 357
cuInit
 CUDA_INITIALIZE, 191
CUjit_fallback
 CUDA_TYPES, 181
CUjit_fallback_enum
 CUDA_TYPES, 188
CUjit_option
 CUDA_TYPES, 181
CUjit_option_enum
 CUDA_TYPES, 189
CUjit_target
 CUDA_TYPES, 181
CUjit_target_enum
 CUDA_TYPES, 189
cuLaunch
 CUDA_EXEC_DEPRECATED, 283
cuLaunchGrid
 CUDA_EXEC_DEPRECATED, 284
cuLaunchGridAsync
 CUDA_EXEC_DEPRECATED, 284
cuLaunchKernel
 CUDA_EXEC, 280
CUlimit
 CUDA_TYPES, 181
CUlimit_enum
 CUDA_TYPES, 190
cuMemAlloc
 CUDA_MEM, 225
cuMemAllocHost
 CUDA_MEM, 226
cuMemAllocPitch
 CUDA_MEM, 226
cuMemcpy
 CUDA_MEM, 227
cuMemcpy2D
 CUDA_MEM, 228
cuMemcpy2DAsync
 CUDA_MEM, 230
cuMemcpy2DUnaligned
 CUDA_MEM, 233
cuMemcpy3D
 CUDA_MEM, 235
cuMemcpy3DAsync
 CUDA_MEM, 238
cuMemcpy3DPeer
 CUDA_MEM, 240
cuMemcpy3DPeerAsync
 CUDA_MEM, 241
cuMemcpyAsync
 CUDA_MEM, 241
cuMemcpyAtoA
 CUDA_MEM, 242
cuMemcpyAtoD
 CUDA_MEM, 243
cuMemcpyAtoH
 CUDA_MEM, 243
cuMemcpyAtoHAsync
 CUDA_MEM, 244
cuMemcpyDtoA
 CUDA_MEM, 244
cuMemcpyDtoD
 CUDA_MEM, 245
cuMemcpyDtoDAsync
 CUDA_MEM, 246
cuMemcpyDtoH
 CUDA_MEM, 246
cuMemcpyDtoHAsync
 CUDA_MEM, 247
cuMemcpyHtoA
 CUDA_MEM, 247
cuMemcpyHtoAAsync

CUDA_MEMORY, 248
 cuMemcpyHtoD
 CUDA_MEMORY, 249
 cuMemcpyHtoDAsync
 CUDA_MEMORY, 249
 cuMemcpyPeer
 CUDA_MEMORY, 250
 cuMemcpyPeerAsync
 CUDA_MEMORY, 251
 cuMemFree
 CUDA_MEMORY, 251
 cuMemFreeHost
 CUDA_MEMORY, 252
 cuMemGetAddressRange
 CUDA_MEMORY, 252
 cuMemGetInfo
 CUDA_MEMORY, 253
 cuMemHostAlloc
 CUDA_MEMORY, 253
 cuMemHostGetDevicePointer
 CUDA_MEMORY, 254
 cuMemHostGetFlags
 CUDA_MEMORY, 255
 cuMemHostRegister
 CUDA_MEMORY, 255
 cuMemHostUnregister
 CUDA_MEMORY, 256
 CUmemorytype
 CUDA_TYPES, 181
 CUmemorytype_enum
 CUDA_TYPES, 190
 cuMemPeerGetDevicePointer
 CUDA_PEER_ACCESS, 303
 cuMemPeerRegister
 CUDA_PEER_ACCESS, 304
 cuMemPeerUnregister
 CUDA_PEER_ACCESS, 305
 cuMemsetD16
 CUDA_MEMORY, 257
 cuMemsetD16Async
 CUDA_MEMORY, 257
 cuMemsetD2D16
 CUDA_MEMORY, 258
 cuMemsetD2D16Async
 CUDA_MEMORY, 259
 cuMemsetD2D32
 CUDA_MEMORY, 259
 cuMemsetD2D32Async
 CUDA_MEMORY, 260
 cuMemsetD2D8
 CUDA_MEMORY, 261
 cuMemsetD2D8Async
 CUDA_MEMORY, 261
 cuMemsetD32
 CUDA_MEMORY, 262
 cuMemsetD32Async
 CUDA_MEMORY, 262
 cuMemsetD8
 CUDA_MEMORY, 263
 cuMemsetD8Async
 CUDA_MEMORY, 264
 CUmodule
 CUDA_TYPES, 181
 cuModuleGetFunction
 CUDA_MODULE, 210
 cuModuleGetGlobal
 CUDA_MODULE, 211
 cuModuleGetSurfRef
 CUDA_MODULE, 211
 cuModuleGetTexRef
 CUDA_MODULE, 212
 cuModuleLoad
 CUDA_MODULE, 212
 cuModuleLoadData
 CUDA_MODULE, 213
 cuModuleLoadDataEx
 CUDA_MODULE, 213
 cuModuleLoadFatBinary
 CUDA_MODULE, 215
 cuModuleUnload
 CUDA_MODULE, 215
 CUOutputMode
 CUDA_TYPES, 181
 CUOutputMode_st
 CUDA_TYPES, 190
 cuParamSetf
 CUDA_EXEC_DEPRECATED, 285
 cuParamSeti
 CUDA_EXEC_DEPRECATED, 286
 cuParamSetSize
 CUDA_EXEC_DEPRECATED, 286
 cuParamSetTexRef
 CUDA_EXEC_DEPRECATED, 287
 cuParamSetv
 CUDA_EXEC_DEPRECATED, 287
 CUpointer_attribute
 CUDA_TYPES, 181
 CUpointer_attribute_enum
 CUDA_TYPES, 190
 cuPointerGetAttribute
 CUDA_UNIFIED, 268
 cuProfilerInitialize
 CUDA_MEMORY, 264
 cuProfilerStart
 CUDA_MEMORY, 265
 cuProfilerStop
 CUDA_MEMORY, 265
 CUresult

CUDA_TYPES, 181
CUstream
 CUDA_TYPES, 181
cuStreamCreate
 CUDA_STREAM, 271
cuStreamDestroy
 CUDA_STREAM, 271
cuStreamQuery
 CUDA_STREAM, 272
cuStreamSynchronize
 CUDA_STREAM, 272
cuStreamWaitEvent
 CUDA_STREAM, 273
CUSurfref
 CUDA_TYPES, 181
cuSurfRefGetArray
 CUDA_SURFREF, 299
cuSurfRefSetArray
 CUDA_SURFREF, 299
CUtexref
 CUDA_TYPES, 182
cuTexRefCreate
 CUDA_TEXREF_DEPRECATED, 297
cuTexRefDestroy
 CUDA_TEXREF_DEPRECATED, 297
cuTexRefGetAddress
 CUDA_TEXREF, 290
cuTexRefGetAddressMode
 CUDA_TEXREF, 290
cuTexRefGetArray
 CUDA_TEXREF, 290
cuTexRefGetFilterMode
 CUDA_TEXREF, 291
cuTexRefGetFlags
 CUDA_TEXREF, 291
cuTexRefGetFormat
 CUDA_TEXREF, 292
cuTexRefSetAddress
 CUDA_TEXREF, 292
cuTexRefSetAddress2D
 CUDA_TEXREF, 293
cuTexRefSetAddressMode
 CUDA_TEXREF, 293
cuTexRefSetArray
 CUDA_TEXREF, 294
cuTexRefSetFilterMode
 CUDA_TEXREF, 294
cuTexRefSetFlags
 CUDA_TEXREF, 295
cuTexRefSetFormat
 CUDA_TEXREF, 295
cuVDPAUCtxCreate
 CUDA_VDPAU, 358
cuVDPAUGetDevice
 CUDA_VDPAU, 358
cuWGLGetDevice
 CUDA_GL, 313

Data types used by CUDA driver, 171
Data types used by CUDA Runtime, 156
Depth
 CUDA_ARRAY3D_DESCRIPTOR_st, 361
 CUDA_MEMCPY3D_PEER_st, 366
 CUDA_MEMCPY3D_st, 369
depth
 cudaExtent, 377
device
 cudaPointerAttributes, 385
Device Management, 13, 193
deviceOverlap
 cudaDeviceProp, 374
devicePointer
 cudaPointerAttributes, 385
Direct3D 10 Interoperability, 93, 335
Direct3D 11 Interoperability, 98, 350
Direct3D 9 Interoperability, 88, 320
dstArray
 CUDA_MEMCPY2D_st, 364
 CUDA_MEMCPY3D_PEER_st, 366
 CUDA_MEMCPY3D_st, 369
 cudaMemcpy3DParms, 380
 cudaMemcpy3DPeerParms, 382
dstContext
 CUDA_MEMCPY3D_PEER_st, 366
dstDevice
 CUDA_MEMCPY2D_st, 364
 CUDA_MEMCPY3D_PEER_st, 366
 CUDA_MEMCPY3D_st, 369
 cudaMemcpy3DPeerParms, 382
dstHeight
 CUDA_MEMCPY3D_PEER_st, 367
 CUDA_MEMCPY3D_st, 369
dstHost
 CUDA_MEMCPY2D_st, 364
 CUDA_MEMCPY3D_PEER_st, 367
 CUDA_MEMCPY3D_st, 370
dstLOD
 CUDA_MEMCPY3D_PEER_st, 367
 CUDA_MEMCPY3D_st, 370
dstMemoryType
 CUDA_MEMCPY2D_st, 364
 CUDA_MEMCPY3D_PEER_st, 367
 CUDA_MEMCPY3D_st, 370
dstPitch
 CUDA_MEMCPY2D_st, 364
 CUDA_MEMCPY3D_PEER_st, 367
 CUDA_MEMCPY3D_st, 370
dstPos

cudaMemcpy3DParms, 380
 cudaMemcpy3DPeerParms, 382
 dstPtr
 cudaMemcpy3DParms, 380
 cudaMemcpy3DPeerParms, 382
 dstXInBytes
 CUDA_MEMCPY2D_st, 364
 CUDA_MEMCPY3D_PEER_st, 367
 CUDA_MEMCPY3D_st, 370
 dstY
 CUDA_MEMCPY2D_st, 365
 CUDA_MEMCPY3D_PEER_st, 367
 CUDA_MEMCPY3D_st, 370
 dstZ
 CUDA_MEMCPY3D_PEER_st, 367
 CUDA_MEMCPY3D_st, 370

 ECCEnabled
 cudaDeviceProp, 374

 Error Handling, 28

 Event Management, 33, 274

 Execution Control, 37, 278

 extent
 cudaMemcpy3DParms, 380
 cudaMemcpy3DPeerParms, 382

 f
 cudaChannelFormatDesc, 372

 filterMode
 textureReference, 390

 Flags
 CUDA_ARRAY3D_DESCRIPTOR_st, 361

 Format
 CUDA_ARRAY3D_DESCRIPTOR_st, 361
 CUDA_ARRAY_DESCRIPTOR_st, 363

 Graphics Interoperability, 106, 306

 Height
 CUDA_ARRAY3D_DESCRIPTOR_st, 361
 CUDA_ARRAY_DESCRIPTOR_st, 363
 CUDA_MEMCPY2D_st, 365
 CUDA_MEMCPY3D_PEER_st, 367
 CUDA_MEMCPY3D_st, 370

 height
 cudaExtent, 377

 hostPointer
 cudaPointerAttributes, 385

 Initialization, 191

 integrated
 cudaDeviceProp, 374

 Interactions with the CUDA Driver API, 131

 kernelExecTimeoutEnabled

cudaDeviceProp, 374
 kind
 cudaMemcpy3DParms, 380

 localSizeBytes
 cudaFuncAttributes, 378

 major
 cudaDeviceProp, 374

 make_cudaExtent
 CUDART_MEMORY, 75

 make_cudaPitchedPtr
 CUDART_MEMORY, 75

 make_cudaPos
 CUDART_MEMORY, 76

 maxGridSize
 cudaDeviceProp, 374
 CUdevprop_st, 387

 maxTexture1D
 cudaDeviceProp, 374

 maxTexture1DLayered
 cudaDeviceProp, 374

 maxTexture2D
 cudaDeviceProp, 374

 maxTexture2DLayered
 cudaDeviceProp, 375

 maxTexture3D
 cudaDeviceProp, 375

 maxThreadsDim
 cudaDeviceProp, 375
 CUdevprop_st, 387

 maxThreadsPerBlock
 cudaDeviceProp, 375
 cudaFuncAttributes, 378
 CUdevprop_st, 387

 Memory Management, 41, 217

 memoryType
 cudaPointerAttributes, 385

 memPitch
 cudaDeviceProp, 375
 CUdevprop_st, 387

 minor
 cudaDeviceProp, 375

 Module Management, 210

 multiProcessorCount
 cudaDeviceProp, 375

 name
 cudaDeviceProp, 375

 normalized
 textureReference, 390

 NumChannels
 CUDA_ARRAY3D_DESCRIPTOR_st, 361
 CUDA_ARRAY_DESCRIPTOR_st, 363

 numRegs

cudaFuncAttributes, 378
OpenGL Interoperability, 85, 311
pciBusID
 cudaDeviceProp, 375
pciDeviceID
 cudaDeviceProp, 375
Peer Context Memory Access, 301
Peer Device Memory Access, 80
pitch
 cudaPitchedPtr, 384
ptr
 cudaPitchedPtr, 384
ptxVersion
 cudaFuncAttributes, 378
regsPerBlock
 cudaDeviceProp, 375
 CUdevprop_st, 387
reserved0
 CUDA_MEMCPY3D_st, 370
reserved1
 CUDA_MEMCPY3D_st, 370
sharedMemPerBlock
 cudaDeviceProp, 375
 CUdevprop_st, 387
sharedSizeBytes
 cudaFuncAttributes, 378
SIMDWidth
 CUdevprop_st, 387
srcArray
 CUDA_MEMCPY2D_st, 365
 CUDA_MEMCPY3D_PEER_st, 367
 CUDA_MEMCPY3D_st, 370
 cudaMemcpy3DParms, 380
 cudaMemcpy3DPeerParms, 382
srcContext
 CUDA_MEMCPY3D_PEER_st, 367
srcDevice
 CUDA_MEMCPY2D_st, 365
 CUDA_MEMCPY3D_PEER_st, 367
 CUDA_MEMCPY3D_st, 370
 cudaMemcpy3DPeerParms, 382
srcHeight
 CUDA_MEMCPY3D_PEER_st, 368
 CUDA_MEMCPY3D_st, 371
srcHost
 CUDA_MEMCPY2D_st, 365
 CUDA_MEMCPY3D_PEER_st, 368
 CUDA_MEMCPY3D_st, 371
srcLOD
 CUDA_MEMCPY3D_PEER_st, 368
 CUDA_MEMCPY3D_st, 371
srcMemoryType
 CUDA_MEMCPY2D_st, 365
 CUDA_MEMCPY3D_PEER_st, 368
 CUDA_MEMCPY3D_st, 371
srcPitch
 CUDA_MEMCPY2D_st, 365
 CUDA_MEMCPY3D_PEER_st, 368
 CUDA_MEMCPY3D_st, 371
srcPos
 cudaMemcpy3DParms, 380
 cudaMemcpy3DPeerParms, 382
srcPtr
 cudaMemcpy3DParms, 380
 cudaMemcpy3DPeerParms, 383
srcXInBytes
 CUDA_MEMCPY2D_st, 365
 CUDA_MEMCPY3D_PEER_st, 368
 CUDA_MEMCPY3D_st, 371
srcY
 CUDA_MEMCPY2D_st, 365
 CUDA_MEMCPY3D_PEER_st, 368
 CUDA_MEMCPY3D_st, 371
srcZ
 CUDA_MEMCPY3D_PEER_st, 368
 CUDA_MEMCPY3D_st, 371
sRGB
 textureReference, 390
Stream Management, 30, 271
Surface Reference Management, 115, 299
surfaceAlignment
 cudaDeviceProp, 376
surfaceReference, 389
 channelDesc, 389
tccDriver
 cudaDeviceProp, 376
Texture Reference Management, 110, 289
textureAlign
 CUdevprop_st, 388
textureAlignment
 cudaDeviceProp, 376
textureReference, 390
 addressMode, 390
 channelDesc, 390
 filterMode, 390
 normalized, 390
 sRGB, 390
totalConstantMemory
 CUdevprop_st, 388
totalConstMem
 cudaDeviceProp, 376
totalGlobalMem
 cudaDeviceProp, 376
Unified Addressing, 77, 267

unifiedAddressing

 cudaDeviceProp, [376](#)

VDPAU Interoperability, [103](#), [356](#)

Version Management, [117](#), [192](#)

w

 cudaChannelFormatDesc, [372](#)

warpSize

 cudaDeviceProp, [376](#)

Width

 CUDA_ARRAY3D_DESCRIPTOR_st, [362](#)

 CUDA_ARRAY_DESCRIPTOR_st, [363](#)

width

 cudaExtent, [377](#)

WidthInBytes

 CUDA_MEMCPY2D_st, [365](#)

 CUDA_MEMCPY3D_PEER_st, [368](#)

 CUDA_MEMCPY3D_st, [371](#)

x

 cudaChannelFormatDesc, [372](#)

 cudaPos, [386](#)

xsize

 cudaPitchedPtr, [384](#)

y

 cudaChannelFormatDesc, [372](#)

 cudaPos, [386](#)

ysize

 cudaPitchedPtr, [384](#)

z

 cudaChannelFormatDesc, [372](#)

 cudaPos, [386](#)

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 in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2007-2011 NVIDIA Corporation. All rights reserved.



NVIDIA.

NVIDIA Corporation

2701 San Tomas Expressway

Santa Clara, CA 95050

www.nvidia.com