Table of Contents

Chapter 1. Difference between the driver and runtime APIs .............................................. 1
Chapter 2. API synchronization behavior ............................................................................. 3
Chapter 3. Stream synchronization behavior ....................................................................... 5
Chapter 4. Graph object thread safety ............................................................................... 7
Chapter 5. Modules ............................................................................................................... 8

5.1. Data types used by CUDA driver .................................................................................. 9
   CUaccessPolicyWindow ........................................................................................................ 10
   CUarrayMapInfo ............................................................................................................... 10
   CUDA_ARRAY3D_DESCRIPTOR .......................................................................................... 10
   CUDA_ARRAY_DESCRIPTOR ............................................................................................. 10
   CUDA_ARRAY_SPARSE_PROPERTIES .............................................................................. 10
   CUDA_EXTERNAL_MEMORY_BUFFER_DESC .................................................................. 10
   CUDA_EXTERNAL_MEMORY_HANDLE_DESC ............................................................... 10
   CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC ............................................. 10
   CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC ......................................................... 10
   CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS ...................................................... 10
   CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS ......................................................... 10
   CUDA_HOST_NODE_PARAMS ......................................................................................... 10
   CUDA_KERNEL_NODE_PARAMS ..................................................................................... 10
   CUDA_LAUNCH_PARAMS ............................................................................................... 10
   CUDA_MEMCPY2D .......................................................................................................... 10
   CUDA_MEMCPY3D .......................................................................................................... 11
   CUDA_MEMCPY3D_PEER ............................................................................................... 11
   CUDA_MEMSET_NODE_PARAMS .................................................................................... 11
   CUDA_POINTER_ATTRIBUTE_P2P_TOKENS .................................................................. 11
   CUDARESOURCE_DESC ................................................................................................. 11
   CUDA_RESOURCE_VIEW_DESC ....................................................................................... 11
   CUDA_TEXTURE_DESC .................................................................................................. 11
   CUdevprop .................................................................................................................... 11
   CUeglFrame ................................................................................................................... 11
   CUIpcEventHandle ......................................................................................................... 11
   CUIpcMemHandle ............................................................................................................ 11
   CUKernelNodeAttrValue ............................................................................................... 11
   CUMemAccessDesc ....................................................................................................... 11
   CUMemAllocationProp ................................................................................................. 11
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUmemLocation</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>CUstreamAttrValue</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>CUstreamBatchMemOpParams</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>CUaccessProperty</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>CUaddress_mode</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>CUarray_cubemap_face</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>CUarray_format</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>CUarraySparseSubresourceType</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>CUncomputeMode</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>CUCtxx_flags</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>CUDA_POINTER_ATTRIBUTE_ACCESS_FLAGS</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>CUdevice_attribute</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>CUdevice_P2PAttribute</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>CUeglColorFormat</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>CUeglFrameType</td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>CUeglResourceLocationFlags</td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>CUevent_flags</td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>CUevent_record_flags</td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>CUevent_wait_flags</td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>CUexternalMemoryHandleType</td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>CUexternalSemaphoreHandleType</td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>CUfilter_mode</td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>CUfunc_cache</td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>CUfunction_attribute</td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>CUgraphicsMapResourceFlags</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>CUgraphicsRegisterFlags</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>CUgraphNodeType</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>CUipcMem_flags</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>CUjit_cacheMode</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>CUjit_fallback</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>CUjit_option</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>CUjit_target</td>
<td></td>
<td>34</td>
</tr>
<tr>
<td>CUjitInputType</td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>CKernelNodeAttrID</td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>CULimit</td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>CUMem_advise</td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>CUMemAccess_flags</td>
<td></td>
<td>37</td>
</tr>
<tr>
<td>CUMemAllocationCompType</td>
<td></td>
<td>37</td>
</tr>
<tr>
<td>CUDA Driver API</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>CUmemAllocationGranularity_flags</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>CUmemAllocationHandleType</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>CUmemAllocationType</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>CUmemAttach_flags</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>CUmemHandleType</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>CUmemLocationType</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>CUmemOperationType</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>CUmemorytype</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>CUoccupancy_flags</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>CUpointer_attribute</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>CUresourceType</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>CUresourceViewFormat</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>CUresult</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>CUshared_carveout</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>CUsharedconfig</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>CUstream_flags</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>CUstreamAttrID</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>CUstreamBatchMemOpType</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>CUstreamCaptureMode</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>CUstreamCaptureStatus</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>CUstreamWaitValue_flags</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>CUstreamWriteValue_flags</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>CUarray</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>CUcontext</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>CUdevice</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>CUdeviceptr</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>CUeglStreamConnection</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>CUevent</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>CUexternalMemory</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>CUexternalSemaphore</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>CUfunction</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>CUgraph</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>CUgraphExec</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>CUgraphicsResource</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>CUgraphNode</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>CUhostFn</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>CUmipmappedArray</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>CUModule</td>
<td>53</td>
<td></td>
</tr>
</tbody>
</table>
CUoccupancyB2DSize.................................................................53
CUstream..........................................................53
CUstreamCallback..................................................53
CUsurfObject..........................................................53
CUsurfref...............................................................54
CUtexObject.............................................................54
CUtexref...............................................................54
CU_ARRAY_SPARSE_PROPERTIES_SINGLE_MIPTAIL........................54
CU_DEVICE_CPU..........................................................54
CU_DEVICE_INVALID.....................................................54
CU_IPC_HANDLE_SIZE..................................................54
CU_LAUNCH_PARAM_BUFFER_POINTER..................................54
CU_LAUNCH_PARAM_BUFFER_SIZE.......................................55
CU_LAUNCH_PARAM_END................................................55
CU_MEM_CREATE_USAGE_TILE_POOL....................................55
CU_MEMHOSTALLOC_DEVICEMAP.........................................55
CU_MEMHOSTALLOC_PORTABLE.........................................55
CU_MEMHOSTALLOC_WRITECOMBINED..................................55
CU_MEMHOSTREGISTER_DEVICEMAP....................................55
CU_MEMHOSTREGISTER_IOMEMORY......................................55
CU_MEMHOSTREGISTER_PORTABLE.......................................55
CU_MEMHOSTREGISTER_READ_ONLY.....................................56
CU_PARAM_TR_DEFAULT..................................................56
CU_STREAM_LEGACY.......................................................56
CU_STREAM_PER_THREAD................................................56
CU_TRSA_OVERRIDE_FORMAT..............................................56
CU_TRSF_DISABLE_TRILINEAR_OPTIMIZATION..............................57
CU_TRSF_NORMALIZED_COORDINATES..................................57
CU_TRSF_READ_AS_INTEGER..............................................57
CU_TRSF_SRGB..............................................................57
CUDA_ARRAY3D_2DARRAY....................................................57
CUDA_ARRAY3D_COLOR_ATTACHMENT.....................................57
CUDA_ARRAY3D_CUBEMAP..................................................57
CUDA_ARRAY3D_DEPTH_TEXTURE.......................................57
CUDA_ARRAY3D_LAYERED................................................58
CUDA_ARRAY3D_SPARSE...................................................58
CUDA_ARRAY3D_SURFACE_LDST...........................................58
CUDA_ARRAY3D_TEXTURE_GATHER.......................................58
CUDA COOPERATIVE_LAUNCH_MULTI_DEVICE_NO_POST_LAUNCH_SYNC..............58
CUDA COOPERATIVE_LAUNCH_MULTI DEVICE_NO_PRE_LAUNCH_SYNC..............58
CUDA_EGL_INFINITE_TIMEOUT..............................................................58
CUDA_EXTERNAL_MEMORY_Dedicated..........................................................59
CUDA_EXTERNAL_SEMAPHORE_SIGNAL_SKIP_NVSCIBUF_MEMSYNC..................59
CUDA_EXTERNAL_SEMAPHORE_WAIT_SKIP_NVSCIBUF_MEMSYNC........................59
CUDA_NVSCISYNC_ATTR_SIGNAL..................................................................59
CUDA_NVSCISYNC_ATTR_WAIT......................................................................59
CUDA_VERSION.......................................................................................59
MAX_PLANES............................................................................................60

5.2. Error Handling..................................................................................60
cuGetErrorName.......................................................................................60
cuGetErrorString.....................................................................................60

5.3. Initialization......................................................................................61
cuInit.........................................................................................................61

5.4. Version Management........................................................................62
cuDriverGetVersion..................................................................................62

5.5. Device Management..........................................................................62
cuDeviceGet.............................................................................................63
cuDeviceGetAttribute..............................................................................63
cuDeviceGetCount...................................................................................69
cuDeviceGetLuid......................................................................................70
cuDeviceGetName....................................................................................70
cuDeviceGetNvSciSyncAttributes...............................................................71
cuDeviceGetTexture1DLinearMaxWidth....................................................72
cuDeviceGetUuid......................................................................................73
cuDeviceTotalMem...................................................................................74

5.6. Device Management [DEPRECATED]..................................................74
cuDeviceComputeCapability.....................................................................75
cuDeviceGetProperties............................................................................76

5.7. Primary Context Management............................................................77
cuDevicePrimaryCtxGetState...................................................................77
cuDevicePrimaryCtxRelease....................................................................78
cuDevicePrimaryCtxReset........................................................................79
cuDevicePrimaryCtxRetain.......................................................................79
cuDevicePrimaryCtxSetFlags...................................................................80

5.8. Context Management.........................................................................82
cuCtxCreate............................................................................................82
5.9. Context Management [DEPRECATED] ................................................................. 98
  cuCtxAttach .................................................. 98
  cuCtxDetach .................................................. 99
5.10. Module Management ...................................................................................... 100
  cuLinkAddData .............................................. 100
  cuLinkAddFile ................................................ 101
  cuLinkComplete ............................................ 102
  cuLinkCreate ............................................... 103
  cuLinkDestroy ............................................... 104
  cuModuleGetFunction ..................................... 104
  cuModuleGetGlobal ....................................... 105
  cuModuleGetSurfRef ....................................... 106
  cuModuleGetTexRef ....................................... 106
  cuModuleLoad ............................................... 107
  cuModuleLoadData ......................................... 108
  cuModuleLoadDataEx ...................................... 109
  cuModuleLoadFatBinary ............................... 110
  cuModuleUnload ............................................ 111
5.11. Memory Management .................................................................................. 112
  cuArray3DCreate ......................................... 112
  cuArray3DGetDescriptor ................................ 115
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuArrayCreate</td>
<td>116</td>
</tr>
<tr>
<td>cuArrayDestroy</td>
<td>118</td>
</tr>
<tr>
<td>cuArrayGetDescriptor</td>
<td>119</td>
</tr>
<tr>
<td>cuArrayGetSparseProperties</td>
<td>120</td>
</tr>
<tr>
<td>cuDeviceGetByPCIBusId</td>
<td>121</td>
</tr>
<tr>
<td>cuDeviceGetPCIBusId</td>
<td>121</td>
</tr>
<tr>
<td>cuIpcCloseMemHandle</td>
<td>122</td>
</tr>
<tr>
<td>cuIpcGetEventHandle</td>
<td>123</td>
</tr>
<tr>
<td>cuIpcGetMemHandle</td>
<td>124</td>
</tr>
<tr>
<td>cuIpcOpenEventHandle</td>
<td>124</td>
</tr>
<tr>
<td>cuIpcOpenMemHandle</td>
<td>125</td>
</tr>
<tr>
<td>cuMemAlloc</td>
<td>126</td>
</tr>
<tr>
<td>cuMemAllocHost</td>
<td>127</td>
</tr>
<tr>
<td>cuMemAllocManaged</td>
<td>128</td>
</tr>
<tr>
<td>cuMemAllocPitch</td>
<td>129</td>
</tr>
<tr>
<td>cuMemcpy</td>
<td>130</td>
</tr>
<tr>
<td>cuMemcpy2D</td>
<td>132</td>
</tr>
<tr>
<td>cuMemcpy2DAsync</td>
<td>134</td>
</tr>
<tr>
<td>cuMemcpy2DUnaligned</td>
<td>136</td>
</tr>
<tr>
<td>cuMemcpy3D</td>
<td>137</td>
</tr>
<tr>
<td>cuMemcpy3DAsync</td>
<td>139</td>
</tr>
<tr>
<td>cuMemcpy3DPeer</td>
<td>141</td>
</tr>
<tr>
<td>cuMemcpy3DPeerAsync</td>
<td>143</td>
</tr>
<tr>
<td>cuMemcpyAsync</td>
<td>145</td>
</tr>
<tr>
<td>cuMemcpyAtoA</td>
<td>146</td>
</tr>
<tr>
<td>cuMemcpyAtoD</td>
<td>148</td>
</tr>
<tr>
<td>cuMemcpyAtoH</td>
<td>150</td>
</tr>
<tr>
<td>cuMemcpyAtoHAsync</td>
<td>152</td>
</tr>
<tr>
<td>cuMemcpyDtoA</td>
<td>153</td>
</tr>
<tr>
<td>cuMemcpyDtoD</td>
<td>155</td>
</tr>
<tr>
<td>cuMemcpyDtoDAsync</td>
<td>157</td>
</tr>
<tr>
<td>cuMemcpyDtoH</td>
<td>159</td>
</tr>
<tr>
<td>cuMemcpyDtoHAsync</td>
<td>161</td>
</tr>
<tr>
<td>cuMemcpyHtoA</td>
<td>162</td>
</tr>
<tr>
<td>cuMemcpyHtoAAsync</td>
<td>164</td>
</tr>
<tr>
<td>cuMemcpyHtoD</td>
<td>166</td>
</tr>
<tr>
<td>cuMemcpyHtoDAsync</td>
<td>168</td>
</tr>
<tr>
<td>cuMemcpyPeer</td>
<td>170</td>
</tr>
</tbody>
</table>
cuMemcpyPeerAsync................................................................................................................166
cuMemFree............................................................................................................................... 167
cuMemFreeHost........................................................................................................................167
cuMemGetAddressRange......................................................................................................... 168
cuMemGetInfo........................................................................................................................... 169
cuMemHostAlloc.......................................................................................................................170
cuMemHostGetDevicePointer.................................................................................................. 172
cuMemHostGetFlags................................................................................................................ 173
cuMemHostRegister................................................................................................................. 174
cuMemHostUnregister................................................................................................................ 176
cuMemsetD16........................................................................................................................... 176
cuMemsetD16Async..................................................................................................................177
cuMemsetD2D16.......................................................................................................................178
cuMemsetD2D16Async............................................................................................................. 180
cuMemsetD2D32.......................................................................................................................181
cuMemsetD2D32Async............................................................................................................. 182
cuMemsetD2D8.........................................................................................................................183
cuMemsetD2D8Async............................................................................................................... 184
cuMemsetD32........................................................................................................................... 184
cuMemsetD32Async..................................................................................................................185
cuMemsetD32Async.................................................................................................................. 186
cuMemsetD8............................................................................................................................. 187
cuMemsetD8Async....................................................................................................................188
cuMipmappedArrayCreate........................................................................................................189
cuMipmappedArrayDestroy..................................................................................................... 192
cuMipmappedArrayGetLevel.................................................................................................... 193
cuMipmappedArrayGetSparseProperties................................................................................ 194
5.12. Virtual Memory Management............................................................................................. 195
cuMemAddressFree..................................................................................................................195
cuMemAddressReserve............................................................................................................ 196
cuMemCreate............................................................................................................................ 197
cuMemExportToShareableHandle........................................................................................... 198
cuMemGetAccess......................................................................................................................199
cuMemGetAllocationGranularity.............................................................................................. 199
cuMemGetAllocationPropertiesFromHandle.......................................................................... 199
cuMemImportFromShareableHandle......................................................................................201
cuMemMap................................................................................................................................202
cuMemMapArrayAsync............................................................................................................. 203
cuMemRelease..........................................................................................................................206
cuMemRetainAllocationHandle.................................................................207
cuMemSetAccess......................................................................................207
cuMemUnmap.........................................................................................208

5.13. Unified Addressing........................................................................209
      cuMemAdvise.....................................................................................211
      cuMemPrefetchAsync..........................................................................214
      cuMemRangeGetAttribute..................................................................216
      cuMemRangeGetAttributes..............................................................218
      cuPointerGetAttribute.....................................................................219
      cuPointerGetAttributes..................................................................222
      cuPointerSetAttribute.....................................................................223

5.14. Stream Management.......................................................................224
      cuStreamAddCallback.......................................................................225
      cuStreamAttachMemAsync...............................................................226
      cuStreamBeginCapture.....................................................................228
      cuStreamCopyAttributes..................................................................229
      cuStreamCreate................................................................................230
      cuStreamCreateWithPriority............................................................231
      cuStreamDestroy.............................................................................232
      cuStreamEndCapture........................................................................233
      cuStreamGetAttribute.....................................................................233
      cuStreamGetCaptureInfo..................................................................234
      cuStreamGetCtx...............................................................................235
      cuStreamGetFlags...........................................................................236
      cuStreamGetPriority.......................................................................236
      cuStreamIsCapturing.......................................................................237
      cuStreamQuery................................................................................238
      cuStreamSetAttribute.......................................................................239
      cuStreamSynchronize.......................................................................239
      cuStreamWaitEvent.........................................................................240
      cuThreadExchangeStreamCaptureMode.............................................241

5.15. Event Management.......................................................................242
      cuEventCreate..................................................................................243
      cuEventDestroy...............................................................................244
      cuEventElapsedTime........................................................................244
      cuEventQuery.................................................................................245
      cuEventRecord...............................................................................246
      cuEventRecordWithFlags..................................................................247
5.16. External Resource Interoperability
- cuEventSynchronize.................................................................................................................. 248
- cuDestroyExternalMemory......................................................................................................... 249
- cuDestroyExternalSemaphore.................................................................................................... 250
- cuExternalMemoryGetMappedBuffer........................................................................................ 251
- cuExternalMemoryGetMappedMipmappedArray......................................................................... 252
- cuImportExternalMemory.......................................................................................................... 253
- cuImportExternalSemaphore...................................................................................................... 256
- cuSignalExternalSemaphoresAsync.......................................................................................... 259
- cuWaitExternalSemaphoresAsync............................................................................................. 261

5.17. Stream memory operations
- cuStreamBatchMemOp............................................................................................................. 263
- cuStreamWaitValue32............................................................................................................... 264
- cuStreamWaitValue64............................................................................................................... 265
- cuStreamWriteValue32............................................................................................................. 266
- cuStreamWriteValue64............................................................................................................. 267

5.18. Execution Control
- cuFuncGetAttribute.................................................................................................................. 268
- cuFuncSetAttribute................................................................................................................... 270
- cuFuncSetCacheConfig............................................................................................................. 271
- cuFuncSetSharedMemConfig................................................................................................... 272
- cuLaunchCooperativeKernel.................................................................................................... 273
- cuLaunchCooperativeKernelMultiDevice............................................................................... 275
- cuLaunchHostFunc................................................................................................................... 278
- cuLaunchKernel....................................................................................................................... 280

5.19. Execution Control [DEPRECATED]
- cuFuncSetBlockShape.............................................................................................................. 282
- cuFuncSetSharedSize................................................................................................................ 283
- cuLaunch................................................................................................................................. 284
- cuLaunchGrid........................................................................................................................... 285
- cuLaunchGridAsync.................................................................................................................. 286
- cuParamSetf.............................................................................................................................. 287
- cuParamSeti.............................................................................................................................. 288
- cuParamSetSize........................................................................................................................ 288
- cuParamSetTexRef..................................................................................................................... 289
- cuParamSetv............................................................................................................................. 290

5.20. Graph Management
- cuGraphAddChildGraphNode.................................................................................................. 291
cuGraphAddDependencies....................................................................................................... 292
cuGraphAddEmptyNode............................................................................................................293
cuGraphAddEventRecordNode.................................................................................................294
cuGraphAddEventWaitNode..................................................................................................... 295
cuGraphAddHostNode..............................................................................................................296
cuGraphAddKernelNode...........................................................................................................297
cuGraphAddMemcpyNode........................................................................................................ 299
cuGraphAddMemsetNode.........................................................................................................301
cuGraphChildGraphNodeGetGraph..........................................................................................302
cuGraphClone............................................................................................................................303
cuGraphCreate.......................................................................................................................... 303
cuGraphDestroy........................................................................................................................ 304
cuGraphDestroyNode................................................................................................................305
cuGraphEventRecordNodeGetEvent........................................................................................ 305
cuGraphEventRecordNodeSetEvent..........................................................................................306
cuGraphEventWaitNodeGetEvent.............................................................................................307
cuGraphEventWaitNodeSetEvent............................................................................................. 307
cuGraphExecChildGraphNodeSetParams............................................................................... 308
cuGraphExecDestroy.................................................................................................................309
cuGraphExecEventRecordNodeSetEvent.................................................................................310
cuGraphExecEventWaitNodeSetEvent......................................................................................311
cuGraphExecHostNodeSetParams............................................................................................312
cuGraphExecKernelNodeSetParams........................................................................................313
cuGraphExecMemcpyNodeSetParams......................................................................................314
cuGraphExecMemsetNodeSetParams........................................................................................315
cuGraphExecUpdate................................................................................................................. 316
cuGraphGetEdges.....................................................................................................................318
cuGraphGetNodes.....................................................................................................................319
cuGraphGetRootNodes.............................................................................................................320
cuGraphHostNodeGetParams..................................................................................................321
cuGraphHostNodeSetParams.................................................................................................. 321
cuGraphInstantiate....................................................................................................................322
cuGraphKernelNodeCopyAttributes.........................................................................................323
cuGraphKernelNodeGetAttribute.............................................................................................324
cuGraphKernelNodeGetParams...............................................................................................324
cuGraphKernelNodeSetAttribute.............................................................................................325
cuGraphKernelNodeSetParams...............................................................................................326
cuGraphLaunch.........................................................................................................................327
cuGraphMemcpyNodeGetParams.................................................................327
cuGraphMemcpyNodeSetParams.................................................................328
cuGraphMemsetNodeGetParams.................................................................329
cuGraphMemsetNodeSetParams.................................................................330
cuGraphNodeFindInClone...........................................................................330
cuGraphNodeGetDependencies....................................................................331
cuGraphNodeGetDependentNodes..............................................................332
cuGraphNodeGetType..................................................................................333
cuGraphRemoveDependencies....................................................................334
cuGraphUpload..........................................................................................335

5.21. Occupancy.........................................................................................335

cuOccupancyAvailableDynamicSMemPerBlock..........................................336

cuOccupancyMaxActiveBlocksPerMultiprocessor........................................337

cuOccupancyMaxActiveBlocksPerMultiprocessorWithFlags........................338

cuOccupancyMaxPotentialBlockSize..........................................................339

cuOccupancyMaxPotentialBlockSizeWithFlags..........................................340

5.22. Texture Reference Management [DEPRECATED].................................342

cuTexRefCreate.........................................................................................342

cuTexRefDestroy.......................................................................................342

cuTexRefGetAddress..................................................................................343

cuTexRefGetAddressMode..........................................................................344

cuTexRefGetArray......................................................................................344

cuTexRefGetBorderColor............................................................................345

cuTexRefGetFilterMode..............................................................................346

cuTexRefGetFlags......................................................................................346

cuTexRefGetFormat....................................................................................347

cuTexRefGetMaxAnisotropy.......................................................................348

cuTexRefGetMipmapFilterMode...................................................................348

cuTexRefGetMipmapLevelBias...................................................................349

cuTexRefGetMipmapLevelClamp..................................................................350

cuTexRefGetMipmappedArray.....................................................................350

cuTexRefSetAddress...................................................................................351

cuTexRefSetAddress2D..............................................................................352

cuTexRefSetAddressMode...........................................................................353

cuTexRefSetArray......................................................................................354

cuTexRefSetBorderColor............................................................................355

cuTexRefSetFilterMode..............................................................................356

cuTexRefSetFlags......................................................................................357
cuTexRefSetFormat.................................................................................................................. 358

cuTexRefSetMaxAnisotropy........................................................................................................... 358

cuTexRefSetMipmapFilterMode.................................................................................................. 359

cuTexRefSetMipmapLevelBias.................................................................................................. 360

cuTexRefSetMipmapLevelClamp............................................................................................... 361

cuTexRefSetMipmappedArray.................................................................................................. 362

5.23. Surface Reference Management [DEPRECATED]............................................................. 362

cuSurfRefGetArray.................................................................................................................... 363

cuSurfRefSetArray.................................................................................................................... 363

5.24. Texture Object Management............................................................................................. 364

cuTexObjectCreate.................................................................................................................... 364

cuTexObjectDestroy.................................................................................................................. 369

cuTexObjectGetResourceDesc................................................................................................. 369

cuTexObjectGetResourceViewDesc.......................................................................................... 370

cuTexObjectGetTextureDesc.................................................................................................... 370

5.25. Surface Object Management............................................................................................. 371

cuSurfObjectCreate................................................................................................................... 371

cuSurfObjectDestroy................................................................................................................. 372

cuSurfObjectGetResourceDesc................................................................................................ 372

5.26. Peer Context Memory Access............................................................................................ 373

cuCtxDisablePeerAccess........................................................................................................... 373

cuCtxEnablePeerAccess............................................................................................................ 374

cuDeviceCanAccessPeer.......................................................................................................... 375

cuDeviceGetP2PAttribute......................................................................................................... 376

5.27. Graphics Interoperability................................................................................................. 377

cuGraphicsMapResources......................................................................................................... 377

cuGraphicsResourceGetMappedMipmappedArray.................................................................... 378

cuGraphicsResourceGetMappedPointer.................................................................................. 379

cuGraphicsResourceSetMapFlags........................................................................................... 380

cuGraphicsSubResourceGetMappedArray.............................................................................. 381

cuGraphicsUnmapResources................................................................................................... 382

cuGraphicsUnregisterResource............................................................................................... 383

5.28. Profiler Control [DEPRECATED]....................................................................................... 383

cuProfilerInitialize.................................................................................................................. 384

5.29. Profiler Control.................................................................................................................... 385

cuProfilerStart........................................................................................................................ 385

cuProfilerStop........................................................................................................................ 385

5.30. OpenGL Interoperability................................................................................................... 386
OpenGL Interoperability [DEPRECATED] ................................................................. 386
CUGLDeviceList ................................................................................................. 386
cuGLGetDevices ............................................................................................... 386
cuGraphicsGLRegisterBuffer ........................................................................ 388
cuGraphicsGLRegisterImage ........................................................................... 389
cuWGLGetDevice ............................................................................................... 390
5.30.1. OpenGL Interoperability [DEPRECATED] .................................................. 391
CUGLmap_flags ............................................................................................... 391
cuGLCtxCreate ............................................................................................... 391
cuGLInit ......................................................................................................... 392
cuGLMapBufferObject ..................................................................................... 393
cuGLMapBufferObjectAsync ........................................................................... 394
cuGLRegisterBufferObject ............................................................................ 395
cuGLSetBufferObjectMapFlags ..................................................................... 395
cuGLUnmapBufferObject ................................................................................ 396
cuGLUnmapBufferObjectAsync ..................................................................... 397
cuGLUnregisterBufferObject .......................................................................... 398
5.31. VDPAU Interoperability .............................................................................. 398
cuGraphicsVDPAURegisterOutputSurface ....................................................... 399
cuGraphicsVDPAURegisterVideoSurface ....................................................... 400
cuVDPAUCtxCreate ......................................................................................... 401
cuVDPAUGetDevice ......................................................................................... 402
5.32. EGL Interoperability ................................................................................. 403
cuEGLStreamConsumerAcquireFrame .............................................................. 403
cuEGLStreamConsumerConnect ....................................................................... 404
cuEGLStreamConsumerConnectWithFlags .................................................. 404
cuEGLStreamConsumerDisconnect ................................................................. 405
cuEGLStreamConsumerReleaseFrame ............................................................ 406
cuEGLStreamProducerConnect ....................................................................... 406
cuEGLStreamProducerDisconnect ................................................................. 407
cuEGLStreamProducerPresentFrame ............................................................... 408
cuEGLStreamProducerReturnFrame ............................................................... 409
cuEventCreateFromEGLSync ......................................................................... 409
cuGraphicsEGLRegisterImage ....................................................................... 410
cuGraphicsResourceGetMappedEglFrame ....................................................... 412
Chapter 6. Data Structures .............................................................................. 414
CUaccessPolicyWindow .................................................................................... 415
base_ptr ........................................................................................................... 415
hitProp....................................................................................................................................... 415
hitRatio...................................................................................................................................... 415
missProp................................................................................................................................... 415
num_bytes................................................................................................................................ 415
CUarrayMapInfo............................................................................................................................415
deviceBitMask........................................................................................................................... 415
extentDepth............................................................................................................................... 416
extentHeight.............................................................................................................................. 416
extentWidth................................................................................................................................416
flags........................................................................................................................................... 416
layer........................................................................................................................................... 416
level............................................................................................................................................416
memHandleType....................................................................................................................... 416
memOperationType...................................................................................................................416
offset.......................................................................................................................................... 416
offsetX........................................................................................................................................ 416
offsetY........................................................................................................................................ 417
offsetZ........................................................................................................................................417
reserved.....................................................................................................................................417
resourceType.............................................................................................................................417
size.............................................................................................................................................417
subresourceType.......................................................................................................................417
CUDA_ARRAY3D_DESCRIPTOR...................................................................................................417
Depth......................................................................................................................................... 417
Flags.......................................................................................................................................... 417
Format....................................................................................................................................... 418
Height........................................................................................................................................ 418
NumChannels........................................................................................................................... 418
Width..........................................................................................................................................418
CUDA_ARRAY_DESCRIPTOR.......................................................................................................418
Format....................................................................................................................................... 418
Height........................................................................................................................................ 418
NumChannels........................................................................................................................... 418
Width..........................................................................................................................................418
CUDA_ARRAY_SPARSE_PROPERTIES........................................................................................ 419
depth..........................................................................................................................................419
flags..........................................................................................................................................419
height.......................................................................................................................................419
miptailFirstLevel....................................................................................................................... 419
miptailSize.................................................................................................................................... 419
width........................................................................................................................................... 419
CUDA_EXTERNAL_MEMORY_BUFFER_DESC.................................................................................. 420
flags........................................................................................................................................... 420
offset......................................................................................................................................... 420
size............................................................................................................................................. 420
CUDA_EXTERNAL_MEMORY_HANDLE_DESC.................................................................................. 420
fd................................................................................................................................................ 420
flags........................................................................................................................................... 420
handle........................................................................................................................................ 421
name.......................................................................................................................................... 421
nvSciBufObject.......................................................................................................................... 421
size............................................................................................................................................. 421
type............................................................................................................................................ 421
win32.......................................................................................................................................... 421
CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC............................................................... 422
arrayDesc.................................................................................................................................. 422
numLevels................................................................................................................................. 422
offset.......................................................................................................................................... 422
CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC......................................................................... 422
fd................................................................................................................................................ 422
flags........................................................................................................................................... 423
handle........................................................................................................................................ 423
name.......................................................................................................................................... 423
nvSciSyncObj............................................................................................................................. 423
type............................................................................................................................................ 423
win32.......................................................................................................................................... 423
CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS................................................................. 424
fence.......................................................................................................................................... 424
flags........................................................................................................................................... 424
key.............................................................................................................................................. 424
keyedMutex................................................................................................................................ 424
value........................................................................................................................................... 425
CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS....................................................................... 425
fence.......................................................................................................................................... 425
flags........................................................................................................................................... 425
Height........................................................................................................................................ 430
srcArray..................................................................................................................................... 430
srcDevice................................................................................................................................... 430
srcHost...................................................................................................................................... 430
srcMemoryType.........................................................................................................................430
srcPitch......................................................................................................................................430
srcXInBytes................................................................................................................................430
srcY............................................................................................................................................ 430
WidthInBytes............................................................................................................................. 430
CUDA_MEMCPY3D........................................................................................................................431
Depth......................................................................................................................................... 431
dstArray..................................................................................................................................... 431
dstDevice................................................................................................................................... 431
dstHeight................................................................................................................................... 431
dstHost...................................................................................................................................... 431
dstLOD.......................................................................................................................................431
dstMemoryType.........................................................................................................................431
dstPitch......................................................................................................................................431
dstXInBytes................................................................................................................................431
dstY............................................................................................................................................ 432
dstZ............................................................................................................................................432
Height........................................................................................................................................ 432
reserved0...................................................................................................................................432
reserved1...................................................................................................................................432
srcArray..................................................................................................................................... 432
srcDevice................................................................................................................................... 432
srcHeight................................................................................................................................... 432
srcHost...................................................................................................................................... 432
srcLOD.......................................................................................................................................432
srcMemoryType.........................................................................................................................432
srcPitch......................................................................................................................................432
srcXInBytes................................................................................................................................432
srcY............................................................................................................................................ 433
srcZ............................................................................................................................................433
WidthInBytes............................................................................................................................. 433
CUDA_MEMCPY3D_PEER............................................................................................................ 433
Depth......................................................................................................................................... 433
dstArray..................................................................................................................................... 433
numChannels............................................................................................................................ 437
pitchInBytes...............................................................................................................................437
resType...................................................................................................................................... 437
sizeInBytes................................................................................................................................ 438
width.......................................................................................................................................... 438
CUDA_RESOURCE_VIEW_DESC.................................................................................................. 438
depth.......................................................................................................................................... 438
firstLayer................................................................................................................................... 438
firstMipmapLevel...................................................................................................................... 438
format........................................................................................................................................ 438
height.........................................................................................................................................438
lastLayer....................................................................................................................................439
lastMipmapLevel.......................................................................................................................439
width.......................................................................................................................................... 439
CUDA_TEXTURE_DESC................................................................................................................ 439
addressMode.............................................................................................................................439
borderColor............................................................................................................................... 439
filterMode.................................................................................................................................. 439
flags........................................................................................................................................... 439
maxAnisotropy...........................................................................................................................439
maxMipmapLevelClamp........................................................................................................... 440
minMipmapLevelClamp............................................................................................................440
mipmapFilterMode....................................................................................................................440
mipmapLevelBias......................................................................................................................440
CUdevprop.....................................................................................................................................440
clockRate...................................................................................................................................440
maxGridSize.............................................................................................................................. 440
maxThreadsDim........................................................................................................................440
maxThreadsPerBlock.................................................................................................................440
memPitch.................................................................................................................................. 440
regsPerBlock.............................................................................................................................441
sharedMemPerBlock................................................................................................................ 441
SIMDWidth.................................................................................................................................441
textureAlign............................................................................................................................... 441
totalConstantMemory............................................................................................................... 441
CUeglFrame..................................................................................................................................441
cuFormat................................................................................................................................... 441
depth.......................................................................................................................................... 441
<table>
<thead>
<tr>
<th>CUDA Driver API</th>
<th>Chapter 7. Data Fields</th>
<th>Chapter 8. Deprecated List</th>
</tr>
</thead>
<tbody>
<tr>
<td>eglColorFormat</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>frameType</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>height</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>numChannels</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>pArray</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>pitch</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>planeCount</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>pPitch</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>width</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>CUipcEventHandle</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>CUipcMemHandle</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>CUkernelNodeAttrValue</td>
<td>.........................</td>
<td>................................</td>
</tr>
<tr>
<td>accessPolicyWindow</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>cooperative</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>CUmemAccessDesc</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>flags</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>location</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>CUmemAllocationProp</td>
<td>.........................</td>
<td>................................</td>
</tr>
<tr>
<td>compressionType</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>location</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>requestedHandleTypes</td>
<td>......................</td>
<td>................................</td>
</tr>
<tr>
<td>type</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>usage</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>win32HandleMetaData</td>
<td>....................</td>
<td>................................</td>
</tr>
<tr>
<td>CUmemLocation</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>id</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>type</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>CUstreamAttrValue</td>
<td>.........................</td>
<td>................................</td>
</tr>
<tr>
<td>accessPolicyWindow</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>syncPolicy</td>
<td>..........................</td>
<td>................................</td>
</tr>
<tr>
<td>CUstreamBatchMemOpParams</td>
<td>.................</td>
<td>................................</td>
</tr>
</tbody>
</table>
Chapter 1. Difference between the driver and runtime APIs

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

**Complexity vs. control**

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

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

**Context management**

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

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

### Memcpy

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

#### Synchronous

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

#### Asynchronous

1. For transfers from device memory to pageable host memory, the function will return only once the copy has completed.
2. For transfers from any host memory to any host memory, the function is fully synchronous with respect to the host.
3. For all other transfers, the function is fully asynchronous. If pageable memory must first be staged to pinned memory, this will be handled asynchronously with a worker thread.

**Memset**

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

**Kernel Launches**

Kernel launches are asynchronous with respect to the host. Details of concurrent kernel execution and data transfers can be found in the CUDA Programmers Guide.
Chapter 3. Stream synchronization behavior

Default stream

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

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

Legacy default stream

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

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

```c
k_1<<<1, 1, 0, s>>;() ;
k_2<<<1, 1>>;() ;
k_3<<<1, 1, 0, s>>;() ;
```

The resulting behavior is that `k_2` will block on `k_1` and `k_3` will block on `k_2`.

Non-blocking streams which do not synchronize with the legacy stream can be created using the `cudaStreamNonBlocking` flag with the stream creation APIs.

The legacy default stream can be used explicitly with the `CUstream` `cudaStream_t` handle `CU_STREAM_LEGACY` `cudaStreamLegacy`.
Per-thread default stream

The per-thread default stream is an implicit stream local to both the thread and the 
CUcontext, and which does not synchronize with other streams (just like explicitly created 
streams). The per-thread default stream is not a non-blocking stream and will synchronize 
with the legacy default stream if both are used in a program.

The per-thread default stream can be used explicitly with the CUstream [cudaStream_t] 
handle CU_STREAM_PER_THREAD [cudaStreamPerThread].
Chapter 4. Graph object thread safety

Graph objects (cudaGraph_t, CUgraph) are not internally synchronized and must not be accessed concurrently from multiple threads. API calls accessing the same graph object must be serialized externally.

Note that this includes APIs which may appear to be read-only, such as cudaGraphClone() (cuGraphClone()) and cudaGraphInstantiate() (cuGraphInstantiate()). No API or pair of APIs is guaranteed to be safe to call on the same graph object from two different threads without serialization.
Chapter 5. Modules

Here is a list of all modules:

- Data types used by CUDA driver
- Error Handling
- Initialization
- Version Management
- Device Management
- Device Management [DEPRECATED]
- Primary Context Management
- Context Management
- Context Management [DEPRECATED]
- Module Management
- Memory Management
- Virtual Memory Management
- Unified Addressing
- Stream Management
- Event Management
- External Resource Interoperability
- Stream memory operations
- Execution Control
- Execution Control [DEPRECATED]
- Graph Management
- Occupancy
- Texture Reference Management [DEPRECATED]
- Surface Reference Management [DEPRECATED]
- Texture Object Management
- Surface Object Management
- Peer Context Memory Access
5.1. Data types used by CUDA driver
struct CUaccessPolicyWindow
struct CUarrayMapInfo
struct CUDA_ARRAY3D_DESCRIPTOR
struct CUDA_ARRAY_DESCRIPTOR
struct CUDA_ARRAY_SPARSE_PROPERTIES
struct CUDA_EXTERNAL_MEMORY_BUFFER_DESC
struct CUDA_EXTERNAL_MEMORY_HANDLE_DESC
struct CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC
struct CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC
struct CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS
struct CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS
struct CUDA_HOST_NODE_PARAMS
struct CUDA_KERNEL_NODE_PARAMS
struct CUDA_LAUNCH_PARAMS
struct CUDA_MEMCPY2D
struct CUDA_MEMCPY3D
struct CUDA_MEMCPY3D_PEER
struct CUDA_MEMSET_NODE_PARAMS
struct CUDA_POINTER_ATTRIBUTE_P2P_TOKENS
struct CUDA_RESOURCE_DESC
struct CUDA_RESOURCE_VIEW_DESC
struct CUDA_TEXTURE_DESC
struct CUdevprop
struct CUeglFrame
struct CUipcEventHandle
struct CUipcMemHandle
union CUkernelNodeAttrValue
struct CUmemAccessDesc
struct CUmemAllocationProp
struct CUmemLocation
union CUstreamAttrValue
union CUstreamBatchMemOpParams
enum CUaccessProperty

Specifies performance hint with CUaccessPolicyWindow for hitProp and missProp members.

Values

CU_ACCESS_PROPERTY_NORMAL = 0
Normal cache persistence.

CU_ACCESS_PROPERTY_STREAMING = 1
Streaming access is less likely to persist from cache.

CU_ACCESS_PROPERTY_PERSISTING = 2
Persisting access is more likely to persist in cache.

enum CUaddress_mode

Texture reference addressing modes

Values

CU_TR_ADDRESS_MODE_WRAP = 0
Wrapping address mode

CU_TR_ADDRESS_MODE_CLAMP = 1
Clamp to edge address mode

CU_TR_ADDRESS_MODE_MIRROR = 2
Mirror address mode

CU_TR_ADDRESS_MODE_BORDER = 3
Border address mode

enum CUarray_cubemap_face

Array indices for cube faces

Values

CU_CUBEMAP_FACE_POSITIVE_X = 0x00
Positive X face of cubemap

CU_CUBEMAP_FACE_NEGATIVE_X = 0x01
Negative X face of cubemap

CU_CUBEMAP_FACE_POSITIVE_Y = 0x02
Positive Y face of cubemap

CU_CUBEMAP_FACE_NEGATIVE_Y = 0x03
Negative Y face of cubemap

CU_CUBEMAP_FACE_POSITIVE_Z = 0x04
Positive Z face of cubemap

CU_CUBEMAP_FACE_NEGATIVE_Z = 0x05
Negative Z face of cubemap
Negative Z face of cubemap

enum CUarray_format

Array formats

Values

CU_AD_FORMAT_UNSIGNED_INT8 = 0x01
Unsigned 8-bit integers

CU_AD_FORMAT_UNSIGNED_INT16 = 0x02
Unsigned 16-bit integers

CU_AD_FORMAT_UNSIGNED_INT32 = 0x03
Unsigned 32-bit integers

CU_AD_FORMAT_SIGNED_INT8 = 0x08
Signed 8-bit integers

CU_AD_FORMAT_SIGNED_INT16 = 0x09
Signed 16-bit integers

CU_AD_FORMAT_SIGNED_INT32 = 0x0a
Signed 32-bit integers

CU_AD_FORMAT_HALF = 0x10
16-bit floating point

CU_AD_FORMAT_FLOAT = 0x20
32-bit floating point

enum CUarraySparseSubresourceType

Sparse subresource types

Values

CU_ARRAY_SPARSE_SUBRESOURCE_TYPE_SPARSE_LEVEL = 0
CU_ARRAY_SPARSE_SUBRESOURCE_TYPE_MIPTAIL = 1

enum CUcomputemode

Compute Modes

Values

CU_COMPUTEMODE_DEFAULT = 0
Default compute mode (Multiple contexts allowed per device)

CU_COMPUTEMODE_PROHIBITED = 2
Compute-prohibited mode (No contexts can be created on this device at this time)

CU_COMPUTEMODE_EXCLUSIVE_PROCESS = 3
Compute-exclusive-process mode (Only one context used by a single process can be present on this device at a time)

**enum CUctx_flags**

Context creation flags

**Values**

- **CU_CTX_SCHED_AUTO** = 0x00
  Automatic scheduling
- **CU_CTX_SCHED_SPIN** = 0x01
  Set spin as default scheduling
- **CU_CTX_SCHED_YIELD** = 0x02
  Set yield as default scheduling
- **CU_CTX_SCHED_BLOCKING_SYNC** = 0x04
  Set blocking synchronization as default scheduling
  
  *Deprecated* This flag was deprecated as of CUDA 4.0 and was replaced with **CU_CTX_SCHED_BLOCKING_SYNC**.
- **CU_CTX_SCHED_MASK** = 0x07
- **CU_CTX_MAP_HOST** = 0x08
  *Deprecated* This flag was deprecated as of CUDA 11.0 and it no longer has any effect. All contexts as of CUDA 3.2 behave as though the flag is enabled.
- **CU_CTX_LMEM_RESIZE_TO_MAX** = 0x10
  Keep local memory allocation after launch
- **CU_CTX_FLAGS_MASK** = 0x1f

**enum CUDA_POINTER_ATTRIBUTE_ACCESS_FLAGS**

Access flags that specify the level of access the current context’s device has on the memory referenced.

**Values**

- **CU_POINTER_ATTRIBUTE_ACCESS_FLAG_NONE** = 0x0
  No access, meaning the device cannot access this memory at all, thus must be staged through accessible memory in order to complete certain operations
- **CU_POINTER_ATTRIBUTE_ACCESS_FLAG_READ** = 0x1
  Read-only access, meaning writes to this memory are considered invalid accesses and thus return error in that case.
- **CU_POINTER_ATTRIBUTE_ACCESS_FLAG_READWRITE** = 0x3
  Read-write access, the device has full read-write access to the memory
enum CUdevice_attribute

Device properties

Values

<table>
<thead>
<tr>
<th>Enum Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK = 1</td>
<td>Maximum number of threads per block</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X = 2</td>
<td>Maximum block dimension X</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y = 3</td>
<td>Maximum block dimension Y</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z = 4</td>
<td>Maximum block dimension Z</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X = 5</td>
<td>Maximum grid dimension X</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y = 6</td>
<td>Maximum grid dimension Y</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z = 7</td>
<td>Maximum grid dimension Z</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK = 8</td>
<td>Maximum shared memory available per block in bytes</td>
</tr>
<tr>
<td>CUDEVICE_ATTRIBUTE_SHARED_MEMORY_PER_BLOCK = 8</td>
<td>Deprecated, use CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY = 9</td>
<td>Memory available on device for <strong>constant</strong> variables in a CUDA C kernel in bytes</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_WARP_SIZE = 10</td>
<td>Warp size in threads</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_PITCH = 11</td>
<td>Maximum pitch in bytes allowed by memory copies</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK = 12</td>
<td>Maximum number of 32-bit registers available per block</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_REGISTERS_PER_BLOCK = 12</td>
<td>Deprecated, use CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_CLOCK_RATE = 13</td>
<td>Typical clock frequency in kilohertz</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT = 14</td>
<td>Alignment requirement for textures</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_GPU_OVERLAP = 15</td>
<td>Device can possibly copy memory and execute a kernel concurrently. Deprecated. Use instead CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT.</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT = 16</td>
<td>Number of multiprocessors on device</td>
</tr>
</tbody>
</table>
CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT = 17
   Specifies whether there is a run time limit on kernels

CU_DEVICE_ATTRIBUTE_INTEGRATED = 18
   Device is integrated with host memory

CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY = 19
   Device can map host memory into CUDA address space

CU_DEVICE_ATTRIBUTE_COMPUTE_MODE = 20
   Compute mode (See CUcomputemode for details)

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH = 21
   Maximum 1D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_WIDTH = 22
   Maximum 2D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_HEIGHT = 23
   Maximum 2D texture height

CUDEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH = 24
   Maximum 3D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT = 25
   Maximum 3D texture height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH = 26
   Maximum 3D texture depth

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH = 27
   Maximum 2D layered texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT = 28
   Maximum 2D layered texture height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS = 29
   Maximum layers in a 2D layered texture

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_WIDTH = 27
   Deprecated, use CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_HEIGHT = 28
   Deprecated, use CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_NUMSLICES = 29
   Deprecated, use CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS

CU_DEVICE_ATTRIBUTE_SURFACE_ALIGNMENT = 30
   Alignment requirement for surfaces

CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS = 31
   Device can possibly execute multiple kernels concurrently

CU_DEVICE_ATTRIBUTE_ECC_ENABLED = 32
   Device has ECC support enabled

CU_DEVICE_ATTRIBUTE_PCI_BUS_ID = 33
   PCI bus ID of the device

CU_DEVICE_ATTRIBUTE_PCI_DEVICE_ID = 34
   PCI device ID of the device

CU_DEVICE_ATTRIBUTE_TCC_DRIVER = 35

Device is using TCC driver model

CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE = 36
Peak memory clock frequency in kilohertz

CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH = 37
Global memory bus width in bits

CU_DEVICE_ATTRIBUTE_L2_CACHE_SIZE = 38
Size of L2 cache in bytes

CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR = 39
Maximum resident threads per multiprocessor

CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT = 40
Number of asynchronous engines

CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING = 41
Device shares a unified address space with the host

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH = 42
Maximum 1D layered texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED LAYERS = 43
Maximum layers in a 1D layered texture

CU_DEVICE_ATTRIBUTE_CAN_TEX2D_GATHER = 44
Deprecated, do not use.

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_GATHER_WIDTH = 45
Maximum 2D texture width if CUDA_ARRAY3D_TEXTURE_GATHER is set

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_GATHER_HEIGHT = 46
Maximum 2D texture height if CUDA_ARRAY3D_TEXTURE_GATHER is set

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH_ALTERNATE = 47
Alternate maximum 3D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT_ALTERNATE = 48
Alternate maximum 3D texture height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH_ALTERNATE = 49
Alternate maximum 3D texture depth

CU_DEVICE_ATTRIBUTE_PCI_DOMAIN_ID = 50
PCI domain ID of the device

CU_DEVICE_ATTRIBUTE_TEXTURE_PITCH_ALIGNMENT = 51
Pitch alignment requirement for textures

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_WIDTH = 52
Maximum cubemap texture width/height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_LAYERED_WIDTH = 53
Maximum cubemap layered texture width/height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_LAYERED LAYERS = 54
Maximum layers in a cubemap layered texture

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_WIDTH = 55
Maximum 1D surface width

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_WIDTH = 56
Maximum 2D surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_HEIGHT = 57
  Maximum 2D surface height
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_WIDTH = 58
  Maximum 3D surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_HEIGHT = 59
  Maximum 3D surface height
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_DEPTH = 60
  Maximum 3D surface depth
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_WIDTH = 61
  Maximum 1D layered surface width
CUDEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_LAYERS = 62
  Maximum layers in a 1D layered surface
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_WIDTH = 63
  Maximum 2D layered surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_HEIGHT = 64
  Maximum 2D layered surface height
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_LAYERS = 65
  Maximum layers in a 2D layered surface
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_WIDTH = 66
  Maximum cubemap surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED_WIDTH = 67
  Maximum cubemap layered surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED_LAYERS = 68
  Maximum layers in a cubemap layered surface
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LINEAR_WIDTH = 69
  Maximum 1D linear texture width
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_WIDTH = 70
  Maximum 2D linear texture width
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_HEIGHT = 71
  Maximum 2D linear texture height
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_PITCH = 72
  Maximum 2D linear texture pitch in bytes
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_WIDTH = 73
  Maximum mipmapped 2D texture width
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_HEIGHT = 74
  Maximum mipmapped 2D texture height
CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MAJOR = 75
  Major compute capability version number
CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MINOR = 76
  Minor compute capability version number
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_MIPMAPPED_WIDTH = 77
  Maximum mipmapped 1D texture width
CU_DEVICE_ATTRIBUTE_STREAM_PRIORITIES_SUPPORTED = 78
Device supports stream priorities

```
CU_DEVICE_ATTRIBUTE_GLOBAL_L1_CACHE_SUPPORTED = 79
Device supports caching globals in L1
```

```
CU_DEVICE_ATTRIBUTE_LOCAL_L1_CACHE_SUPPORTED = 80
Device supports caching locals in L1
```

```
CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_MULTIPROCESSOR = 81
Maximum shared memory available per multiprocessor in bytes
```

```
CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_MULTIPROCESSOR = 82
Maximum number of 32-bit registers available per multiprocessor
```

```
CU_DEVICE_ATTRIBUTE_MANAGED_MEMORY = 83
Device can allocate managed memory on this system
```

```
CU_DEVICE_ATTRIBUTE_MULTI_GPU_BOARD = 84
Device is on a multi-GPU board
```

```
CU_DEVICE_ATTRIBUTE_MULTI_GPU_BOARD_GROUP_ID = 85
Unique id for a group of devices on the same multi-GPU board
```

```
CU_DEVICE_ATTRIBUTE_HOST_NATIVE_ATOMIC_SUPPORTED = 86
Link between the device and the host supports native atomic operations (this is a
placeholder attribute, and is not supported on any current hardware)
```

```
CU_DEVICE_ATTRIBUTE_SINGLE_TO_DOUBLE_PRECISION_PERF_RATIO = 87
Ratio of single precision performance [in floating-point operations per second] to double
precision performance
```

```
CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS = 88
Device supports coherently accessing pageable memory without calling cudaHostRegister
on it
```

```
CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS = 89
Device can coherently access managed memory concurrently with the CPU
```

```
CU_DEVICE_ATTRIBUTE_COMPUTE_PREEMPTION_SUPPORTED = 90
Device supports compute preemption.
```

```
CU_DEVICE_ATTRIBUTE_CAN_USE_HOST_POINTER_FOR_REGISTERED_MEM = 91
Device can access host registered memory at the same virtual address as the CPU
```

```
CU_DEVICE_ATTRIBUTE_CAN_USE_STREAM_MEM_OPS = 92
cuStreamBatchMemOp and related APIs are supported.
```

```
CU_DEVICE_ATTRIBUTE_CAN_USE_64_BIT_STREAM_MEM_OPS = 93
64-bit operations are supported in cuStreamBatchMemOp and related APIs.
```

```
CU_DEVICE_ATTRIBUTE_CAN_USE_STREAM_WAIT_VALUE_NOR = 94
CU_STREAM_WAIT_VALUE_NOR is supported.
```

```
CU_DEVICE_ATTRIBUTE_COOPERATIVE_LAUNCH = 95
Device supports launching cooperative kernels via cuLaunchCooperativeKernel
```

```
CU_DEVICE_ATTRIBUTE_COOPERATIVE_MULTI_DEVICE_LAUNCH = 96
Device can participate in cooperative kernels launched via
cuLaunchCooperativeKernelMultiDevice
```

```
CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK_OPTIN = 97
Maximum optin shared memory per block
```
CU_DEVICE_ATTRIBUTE_CAN_FLUSH_REMOTE_WRITES = 98
Both the CU_STREAM_WAIT_VALUE_FLUSH flag and the
CU_STREAM_MEM_OP_FLUSH_REMOTE_WRITES MemOp are supported on the device.
See Stream memory operations for additional details.

CU_DEVICE_ATTRIBUTE_HOST_REGISTER_SUPPORTED = 99
Device supports host memory registration via cudaHostRegister.

CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES = 100
Device accesses pageable memory via the host’s page tables.

CU_DEVICE_ATTRIBUTE_DIRECT_MANAGED_MEM_ACCESS_FROM_HOST = 101
The host can directly access managed memory on the device without migration.

CU_DEVICE_ATTRIBUTE_VIRTUAL_ADDRESS_MANAGEMENT_SUPPORTED = 102
Device supports virtual address management APIs like cuMemAddressReserve, cuMemCreate, cuMemMap and related APIs.

CU_DEVICE_ATTRIBUTE_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR_SUPPORTED = 103
Device supports exporting memory to a posix file descriptor with cuMemExportToShareableHandle, if requested via cuMemCreate.

CU_DEVICE_ATTRIBUTE_HANDLE_TYPE_WIN32_HANDLE_SUPPORTED = 104
Device supports exporting memory to a Win32 NT handle with cuMemExportToShareableHandle, if requested via cuMemCreate.

CU_DEVICE_ATTRIBUTE_HANDLE_TYPE_WIN32_KMT_HANDLE_SUPPORTED = 105
Device supports exporting memory to a Win32 KMT handle with cuMemExportToShareableHandle, if requested cuMemCreate.

CU_DEVICE_ATTRIBUTE_MAX_BLOCKS_PER_MULTIPROCESSOR = 106
Maximum number of blocks per multiprocessor.

CU_DEVICE_ATTRIBUTE_GENERIC_COMPRESSION_SUPPORTED = 107
Device supports compression of memory.

CU_DEVICE_ATTRIBUTE_MAX_PERSISTING_L2_CACHE_SIZE = 108
Device’s maximum L2 persisting lines capacity setting in bytes.

CU_DEVICE_ATTRIBUTE_MAX_ACCESS_POLICY_WINDOW_SIZE = 109
The maximum value of CUaccessPolicyWindow::num_bytes.

CU_DEVICE_ATTRIBUTE_GPU_DIRECT_RDMA_WITH_CUDA_VMM_SUPPORTED = 110
Device supports specifying the GPUDirect RDMA flag with cuMemCreate.

CU_DEVICE_ATTRIBUTE_RESERVED_SHARED_MEMORY_PER_BLOCK = 111
Shared memory reserved by CUDA driver per block in bytes.

CU_DEVICE_ATTRIBUTE_SPARSE_CUDA_ARRAY_SUPPORTED = 112
Device supports sparse CUDA arrays and sparse CUDA mipmapped arrays.

CU_DEVICE_ATTRIBUTE_READ_ONLY_HOST_REGISTER_SUPPORTED = 113
Device supports using the cuMemHostRegister flag CU_MEMHOSTREGISTER_READ_ONLY to register memory that must be mapped as read-only to the GPU.

CU_DEVICE_ATTRIBUTE_MAX
enum CUdevice_P2PAttribute

P2P Attributes

Values

CU_DEVICE_P2P_ATTRIBUTE_PERFORMANCE_RANK = 0x01
A relative value indicating the performance of the link between two devices

CU_DEVICE_P2P_ATTRIBUTE_ACCESS_SUPPORTED = 0x02
P2P Access is enable

CU_DEVICE_P2P_ATTRIBUTE_NATIVE_ATOMIC_SUPPORTED = 0x03
Atomic operation over the link supported

CU_DEVICE_P2P_ATTRIBUTE_ACCESS_ACCESS_SUPPORTED = 0x04
 Deprecated use CU_DEVICE_P2P_ATTRIBUTE_CUDA_ARRAY_ACCESS_SUPPORTED instead

CU_DEVICE_P2P_ATTRIBUTE_CUDA_ARRAY_ACCESS_SUPPORTED = 0x04
Accessing CUDA arrays over the link supported

enum CUeglColorFormat

CUDA EGL Color Format - The different planar and multiplanar formats currently supported for CUDA_EGL interops. Three channel formats are currently not supported for CU_EGL_FRAME_TYPE_ARRAY

Values

CU_EGL_COLOR_FORMAT_YUV420_PLANAR = 0x00
Y, U, V in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_YUV420_SEMIPLANAR = 0x01
Y, UV in two surfaces (UV as one surface) with VU byte ordering, width, height ratio same as YUV420Planar.

CU_EGL_COLOR_FORMAT_YUV422_PLANAR = 0x02
Y, U, V each in a separate surface, U/V width = 1/2 Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YUV422_SEMIPLANAR = 0x03
Y, UV in two surfaces with VU byte ordering, width, height ratio same as YUV422Planar.

CU_EGL_COLOR_FORMAT_RGB = 0x04
R/G/B three channels in one surface with BGR byte ordering. Only pitch linear format supported.

CU_EGL_COLOR_FORMAT_BGR = 0x05
R/G/B three channels in one surface with RGB byte ordering. Only pitch linear format supported.

CU_EGL_COLOR_FORMAT_ARGB = 0x06
R/G/B/A four channels in one surface with BGRA byte ordering.
CU_EGL_COLOR_FORMAT_RGBA = 0x07
    R/G/B/A four channels in one surface with ABGR byte ordering.
CU_EGL_COLOR_FORMAT_L = 0x08
    single luminance channel in one surface.
CU_EGL_COLOR_FORMAT_R = 0x09
    single color channel in one surface.
CU_EGL_COLOR_FORMAT_YUV444_PLANAR = 0x0A
    Y, U, V in three surfaces, each in a separate surface, U/V width = Y width, U/V height = Y height.
CU_EGL_COLOR_FORMAT_YUV444_SEMIPLANAR = 0x0B
    Y, UV in two surfaces (UV as one surface) with VU byte ordering, width, height ratio same as YUV444Planar.
CU_EGL_COLOR_FORMAT_YUYV_422 = 0x0C
    Y, U, V in one surface, interleaved as UYVY.
CU_EGL_COLOR_FORMAT_UYVY_422 = 0x0D
    Y, U, V in one surface, interleaved as YUYV.
CU_EGL_COLOR_FORMAT_ABGR = 0x0E
    R/G/B/A four channels in one surface with RGBA byte ordering.
CU_EGL_COLOR_FORMAT_BGRA = 0x0F
    R/G/B/A four channels in one surface with ARGB byte ordering.
CU_EGL_COLOR_FORMAT_A = 0x10
    Alpha color format - one channel in one surface.
CU_EGL_COLOR_FORMAT_RG = 0x11
    R/G color format - two channels in one surface with GR byte ordering.
CU_EGL_COLOR_FORMAT_AYUV = 0x12
    Y, U, V, A four channels in one surface, interleaved as VUYA.
CU_EGL_COLOR_FORMAT_YVU444_SEMIPLANAR = 0x13
    Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = Y width, U/V height = Y height.
CU_EGL_COLOR_FORMAT_YVU422_SEMIPLANAR = 0x14
    Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.
CU_EGL_COLOR_FORMAT_YVU420_SEMIPLANAR = 0x15
    Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.
CU_EGL_COLOR_FORMAT_Y10V10U10_444_SEMIPLANAR = 0x16
    Y10, V10U10 in two surfaces (VU as one surface) with UV byte ordering, U/V width = Y width, U/V height = Y height.
CU_EGL_COLOR_FORMAT_Y10V10U10_420_SEMIPLANAR = 0x17
    Y10, V10U10 in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.
CU_EGL_COLOR_FORMAT_Y12V12U12_444_SEMIPLANAR = 0x18

Y12, V12U12 in two surfaces (VU as one surface) with UV byte ordering, U/V width = Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_Y12V12U12_420_SEMIPLANAR = 0x19
Y12, V12U12 in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_VYUY_ER = 0x1A
Extended Range Y, U, V in one surface, interleaved as YVYU.

CU_EGL_COLOR_FORMAT_UYVY_ER = 0x1B
Extended Range Y, U, V in one surface, interleaved as YUYV.

CU_EGL_COLOR_FORMAT_YUYV_ER = 0x1C
Extended Range Y, U, V in one surface, interleaved as UYVY.

CU_EGL_COLOR_FORMAT_VYUYU_ER = 0x1D
Extended Range Y, U, V in one surface, interleaved as VYUY.

CU_EGL_COLOR_FORMAT_VYUV_ER = 0x1E
Extended Range Y, U, V three channels in one surface, interleaved as VUY. Only pitch linear format supported.

CU_EGL_COLOR_FORMAT_YUVA_ER = 0x1F
Extended Range Y, U, V, A four channels in one surface, interleaved as AVUY.

CU_EGL_COLOR_FORMAT_AYUV_ER = 0x20
Extended Range Y, U, V, A four channels in one surface, interleaved as VUYA.

CU_EGL_COLOR_FORMAT_YUV444_PLANAR_ER = 0x21
Extended Range Y, U, V in three surfaces, U/V width = Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YUV422_PLANAR_ER = 0x22
Extended Range Y, U, V in three surfaces, U/V width = 1/2 Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YUV420_PLANAR_ER = 0x23
Extended Range Y, U, V in three surfaces, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_YUV444_SEMIPLANAR_ER = 0x24
Extended Range Y, UV in two surfaces (UV as one surface) with VU byte ordering, U/V width = Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YUV422_SEMIPLANAR_ER = 0x25
Extended Range Y, UV in two surfaces (UV as one surface) with VU byte ordering, U/V width = 1/2 Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YUV420_SEMIPLANAR_ER = 0x26
Extended Range Y, UV in two surfaces (UV as one surface) with VU byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_YUV444_PLANAR_ER = 0x27
Extended Range Y, V, U in three surfaces, U/V width = Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YUV422_PLANAR_ER = 0x28
Extended Range Y, V, U in three surfaces, U/V width = 1/2 Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YUV420_PLANAR_ER = 0x29
Extended Range Y, V, U in three surfaces, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_YUV444_SEMIPLANAR_ER = 0x2A
Extended Range Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YUV422_SEMIPLANAR_ER = 0x2B

Extended Range Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YUV420_SEMIPLANAR_ER = 0x2C

Extended Range Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_BAYER_RGGB = 0x2D

Bayer format - one channel in one surface with interleaved RGGB ordering.

CU_EGL_COLOR_FORMAT_BAYER_BGGR = 0x2E

Bayer format - one channel in one surface with interleaved BGGR ordering.

CU_EGL_COLOR_FORMAT_BAYER_GRBG = 0x2F

Bayer format - one channel in one surface with interleaved GRBG ordering.

CU_EGL_COLOR_FORMAT_BAYER_GBRG = 0x30

Bayer format - one channel in one surface with interleaved GBRG ordering.

CU_EGL_COLOR_FORMAT_BAYER10_RGGB = 0x31

Bayer10 format - one channel in one surface with interleaved RGGB ordering. Out of 16 bits, 10 bits used 6 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER10_BGGR = 0x32

Bayer10 format - one channel in one surface with interleaved BGGR ordering. Out of 16 bits, 10 bits used 6 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER10_GRBG = 0x33

Bayer10 format - one channel in one surface with interleaved GRBG ordering. Out of 16 bits, 10 bits used 6 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER10_GBRG = 0x34

Bayer10 format - one channel in one surface with interleaved GBRG ordering. Out of 16 bits, 10 bits used 6 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_RGGB = 0x35

Bayer12 format - one channel in one surface with interleaved RGGB ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_BGGR = 0x36

Bayer12 format - one channel in one surface with interleaved BGGR ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_GRBG = 0x37

Bayer12 format - one channel in one surface with interleaved GRBG ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_GBRG = 0x38

Bayer12 format - one channel in one surface with interleaved GBRG ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER14_RGGB = 0x39

Bayer14 format - one channel in one surface with interleaved RGGB ordering. Out of 16 bits, 14 bits used 2 bits No-op.
CU_EGL_COLOR_FORMAT_BAYER14_BGGR = 0x3A
Bayer14 format - one channel in one surface with interleaved BGGR ordering. Out of 16 bits, 14 bits used 2 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER14_GRBG = 0x3B
Bayer14 format - one channel in one surface with interleaved GRBG ordering. Out of 16 bits, 14 bits used 2 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER14_GBRG = 0x3C
Bayer14 format - one channel in one surface with interleaved GBRG ordering. Out of 16 bits, 14 bits used 2 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER20_RGGB = 0x3D
Bayer20 format - one channel in one surface with interleaved RGGB ordering. Out of 32 bits, 20 bits used 12 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER20_BGGR = 0x3E
Bayer20 format - one channel in one surface with interleaved BGGR ordering. Out of 32 bits, 20 bits used 12 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER20_GRBG = 0x3F
Bayer20 format - one channel in one surface with interleaved GRBG ordering. Out of 32 bits, 20 bits used 12 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER20_GBRG = 0x40
Bayer20 format - one channel in one surface with interleaved GBRG ordering. Out of 32 bits, 20 bits used 12 bits No-op.

CU_EGL_COLOR_FORMAT_YVU444_PLANAR = 0x41
Y, V, U in three surfaces, each in a separate surface, U/V width = Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YVU422_PLANAR = 0x42
Y, V, U in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = Y height.

CU_EGL_COLOR_FORMAT_YVU420_PLANAR = 0x43
Y, V, U in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

CU_EGL_COLOR_FORMAT_BAYER_ISP_RGGB = 0x44
Nvidia proprietary Bayer ISP format - one channel in one surface with interleaved RGGB ordering and mapped to opaque integer datatype.

CU_EGL_COLOR_FORMAT_BAYER_ISP_BGGR = 0x45
Nvidia proprietary Bayer ISP format - one channel in one surface with interleaved BGGR ordering and mapped to opaque integer datatype.

CU_EGL_COLOR_FORMAT_BAYER_ISP_GRBG = 0x46
Nvidia proprietary Bayer ISP format - one channel in one surface with interleaved GRBG ordering and mapped to opaque integer datatype.

CU_EGL_COLOR_FORMAT_BAYER_ISP_GBRG = 0x47
Nvidia proprietary Bayer ISP format - one channel in one surface with interleaved GBRG ordering and mapped to opaque integer datatype.

CU_EGL_COLOR_FORMAT_BAYER_BCCR = 0x48
Bayer format - one channel in one surface with interleaved BCCR ordering.

CU_EGL_COLOR_FORMAT_BAYER_RCCB = 0x49
Bayer format - one channel in one surface with interleaved RCCB ordering.

CU_EGL_COLOR_FORMAT_BAYER_CRBC = 0x4A
Bayer format - one channel in one surface with interleaved CRBC ordering.

CU_EGL_COLOR_FORMAT_BAYER_CBRC = 0x4B
Bayer format - one channel in one surface with interleaved CBRC ordering.

CU_EGL_COLOR_FORMAT_BAYER10_CCCC = 0x4C
Bayer10 format - one channel in one surface with interleaved CCCC ordering. Out of 16 bits, 10 bits used 6 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_BCCR = 0x4D
Bayer12 format - one channel in one surface with interleaved BCCR ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_RCCB = 0x4E
Bayer12 format - one channel in one surface with interleaved RCCB ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_CRBC = 0x4F
Bayer12 format - one channel in one surface with interleaved CRBC ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_CBRC = 0x50
Bayer12 format - one channel in one surface with interleaved CBRC ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_BAYER12_CCCC = 0x51
Bayer12 format - one channel in one surface with interleaved CCCC ordering. Out of 16 bits, 12 bits used 4 bits No-op.

CU_EGL_COLOR_FORMAT_MAX

enum CUeglFrameType
CUDA EglFrame type - array or pointer

Values

CU_EGL_FRAME_TYPE_ARRAY = 0
Frame type CUDA array

CU_EGL_FRAME_TYPE_PITCH = 1
Frame type pointer

enum CUeglResourceLocationFlags
Resource location flags- sysmem or vidmem

For CUDA context on iGPU, since video and system memory are equivalent - these flags will not have an effect on the execution.
For CUDA context on dGPU, applications can use the flag `CUEglResourceLocationFlags` to give a hint about the desired location.

`CU_EGLRESOURCELOCATION_SYSMEM` - the frame data is made resident on the system memory to be accessed by CUDA.

`CU_EGLRESOURCELOCATION_VIDMEM` - the frame data is made resident on the dedicated video memory to be accessed by CUDA.

There may be an additional latency due to new allocation and data migration, if the frame is produced on a different memory.

**Values**

`CU_EGLRESOURCELOCATION_SYSMEM = 0x00`
- Resource location sysmem

`CU_EGLRESOURCELOCATION_VIDMEM = 0x01`
- Resource location vidmem

**enum CUevent_flags**

Event creation flags

**Values**

`CU_EVENT_DEFAULT = 0x0`
- Default event flag

`CU_EVENT_BLOCKING_SYNC = 0x1`
- Event uses blocking synchronization

`CU_EVENT_DISABLE_TIMING = 0x2`
- Event will not record timing data

`CU_EVENT_INTERPROCESS = 0x4`
- Event is suitable for interprocess use. `CU_EVENT_DISABLE_TIMING` must be set

**enum CUevent_record_flags**

Event record flags

**Values**

`CU_EVENT_RECORD_DEFAULT = 0x0`
- Default event record flag

`CU_EVENT_RECORD_EXTERNAL = 0x1`
- When using stream capture, create an event record node instead of the default behavior.
  This flag is invalid when used outside of capture.
enum CUevent_wait_flags

Event wait flags

Values

CU_EVENT_WAIT_DEFAULT = 0x0
  Default event wait flag
CU_EVENT_WAIT_EXTERNAL = 0x1
  When using stream capture, create an event wait node instead of the default behavior. This flag is invalid when used outside of capture.

enum CUexternalMemoryHandleType

External memory handle types

Values

CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD = 1
  Handle is an opaque file descriptor
CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32 = 2
  Handle is an opaque shared NT handle
CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT = 3
  Handle is an opaque, globally shared handle
CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP = 4
  Handle is a D3D12 heap object
CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12RESOURCE = 5
  Handle is a D3D12 committed resource
CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11RESOURCE = 6
  Handle is a shared NT handle to a D3D11 resource
CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11RESOURCE_KMT = 7
  Handle is a globally shared handle to a D3D11 resource
CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIbuf = 8
  Handle is an NvSciBuf object

enum CUexternalSemaphoreHandleType

External semaphore handle types

Values

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD = 1
  Handle is an opaque file descriptor
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32 = 2
  Handle is an opaque shared NT handle
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT = 3
   Handle is an opaque, globally shared handle
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE = 4
   Handle is a shared NT handle referencing a D3D12 fence object
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE = 5
   Handle is a shared NT handle referencing a D3D11 fence object
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC = 6
   Opaque handle to NvSciSync Object
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX = 7
   Handle is a shared NT handle referencing a D3D11 keyed mutex object
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX_KMT = 8
   Handle is a globally shared handle referencing a D3D11 keyed mutex object

enum CUfilter_mode
Texture reference filtering modes

Values
CU_TR_FILTER_MODE_POINT = 0
   Point filter mode
CU_TR_FILTER_MODE_LINEAR = 1
   Linear filter mode

enum CUfunc_cache
Function cache configurations

Values
CU_FUNC_CACHE_PREFER_NONE = 0x00
   no preference for shared memory or L1 [default]
CU_FUNC_CACHE_PREFER_SHARED = 0x01
   prefer larger shared memory and smaller L1 cache
CU_FUNC_CACHE_PREFER_L1 = 0x02
   prefer larger L1 cache and smaller shared memory
CU_FUNC_CACHE_PREFER_EQUAL = 0x03
   prefer equal sized L1 cache and shared memory

enum CUfunction_attribute
Function properties

Values
CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK = 0
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 = 1**

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 = 2**

The size in bytes of user-allocated constant memory required by this function.

**CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES = 3**

The size in bytes of local memory used by each thread of this function.

**CU_FUNC_ATTRIBUTE_NUM_REGS = 4**

The number of registers used by each thread of this function.

**CU_FUNC_ATTRIBUTE_PTX_VERSION = 5**

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 = 6**

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.

**CU_FUNC_ATTRIBUTE_CACHE_MODE_CA = 7**

The attribute to indicate whether the function has been compiled with user specified option “-Xptxas --dlcm=ca” set.

**CU_FUNC_ATTRIBUTE_MAX_DYNAMIC_SHARED_SIZE_BYTES = 8**

The maximum size in bytes of dynamically-allocated shared memory that can be used by this function. If the user-specified dynamic shared memory size is larger than this value, the launch will fail. See cuFuncSetAttribute

**CU_FUNC_ATTRIBUTE_PREFERRED_SHARED_MEMORY_CARVEOUT = 9**

On devices where the L1 cache and shared memory use the same hardware resources, this sets the shared memory carveout preference, in percent of the total shared memory. Refer toCU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_MULTIPROCESSOR. This is only a hint, and the driver can choose a different ratio if required to execute the function. See cuFuncSetAttribute

**CU_FUNC_ATTRIBUTE_MAX**

enum CUgraphicsMapResourceFlags

Flags for mapping and unmapping interop resources

**Values**

**CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE = 0x00**
enum CUgraphicsRegisterFlags
Flags to register a graphics resource

Values
CU_GRAPHICS_REGISTER_FLAGS_NONE = 0x00
CU_GRAPHICS_REGISTER_FLAGS_READ_ONLY = 0x01
CU_GRAPHICS_REGISTER_FLAGS_WRITE_DISCARD = 0x02
CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST = 0x04
CU_GRAPHICS_REGISTER_FLAGS_TEXTURE_GATHER = 0x08

enum CUgraphNodeType
Graph node types

Values
CU_GRAPH_NODE_TYPE_KERNEL = 0
GPU kernel node
CU_GRAPH_NODE_TYPE_MEMCPY = 1
Memcpy node
CU_GRAPH_NODE_TYPE_MEMSET = 2
Memset node
CU_GRAPH_NODE_TYPE_HOST = 3
Host (executable) node
CU_GRAPH_NODE_TYPE_GRAPH = 4
Node which executes an embedded graph
CU_GRAPH_NODE_TYPE_EMPTY = 5
Empty (no-op) node
CU_GRAPH_NODE_TYPE_WAIT_EVENT = 6
External event wait node
CU_GRAPH_NODE_TYPE_EVENT_RECORD = 7
External event record node

enum CUipcMem_flags
CUDA Ipc Mem Flags

Values
CU_IPC_MEM_LAZY_ENABLE_PEER_ACCESS = 0x1
Automatically enable peer access between remote devices as needed
enum CUjit_cacheMode

Caching modes for dlcm

Values

CU_JIT_CACHE_OPTION_NONE = 0
  Compile with no -dlcm flag specified
CU_JIT_CACHE_OPTION_CG
  Compile with L1 cache disabled
CU_JIT_CACHE_OPTION_CA
  Compile with L1 cache enabled

enum CUjit_fallback

Cubin matching fallback strategies

Values

CU_PREFER_PTX = 0
  Prefer to compile ptx if exact binary match not found
CU_PREFER_BINARY
  Prefer to fall back to compatible binary code if exact match not found

enum CUjit_option

Online compiler and linker options

Values

CU_JIT_MAX_REGISTERS = 0
  Max number of registers that a thread may use. Option type: unsigned int Applies to: compiler only
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 to 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. Cannot be combined with CU_JIT_TARGET. Option type: unsigned int Applies to: compiler only
CU_JIT_WALL_TIME
  Overwrites the option value with the total wall clock time, in milliseconds, spent in the compiler and linker Option type: float Applies to: compiler and linker
CU_JIT_INFO_LOG_BUFFER
Modules

Pointer to a buffer in which to print any log messages that are informational in nature (the buffer size is specified via option `CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES`) Option type: char * Applies to: compiler and linker

**CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES**
- IN: Log buffer size in bytes. Log messages will be capped at this size (including null terminator)
- OUT: Amount of log buffer filled with messages Option type: unsigned int
- Applies to: compiler and linker

**CU_JIT_ERROR_LOG_BUFFER**
- Pointer to a buffer in which to print any log messages that reflect errors (the buffer size is specified via option `CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES`) Option type: char *
- Applies to: compiler and linker

**CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES**
- IN: Log buffer size in bytes. Log messages will be capped at this size (including null terminator)
- OUT: Amount of log buffer filled with messages Option type: unsigned int
- Applies to: compiler and linker

**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
- Applies to: compiler only

**CU_JIT_TARGET_FROM_CUCONTX**
- No option value required. Determines the target based on the current attached context (default) Option type: No option value needed
- Applies to: compiler and linker

**CU_JIT_TARGET**
- Target is chosen based on supplied `CUjit_target`. Cannot be combined with `CU_JIT_THREADS_PER_BLOCK`. Option type: unsigned int for enumerated type `CUjit_target`
- Applies to: compiler and linker

**CU_JIT_FALLBACK_STRATEGY**
- Specifies choice of fallback strategy if matching cubin is not found. Choice is based on supplied `CUjit_fallback`. This option cannot be used with cuLink* APIs as the linker requires exact matches. Option type: unsigned int for enumerated type `CUjit_fallback`
- Applies to: compiler only

**CU_JIT_GENERATE_DEBUG_INFO**
- Specifies whether to create debug information in output [-g] [0: false, default] Option type: int
- Applies to: compiler and linker

**CU_JIT_LOG_VERBOSE**
- Generate verbose log messages [0: false, default] Option type: int
- Applies to: compiler and linker

**CU_JIT_GENERATE_LINE_INFO**
- Generate line number information [-lineinfo] [0: false, default] Option type: int
- Applies to: compiler only

**CU_JIT_CACHE_MODE**
- Specifies whether to enable caching explicitly [-dlcm]. Choice is based on supplied `CUjit_cacheMode_enum`. Option type: unsigned int for enumerated type `CUjit_cacheMode_enum`
- Applies to: compiler only
**CU_JIT_NEW_SM3X_OPT**

The below jit options are used for internal purposes only, in this version of CUDA.

**CU_JIT_FAST_COMPILE**

**CU_JIT_GLOBAL_SYMBOL_NAMES**

Array of device symbol names that will be relocated to the corresponding host addresses stored in `CU_JIT_GLOBAL_SYMBOL_ADDRESSES`. Must contain `CU_JIT_GLOBAL_SYMBOL_COUNT` entries. When loading a device module, driver will relocate all encountered unresolved symbols to the host addresses. It is only allowed to register symbols that correspond to unresolved global variables. It is illegal to register the same device symbol at multiple addresses. Option type: `const char **` Applies to: dynamic linker only

**CU_JIT_GLOBAL_SYMBOL_ADDRESSES**

Array of host addresses that will be used to relocate corresponding device symbols stored in `CU_JIT_GLOBAL_SYMBOL_NAMES`. Must contain `CU_JIT_GLOBAL_SYMBOL_COUNT` entries. Option type: `void **` Applies to: dynamic linker only

**CU_JIT_GLOBAL_SYMBOL_COUNT**

Number of entries in `CU_JIT_GLOBAL_SYMBOL_NAMES` and `CU_JIT_GLOBAL_SYMBOL_ADDRESSES` arrays. Option type: `unsigned int` Applies to: dynamic linker only

**CU_JIT_NUM_OPTIONS**

---

**enum CUjit_target**

Online compilation targets

**Values**

- **CU_TARGET_COMPUTE_20 = 20**
  
  Compute device class 2.0

- **CU_TARGET_COMPUTE_21 = 21**
  
  Compute device class 2.1

- **CU_TARGET_COMPUTE_30 = 30**
  
  Compute device class 3.0

- **CU_TARGET_COMPUTE_32 = 32**
  
  Compute device class 3.2

- **CU_TARGET_COMPUTE_35 = 35**
  
  Compute device class 3.5

- **CU_TARGET_COMPUTE_37 = 37**
  
  Compute device class 3.7

- **CU_TARGET_COMPUTE_50 = 50**
  
  Compute device class 5.0

- **CU_TARGET_COMPUTE_52 = 52**
  
  Compute device class 5.2

- **CU_TARGET_COMPUTE_53 = 53**
  
  Compute device class 5.3
Compute device class 5.3

CU_TARGET_COMPUTE_60 = 60
    Compute device class 6.0.
CU_TARGET_COMPUTE_61 = 61
    Compute device class 6.1.
CU_TARGET_COMPUTE_62 = 62
    Compute device class 6.2.
CU_TARGET_COMPUTE_70 = 70
    Compute device class 7.0.
CU_TARGET_COMPUTE_72 = 72
    Compute device class 7.2.
CU_TARGET_COMPUTE_75 = 75
    Compute device class 7.5.
CU_TARGET_COMPUTE_80 = 80
    Compute device class 8.0.
CU_TARGET_COMPUTE_86 = 86
    Compute device class 8.6.

enum CUjitInputType

Device code formats

Values

CU_JIT_INPUT_CUBIN = 0
    Compiled device-class-specific device code Applicable options: none
CU_JIT_INPUT_PTX
    PTX source code Applicable options: PTX compiler options
CU_JIT_INPUT_FATBINARY
    Bundle of multiple cubins and/or PTX of some device code Applicable options: PTX compiler options, CU_JIT_FALLBACK_STRATEGY
CU_JIT_INPUT_OBJECT
    Host object with embedded device code Applicable options: PTX compiler options, CU_JIT_FALLBACK_STRATEGY
CU_JIT_INPUT_LIBRARY
    Archive of host objects with embedded device code Applicable options: PTX compiler options, CU_JIT_FALLBACK_STRATEGY
CU_JIT_NUM_INPUT_TYPES

enum CUKernelNodeAttrID

Graph kernel node Attributes
Values

**CU_KERNEL_NODE_ATTRIBUTE_ACCESS_POLICY_WINDOW = 1**
Identifier for `CUkernelNodeAttrValue::accessPolicyWindow`.

**CU_KERNEL_NODE_ATTRIBUTE_COOPERATIVE = 2**
Allows a kernel node to be cooperative [see `cuLaunchCooperativeKernel`].

---

**enum CUlimit**

Limits

Values

**CU_LIMIT_STACK_SIZE = 0x00**
- GPU thread stack size

**CU_LIMIT_PRINTF_FIFO_SIZE = 0x01**
- GPU printf FIFO size

**CU_LIMIT_MALLOC_HEAP_SIZE = 0x02**
- GPU malloc heap size

**CU_LIMIT_DEV_RUNTIME_SYNC_DEPTH = 0x03**
- GPU device runtime launch synchronize depth

**CU_LIMIT_DEV_RUNTIME_PENDING_LAUNCH_COUNT = 0x04**
- GPU device runtime pending launch count

**CU_LIMIT_MAX_L2_FETCH_GRANULARITY = 0x05**
- A value between 0 and 128 that indicates the maximum fetch granularity of L2 (in Bytes).
  - This is a hint

**CU_LIMIT_PERSISTING_L2_CACHE_SIZE = 0x06**
- A size in bytes for L2 persisting lines cache size

**CU_LIMIT_MAX**

---

**enum CUmem_advise**

Memory advise values

Values

**CU_MEM_ADVISE_SET_READ_MOSTLY = 1**
- Data will mostly be read and only occasionally be written to

**CU_MEM_ADVISE_UNSET_READ_MOSTLY = 2**
- Undo the effect of `CU_MEM_ADVISE_SET_READ_MOSTLY`

**CU_MEM_ADVISE_SET_PREFERRED_LOCATION = 3**
- Set the preferred location for the data as the specified device

**CU_MEM_ADVISE_UNSET_PREFERRED_LOCATION = 4**
- Clear the preferred location for the data

**CU_MEM_ADVISE_SET_ACCESSED_BY = 5**
Data will be accessed by the specified device, so prevent page faults as much as possible

\[
\text{CU_MEM_ADVISE_UNSET_ACCESSIBILITY} = 6
\]

Let the Unified Memory subsystem decide on the page faulting policy for the specified device

**enum CUmemAccess_flags**

Specifies the memory protection flags for mapping.

**Values**

- **CU_MEM_ACCESS_FLAGS_PROT_NONE** = 0x0
  - Default, make the address range not accessible
- **CU_MEM_ACCESS_FLAGS_PROT_READ** = 0x1
  - Make the address range read accessible
- **CU_MEM_ACCESS_FLAGS_PROT_READWRITE** = 0x3
  - Make the address range read-write accessible
- **CU_MEM_ACCESS_FLAGS_PROT_MAX** = 0xFFFFFFFF

**enum CUmemAllocationCompType**

Specifies compression attribute for an allocation.

**Values**

- **CU_MEM_ALLOCATION_COMP_NONE** = 0x0
  - Allocating non-compressible memory
- **CU_MEM_ALLOCATION_COMP_GENERIC** = 0x1
  - Allocating compressible memory

**enum CUmemAllocationGranularity_flags**

Flag for requesting different optimal and required granularities for an allocation.

**Values**

- **CU_MEM_ALLOC_GRANULARITY_MINIMUM** = 0x0
  - Minimum required granularity for allocation
- **CU_MEM_ALLOC_GRANULARITY_RECOMMENDED** = 0x1
  - Recommended granularity for allocation for best performance

**enum CUmemAllocationHandleType**

Flags for specifying particular handle types
Values

CU_MEM_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR = 0x1
  Allows a file descriptor to be used for exporting. Permitted only on POSIX systems. (int)
CU_MEM_HANDLE_TYPE_WIN32 = 0x2
  Allows a Win32 NT handle to be used for exporting. (HANDLE)
CU_MEM_HANDLE_TYPE_WIN32_KMT = 0x4
  Allows a Win32 KMT handle to be used for exporting. (D3DKMT_HANDLE)
CU_MEM_HANDLE_TYPE_MAX = 0xFFFFFFFF

enum CUmemAllocationType

Defines the allocation types available

Values

CU_MEM_ALLOCATION_TYPE_INVALID = 0x0
CU_MEM_ALLOCATION_TYPE_PINNED = 0x1
  This allocation type is ‘pinned’, i.e. cannot migrate from its current location while the application is actively using it
CU_MEM_ALLOCATION_TYPE_MAX = 0xFFFFFFFF

enum CUmemAttach_flags

CUDA Mem Attach Flags

Values

CU_MEM_ATTACH_GLOBAL = 0x1
  Memory can be accessed by any stream on any device
CU_MEM_ATTACH_HOST = 0x2
  Memory cannot be accessed by any stream on any device
CU_MEM_ATTACH_SINGLE = 0x4
  Memory can only be accessed by a single stream on the associated device

enum CUmemHandleType

Memory handle types

Values

CU_MEM_HANDLE_TYPE_GENERIC = 0

enum CUmemLocationType

Specifies the type of location
Values
CU_MEM_LOCATION_TYPE_INVALID = 0x0
CU_MEM_LOCATION_TYPE_DEVICE = 0x1
   Location is a device location, thus id is a device ordinal
CU_MEM_LOCATION_TYPE_MAX = 0xFFFFFFFF

enum CUmemOperationType
Memory operation types

Values
CU_MEM_OPERATION_TYPE_MAP = 1
CU_MEM_OPERATION_TYPE_UNMAP = 2

enum CUmemorytype
Memory types

Values
CU_MEMORYTYPE_HOST = 0x01
   Host memory
CU_MEMORYTYPE_DEVICE = 0x02
   Device memory
CU_MEMORYTYPE_ARRAY = 0x03
   Array memory
CU_MEMORYTYPE_UNIFIED = 0x04
   Unified device or host memory

enum CUoccupancy_flags
Occupancy calculator flag

Values
CU_OCCUPANCY_DEFAULT = 0x0
   Default behavior
CU_OCCUPANCY_DISABLE_CACHING_OVERRIDE = 0x1
   Assume global caching is enabled and cannot be automatically turned off

enum CUpointer_attribute
Pointer information
Values

CU_POINTER_ATTRIBUTE_CONTEXT = 1
- The CUcontext on which a pointer was allocated or registered

CU_POINTER_ATTRIBUTE_MEMORY_TYPE = 2
- The CUmemorytype describing the physical location of a pointer

CU_POINTER_ATTRIBUTE_DEVICE_POINTER = 3
- The address at which a pointer’s memory may be accessed on the device

CU_POINTER_ATTRIBUTE_HOST_POINTER = 4
- The address at which a pointer’s memory may be accessed on the host

CU_POINTER_ATTRIBUTE_P2P_TOKENS = 5
- A pair of tokens for use with the nv-p2p.h Linux kernel interface

CU_POINTER_ATTRIBUTE_SYNC_MEMOPS = 6
- Synchronize every synchronous memory operation initiated on this region

CU_POINTER_ATTRIBUTE_BUFFER_ID = 7
- A process-wide unique ID for an allocated memory region

CU(Pointer_Attribute_ATTR_VALUE) IS_MANAGED = 8
- Indicates if the pointer points to managed memory

CU_POINTER_ATTRIBUTE_DEVICE_ORDINAL = 9
- A device ordinal of a device on which a pointer was allocated or registered

CU_POINTER_ATTRIBUTE_IS_LEGACY_CUDA_IPC_CAPABLE = 10
- 1 if this pointer maps to an allocation that is suitable for `cudaIpcGetMemHandle`, 0 otherwise

CU_POINTER_ATTRIBUTE_RANGE_START_ADDR = 11
- Starting address for this requested pointer

CU_POINTER_ATTRIBUTE_RANGE_SIZE = 12
- Size of the address range for this requested pointer

CU_POINTER_ATTRIBUTE_MAPPED = 13
- 1 if this pointer is in a valid address range that is mapped to a backing allocation, 0 otherwise

CU_POINTER_ATTRIBUTE_ALLOWED_HANDLE_TYPES = 14
- Bitmask of allowed `CUnionAllocationHandleType` for this allocation

CU_POINTER_ATTRIBUTE_IS_GPU_DIRECT_RDMA_CAPABLE = 15
- 1 if the memory this pointer is referencing can be used with the GPUDirect RDMA API

CU_POINTER_ATTRIBUTE_ACCESS_FLAGS = 16
- Returns the access flags the device associated with the current context has on the corresponding memory referenced by the pointer given

enum CUresourceType

Resource types
Values

CURESOURCE_TYPE_ARRAY = 0x00
Array resource

CURESOURCE_TYPE_MIPMAPPED_ARRAY = 0x01
Mipmapped array resource

CURESOURCE_TYPE_LINEAR = 0x02
Linear resource

CURESOURCE_TYPE_PITCH2D = 0x03
Pitch 2D resource

definition enum CUresourceViewFormat

Resource view format

Values

CU_RES_VIEW_FORMAT_NONE = 0x00
No resource view format (use underlying resource format)

CU_RES_VIEW_FORMAT_UINT_1X8 = 0x01
1 channel unsigned 8-bit integers

CU_RES_VIEW_FORMAT_UINT_2X8 = 0x02
2 channel unsigned 8-bit integers

CU_RES_VIEW_FORMAT_UINT_4X8 = 0x03
4 channel unsigned 8-bit integers

CU_RES_VIEW_FORMAT_SINT_1X8 = 0x04
1 channel signed 8-bit integers

CU_RES_VIEW_FORMAT_SINT_2X8 = 0x05
2 channel signed 8-bit integers

CU_RES_VIEW_FORMAT_SINT_4X8 = 0x06
4 channel signed 8-bit integers

CU_RES_VIEW_FORMAT_UINT_1X16 = 0x07
1 channel unsigned 16-bit integers

CU_RES_VIEW_FORMAT_UINT_2X16 = 0x08
2 channel unsigned 16-bit integers

CU_RES_VIEW_FORMAT_UINT_4X16 = 0x09
4 channel unsigned 16-bit integers

CU_RES_VIEW_FORMAT_SINT_1X16 = 0x0a
1 channel signed 16-bit integers

CU_RES_VIEW_FORMAT_SINT_2X16 = 0x0b
2 channel signed 16-bit integers

CU_RES_VIEW_FORMAT_SINT_4X16 = 0x0c
4 channel signed 16-bit integers

CU_RES_VIEW_FORMAT_UINT_1X32 = 0x0d
1 channel unsigned 32-bit integers
CU_RES_VIEW_FORMAT_UINT_2X32 = 0x0e
2 channel unsigned 32-bit integers
CU_RES_VIEW_FORMAT_UINT_4X32 = 0x0f
4 channel unsigned 32-bit integers
CU_RES_VIEW_FORMAT_SINT_1X32 = 0x10
1 channel signed 32-bit integers
CU_RES_VIEW_FORMAT_SINT_2X32 = 0x11
2 channel signed 32-bit integers
CU_RES_VIEW_FORMAT_SINT_4X32 = 0x12
4 channel signed 32-bit integers
CU_RES_VIEW_FORMAT_FLOAT_1X16 = 0x13
1 channel 16-bit floating point
CU_RES_VIEW_FORMAT_FLOAT_2X16 = 0x14
2 channel 16-bit floating point
CU_RES_VIEW_FORMAT_FLOAT_4X16 = 0x15
4 channel 16-bit floating point
CU_RES_VIEW_FORMAT_FLOAT_1X32 = 0x16
1 channel 32-bit floating point
CU_RES_VIEW_FORMAT_FLOAT_2X32 = 0x17
2 channel 32-bit floating point
CU_RES_VIEW_FORMAT_FLOAT_4X32 = 0x18
4 channel 32-bit floating point
CU_RES_VIEW_FORMAT_UNSIGNED_BC1 = 0x19
Block compressed 1
CU_RES_VIEW_FORMAT_UNSIGNED_BC2 = 0x1a
Block compressed 2
CU_RES_VIEW_FORMAT_UNSIGNED_BC3 = 0x1b
Block compressed 3
CU_RES_VIEW_FORMAT_UNSIGNED_BC4 = 0x1c
Block compressed 4 unsigned
CU_RES_VIEW_FORMAT_SIGNED_BC4 = 0x1d
Block compressed 4 signed
CU_RES_VIEW_FORMAT_UNSIGNED_BC5 = 0x1e
Block compressed 5 unsigned
CU_RES_VIEW_FORMAT_SIGNED_BC5 = 0x1f
Block compressed 5 signed
CU_RES_VIEW_FORMAT_UNSIGNED_BC6H = 0x20
Block compressed 6 unsigned half-float
CU_RES_VIEW_FORMAT_SIGNED_BC6H = 0x21
Block compressed 6 signed half-float
CU_RES_VIEW_FORMAT_UNSIGNED_BC7 = 0x22
Block compressed 7
enum CUresult

Error codes

Values

CUDA_SUCCESS = 0
The API call returned with no errors. In the case of query calls, this also means that the
operation being queried is complete (see cuEventQuery() and cuStreamQuery()).

CUDA_ERROR_INVALID_VALUE = 1
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 = 2
The API call failed because it was unable to allocate enough memory to perform the
requested operation.

CUDA_ERROR_NOT_INITIALIZED = 3
This indicates that the CUDA driver has not been initialized with cuInit() or that initialization
has failed.

CUDA_ERROR_DEINITIALIZED = 4
This indicates that the CUDA driver is in the process of shutting down.

CUDA_ERROR_PROFILER_DISABLED = 5
This indicates profiler is not initialized for this run. This can happen when the application is
running with external profiling tools like visual profiler.

CUDA_ERROR_PROFILER_NOT_INITIALIZED = 6
Deprecated This error return is deprecated as of CUDA 5.0. It is no longer an error to
attempt to enable/disable the profiling via cuProfilerStart or cuProfilerStop without
initialization.

CUDA_ERROR_PROFILER_ALREADY_STARTED = 7
Deprecated This error return is deprecated as of CUDA 5.0. It is no longer an error to call
cuProfilerStart() when profiling is already enabled.

CUDA_ERROR_PROFILER_ALREADY_STOPPED = 8
Deprecated This error return is deprecated as of CUDA 5.0. It is no longer an error to call
cuProfilerStop() when profiling is already disabled.

CUDA_ERROR_STUB_LIBRARY = 34
This indicates that the CUDA driver that the application has loaded is a stub library.
Applications that run with the stub rather than a real driver loaded will result in CUDA API
returning this error.

CUDA_ERROR_NO_DEVICE = 100
This indicates that no CUDA-capable devices were detected by the installed CUDA driver.

CUDA_ERROR_INVALID_DEVICE = 101
This indicates that the device ordinal supplied by the user does not correspond to a valid
CUDA device.

CUDA_ERROR_DEVICE_NOT_LICENSED = 102
This error indicates that the Grid license is not applied.

**CUDA_ERROR_INVALID_IMAGE = 200**
This indicates that the device kernel image is invalid. This can also indicate an invalid CUDA module.

**CUDA_ERROR_INVALID_CONTEXT = 201**
This most frequently indicates that there is no context bound to the current thread. This can also be returned if the context passed to an API call is not a valid handle (such as a context that has had `cuCtxDestroy()` invoked on it). This can also be returned if a user mixes different API versions (i.e. 3010 context with 3020 API calls). See `cuCtxGetApiVersion()` for more details.

**CUDA_ERROR_CONTEXT_ALREADY_CURRENT = 202**
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 = 205**
This indicates that a map or register operation has failed.

**CUDA_ERROR_UNMAP_FAILED = 206**
This indicates that an unmap or unregister operation has failed.

**CUDA_ERROR_ARRAY_IS_MAPPED = 207**
This indicates that the specified array is currently mapped and thus cannot be destroyed.

**CUDA_ERROR_ALREADY_MAPPED = 208**
This indicates that the resource is already mapped.

**CUDA_ERROR_NO_BINARY_FOR_GPU = 209**
This indicates that there is no kernel image available that is suitable for the device. This can occur when a user specifies code generation options for a particular CUDA source file that do not include the corresponding device configuration.

**CUDA_ERROR_ALREADY_ACQUIRED = 210**
This indicates that a resource has already been acquired.

**CUDA_ERROR_NOT_MAPPED = 211**
This indicates that a resource is not mapped.

**CUDA_ERROR_NOT_MAPPED_AS_ARRAY = 212**
This indicates that a mapped resource is not available for access as an array.

**CUDA_ERROR_NOT_MAPPED_AS_POINTER = 213**
This indicates that a mapped resource is not available for access as a pointer.

**CUDA_ERROR_ECC_UNCORRECTABLE = 214**
This indicates that an uncorrectable ECC error was detected during execution.

**CUDA_ERROR_UNSUPPORTED_LIMIT = 215**
This indicates that the `CUlimit` passed to the API call is not supported by the active device.

**CUDA_ERROR_CONTEXT_ALREADY_IN_USE = 216**
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_PEER_ACCESS_UNSUPPORTED = 217**
This indicates that peer access is not supported across the given devices.
CUDA_ERROR_INVALID_PTX = 218
This indicates that a PTX JIT compilation failed.
CUDA_ERROR_INVALID_GRAPHICS_CONTEXT = 219
This indicates an error with OpenGL or DirectX context.
CUDA_ERROR_NVLINK_UNCORRECTABLE = 220
This indicates that an uncorrectable NVLink error was detected during the execution.
CUDA_ERROR_JIT_COMPILER_NOT_FOUND = 221
This indicates that the PTX JIT compiler library was not found.
CUDA_ERROR_UNSUPPORTED_PTX_VERSION = 222
This indicates that the provided PTX was compiled with an unsupported toolchain.
CUDA_ERROR_INVALID_SOURCE = 300
This indicates that the device kernel source is invalid.
CUDA_ERROR_FILE_NOT_FOUND = 301
This indicates that the file specified was not found.
CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND = 302
This indicates that a link to a shared object failed to resolve.
CUDA_ERROR_SHARED_OBJECT_INIT_FAILED = 303
This indicates that initialization of a shared object failed.
CUDA_ERROR_OPERATING_SYSTEM = 304
This indicates that an OS call failed.
CUDA_ERROR_INVALID_HANDLE = 400
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_ILLEGAL_STATE = 401
This indicates that a resource required by the API call is not in a valid state to perform the requested operation.
CUDA_ERROR_NOT_FOUND = 500
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 = 600
This indicates that asynchronous operations issued previously have not completed yet. This result is not actually an error, but must be indicated differently than CUDA_SUCCESS (which indicates completion). Calls that may return this value include cuEventQuery() and cuStreamQuery().
CUDA_ERROR_ILLEGAL_ADDRESS = 700
While executing a kernel, the device encountered a load or store instruction on an invalid memory address. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.
CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES = 701
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 = 702
This indicates that the device kernel took too long to execute. This can only occur if timeouts are enabled - see the device attribute

CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING = 703
This indicates a kernel launch that uses an incompatible texturing mode.

CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED = 704
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 = 705
This error indicates that cuCtxDisablePeerAccess() is trying to disable peer access which has not been enabled yet via cuCtxEnablePeerAccess().

CUDA_ERROR_PRIMARY_CONTEXT_ACTIVE = 708
This error indicates that the primary context for the specified device has already been initialized.

CUDA_ERROR_CONTEXT_IS_DESTROYED = 709
This error indicates that the context current to the calling thread has been destroyed using cuCtxDestroy, or is a primary context which has not yet been initialized.

CUDA_ERROR_ASSERT = 710
A device-side assert triggered during kernel execution. The context cannot be used anymore, and must be destroyed. All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.

CUDA_ERROR_TOO_MANY_PEERS = 711
This error indicates that the hardware resources required to enable peer access have been exhausted for one or more of the devices passed to cuCtxEnablePeerAccess().

CUDA_ERROR_HOST_MEMORY_ALREADY_REGISTERED = 712
This error indicates that the memory range passed to cuMemHostRegister() has already been registered.

CUDA_ERROR_HOST_MEMORY_NOT_REGISTERED = 713
This error indicates that the pointer passed to cuMemHostUnregister() does not correspond to any currently registered memory region.

CUDA_ERROR_HARDWARE_STACK_ERROR = 714
While executing a kernel, the device encountered a stack error. This can be due to stack corruption or exceeding the stack size limit. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

CUDA_ERROR_ILLEGAL_INSTRUCTION = 715
While executing a kernel, the device encountered an illegal instruction. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

**CUDA_ERROR_MISALIGNED_ADDRESS = 716**
While executing a kernel, the device encountered a load or store instruction on a memory address which is not aligned. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

**CUDA_ERROR_INVALID_ADDRESS_SPACE = 717**
While executing a kernel, the device encountered an instruction which can only operate on memory locations in certain address spaces (global, shared, or local), but was supplied a memory address not belonging to an allowed address space. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

**CUDA_ERROR_INVALID_PC = 718**
While executing a kernel, the device program counter wrapped its address space. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

**CUDA_ERROR_LAUNCH_FAILED = 719**
An exception occurred on the device while executing a kernel. Common causes include dereferencing an invalid device pointer and accessing out of bounds shared memory. Less common cases can be system specific - more information about these cases can be found in the system specific user guide. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

**CUDA_ERROR_COOPERATIVE_LAUNCH_TOO_LARGE = 720**
This error indicates that the number of blocks launched per grid for a kernel that was launched via either `cuLaunchCooperativeKernel` or `cuLaunchCooperativeKernelMultiDevice` exceeds the maximum number of blocks as allowed by `cuOccupancyMaxActiveBlocksPerMultiprocessor` or `cuOccupancyMaxActiveBlocksPerMultiprocessorWithFlags` times the number of multiprocessors as specified by the device attribute `CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT`.

**CUDA_ERROR_NOT_PERMITTED = 800**
This error indicates that the attempted operation is not permitted.

**CUDA_ERROR_NOT_SUPPORTED = 801**
This error indicates that the attempted operation is not supported on the current system or device.

**CUDA_ERROR_SYSTEM_NOT_READY = 802**
This error indicates that the system is not yet ready to start any CUDA work. To continue using CUDA, verify the system configuration is in a valid state and all required driver daemons are actively running. More information about this error can be found in the system specific user guide.
CUDA_ERROR_SYSTEM_DRIVER_MISMATCH = 803
This error indicates that there is a mismatch between the versions of the display driver and the CUDA driver. Refer to the compatibility documentation for supported versions.

CUDA_ERROR_COMPAT_NOT_SUPPORTED_ON_DEVICE = 804
This error indicates that the system was upgraded to run with forward compatibility but the visible hardware detected by CUDA does not support this configuration. Refer to the compatibility documentation for the supported hardware matrix or ensure that only supported hardware is visible during initialization via the CUDA_VISIBLE_DEVICES environment variable.

CUDA_ERROR_STREAM_CAPTURE_UNSUPPORTED = 900
This error indicates that the operation is not permitted when the stream is capturing.

CUDA_ERROR_STREAM_CAPTURE_INVALIDATED = 901
This error indicates that the current capture sequence on the stream has been invalidated due to a previous error.

CUDA_ERROR_STREAM_CAPTURE_MERGE = 902
This error indicates that the operation would have resulted in a merge of two independent capture sequences.

CUDA_ERROR_STREAM_CAPTURE_UNMATCHED = 903
This error indicates that the capture was not initiated in this stream.

CUDA_ERROR_STREAM_CAPTURE_UNJOINED = 904
This error indicates that the capture sequence contains a fork that was not joined to the primary stream.

CUDA_ERROR_STREAM_CAPTURE_ISOLATION = 905
This error indicates that a dependency would have been created which crosses the capture sequence boundary. Only implicit in-stream ordering dependencies are allowed to cross the boundary.

CUDA_ERROR_STREAM_CAPTURE_IMPLICIT = 906
This error indicates a disallowed implicit dependency on a current capture sequence from cudaStreamLegacy.

CUDA_ERROR_CAPTURED_EVENT = 907
This error indicates that the operation is not permitted on an event which was last recorded in a capturing stream.

CUDA_ERROR_STREAM_CAPTURE_WRONG_THREAD = 908
A stream capture sequence not initiated with the CU_STREAM_CAPTURE_MODE_RELAXED argument to cuStreamBeginCapture was passed to cuStreamEndCapture in a different thread.

CUDA_ERROR_TIMEOUT = 909
This error indicates that the timeout specified for the wait operation has lapsed.

CUDA_ERROR_GRAPH_EXEC_UPDATE_FAILURE = 910
This error indicates that the graph update was not performed because it included changes which violated constraints specific to instantiated graph update.

CUDA_ERROR_UNKNOWN = 999
This indicates that an unknown internal error has occurred.
enum CUshared_carveout

Shared memory carveout configurations. These may be passed to cuFuncSetAttribute

Values

CU_SHARED_MEM_CARVEOUT_DEFAULT = -1
   No preference for shared memory or L1 (default)
CU_SHARED_MEM_CARVEOUT_MAX_SHARED = 100
   Prefer maximum available shared memory, minimum L1 cache
CU_SHARED_MEM_CARVEOUT_MAX_L1 = 0
   Prefer maximum available L1 cache, minimum shared memory

enum CUsharedconfig

Shared memory configurations

Values

CU_SHARED_MEM_CONFIG_DEFAULT_BANK_SIZE = 0x00
   set default shared memory bank size
CU_SHARED_MEM_CONFIG_FOUR_BYTE_BANK_SIZE = 0x01
   set shared memory bank width to four bytes
CU_SHARED_MEM_CONFIG_EIGHT_BYTE_BANK_SIZE = 0x02
   set shared memory bank width to eight bytes

enum CUstream_flags

Stream creation flags

Values

CU_STREAM_DEFAULT = 0x0
   Default stream flag
CU_STREAM_NON_BLOCKING = 0x1
   Stream does not synchronize with stream 0 (the NULL stream)

enum CUstreamAttrID

Stream Attributes

Values

CU_STREAM_ATTRIBUTE_ACCESS_POLICY_WINDOW = 1
   Identifier for CUstreamAttrValue::accessPolicyWindow.
CU_STREAM_ATTRIBUTE_SYNCHRONIZATION_POLICY = 3
CU synchronization policy for work queued up in this stream

**enum CUstreamBatchMemOpType**

Operations for `cuStreamBatchMemOp`

**Values**

- **CU_STREAM_MEM_OP_WAIT_VALUE_32** = 1
  - Represents a `cuStreamWaitValue32` operation
- **CU_STREAM_MEM_OP_WRITE_VALUE_32** = 2
  - Represents a `cuStreamWriteValue32` operation
- **CU_STREAM_MEM_OP_WAIT_VALUE_64** = 4
  - Represents a `cuStreamWaitValue64` operation
- **CU_STREAM_MEM_OP_WRITE_VALUE_64** = 5
  - Represents a `cuStreamWriteValue64` operation
- **CU_STREAM_MEM_OP_FLUSH_REMOTE_WRITES** = 3
  - This has the same effect as `CU_STREAM_WAIT_VALUE_FLUSH`, but as a standalone operation.

**enum CUstreamCaptureMode**

Possible modes for stream capture thread interactions. For more details see `cuStreamBeginCapture` and `cuThreadExchangeStreamCaptureMode`

**Values**

- **CU_STREAM_CAPTURE_MODE_GLOBAL** = 0
- **CU_STREAM_CAPTURE_MODE_THREAD_LOCAL** = 1
- **CU_STREAM_CAPTURE_MODE_RELAXED** = 2

**enum CUstreamCaptureStatus**

Possible stream capture statuses returned by `cuStreamIsCapturing`

**Values**

- **CU_STREAM_CAPTURE_STATUS_NONE** = 0
  - Stream is not capturing
- **CU_STREAM_CAPTURE_STATUS_ACTIVE** = 1
  - Stream is actively capturing
- **CU_STREAM_CAPTURE_STATUS_INVALIDATED** = 2
  - Stream is part of a capture sequence that has been invalidated, but not terminated
enum CUstreamWaitValue_flags

Flags for cuStreamWaitValue32 and cuStreamWaitValue64

Values

CU_STREAM_WAIT_VALUE_GEQ = 0x0
Wait until (int32_t)*addr - value) >= 0 (or int64_t for 64 bit values). Note this is a cyclic comparison which ignores wraparound. (Default behavior.)

CU_STREAM_WAIT_VALUE_EQ = 0x1
Wait until *addr == value.

CU_STREAM_WAIT_VALUE_AND = 0x2
Wait until (*addr & value) != 0.

CU_STREAM_WAIT_VALUE_NOR = 0x3
Wait until ~(*addr | value) != 0. Support for this operation can be queried with cuDeviceGetAttribute() and CU_DEVICE_ATTRIBUTE_CAN_USE_STREAM_WAIT_VALUE_NOR.

CU_STREAM_WAIT_VALUE_FLUSH = 1<<30
Follow the wait operation with a flush of outstanding remote writes. This means that, if a remote write operation is guaranteed to have reached the device before the wait can be satisfied, that write is guaranteed to be visible to downstream device work. The device is permitted to reorder remote writes internally. For example, this flag would be required if two remote writes arrive in a defined order, the wait is satisfied by the second write, and downstream work needs to observe the first write. Support for this operation is restricted to selected platforms and can be queried with CU_DEVICE_ATTRIBUTE_CAN_USE_WAIT_VALUE_FLUSH.

enum CUstreamWriteValue_flags

Flags for cuStreamWriteValue32

Values

CU_STREAM_WRITE_VALUE_DEFAULT = 0x0
Default behavior

CU_STREAM_WRITE_VALUE_NO_MEMORY_BARRIER = 0x1
Permits the write to be reordered with writes which were issued before it, as a performance optimization. Normally, cuStreamWriteValue32 will provide a memory fence before the write, which has similar semantics to __threadfence_system() but is scoped to the stream rather than a CUDA thread.

typedef struct CUarray_st *CUarray

CUDA array
typedef struct CUctx_st *CUcontext
CUDA context

typedef int CUdevice
CUDA device

typedef unsigned int CUdeviceptr
CUDA device pointer CUdeviceptr is defined as an unsigned integer type whose size matches the size of a pointer on the target platform.

typedef struct CUeglStreamConnection_st *CUeglStreamConnection
CUDA EGLStream Connection

typedef struct CUevent_st *CUevent
CUDA event

typedef struct CUextMemory_st *CUexternalMemory
CUDA external memory

typedef struct CUextSemaphore_st *CUexternalSemaphore
CUDA external semaphore

typedef struct CUfunc_st *CUfunction
CUDA function

typedef struct CUgraph_st *CUgraph
CUDA graph

typedef struct CUgraphExec_st *CUgraphExec
CUDA executable graph
typedef struct CUgraphicsResource_st *CUgraphicsResource
CUDA graphics interop resource

typedef struct CUgraphNode_st *CUgraphNode
CUDA graph node

typedef void (CUDA_CB *CUhostFn) (void* userData)
CUDA host function

typedef struct CUmipmappedArray_st *CUmipmappedArray
CUDA mipmapped array

typedef struct CUmod_st *CUmodule
CUDA module

typedef size_t (CUDA_CB *CUoccupancyB2DSize) (int blockSize)
Block size to per-block dynamic shared memory mapping for a certain kernel

typedef struct CUstream_st *CUstream
CUDA stream

typedef void (CUDA_CB *CUstreamCallback) (CUstream hStream, CUresult status, void* userData)
CUDA stream callback

typedef unsigned long long CUsurfObject
An opaque value that represents a CUDA surface object
typedef struct CUsurfref_st *CUsurfref
CUDA surface reference

typedef unsigned long long CUtexObject
An opaque value that represents a CUDA texture object

typedef struct CUtexref_st *CUtexref
CUDA texture reference

#define CU_ARRAY_SPARSE_PROPERTIES_SINGLE_MIPTAIL 0x1
Indicates that the layered sparse CUDA array or CUDA mipmapped array has a single mip tail region for all layers

#define CU_DEVICE_CPU ((CUdevice)-1)
Device that represents the CPU

#define CU_DEVICE_INVALID ((CUdevice)-2)
Device that represents an invalid device

#define CU_IPC_HANDLE_SIZE 64
CUDA IPC handle size

#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. This buffer needs to honor all alignment/padding requirements of the individual parameters. If CU_LAUNCH_PARAM_BUFFER_SIZE is not specified in the extra array, then CU_LAUNCH_PARAM_BUFFER_POINTER will have no effect.
#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.

#define CU_LAUNCH_PARAM_END ((void*)0x00)

End of array terminator for the extra parameter to cuLaunchKernel

#define CU_MEM_CREATE_USAGE_TILE_POOL 0x1

This flag if set indicates that the memory will be used as a tile pool.

#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()

#define CU_MEMHOSTALLOC_PORTABLE 0x01

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

#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()

#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()

#define CU_MEMHOSTREGISTER_IOMEMORY 0x04

If set, the passed memory pointer is treated as pointing to some memory-mapped I/O space, e.g. belonging to a third-party PCIe device. On Windows the flag is a no-op. On Linux that memory is marked as non cache-coherent for the GPU and is expected to be...
physically contiguous. It may return CUDA_ERROR_NOT_PERMITTED if run as an unprivileged user, CUDA_ERROR_NOT_SUPPORTED on older Linux kernel versions. On all other platforms, it is not supported and CUDA_ERROR_NOT_SUPPORTED is returned. Flag for `cuMemHostRegister`

```c
#define CU_MEMHOSTREGISTER_PORTABLE 0x01
```

If set, host memory is portable between CUDA contexts. Flag for `cuMemHostRegister`

```c
#define CU_MEMHOSTREGISTER_READ_ONLY 0x08
```

If set, the passed memory pointer is treated as pointing to memory that is considered read-only by the device. On platforms without `CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES`, this flag is required in order to register memory mapped to the CPU as read-only. Support for the use of this flag can be queried from the device attribute `CU_DEVICE_ATTRIBUTE_READ ONLY HOST REGISTER_SUPPORTED`. Using this flag with a current context associated with a device that does not have this attribute set will cause `cuMemHostRegister` to error with CUDA_ERROR NOT_SUPPORTED.

```c
#define CU_PARAM_TR_DEFAULT -1
```

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

```c
#define CU_STREAM_LEGACY ((CUstream)0x1)
```

Legacy stream handle

Stream handle that can be passed as a CUstream to use an implicit stream with legacy synchronization behavior.

See details of the [synchronization behavior](#).

```c
#define CU_STREAM_PER_THREAD ((CUstream)0x2)
```

Per-thread stream handle

Stream handle that can be passed as a CUstream to use an implicit stream with per-thread synchronization behavior.

See details of the [synchronization behavior](#).

```c
#define CU_TRSA_OVERRIDE_FORMAT 0x01
```

Override the texref format with a format inferred from the array. Flag for `cuTexRefSetArray`
#define CU_TRSF_DISABLE_TRILINEAR_OPTIMIZATION 0x20
Disable any trilinear filtering optimizations. Flag for cuTexRefSetFlags[] and cuTexObjectCreate[]

#define CU_TRSF_NORMALIZEDCOORDINATES 0x02
Use normalized texture coordinates in the range [0,1) instead of [0,dim). Flag for cuTexRefSetFlags[] and cuTexObjectCreate[]

#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[] and cuTexObjectCreate[]

#define CU_TRSF_SRGB 0x10
Perform sRGB->linear conversion during texture read. Flag for cuTexRefSetFlags[] and cuTexObjectCreate[]

#define CUDA_ARRAY3D_2DARRAY 0x01
Deprecated, use CUDA_ARRAY3D_LAYERED

#define CUDA_ARRAY3D_COLOR_ATTACHMENT 0x20
This flag indicates that the CUDA array may be bound as a color target in an external graphics API

#define CUDA_ARRAY3D_CUBEMAP 0x04
If set, the CUDA array is a collection of six 2D arrays, representing faces of a cube. The width of such a CUDA array must be equal to its height, and Depth must be six. If CUDA ARRAY3D_LAYERED flag is also set, then the CUDA array is a collection of cubemaps and Depth must be a multiple of six.

#define CUDA_ARRAY3D_DEPTH_TEXTURE 0x10
This flag if set indicates that the CUDA array is a DEPTH_TEXTURE.
#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.

#define CUDA_ARRAY3D_SPARSE 0x40

This flag if set indicates that the CUDA array or CUDA mipmapped array is a sparse CUDA array or CUDA mipmapped array respectively

#define CUDA_ARRAY3D_SURFACE_LDST 0x02

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

#define CUDA_ARRAY3D_TEXTURE_GATHER 0x08

This flag must be set in order to perform texture gather operations on a CUDA array.

#define CUDA_COOPERATIVE_LAUNCH_MULTI_DEVICE_NO_POST_LAUNCH_SYNC 0x02

If set, any subsequent work pushed in a stream that participated in a call to cuLaunchCooperativeKernelMultiDevice will only wait for the kernel launched on the GPU corresponding to that stream to complete before it begins execution.

#define CUDA_COOPERATIVE_LAUNCH_MULTI_DEVICE_NO_PRE_LAUNCH_SYNC 0x01

If set, each kernel launched as part of cuLaunchCooperativeKernelMultiDevice only waits for prior work in the stream corresponding to that GPU to complete before the kernel begins execution.

#define CUDA_EGL_INFINITE_TIMEOUT 0xFFFFFFFF

Indicates that timeout for cuEGLStreamConsumerAcquireFrame is infinite.
#define CUDA_EXTERNAL_MEMORY_DEDICATED 0x1
Indicates that the external memory object is a dedicated resource

#define CUDA_EXTERNAL_SEMAPHORE_SIGNAL_SKIP_NVSCIBUF_MEMSYNC 0x01
When the /p flags parameter of CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS contains this flag, it indicates that signaling an external semaphore object should skip performing appropriate memory synchronization operations over all the external memory objects that are imported as CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF, which otherwise are performed by default to ensure data coherency with other importers of the same NvSciBuf memory objects.

#define CUDA_EXTERNAL_SEMAPHORE_WAIT_SKIP_NVSCIBUF_MEMSYNC 0x02
When the /p flags parameter of CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS contains this flag, it indicates that waiting on an external semaphore object should skip performing appropriate memory synchronization operations over all the external memory objects that are imported as CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF, which otherwise are performed by default to ensure data coherency with other importers of the same NvSciBuf memory objects.

#define CUDA_NVSCISYNC_ATTR_SIGNAL 0x1
When /p flags of cuDeviceGetNvSciSyncAttributes is set to this, it indicates that application needs signaler specific NvSciSyncAttr to be filled by cuDeviceGetNvSciSyncAttributes.

#define CUDA_NVSCISYNC_ATTR_WAIT 0x2
When /p flags of cuDeviceGetNvSciSyncAttributes is set to this, it indicates that application needs waiter specific NvSciSyncAttr to be filled by cuDeviceGetNvSciSyncAttributes.

#define CUDA_VERSION 11010
CUDA API version number
#define MAX_PLANES 3
Maximum number of planes per frame

5.2.  Error Handling

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

CUresult cuGetErrorName (CUresult error, const char **pStr)
Gets the string representation of an error code enum name.

Parameters

error
- Error code to convert to string

pStr
- Address of the string pointer.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE

Description

Sets *pStr to the address of a NULL-terminated string representation of the name of the enum error code error. If the error code is not recognized, CUDA_ERROR_INVALID_VALUE will be returned and *pStr will be set to the NULL address.

See also:

CUresult, cudaGetErrorName

CUresult cuGetErrorString (CUresult error, const char **pStr)
Gets the string description of an error code.

Parameters

error
- Error code to convert to string
5.3. Initialization

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

CUresult cuInit (unsigned int Flags)
Initialize the CUDA driver API.

Parameters

Flags
- Initialization flag for CUDA.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE,
CUDA_ERROR_SYSTEM_DRIVER_MISMATCH,
CUDA_ERROR_COMPAT_NOT_SUPPORTED_ON_DEVICE

Description

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.

Note:
Note that this function may also return error codes from previous, asynchronous launches.
5.4. Version Management

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

CUresult cuDriverGetVersion (int *driverVersion)

Returns the latest CUDA version supported by driver.

Parameters

driverVersion
  - Returns the CUDA driver version

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE

Description

Returns in *driverVersion the version of CUDA supported by the driver. The version is returned as (1000 major + 10 minor). For example, CUDA 9.2 would be represented by 9020. This function automatically returns CUDA_ERROR_INVALID_VALUE if driverVersion is NULL.

Note:

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

See also:

cudaDriverGetVersion, cudaRuntimeGetVersion

5.5. Device Management

This section describes the device management functions of the low-level CUDA driver application programming interface.
CUresult cuDeviceGet (CUdevice *device, int ordinal)

Returns a handle to a compute device.

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

Description

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

Note:

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

See also:

cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGetName, cuDeviceGetUuid,
cuDeviceGetLuid, cuDeviceTotalMem

CUresult cuDeviceGetAttribute (int *pi, 
CUdevice_attribute attrib, CUdevice dev)

Returns information about the device.

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

Description
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;
- **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_TEXTURE1D_LINEAR_WIDTH**: Maximum width for a 1D texture bound to linear memory;
- **CU DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_MIPMAPPED_WIDTH**: Maximum mipmapped 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_TEXTURE2D_LINEAR_WIDTH**: Maximum width for a 2D texture bound to linear memory;
- **CU DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_HEIGHT**: Maximum height for a 2D texture bound to linear memory;
- **CU DEVICEATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_PITCH**: Maximum pitch in bytes for a 2D texture bound to linear memory;
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_WIDTH: Maximum mipmapped 2D texture width;
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_HEIGHT: Maximum mipmapped 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_TEXTURE3D_WIDTH_ALTERNATE: Alternate maximum 3D texture width, 0 if no alternate maximum 3D texture size is supported;
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT_ALTERNATE: Alternate maximum 3D texture height, 0 if no alternate maximum 3D texture size is supported;
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH_ALTERNATE: Alternate maximum 3D texture depth, 0 if no alternate maximum 3D texture size is supported;
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_WIDTH: Maximum cubemap texture width or height;
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_MAXIMUM_TEXTURECUBEMAP_LAYERED_WIDTH: Maximum cubemap layered texture width or height;
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_LAYERED_LAYERS: Maximum layers in a cubemap layered texture;
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_WIDTH: Maximum 1D surface width;
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_WIDTH: Maximum 2D surface width;
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_HEIGHT: Maximum 2D surface height;
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_WIDTH: Maximum 3D surface width;
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_WIDTH: Maximum 3D surface width;
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_HEIGHT: Maximum 3D surface height;
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_DEPTH: Maximum 3D surface depth;
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_WIDTH: Maximum 1D layered surface width;
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_LAYERS: Maximum layers in a 1D layered surface;
- **CU DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_WIDTH**: Maximum 2D layered surface width;
- **CU DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_HEIGHT**: Maximum 2D layered surface height;
- **CU DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED LAYERS**: Maximum layers in a 2D layered surface;
- **CU DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_WIDTH**: Maximum cubemap surface width;
- **CU DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED_WIDTH**: Maximum cubemap layered surface width;
- **CU DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED LAYERS**: Maximum layers in a cubemap layered surface;
- **CU DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK**: Maximum number of 32-bit registers available to a thread block;
- **CU DEVICE_ATTRIBUTE_CLOCK_RATE**: The typical 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_TEXTURE_PITCH_ALIGNMENT**: Pitch alignment requirement for 2D texture references bound to pitched memory;
- **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_PROHIBITED**: Compute-prohibited mode - Device is prohibited from creating new CUDA contexts.
  - **CU COMPUTEMODE_EXCLUSIVE_PROCESS**: Compute-exclusive-process mode - Device can have only one context used by a single process at a time.
  - **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_PCI_DOMAIN_ID**: PCI domain 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;
- **CUDEVICE_ATTRIBUTE_UNIFIED_ADDRESSING**: 1 if the device shares a unified address space with the host, or 0 if not;
- **CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MAJOR**: Major compute capability version number;
- **CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MINOR**: Minor compute capability version number;
- **CU_DEVICE_ATTRIBUTE_GLOBAL_L1_CACHE_SUPPORTED**: 1 if device supports caching globals in L1 cache, 0 if caching globals in L1 cache is not supported by the device;
- **CU_DEVICE_ATTRIBUTE_LOCAL_L1_CACHE_SUPPORTED**: 1 if device supports caching locals in L1 cache, 0 if caching locals in L1 cache is not supported by the device;
- **CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_MULTIPROCESSOR**: Maximum amount of shared memory available to a multiprocessor in bytes; this amount is shared by all thread blocks simultaneously resident on a multiprocessor;
- **CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_MULTIPROCESSOR**: Maximum number of 32-bit registers available to a multiprocessor; this number is shared by all thread blocks simultaneously resident on a multiprocessor;
- **CU_DEVICE_ATTRIBUTE_MANAGED_MEMORY**: 1 if device supports allocating managed memory on this system, 0 if allocating managed memory is not supported by the device on this system;
- **CU_DEVICE_ATTRIBUTE_MULTI_GPU_BOARD**: 1 if device is on a multi-GPU board, 0 if not.
- **CU_DEVICE_ATTRIBUTE_MULTI_GPU_BOARD_GROUP_ID**: Unique identifier for a group of devices associated with the same board. Devices on the same multi-GPU board will share the same identifier.

- **CU_DEVICE_ATTRIBUTE_HOST_NATIVE_ATOMIC_SUPPORTED**: 1 if Link between the device and the host supports native atomic operations.

- **CU_DEVICE_ATTRIBUTE_SINGLE_TO_DOUBLE_PRECISION_PERF_RATIO**: Ratio of single precision performance (in floating-point operations per second) to double precision performance.

- **CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS**: Device supports coherently accessing pageable memory without calling cudaHostRegister on it.

- **CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS**: Device can coherently access managed memory concurrently with the CPU.

- **CU_DEVICE_ATTRIBUTE_COMPUTE_PREEMPTION_SUPPORTED**: Device supports Compute Preemption.

- **CU_DEVICE_ATTRIBUTE_CAN_USE_HOST_POINTER_FOR_REGISTERED_MEM**: Device can access host registered memory at the same virtual address as the CPU.

- **CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK_OPTIN**: The maximum per block shared memory size supported on this device. This is the maximum value that can be opted into when using the `cuFuncSetAttribute()` call. For more details see `CU_FUNC_ATTRIBUTE_MAX_DYNAMIC_SHARED_SIZE_BYTES`.

- **CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES**: Device accesses pageable memory via the host’s page tables.

- **CU_DEVICE_ATTRIBUTE_DIRECT_MANAGED_MEM_ACCESS_FROM_HOST**: The host can directly access managed memory on the device without migration.

- **CU_DEVICE_ATTRIBUTE_VIRTUAL_ADDRESS_MANAGEMENT_SUPPORTED**: Device supports virtual address management APIs like `cuMemAddressReserve`, `cuMemCreate`, `cuMemMap` and related APIs.

- **CU_DEVICE_ATTRIBUTE_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR_SUPPORTED**: Device supports exporting memory to a posix file descriptor with `cuMemExportToShareableHandle`, if requested via `cuMemCreate`.

- **CU_DEVICE_ATTRIBUTE_HANDLE_TYPE_WIN32 HANDLE_SUPPORTED**: Device supports exporting memory to a Win32 NT handle with `cuMemExportToShareableHandle`, if requested via `cuMemCreate`.

- **CU_DEVICE_ATTRIBUTE_HANDLE_TYPE_WIN32_KMT_HANDLE_SUPPORTED**: Device supports exporting memory to a Win32 KMT handle with `cuMemExportToShareableHandle`, if requested `cuMemCreate`.

- **CU_DEVICE_ATTRIBUTE_MAX_PERSISTING_L2_CACHE_SIZE**: Maximum L2 cache size in byte for persisting lines.

- **CU DEVICE_ATTRIBUTE_MAX_BLOCKS_PER_MULTIPROCESSOR**: Maximum number of thread blocks that can reside on a multiprocessor.
CU DEVICE ATTRIBUTE GENERIC COMPRESSION SUPPORTED: Device supports compressible memory allocation via cuMemCreate

CU DEVICE ATTRIBUTE RESERVED SHARED MEMORY PER BLOCK: Amount of shared memory per block reserved by CUDA driver in bytes.

CU DEVICE ATTRIBUTE READ ONLY HOST REGISTER SUPPORTED: Device supports using the cuMemHostRegister flag CU_MEMHOSTREGISTER_READ_ONLY to register memory that must be mapped as read-only to the GPU.

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

See also:
cuDeviceGetCount, cuDeviceGetName, cuDeviceGetUuid, cuDeviceGet, cuDeviceTotalMem, cudaDeviceGetAttribute, cudaGetDeviceProperties

CUresult cuDeviceGetCount (int *count)
Returns the number of compute-capable devices.

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

Description
Returns in *count the number of devices with compute capability greater than or equal to 2.0 that are available for execution. If there is no such device, cuDeviceGetCount() returns 0.

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

See also:
cuDeviceGetAttribute, cuDeviceGetName, cuDeviceGetUuid, cuDeviceGetLuid, cuDeviceGet, cuDeviceTotalMem, cudaGetDeviceCount
CUresult cuDeviceGetLuid (char *luid, unsigned int *deviceNodeMask, CUdevice dev)

Return an LUID and device node mask for the device.

Parameters

luid
- Returned LUID
deviceNodeMask
- Returned device node mask
dev
- Device to get identifier string for

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Description
Return identifying information (luid and deviceNodeMask) to allow matching device with graphics APIs.

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

See also:
cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGetName, cuDeviceGet,
cuDeviceTotalMem, cudaGetDeviceProperties

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

Returns an identifier string for the device.

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

**Description**
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.

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

**See also:**
cuDeviceGetAttribute, cuDeviceGetUuid, cuDeviceGetLuid, cuDeviceGetCount, cuDeviceGet,
cuDeviceTotalMem, cudaGetDeviceProperties

**CUresult cuDeviceGetNvSciSyncAttributes (void *nvSciSyncAttrList, CUdevice dev, int flags)**
Return NvSciSync attributes that this device can support.

**Parameters**

- **nvSciSyncAttrList**
  - Return NvSciSync attributes supported.

- **dev**
  - Valid Cuda Device to get NvSciSync attributes for.

- **flags**
  - flags describing NvSciSync usage.

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

If NvSciSyncAttrKey_RequiredPerm field in `nvSciSyncAttrList` is already set this API will return CUDA_ERROR_INVALID_VALUE.
The applications should set `nvSciSyncAttrList` to a valid `NvSciSyncAttrList` failing which this API will return `CUDA_ERROR_INVALID_HANDLE`.

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

- `CUDA_NVSCISYNC_ATTR_SIGNAL`, specifies that the applications intends to signal an `NvSciSync` on this CUDA device.
- `CUDA_NVSCISYNC_ATTR_WAIT`, specifies that the applications intends to wait on an `NvSciSync` on this CUDA device.

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

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_INVALID_HANDLE`, `CUDA_ERROR_INVALID_DEVICE`, `CUDA_ERROR_NOT_SUPPORTED`, `CUDA_ERROR_OUT_OF_MEMORY`

**See also:**

`cuImportExternalSemaphore`, `cuDestroyExternalSemaphore`, `cuSignalExternalSemaphoresAsync`, `cuWaitExternalSemaphoresAsync`

```c
CUresult cuDeviceGetTexture1DLinearMaxWidth(size_t *maxWidthInElements, CUarray_format format, unsigned numChannels, CUdevice dev)
```

Returns the maximum number of elements allocatable in a 1D linear texture for a given texture element size.

**Parameters**

- `maxWidthInElements` - Returned maximum number of texture elements allocatable for given `format` and `numChannels`.
- `format` - Texture format.
- `numChannels` - Number of channels per texture element.
- `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

Description

Returns in `maxWidthInElements` the maximum number of texture elements allocatable in a 1D linear texture for given `format` and `numChannels`.

Note:

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

See also:

cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGetName, cuDeviceGetUuid, cuDeviceGet,
cudaMemGetInfo, cuDeviceTotalMem

CUresult cuDeviceGetUuid (CUuuid *uuid, CUdevice dev)

Return an UUID for the device.

Parameters

`uuid`
- Returned UUID

`dev`
- Device to get identifier string for

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Description

Returns 16-octets identifying the device `dev` in the structure pointed by the `uuid`.

Note:

Note that this function may also return error codes from previous, asynchronous launches.
CUresult cuDeviceTotalMem (size_t *bytes, CUdevice dev)

Returns the total amount of memory on the device.

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

Description

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

Note:

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

5.6. Device Management [DEPRECATED]

This section describes the device management functions of the low-level CUDA driver application programming interface.
CUresult cuDeviceComputeCapability (int *major, int *minor, CUdevice dev)
Returns the compute capability of the device.

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

Description

**Deprecated**

This function was deprecated as of CUDA 5.0 and its functionality superceded by
cuDeviceGetAttribute().

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

---

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

See also:

cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGetName, cuDeviceGetUuid, cuDeviceGet,
cuDeviceTotalMem
CUresult cuDeviceGetProperties (CUdevprop *prop, CUdevice dev)

Returns properties for a selected device.

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

Description

Deprecated

This function was deprecated as of CUDA 5.0 and replaced by cuDeviceGetAttribute().

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

```c
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.

<table>
<thead>
<tr>
<th>Note:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note that this function may also return error codes from previous, asynchronous launches.</td>
</tr>
</tbody>
</table>

See also:
cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGetName, cuDeviceGetUuid, cuDeviceGet, cuDeviceTotalMem

5.7. Primary Context Management

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

The primary context is unique per device and shared with the CUDA runtime API. These functions allow integration with other libraries using CUDA.

**CUresult cuDevicePrimaryCtxGetState (CUdevice dev, unsigned int *flags, int *active)**

Get the state of the primary context.

**Parameters**

- **dev**
  - Device to get primary context flags for

- **flags**
  - Pointer to store flags

- **active**
  - Pointer to store context state; 0 = inactive, 1 = active

**Returns**

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_DEVICE, CUDA_ERROR_INVALID_VALUE.

**Description**

Returns in *flags the flags for the primary context of dev, and in *active whether it is active. See cuDevicePrimaryCtxSetFlags for flag values.
CUresult cuDevicePrimaryCtxRelease (CUdevice dev)
Release the primary context on the GPU.

Parameters

dev
- Device which primary context is released

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_DEVICE, CUDA_ERROR_INVALID_CONTEXT

Description

Releases the primary context interop on the device. A retained context should always be released once the user is done using it. The context is automatically reset once the last reference to it is released. This behavior is different when the primary context was retained by the CUDA runtime from CUDA 4.0 and earlier. In this case, the primary context remains always active.

Releasing a primary context that has not been previously retained will fail with CUDA_ERROR_INVALID_CONTEXT.

Please note that unlike cuCtxDestroy, this method does not pop the context from stack in any circumstances.

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

See also:

cuDevicePrimaryCtxRetain, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig,
cuCtxGetDevice, cuCtxGetFlags, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent,
cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize
CUresult cuDevicePrimaryCtxReset (CUdevice dev)

Destroy all allocations and reset all state on the primary context.

**Parameters**

- **dev**
  - Device for which primary context is destroyed

**Returns**

CUDA SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_DEVICE, CUDA_ERROR_PRIMARY_CONTEXT_ACTIVE

**Description**

Explicitly destroys and cleans up all resources associated with the current device in the current process.

Note that it is responsibility of the calling function to ensure that no other module in the process is using the device any more. For that reason it is recommended to use cuDevicePrimaryCtxRelease() in most cases. However it is safe for other modules to call cuDevicePrimaryCtxRelease() even after resetting the device. Resetting the primary context does not release it, an application that has retained the primary context should explicitly release its usage.

**Note:**

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

**See also:**

cuDevicePrimaryCtxRetain, cuDevicePrimaryCtxRelease, cuCtxGetApiVersion,
cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetFlags, cuCtxGetLimit,
cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit,
cuCtxSynchronize, cudaDeviceReset

CUresult cuDevicePrimaryCtxRetain (CUcontext *pctx,
CUdevice dev)

Retain the primary context on the GPU.

**Parameters**

- **pctx**
  - Returned context handle of the new context
dev
- Device for which primary context is requested

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

Description
Retains the primary context on the device. Once the user successfully retains the primary context, the primary context will be active and available to the user until the user releases it with cuDevicePrimaryCtxRelease() or resets it with cuDevicePrimaryCtxReset(). Unlike cuCtxCreate() the newly retained context is not pushed onto the stack.

Retaining the primary context for the first time will fail with CUDA_ERROR_UNKNOWN if the compute mode of the device is CU_COMPUTEMODE_PROHIBITED. 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.

Please note that the primary context always supports pinned allocations. Other flags can be specified by cuDevicePrimaryCtxSetFlags().

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

See also:
cuDevicePrimaryCtxRelease, cuDevicePrimaryCtxSetFlags, cuCtxCreate, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetFlags, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

CUresult cuDevicePrimaryCtxSetFlags (CUdevice dev, unsigned int flags)
Set flags for the primary context.

Parameters
dev
- Device for which the primary context flags are set
flags

- New flags for the device

Retruns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_DEVICE, CUDA_ERROR_INVALID_VALUE,

Description

Sets the flags for the primary context on the device overwriting perviously set ones.

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_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_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 [CU_CTX_SCHED_YIELD], otherwise CUDA will not yield while waiting for results and actively spin on the processor [CU_CTX_SCHED_SPIN]. Additionally, on Tegra devices, **CU_CTX_SCHED_AUTO** uses a heuristic based on the power profile of the platform and may choose **CU_CTX_SCHED_BLOCKING_SYNC** for low-powered devices.

- **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.

**Deprecated**: This flag is deprecated and the behavior enabled by this flag is now the default and cannot be disabled.
5.8. Context Management

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

Please note that some functions are described in Primary Context Management section.

CUresult cuCtxCreate (CUcontext *pctx, unsigned int flags, CUdevice dev)

Create a CUDA context.

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

Description

Note:

In most cases it is recommended to use cuDevicePrimaryCtxRetain.

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_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_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 (**CU_CTX_SCHED_YIELD**), otherwise CUDA will not yield while waiting for results and actively spin on the processor (**CU_CTX_SCHED_SPIN**). Additionally, on Tegra devices, **CU_CTX_SCHED_AUTO** uses a heuristic based on the power profile of the platform and may choose **CU_CTX_SCHED_BLOCKING_SYNC** for low-powered devices.

- **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.

**Deprecated**: This flag is deprecated and the behavior enabled by this flag is now the default and cannot be disabled. Instead, the per-thread stack size can be controlled with cuCtxSetLimit().

Context creation will fail with CUDA_ERROR_UNKNOWN if the compute mode of the device is **CU_COMPUTEMODE_PROHIBITED**. The function **cuDeviceGetAttribute()** can be used with **CUDEVICE_ATTRIBUTE_COMPUTEMODE** to determine the compute mode of the device.
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.

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

See also:
cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetFlags, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

CUresult cuCtxDestroy (CUcontext ctx)

Destroy a CUDA context.

Parameters

ctx
- Context to destroy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

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 responsibility of the calling function to ensure that no API call issues using 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.

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

See also:
cuCtxCreate, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetFlags, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

CUresult cuCtxGetApiVersion (CUcontext ctx, unsigned int *version)
Gets the context’s API version.

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_INVALID_VALUE, CUDA_ERROR_UNKNOWN

Description

Returns a version number in version corresponding to the capabilities of the context (e.g. 3010 or 3020), which library developers can use to direct callers to a specific API version. If ctx is NULL, returns the API version used to create the currently bound context.

Note that new API versions are only introduced when context capabilities are changed that break binary compatibility, so the API version and driver version may be different. For example, it is valid for the API version to be 3020 while the driver version is 4020.

Note:

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetDevice, cuCtxGetFlags, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize
CUresult cuCtxGetCacheConfig (CUfunc_cache *pconfig)

Returns the preferred cache configuration for the current context.

Parameters

pconfig
  - Returned cache configuration

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

On devices where the L1 cache and shared memory use the same hardware resources, this function 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
- **CU_FUNC_CACHE_PREFER_EQUAL**: prefer equal sized L1 cache and shared memory

Note:

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetDevice, cuCtxGetFlags, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize, cuFuncSetCacheConfig, cudaDeviceGetCacheConfig
CUresult cuCtxGetCurrent (CUcontext *pctx)

Returns the CUDA context bound to the calling CPU thread.

**Parameters**

*pctx*
- Returned context handle

**Returns**

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`.

**Description**

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.

**Note:**

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

**See also:**

`cuCtxSetCurrent`, `cuCtxCreate`, `cuCtxDestroy`, `cudaGetDevice`

CUresult cuCtxGetDevice (CUdevice *device)

Returns the device ID for the current context.

**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`.

**Description**

Returns in *device* the ordinal of the current context’s device.

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.
See also:
cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetFlags, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize, cudaGetDevice

CUresult cuCtxGetFlags (unsigned int *flags)
Returns the flags for the current context.

Parameters
flags
- Pointer to store flags of current context

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE,

Description
Returns in *flags the flags of the current context. See cuCtxCreate for flag values.

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

See also:
cuCtxCreate, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetCurrent, cuCtxGetDevice, cuCtxGetLimit, cuCtxGetSharedMemConfig, cuCtxGetStreamPriorityRange, cudaGetDeviceFlags

CUresult cuCtxGetLimit (size_t *pvalue, CUlimit limit)
Returns resource limits.

Parameters
pvalue
- Returned size of limit
limit
- Limit to query

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_UNSUPPORTED_LIMIT
Description

Returns in *pvalue the current size of limit. The supported **CUlimit** values are:

- **CU_LIMIT_STACK_SIZE**: stack size in bytes of each GPU thread.
- **CU_LIMIT_PRINTF_FIFO_SIZE**: size in bytes of the FIFO used by the printf[] device system call.
- **CU_LIMIT_MALLOC_HEAP_SIZE**: size in bytes of the heap used by the malloc[] and free[] device system calls.
- **CU_LIMIT_DEV_RUNTIME_SYNC_DEPTH**: maximum grid depth at which a thread can issue the device runtime call **cudaDeviceSynchronize[]** to wait on child grid launches to complete.
- **CU_LIMIT_DEV_RUNTIME_PENDING_LAUNCH_COUNT**: maximum number of outstanding device runtime launches that can be made from this context.
- **CU_LIMIT_MAX_L2_FETCH_GRANULARITY**: L2 cache fetch granularity.

**Note:**

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, 
cuCtxGetFlags, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, 
cuCtxSynchronize, cudaDeviceGetLimit

**CUresult cuCtxGetSharedMemConfig**  
(CUsharedconfig *pConfig)

Returns the current shared memory configuration for the current context.

**Parameters**

**pConfig**

- returned shared memory configuration

**Returns**

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, 
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE
Description

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

The returned bank configurations can be either:

- `CU_SHARED_MEM_CONFIG_FOUR_BYTE_BANK_SIZE`: shared memory bank width is four bytes.
- `CU_SHARED_MEM_CONFIG_EIGHT_BYTE_BANK_SIZE`: shared memory bank width will eight bytes.

Note:

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

See also:

`cuCtxCreate`, `cuCtxDestroy`, `cuCtxGetApiVersion`, `cuCtxGetCacheConfig`, `cuCtxGetDevice`, `cuCtxGetFlags`, `cuCtxGetLimit`, `cuCtxPopCurrent`, `cuCtxPushCurrent`, `cuCtxSetLimit`, `cuCtxSynchronize`, `cuCtxGetSharedMemConfig`, `cuFuncSetCacheConfig`, `cudaDeviceGetSharedMemConfig`

`CUresult cuCtxGetStreamPriorityRange (int *leastPriority, int *greatestPriority)`

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

Parameters

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

Returns

`CUDA_SUCCESS`, `CUDA_ERROR_INVALID_VALUE`.

Description

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

This function will return '0' in both \(*\text{leastPriority}\) and \(*\text{greatestPriority}\) if the current context’s device does not support stream priorities (see \texttt{cuDeviceGetAttribute}\).

### Note:

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

### See also:

\texttt{cuStreamCreateWithPriority}, \texttt{cuStreamGetPriority}, \texttt{cuCtxGetDevice}, \texttt{cuCtxGetFlags}, \texttt{cuCtxSetLimit}, \texttt{cuCtxSynchronize}, \texttt{cudaDeviceGetStreamPriorityRange}

### 

**CUresult cuCtxPopCurrent (CUcontext *pctx)**

Pops the current CUDA context from the current CPU thread.

**Parameters**

**pctx**

- Returned new context handle

**Returns**

\texttt{CUDA\ SUCCESS}, \texttt{CUDA\ ERROR\ DEINITIALIZED}, \texttt{CUDA\ ERROR\ NOT\ INITIALIZED}, \texttt{CUDA\ ERROR\ INVALID\ CONTEXT}

**Description**

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

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

### Note:

Note that this function may also return error codes from previous, asynchronous launches.
See also:
cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetFlags, cuCtxGetLimit, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

CUresult cuCtxPushCurrent (CUcontext ctx)
Pushes a context on the current CPU thread.

Parameters
ctx
- Context to push

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description
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().

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

See also:
cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetFlags, cuCtxGetLimit, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

CUresult cuCtxResetPersistingL2Cache (void)
Resets all persisting lines in cache to normal status.

Returns
CUDA_SUCCESS, CUDA_ERROR_NOT_SUPPORTED
Description

cuCtxResetPersistingL2Cache Resets all persisting lines in cache to normal status. Takes effect on function return.

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

See also:
CUaccessPolicyWindow

CResult cuCtxSetCacheConfig (CUfunc_cache config)
Sets the preferred cache configuration for the current context.

Parameters

config
- Requested cache configuration

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

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
CU_FUNC_CACHE_PREFER_EQUAL: prefer equal sized L1 cache and shared memory

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

See also:
cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice,
cuCtxGetFlags, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetLimit,
cuCtxSynchronize, cuFuncSetCacheConfig, cudaDeviceSetCacheConfig

CUresult cuCtxSetCurrent (CUcontext ctx)
Binds the specified CUDA context to the calling CPU thread.

Parameters
ctx
- Context to bind to the calling CPU thread

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT

Description
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).

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

See also:
cuCtxGetCurrent, cuCtxCreate, cuCtxDestroy, cudaSetDevice
CUresult cuCtxSetLimit (CUlimit limit, size_t value)

Set resource limits.

Parameters

limit
- Limit to set

value
- Size of limit

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_UNSUPPORTED_LIMIT, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_INVALID_CONTEXT

Description

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 in bytes of each GPU thread. The driver automatically increases the per-thread stack size for each kernel launch as needed. This size isn’t reset back to the original value after each launch. Setting this value will take effect immediately, and if necessary, the device will block until all preceding requested tasks are complete.

- **CU_LIMIT_PRINTF_FIFO_SIZE** controls the size in bytes 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.

- **CU_LIMIT_MALLOC_HEAP_SIZE** controls the size in bytes 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.

- **CU_LIMIT_DEV_RUNTIME_SYNC_DEPTH** controls the maximum nesting depth of a grid at which a thread can safely call cudaDeviceSynchronize(). Setting this limit must be performed before any launch of a kernel that uses the device runtime and calls cudaDeviceSynchronize() above the default sync depth, two levels of grids. Calls to cudaDeviceSynchronize() will fail with error code cudaErrorSyncDepthExceeded if the limitation is violated. This limit can be set smaller than the default or up to the maximum launch depth of 24. When setting this limit, keep in mind that additional levels of sync
depth require the driver to reserve large amounts of device memory which can no longer be used for user allocations. If these reservations of device memory fail, \texttt{cuCtxSetLimit()} will return \texttt{CUDA\_ERROR\_OUT\_OF\_MEMORY}, and the limit can be reset to a lower value. This limit is only applicable to devices of compute capability 3.5 and higher. Attempting to set this limit on devices of compute capability less than 3.5 will result in the error \texttt{CUDA\_ERROR\_UNSUPPORTED\_LIMIT} being returned.

- \textbf{CU\_LIMIT\_DEV\_RUNTIME\_PENDING\_LAUNCH\_COUNT} controls the maximum number of outstanding device runtime launches that can be made from the current context. A grid is outstanding from the point of launch up until the grid is known to have been completed. Device runtime launches which violate this limitation fail and return \texttt{cudaErrorLaunchPendingCountExceeded} when \texttt{cudaGetLastError()} is called after launch. If more pending launches than the default (2048 launches) are needed for a module using the device runtime, this limit can be increased. Keep in mind that being able to sustain additional pending launches will require the driver to reserve larger amounts of device memory upfront which can no longer be used for allocations. If these reservations fail, \texttt{cuCtxSetLimit()} will return \texttt{CUDA\_ERROR\_OUT\_OF\_MEMORY}, and the limit can be reset to a lower value. This limit is only applicable to devices of compute capability 3.5 and higher. Attempting to set this limit on devices of compute capability less than 3.5 will result in the error \texttt{CUDA\_ERROR\_UNSUPPORTED\_LIMIT} being returned.

- \textbf{CU\_LIMIT\_MAX\_L2\_FETCH\_GRANULARITY} controls the L2 cache fetch granularity. Values can range from 0B to 128B. This is purely a performance hint and it can be ignored or clamped depending on the platform.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Note:} \\
Note that this function may also return error codes from previous, asynchronous launches. \\
\hline
\end{tabular}
\end{center}

\textbf{See also:}
\texttt{cuCtxCreate}, \texttt{cuCtxDestroy}, \texttt{cuCtxGetApiVersion}, \texttt{cuCtxGetCacheConfig}, \texttt{cuCtxGetDevice}, \texttt{cuCtxGetFlags}, \texttt{cuCtxGetLimit}, \texttt{cuCtxPopCurrent}, \texttt{cuCtxPushCurrent}, \texttt{cuCtxSetCacheConfig}, \texttt{cuCtxSynchronize}, \texttt{cudaDeviceSetLimit}

\textbf{CUresult cuCtxSetSharedMemConfig (CUsharedconfig\ config)}

Sets the shared memory configuration for the current context.

\textbf{Parameters}

\textbf{config}
- requested shared memory configuration
Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

On devices with configurable shared memory banks, this function will set the context’s shared memory bank size which is used for subsequent kernel launches.

Changed the shared memory configuration between launches may insert a device side synchronization point between those launches.

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

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

The supported bank configurations are:

- CU_SHARED_MEM_CONFIG_DEFAULT_BANK_SIZE: set bank width to the default initial setting (currently, four bytes).
- CU_SHARED_MEM_CONFIG_FOUR_BYTE_BANK_SIZE: set shared memory bank width to be natively four bytes.
- CU_SHARED_MEM_CONFIG_EIGHT_BYTE_BANK_SIZE: set shared memory bank width to be natively eight bytes.

Note:

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetFlags, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetLimit, cuCtxSynchronize, cuCtxGetSharedMemConfig, cuFuncSetCacheConfig, cudaDeviceSetSharedMemConfig
CUresult cuCtxSynchronize (void)
Block for a context’s tasks to complete.

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT

Description
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.

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

See also:
cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice,
cuCtxGetFlags, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig,
cuCtxSetLimit, cudaDeviceSynchronize

5.9. Context Management [DEPRECATED]
This section describes the deprecated context management functions of the low-level CUDA driver application programming interface.

CUresult cuCtxAttach (CUcontext *pctx, unsigned int flags)
Increment a context’s usage-count.

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

## Description
**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.

---

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

## See also:
- `cuCtxCreate`, `cuCtxDestroy`, `cuCtxDetach`, `cuCtxGetApiVersion`, `cuCtxGetCacheConfig`, `cuCtxGetDevice`, `cuCtxGetFlags`, `cuCtxGetLimit`, `cuCtxPopCurrent`, `cuCtxPushCurrent`, `cuCtxSetCacheConfig`, `cuCtxSetLimit`, `cuCtxSynchronize`

## CUresult cuCtxDetach (CUcontext ctx)
Decrement a context’s usage-count.

## Parameters
- **ctx**
  - Context to destroy

## Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

## Description
**Deprecated**

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

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

See also:
cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetFlags, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

5.10. Module Management

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

CUresult cuLinkAddData (CUlinkState state, CUjitInputType type, void *data, size_t size, const char *name, unsigned int numOptions, CUjit_option *options, void **optionValues)

Add an input to a pending linker invocation.

Parameters

state
   A pending linker action.

type
   The type of the input data.

data
   The input data. PTX must be NULL-terminated.

size
   The length of the input data.

name
   An optional name for this input in log messages.

numOptions
   Size of options.

options
   Options to be applied only for this input [overrides options from cuLinkCreate].

optionValues
   Array of option values, each cast to void *.
Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE,
CUDA_ERROR_INVALID_IMAGE, CUDA_ERROR_INVALID_PTX,
CUDA_ERROR_UNSUPPORTED_PTX_VERSION, CUDA_ERROR_OUT_OF_MEMORY,
CUDA_ERROR_NO_BINARY_FOR_GPU

Description
Ownership of data is retained by the caller. No reference is retained to any inputs after this call returns.

This method accepts only compiler options, which are used if the data must be compiled from PTX, and does not accept any of CU_JIT_WALL_TIME, CU_JIT_INFO_LOG_BUFFER,
CU_JIT_ERROR_LOG_BUFFER, CU_JIT_TARGET_FROM_CUCONTEXT, or CU_JIT_TARGET.

See also:
cuLinkCreate, cuLinkAddFile, cuLinkComplete, cuLinkDestroy

CUresult cuLinkAddFile (CUlinkState state,
CUjitInputType type, const char *path, unsigned
int numOptions, CUjit_option *options, void
**optionValues)
Add a file input to a pending linker invocation.

Parameters
state
A pending linker action
type
The type of the input data
path
Path to the input file
numOptions
Size of options
options
Options to be applied only for this input [overrides options from cuLinkCreate]
optionValues
Array of option values, each cast to void *
Returns
CUDA_SUCCESS, CUDA_ERROR_FILE_NOT_FOUND, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_IMAGE,
CUDA_ERROR_INVALID_PTX, CUDA_ERROR_UNSUPPORTED_PTX_VERSION,
CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_NO_BINARY_FOR_GPU

Description
No reference is retained to any inputs after this call returns.
This method accepts only compiler options, which are used if the input must be compiled
from PTX, and does not accept any of CU_JIT_WALL_TIME, CU_JIT_INFO_LOG_BUFFER,
CU_JIT_ERROR_LOG_BUFFER, CU_JIT_TARGET_FROM_CU_CONTEXT, or CU_JIT_TARGET.
This method is equivalent to invoking cuLinkAddData on the contents of the file.

See also:
cuLinkCreate, cuLinkAddData, cuLinkComplete, cuLinkDestroy

CUresult cuLinkComplete (CUlinkState state, void **cubinOut, size_t *sizeOut)
Complete a pending linker invocation.

Parameters
state
   A pending linker invocation
cubinOut
   On success, this will point to the output image
sizeOut
   Optional parameter to receive the size of the generated image

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY

Description
Completes the pending linker action and returns the cubin image for the linked device code,
which can be used with cuModuleLoadData. The cubin is owned by state, so it should be
loaded before state is destroyed via cuLinkDestroy. This call does not destroy state.

See also:
cuLinkCreate, cuLinkAddData, cuLinkAddFile, cuLinkDestroy, cuModuleLoadData
CUresult cuLinkCreate (unsigned int numOptions,
CUjit_option *options, void **optionValues,
CUlinkState *stateOut)

Creates a pending JIT linker invocation.

Parameters

numOptions
   Size of options arrays

options
   Array of linker and compiler options

optionValues
   Array of option values, each cast to void *

stateOut
   On success, this will contain a CUlinkState to specify and complete this action

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_JIT_COMPILER_NOT_FOUND

Description

If the call is successful, the caller owns the returned CUlinkState, which should eventually
be destroyed with cuLinkDestroy. The device code machine size (32 or 64 bit) will match the
calling application.

Both linker and compiler options may be specified. Compiler options will be applied to inputs
to this linker action which must be compiled from PTX. The options CU_JIT_WALL_TIME,
CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES, and CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES
will accumulate data until the CUlinkState is destroyed.

optionValues must remain valid for the life of the CUlinkState if output options are used.
No other references to inputs are maintained after this call returns.

Note:

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

See also:

cuLinkAddData, cuLinkAddFile, cuLinkComplete, cuLinkDestroy
CUresult cuLinkDestroy (CUlinkState state)
Destroys state for a JIT linker invocation.

Parameters
state
  State object for the linker invocation

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE

Description

See also:
cuLinkCreate

CUresult cuModuleGetFunction (CUfunction *hfunc, 
CUmodule hmod, const char *name)
Returns a function handle.

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

Description

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.

Note:
CUresult cuModuleGetGlobal (CUdeviceptr *dptr, size_t *bytes, CUmodule hmod, const char *name)

Returns a global pointer from a module.

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

Description

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.

Note:

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

See also:

cuParamGetFunction, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadData,
cuParamLoadDataEx, cuModuleLoadFatBinary, cuModuleUnload, cudaGetSymbolAddress,
cudaGetSymbolSize
CUresult cuModuleGetSurfRef (CUsurfref *pSurfRef, CUmodule hmod, const char *name)

Returns a handle to a surface reference.

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

Description

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.

Note:

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

See also:
cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoad,
cuModuleLoadData, cuModuleLoadDataEx, cuModuleLoadFatBinary, cuModuleUnload,
cudaGetSurfaceReference

CUresult cuModuleGetTexRef (CUtexref *pTexRef, CUmodule hmod, const char *name)

Returns a handle to a texture reference.

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

**Description**
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.

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

**See also:**
cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetSurfRef, cuModuleLoad, cuModuleLoadData, cuModuleLoadDataEx, cuModuleLoadFatBinary, cuModuleUnload, cudaGetTextureReference

**CUresult cuModuleLoad (CUmodule *module, const char *fname)**
Loads a compute module.

**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_INVALID_PTX, CUDA_ERROR_UNSUPPORTED_PTX_VERSION, CUDA_ERROR_NOT_FOUND, CUDA_ERROR_OUT_OF_MEMORY,
CUDA_ERROR_FILE_NOT_FOUND, CUDA_ERROR_NO_BINARY_FOR_GPU, CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED, CUDA_ERROR_JIT_COMPILER_NOT_FOUND

Description

Takes a filename \texttt{fname} and loads the corresponding module \texttt{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, \texttt{cuModuleLoad()} fails. The file should be a cubin file as output by \texttt{nvcc}, or a PTX file either as output by \texttt{nvcc} or handwritten, or a fatbin file as output by \texttt{nvcc} from toolchain 4.0 or later.

Note:

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

See also:

\texttt{cuModuleGetFunction}, \texttt{cuModuleGetGlobal}, \texttt{cuModuleGetTexRef}, \texttt{cuModuleLoadData}, \texttt{cuModuleLoadDataEx}, \texttt{cuModuleLoadFatBinary}, \texttt{cuModuleUnload}

\textbf{CUresult cuModuleLoadData (CUmodule *module, const void *image)}

Load a module’s data.

Parameters

\begin{itemize}
  \item \texttt{module} - Returned module
  \item \texttt{image} - Module data to load
\end{itemize}

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_PTX, CUDA_ERROR_UNSUPPORTED_PTX_VERSION, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_NO_BINARY_FOR_GPU, CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED, CUDA_ERROR_JIT_COMPILER_NOT_FOUND
Description

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.

Note:

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

See also:

- `cuModuleGetFunction`
- `cuModuleGetGlobal`
- `cuModuleGetTexRef`
- `cuModuleLoad`
- `cuModuleLoadDataEx`
- `cuModuleLoadFatBinary`
- `cuModuleUnload`

### CUresult cuModuleLoadDataEx (CUmodule *module, const void *image, unsigned int numOptions, CUjit_option *options, void **optionValues)

Load a module’s data with options.

#### 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_INVALID_PTX`
- `CUDA_ERROR_UNSUPPORTED_PTX_VERSION`
- `CUDA_ERROR_OUT_OF_MEMORY`
- `CUDA_ERROR_NO_BINARY_FOR_GPU`
- `CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND`
- `CUDA_ERROR_SHARED_OBJECT_INIT_FAILED`
- `CUDA_ERROR_JIT_COMPILER_NOT_FOUND`
Description

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`.

Note:

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

See also:

`cuModuleGetFunction`, `cuModuleGetGlobal`, `cuModuleGetTexRef`, `cuModuleLoad`, `cuModuleLoadData`, `cuModuleLoadFatBinary`, `cuModuleUnload`

`CUresult cuModuleLoadFatBinary (CUmodule *module, const void *fatCubin)`

Load a module’s data.

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_INVALID_PTX`, `CUDA_ERROR_UNSUPPORTED_PTX_VERSION`, `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`, `CUDA_ERROR_JIT_COMPILER_NOT_FOUND`

Description

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.

Note:

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

See also:

`cuModuleGetFunction`, `cuModuleGetGlobal`, `cuModuleGetTexRef`, `cuModuleLoad`, `cuModuleLoadData`, `cuModuleLoadDataEx`, `cuModuleUnload`

**CUresult cuModuleUnload (CUmodule hmod)**

Unloads a module.

**Parameters**

`hmod`
- Module to unload

**Returns**

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

**Description**

Unloads a module `hmod` from the current context.

Note:

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

See also:

`cuModuleGetFunction`, `cuModuleGetGlobal`, `cuModuleGetTexRef`, `cuModuleLoad`, `cuModuleLoadData`, `cuModuleLoadDataEx`, `cuModuleLoadFatBinary`
5.11. Memory Management

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

CUresult cuArray3DCreate (CUarray *pHandle, const CUDA_ARRAY3D_DESCRIPTOR *pAllocateArray)

Creates a 3D CUDA array.

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

Description

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:

```c
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 following types of CUDA arrays can be allocated:
  - A 1D array is allocated if Height and Depth extents are both zero.
  - A 2D array is allocated if only Depth extent is zero.
  - A 3D array is allocated if all three extents are non-zero.
A 1D layered CUDA array is allocated if only **Height** is zero and the **CUDA_ARRAY3D_LAYERED** flag is set. Each layer is a 1D array. The number of layers is determined by the depth extent.

A 2D layered CUDA array is allocated if all three extents are non-zero and the **CUDA_ARRAY3D_LAYERED** flag is set. Each layer is a 2D array. The number of layers is determined by the depth extent.

A cubemap CUDA array is allocated if all three extents are non-zero and the **CUDA_ARRAY3D_CUBEMAP** flag is set. **Width** must be equal to **Height**, and **Depth** must be six. A cubemap is a special type of 2D layered CUDA array, where the six layers represent the six faces of a cube. The order of the six layers in memory is the same as that listed in **CUarray_cubemap_face**.

A cubemap layered CUDA array is allocated if all three extents are non-zero, and both, **CUDA_ARRAY3D_CUBEMAP** and **CUDA_ARRAY3D_LAYERED** flags are set. **Width** must be equal to **Height**, and **Depth** must be a multiple of six. A cubemap layered CUDA array is a special type of 2D layered CUDA array that consists of a collection of cubemaps. The first six layers represent the first cubemap, the next six layers form the second cubemap, and so on.

**Format** specifies the format of the elements; **CUarray_format** is defined as:

```c
typedef enum CUarray_format_enum {
    CU_AD_FORMAT_UNSIGNED_INT8       = 0x01,
    CU_AD_FORMAT_UNSIGNED_INT16      = 0x02,
    CU_AD_FORMAT_UNSIGNED_INT32      = 0x03,
    CU_AD_FORMAT_SIGNED_INT8        = 0x08,
    CU_AD_FORMAT_SIGNED_INT16       = 0x09,
    CU_AD_FORMAT_SIGNED_INT32       = 0x0a,
    CU_AD_FORMAT_HALF               = 0x10,
    CU_AD_FORMAT_FLOAT              = 0x20
} CUarray_format;
```

**NumChannels** specifies the number of packed components per CUDA array element; it may be 1, 2, or 4;

Flags may be set to

- **CUDA_ARRAY3D_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.
- **CUDA_ARRAY3D_CUBEMAP** to enable creation of cubemaps. If this flag is set, **Width** must be equal to **Height**, and **Depth** must be six. If the **CUDA_ARRAY3D_LAYERED** flag is also set, then **Depth** must be a multiple of six.
- **CUDA_ARRAY3D_TEXTURE_GATHER** to indicate that the CUDA array will be used for texture gather. Texture gather can only be performed on 2D CUDA arrays.

**Width**, **Height** and **Depth** must meet certain size requirements as listed in the following table. All values are specified in elements. Note that for brevity’s sake, the full name of the
device attribute is not specified. For ex., TEXTURE1D_WIDTH refers to the device attribute 
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH.

Note that 2D CUDA arrays have different size requirements if the 
CUDA_ARRAY3D_TEXTURE_GATHER flag is set. Width and Height must not be 
greater than CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_GATHER_WIDTH and 
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_GATHER_HEIGHT respectively, in that case.

<table>
<thead>
<tr>
<th>CUDA array type</th>
<th>Valid extents that must always be met {(width range in elements), (height range), (depth range)}</th>
<th>Valid extents with CUDA_ARRAY3D_SURFACE_LDST set {(width range in elements), (height range), (depth range)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D</td>
<td>{1,TEXTURE1D_WIDTH}, 0, 0</td>
<td>{1,SURFACE1D_WIDTH}, 0, 0</td>
</tr>
<tr>
<td>2D</td>
<td>{1,TEXTURE2D_WIDTH}, 1,TEXTURE2D_HEIGHT}, 0</td>
<td>{1,SURFACE2D_WIDTH}, 1,TEXTURE2D_HEIGHT}, 0</td>
</tr>
<tr>
<td>3D</td>
<td>{1,TEXTURE3D_WIDTH}, 1,TEXTURE3D_HEIGHT}, 1,TEXTURE3D_DEPTH) OR {1,TEXTURE3D_WIDTH_ALTERNATE}, {1,SURFACE3D_WIDTH}, 1,TEXTURE3D_DEPTH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1D Layered</td>
<td>{1,TEXTURE1D_LAYERED_WIDTH},1,TEXTURE1D_LAYERED_WIDTH}, 0, 0, 0, 1,TEXTURE1D_LAYERED_LAYERS}</td>
<td>{1,SURFACE1D_LAYERED_WIDTH}, 0, 0, 0, 1,TEXTURE1D_LAYERED_LAYERS}</td>
</tr>
<tr>
<td>2D Layered</td>
<td>{1,TEXTURE2D_LAYERED_WIDTH},1,TEXTURE2D_LAYERED_WIDTH}, 1,TEXTURE2D_LAYERED_HEIGHT}, 1,TEXTURE2D_LAYERED_HEIGHT}, 1,TEXTURE2D_LAYERED_LAYERS}</td>
<td>{1,SURFACE2D_LAYERED_WIDTH}, 1,TEXTURE2D_LAYERED_HEIGHT}, 1,TEXTURE2D_LAYERED_HEIGHT}, 1,TEXTURE2D_LAYERED_LAYERS}</td>
</tr>
<tr>
<td>Cubemap</td>
<td>{1,TEXTURECUBEMAP_WIDTH}, {1,SURFACECUBEMAP_WIDTH}, 1,TEXTURECUBEMAP_WIDTH}, 6)</td>
<td>{1,SURFACECUBEMAP_WIDTH}, 1,TEXTURECUBEMAP_WIDTH}, 6)</td>
</tr>
<tr>
<td>Cubemap Layered</td>
<td>{1,TEXTURECUBEMAP_LAYERED_WIDTH}, {1,SURFACECUBEMAP_LAYERED_WIDTH}, 1,TEXTURECUBEMAP_LAYERED_WIDTH}, 1,TEXTURECUBEMAP_LAYERED_LAYERS}</td>
<td>{1,SURFACECUBEMAP_LAYERED_WIDTH}, 1,TEXTURECUBEMAP_LAYERED_WIDTH}, 1,TEXTURECUBEMAP_LAYERED_LAYERS}</td>
</tr>
</tbody>
</table>

Here are examples of CUDA array descriptions:

Description for a CUDA array of 2048 floats:

```c
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:

```c
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:

```c
CUDA_ARRAY3D_DESCRIPTOR desc;
desc.FormatFlags = CU_AD_FORMAT_HALF;
desc.NumChannels = 4;
desc.Width = width;
desc.Height = height;
desc.Depth = depth;
```

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
- cuMemFree
- cuMemFreeHost
- cuMemGetAddressRange
- cuMemGetInfo
- cuMemHostAlloc
- cuMemHostGetDevicePointer
- cuMemsetD2D8
- cuMemsetD2D16
- cuMemsetD2D32
- cuMemsetD8
- cuMemsetD16
- cuMemsetD32
- cudaMalloc3DArray

CUresult cuArray3DGetDescriptor
(CUDA_ARRAY3D_DESCRIPTOR *pArrayDescriptor, CUarray hArray)
Get a 3D CUDA array descriptor.

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, CUDA_ERROR_CONTEXT_IS_DESTROYED

**Description**

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.

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, cudaArrayGetInfo

**CUresult cuArrayCreate (CUarray *pHandle, const CUDA_ARRAY_DESCRIPTOR *pAllocateArray)**

Creates a 1D or 2D CUDA array.

**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

Description

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:

```c
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:

```c
typedef enum CUarray_format_enum {
    CU_AD_FORMAT_UNSIGNED_INT8 = 0x01,
    CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
    CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
    CU_AD_FORMAT_SIGNED_INT8 = 0x08,
    CU_AD_FORMAT_SIGNED_INT16 = 0x09,
    CU_AD_FORMAT_SIGNED_INT32 = 0x0a,
    CU_AD_FORMAT_HALF = 0x10,
    CU_AD_FORMAT_FLOAT = 0x20
} CUarray_format;
```

- **NumChannels** specifies the number of packed components per CUDA array element; it may be 1, 2, or 4;

Here are examples of CUDA array descriptions:

**Description for a CUDA array of 2048 floats:**
```c
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:**
```c
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:**
```c
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;
```

<table>
<thead>
<tr>
<th>Note:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note that this function may also return error codes from previous, asynchronous launches.</td>
</tr>
</tbody>
</table>

See also:

- cuArray3DCreate
- cuArray3DGetDescriptor
- cuArrayDestroy
- cuArrayGetDescriptor
- cuMemAlloc
- cuMemAllocHost
- cuMemAllocPitch
- cuMemcpy2D
- cuMemcpy2DAsync
- cuMemcpy2DUnaligned
- cuMemcpy3D
- cuMemcpy3DAsync
- cuMemcpyAtoA
- cuMemcpyAtoD
- cuMemcpyAtoH
- cuMemcpyAtoHAsync
- cuMemcpyDtoA
- cuMemcpyDtoD
- cuMemcpyDtoAAsync
- cuMemcpyDtoDataFrame
- cuMemcpyDtoH
- cuMemcpyDtoHAsync
- cuMemcpyHtoA
- cuMemcpyHtoAAsync
- cuMemcpyHtoD
- cuMemcpyHtoDAsync
- cuMemGetAddressRange
- cuMemGetInfo
- cuMemHostAlloc
- cuMemHostGetDevicePointer
- cuMemsetD2D8
- cuMemsetD2D16
- cuMemsetD2D32
- cuMemsetD8
- cuMemsetD16
- cuMemsetD32
- cudaMallocArray

**CUresult cuArrayDestroy (CUarray hArray)**

Destroys a CUDA array.

**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
- CUDA_ERROR_CONTEXT_IS_DESTROYED

**Description**

Destroys the CUDA array hArray.

<table>
<thead>
<tr>
<th>Note:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note that this function may also return error codes from previous, asynchronous launches.</td>
</tr>
</tbody>
</table>
See also:


**CUresult cuArrayGetDescriptor**

`(CUDA_ARRAY_DESCRIPTOR *pArrayDescriptor, CUarray hArray)`

Get a 1D or 2D CUDA array descriptor.

**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`

**Description**

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.

**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`

CUresult cuArrayGetSparseProperties (CUDA_ARRAY_SPARSE_PROPERTIES *sparseProperties, CUarray array)

Returns the layout properties of a sparse CUDA array.

Parameters

sparseProperties
- Pointer to CUDA_ARRAY_SPARSE_PROPERTIES

array
- CUDA array to get the sparse properties of

Returns

CUDA_SUCCESS CUDA_ERROR_INVALID_VALUE

Description

Returns the layout properties of a sparse CUDA array in sparseProperties. If the CUDA array is not allocated with flag CUDA_ARRAY3D_SPARSE_CUDA_ERROR_INVALID_VALUE will be returned.

If the returned value in CUDA_ARRAY_SPARSE_PROPERTIES::flags contains CU_ARRAY_SPARSE_PROPERTIES_SINGLE_MIPTAIL, then CUDA_ARRAY_SPARSE_PROPERTIES::miptailSize represents the total size of the array. Otherwise, it will be zero. Also, the returned value in CUDA_ARRAY_SPARSE_PROPERTIES::miptailFirstLevel is always zero. Note that the array must have been allocated using cuArrayCreate or cuArray3DCreate. For CUDA arrays obtained using cuMipmappedArrayGetLevel, CUDA_ERROR_INVALID_VALUE will be returned. Instead, cuMipmappedArrayGetSparseProperties must be used to obtain the sparse properties of the entire CUDA mipmapmed array to which array belongs to.

See also:

cuMipmappedArrayGetSparseProperties, cuMemMapArrayAsync
CUresult cuDeviceGetByPCIBusId (CUdevice *dev, const char *pciBusId)
Returns a handle to a compute device.

Parameters

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

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Description

Returns in *device a device handle given a PCI bus ID string.

Note:

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

See also:
cuDeviceGet, cuDeviceGetAttribute, cuDeviceGetPCIBusId, cudaDeviceGetByPCIBusId

CUresult cuDeviceGetPCIBusId (char *pciBusId, int len, CUdevice dev)
Returns a PCI Bus Id string for the device.

Parameters

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

- len
  - Maximum length of string to store in name
**dev**
- Device to get identifier string for

**Returns**
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

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

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

**See also:**
cuParamGet, cuDeviceGetAttribute, cuDeviceGetByPCIBusId, cudaDeviceGetPCIBusId

**CUresult cuIpcCloseMemHandle (CUdeviceptr dptr)**
Attempts to close memory mapped with cuIpcOpenMemHandle.

**Parameters**
**dptr**
- Device pointer returned by cuIpcOpenMemHandle

**Returns**
CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_MAP_FAILED, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE

**Description**
Decrement the reference count of the memory returned by cuIpcOpenMemHandle by 1. When the reference count reaches 0, this API unmaps the memory. The original allocation in the exporting process as well as imported mappings in other processes will be unaffected.

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

IPC functionality is restricted to devices with support for unified addressing on Linux and Windows operating systems. IPC functionality on Windows is restricted to GPUs in TCC mode.

**See also:**
CUresult cuIpcGetEventHandle (CUipcEventHandle *pHandle, CUevent event)

Gets an interprocess handle for a previously allocated event.

Parameters

pHandle
- Pointer to a user allocated CUipcEventHandle in which to return the opaque event handle

event
- Event allocated with CU_EVENT_INTERPROCESS and CU_EVENT_DISABLE_TIMING flags.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_MAP_FAILED, CUDA_ERROR_INVALID_VALUE

Description

Takes as input a previously allocated event. This event must have been created with the
CU_EVENT_INTERPROCESS and CU_EVENT_DISABLE_TIMING flags set. This opaque handle
may be copied into other processes and opened with cuIpcOpenEventHandle to allow efficient
hardware synchronization between GPU work in different processes.

After the event has been opened in the importing process, cuEventRecord,
cuEventSynchronize, cuStreamWaitEvent and cuEventQuery may be used in either process.
Performing operations on the imported event after the exported event has been freed with
cuEventDestroy will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux and
Windows operating systems. IPC functionality on Windows is restricted to GPUs in TCC mode

See also:

cuEventCreate, cuEventDestroy, cuEventSynchronize, cuEventQuery, cuStreamWaitEvent,
cuIpcOpenEventHandle, cuIpcGetMemHandle, cuIpcOpenMemHandle, cuIpcCloseMemHandle,
cuIpcGetEventHandle
CUresult cuIpcGetMemHandle (CUipcMemHandle *pHandle, CUdeviceptr dptr)
Gets an interprocess memory handle for an existing device memory allocation.

Parameters
pHandle
- Pointer to user allocated CUipcMemHandle to return the handle in.

dptr
- Base pointer to previously allocated device memory

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_MAP_FAILED,
CUDA_ERROR_INVALID_VALUE

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

If a region of memory is freed with cuMemFree and a subsequent call to cuMemAlloc returns
memory with the same device address, cuIpcGetMemHandle will return a unique handle for
the new memory.

IPC functionality is restricted to devices with support for unified addressing on Linux and
Windows operating systems. IPC functionality on Windows is restricted to GPUs in TCC mode

See also:
cuMemAlloc, cuMemFree, cuIpcGetEventHandle, cuIpcOpenEventHandle,
cuIpcOpenMemHandle, cuIpcCloseMemHandle, cudaIpcGetMemHandle

CUresult cuIpcOpenEventHandle (CUevent *phEvent, CUipcEventHandle handle)
Opens an interprocess event handle for use in the current process.

Parameters
phEvent
- Returns the imported event
handle
- Interprocess handle to open
Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_MAP_FAILED,
CUDA_ERROR_PEER_ACCESS_UNSUPPORTED, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_INVALID_VALUE

Description
Opens an interprocess event handle exported from another process with
`cuIpcGetEventHandle`. This function returns a CUevent that behaves like a locally created
event with the `CU_EVENT_DISABLE_TIMING` flag specified. This event must be freed with
cuEventDestroy.

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

IPC functionality is restricted to devices with support for unified addressing on Linux and
Windows operating systems. IPC functionality on Windows is restricted to GPUs in TCC mode

See also:
`cuEventCreate`, `cuEventDestroy`, `cuEventSynchronize`, `cuEventQuery`, `cuStreamWaitEvent`,
`cuIpcGetEventHandle`, `cuIpcGetMemHandle`, `cuIpcOpenMemHandle`, `cuIpcCloseMemHandle`,
cudaIpcOpenEventHandle

**CUresult cuIpcOpenMemHandle (CUdeviceptr *pdptr,**
CUipcMemHandle handle, unsigned int Flags)**

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

**Parameters**

**pdptr**
- Returned device pointer

**handle**
- CUipcMemHandle to open

**Flags**
- Flags for this operation. Must be specified as
  - `CU_IPC_MEM_LAZY_ENABLE_PEER_ACCESS`

**Returns**
CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_MAP_FAILED,
CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_TOO_MANY_PEERS,
CUDA_ERROR_INVALID_VALUE
Description
Maps memory exported from another process with `culpcGetMemHandle` into the current
device address space. For contexts on different devices `culpcOpenMemHandle` can attempt
to enable peer access between the devices as if the user called `cuCtxEnablePeerAccess`.
This behavior is controlled by the `CU_IPC_MEM.LAZY_ENABLE_PEER_ACCESS` flag.
`cuDeviceCanAccessPeer` can determine if a mapping is possible.

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

If the memory handle has already been opened by the current context, the reference count on
the handle is incremented by 1 and the existing device pointer is returned.

Memory returned from `culpcOpenMemHandle` must be freed with `culpcCloseMemHandle`.

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

IPC functionality is restricted to devices with support for unified addressing on Linux and
Windows operating systems. IPC functionality on Windows is restricted to GPUs in TCC mode.

Note:
No guarantees are made about the address returned in *dptr. In particular, multiple
processes may not receive the same address for the same handle.

See also:
`cuMemAlloc`, `cuMemFree`, `culpcGetEventHandle`, `culpcOpenEventHandle`,
`culpcGetMemHandle`, `culpcCloseMemHandle`, `cuCtxEnablePeerAccess`,
`cuDeviceCanAccessPeer`, `cudalpcOpenMemHandle`

`CUresult cuMemAlloc (CUdeviceptr *dptr, size_t bytesize)`
Allocates device memory.

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

Description
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.

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, cuMemcpyToA,
cuMemcpyToD, cuMemcpyToH, cuMemcpyToHasync, cuMemcpyDtoA, cuMemcpyDtoD,
cuMemcpyDtoHasync, cuMemcpyDtoH, cuMemcpyDtoHasync, cuMemcpyHtoA,
cuMemcpyHtoAasync, cuMemcpyHtoD, cuMemcpyHtoDasync, cuMemFree, cuMemFreeHost,
cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer,
cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16,
cuMemsetD32, cudaMalloc

CUresult cuMemAllocHost (void **pp, size_t bytesize)
Allocates page-locked host memory.

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
Description

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 `cuMemcpyHostToDevice`. 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.

Note:

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

See also:

- `cuArray3DCreate`, `cuArray3DGetDescriptor`, `cuArrayCreate`, `cuArrayDestroy`,
- `cuArrayGetDescriptor`, `cuMemAlloc`, `cuMemAllocPitch`, `cuMemcpy2D`, `cuMemcpy2DAsync`,
- `cuMemcpy2DUnaligned`, `cuMemcpy3D`, `cuMemcpy3DAsync`, `cuMemcpyAtoA`,
- `cuMemcpyAtoD`, `cuMemcpyAtoH`, `cuMemcpyAtoHAsync`, `cuMemcpyDtoA`, `cuMemcpyDtoD`,
- `cuMemcpyDtoDAsync`, `cuMemcpyDtoH`, `cuMemcpyDtoHAsync`, `cuMemcpyHtoA`,
- `cuMemcpyHtoAAsync`, `cuMemcpyHtoD`, `cuMemcpyHtoDAsync`, `cuMemFree`, `cuMemFreeHost`,
- `cuMemGetAddressRange`, `cuMemGetInfo`, `cuMemHostAlloc`, `cuMemHostGetDevicePointer`,
- `cuMemsetD2D8`, `cuMemsetD2D16`, `cuMemsetD2D32`, `cuMemsetD8`, `cuMemsetD16`,
- `cuMemsetD32`, `cudaMallocHost`

`CUresult cuMemAllocManaged (CUdeviceptr *dptr, size_t bytesize, unsigned int flags)`

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

Parameters

dptr
- Returned device pointer

bytesize
- Requested allocation size in bytes
flags
- Must be one of CU_MEM_ATTACH_GLOBAL or CU_MEM_ATTACH_HOST

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_NOT_SUPPORTED,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Description
Allocates bytesize bytes of managed memory on the device and returns in *dptr a
pointer to the allocated memory. If the device doesn’t support allocating managed memory,
CUDA_ERROR_NOT_SUPPORTED is returned. Support for managed memory can be queried
using the device attribute CU_DEVICE_ATTRIBUTE_MANAGED_MEMORY. The allocated
memory is suitably aligned for any kind of variable. The memory is not cleared. If bytesize
is 0, cuMemAllocManaged returns CUDA_ERROR_INVALID_VALUE. The pointer is valid on
the CPU and on all GPUs in the system that support managed memory. All accesses to this
pointer must obey the Unified Memory programming model.

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

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

Memory allocated with cuMemAllocManaged should be released with cuMemFree.

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

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

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

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

On Linux, the location chosen will be device memory as long as the current set of active contexts are on devices that either have peer-to-peer support with each other or have a non-zero value for the device attribute CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS. If there is an active context on a GPU that does not have a non-zero value for that device attribute and it does not have peer-to-peer support with the other devices that have active contexts on them, then the location for physical storage will be ‘zero-copy’ or host memory. Note that this means that managed memory that is located in device memory is migrated to host memory if a new context is created on a GPU that doesn’t have a non-zero value for the device attribute and does not support peer-to-peer with at least one of the other devices that has an active context. This in turn implies that context creation may fail if there is insufficient host memory to migrate all managed allocations.

On Windows, the physical storage is always created in ‘zero-copy’ or host memory. All GPUs will reference the data at reduced bandwidth over the PCIe bus. In these circumstances, use of the environment variable CUDA_VISIBLE_DEVICES is recommended to restrict CUDA to only use those GPUs that have peer-to-peer support. Alternatively, users can also set CUDA_MANAGED_FORCE_DEVICE_ALLOC to a non-zero value to force the driver to always use device memory for physical storage. When this environment variable is set to a non-zero value, all contexts created in that process on devices that support managed memory have to be peer-to-peer compatible with each other. Context creation will fail if a context is created on a device that supports managed memory and is not peer-to-peer compatible with any of the other managed memory supporting devices on which contexts were previously created, even if those contexts have been destroyed. These environment variables are described in the CUDA programming guide under the “CUDA environment variables” section.

On ARM, managed memory is not available on discrete gpu with Drive PX-2.
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy,
cuArrayGetDescriptor, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync,
cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA,
cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHASync, cuMemcpyDtoA, cuMemcpyDtoD,
cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHASync, cuMemcpyHtoA,
cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost,
cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer,
cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16,
cuMemsetD32, cuDeviceGetAttribute, cuStreamAttachMemAsync, cudaMallocManaged

CUresult cuMemAllocPitch (CUdeviceptr *dptr, size_t *pPitch, size_t WidthInBytes, size_t Height, unsigned int ElementSizeBytes)
Allocates pitched device memory.

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

Description
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. \texttt{ElementSizeBytes} may be 4, 8 or 16 (since coalesced memory transactions are not possible on other data sizes). If \texttt{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 \texttt{*pPitch} by \texttt{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 \texttt{T}, the address is computed as:

\[
T* \texttt{pElement} = (T*)((\texttt{char}*)\texttt{BaseAddress} + \texttt{Row} * \texttt{Pitch}) + \texttt{Column};
\]

The pitch returned by \texttt{cuMemAllocPitch()} is guaranteed to work with \texttt{cuMemcpy2D()} under all circumstances. For allocations of 2D arrays, it is recommended that programmers consider performing pitch allocations using \texttt{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 \texttt{cuMemAllocPitch()} is guaranteed to match or exceed the alignment requirement for texture binding with \texttt{cuTexRefSetAddress2D()}.

\textbf{Note:}

\texttt{Note that this function may also return error codes from previous, asynchronous launches.}

\textbf{See also:}

- \texttt{cuArray3DCreate}, \texttt{cuArray3DGetDescriptor}, \texttt{cuArrayCreate}, \texttt{cuArrayDestroy}, \texttt{cuArrayGetDescriptor}, \texttt{cuMemAlloc}, \texttt{cuMemAllocHost}, \texttt{cuMemcpy2D}, \texttt{cuMemcpy2DAsync}, \texttt{cuMemcpy3D}, \texttt{cuMemcpy3DAsync}, \texttt{cuMemcpyAtoA}, \texttt{cuMemcpyAtoD}, \texttt{cuMemcpyAtoH}, \texttt{cuMemcpyAtoHAsync}, \texttt{cuMemcpyDtoA}, \texttt{cuMemcpyDtoD}, \texttt{cuMemcpyDtoDAsync}, \texttt{cuMemcpyDtoH}, \texttt{cuMemcpyDtoHAsync}, \texttt{cuMemcpyHtoA}, \texttt{cuMemcpyHtoD}, \texttt{cuMemcpyHtoDAsync}, \texttt{cuMemFree}, \texttt{cuMemFreeHost}, \texttt{cuMemGetAddressRange}, \texttt{cuMemGetInfo}, \texttt{cuMemHostAlloc}, \texttt{cuMemHostGetDevicePointer}, \texttt{cuMemsetD2D8}, \texttt{cuMemsetD2D16}, \texttt{cuMemsetD2D32}, \texttt{cuMemsetD8}, \texttt{cuMemsetD16}, \texttt{cuMemsetD32}, \texttt{cudaMallocPitch}

\textbf{CUresult cuMemcpy (CUdeviceptr dst, CUdeviceptr src, size_t ByteCount)}

Copies memory.

\textbf{Parameters}

\texttt{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

Description
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:
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.
- Memory regions requested must be either entirely registered with CUDA, or in the case of
  host pageable transfers, not registered at all. Memory regions spanning over allocations
  that are both registered and not registered with CUDA are not supported and will return
  CUDA_ERROR_INVALID_VALUE.

See also:
cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy,
cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D,
cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync,
cuMemcpyToA, cuMemcpyToAAsync, cuMemcpyToHost, cuMemcpyToHostAsync,
cuMemcpyToHostAsync, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer,
cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16,
cuMemsetD32, cudaMemcpy, cudaMemcpyToSymbol, cudaMemcpyFromSymbol
CUresult cuMemcpy2D (const CUDA_MEMCPY2D *pCopy)
Copies memory for 2D arrays.

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

Description
Perform a 2D memory copy according to the parameters specified in pCopy. The CUDA_MEMCPY2D structure is defined as:

```c
typedef struct CUDA_MEMCPY2D_st {
    unsigned int srcXInBytes, srcY;
    CUmemorytype srcMemoryType;
    const void *srcHost;
    CUDeviceptr srcDevice;
    CUDarray srcArray;
    unsigned int srcPitch;
    
    unsigned int dstXInBytes, dstY;
    CUmemorytype dstMemoryType;
    void *dstHost;
    CUDeviceptr dstDevice;
    CUDarray 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:

```c
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

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

For device pointers, the starting address is

```c
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

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

For device pointers, the starting address is

```c
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.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.

See also:

CUDAResult cuMemcpy2DAsync (const CUDA_MEMCPY2D *pCopy, CUstream hStream)

Copies memory for 2D arrays.

**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, CUDA_ERROR_INVALID_HANDLE
Description

Perform a 2D memory copy according to the parameters specified in `pCopy`. The CUDA `CUDA_MEMCPY2D` structure is defined as:

```c
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:

```c
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
  ```c
  void* Start = (void*)((char*)srcHost+srcY*srcPitch + srcXInBytes);
  ```
- For device pointers, the starting address is
  ```c
  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
  ```c
  void* dstStart = (void*)((char*)dstHost+dstY*dstPitch + dstXInBytes);
  ```
- For device pointers, the starting address is
  ```c
  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.

`cuMemcpy2DAsync()` 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), `cuMemcpy2DAsync()` may fail for pitches not computed by `cuMemAllocPitch()`.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
CUresult cuMemcpy2DUnaligned (const CUDA_MEMCPY2D *pCopy)
Copies memory for 2D arrays.

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

Description

Perform a 2D memory copy according to the parameters specified in pCopy. The CUDA_MEMCPY2D structure is defined as:

```c
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;
```
srcMemoryType and dstMemoryType specify the type of memory of the source and destination, respectively; CUmemorytype_enum is defined as:

```c
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

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

For device pointers, the starting address is

```c
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

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

For device pointers, the starting address is

```c
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.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits [synchronous] behavior for most use cases.

**See also:**

CUresult cuMemcpy3D (const CUDA_MEMCPY3D *pCopy)
Copies memory for 3D arrays.

Parameters
pCopy
- Parameters for the memory copy

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERRORINVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description
Perform a 3D memory copy according to the parameters specified in pCopy. The
CUDA_MEMCPY3D structure is defined as:

```c
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; CUnumber_of_memorytype_enum is defined as:

```c
typedef enum CUnumber_of_memorytype_enum {
    CU_MEMORYTYPE_HOST = 0x01,
    CU_MEMORYTYPE_DEVICE = 0x02,
    CU_MEMORYTYPE_ARRAY = 0x03,
    CU_MEMORYTYPE_UNIFIED = 0x04
} CUnumber_of_memorytype;
```
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

```c
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.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits **synchronous** behavior for most use cases.

**See also:**

- `cuArray3DCreate`, `cuArray3DGetDescriptor`, `cuArrayCreate`, `cuArrayDestroy`
- `cuArrayGetDescriptor`, `cuMemAlloc`, `cuMemAllocHost`, `cuMemAllocPitch`, `cuMemcpy2D`,
- `cuMemcpy2DAsync`, `cuMemcpy2DUnaligned`, `cuMemcpy3DAsync`, `cuMemcpyAtoA`,
- `cuMemcpyAtoD`, `cuMemcpyAtoH`, `cuMemcpyAtoHasync`, `cuMemcpyDtoA`, `cuMemcpyDtoD`,
- `cuMemcpyDtoHasync`, `cuMemcpyDtoH`, `cuMemcpyDtoHAsync`, `cuMemcpyHtoA`,
- `cuMemcpyHtoAAsync`, `cuMemcpyHtoD`, `cuMemcpyHtoHasync`, `cuMemFree`, `cuMemFreeHost`,
- `cuMemGetAddressRange`, `cuMemGetInfo`, `cuMemHostAlloc`, `cuMemHostGetDevicePointer`,
- `cuMemsetD2D8`, `cuMemsetD2D16`, `cuMemsetD2D32`, `cuMemsetD8`, `cuMemsetD16`,
- `cuMemsetD32`, `cudaMemcpy3D`

**CUresult cuMemcpy3DAsync (const CUDA_MEMCPY3D *pCopy, CUstream hStream)**

Copies memory for 3D arrays.

**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,
CUDA_ERROR_INVALID_HANDLE

Description
Perform a 3D memory copy according to the parameters specified in `pCopy`. The
`CUDA_MEMCPY3D` structure is defined as:

```c
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:

```c
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.
Modules

If specified, srcHeight must be greater than or equal to Height + srcY, and dstHeight must be greater than or equal to Height + dstY.

cuMemcpy3DAsync() 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.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits _asynchronous_ behavior for most use cases.
- This function uses standard _default stream_ semantics.

See also:

**CUresult cuMemcpy3DPeer (const CUDA_MEMCPY3D_PEER *pCopy)**

Copies memory between contexts.

**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

**Description**
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:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.

See also:
cuMemcpyDtoD, cuMemcpyPeer, cuMemcpyDtoDAsync, cuMemcpyPeerAsync, cuMemcpy3DPeerAsync, cudaMemcpy3DPeerAsync

CUresult cuMemcpy3DPeerAsync (const CUDA_MEMCPY3D_PEER *pCopy, CUstream hStream)
Copies memory between contexts asynchronously.

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

Description
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:

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

See also:
cuMemcpyDtoD, cuMemcpyPeer, cuMemcpyDtoDAsync, cuMemcpyPeerAsync, cuMemcpy3DPeerAsync, cudaMemcpy3DPeerAsync
CUresult cuMemcpyAsync (CUdeviceptr dst, CUdeviceptr src, size_t ByteCount, CUstream hStream)

Copies memory asynchronously.

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,
CUDA_ERROR_INVALID_HANDLE

Description

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:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
- Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

See also:
CUresult cuMemcpyAtoA (CUarray dstArray, size_t dstOffset, CUarray srcArray, size_t srcOffset, size_t ByteCount)

Copies memory from Array to Array.

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**

Description

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.
Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.

See also:

- cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy,
- cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, 
- cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync,
- cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD,
- cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcppDtoHAsync, cuMemcpyHtoA,
- cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost,
- cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer,
- cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16,
- cuMemsetD32, cudaMemcpyArrayToArray

CUresult cuMemcpyAtoD (CUdeviceptr dstDevice, 
CUarray srcArray, size_t srcOffset, size_t ByteCount)

Copies memory from Array to Device.

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

Description

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.
Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.

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, cudaMemcpyFromArray

CUresult cuMemcpyAtoH (void *dstHost, CUarray srcArray, size_t srcOffset, size_t ByteCount)

Copies memory from Array to Host.

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

Description

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.
CUresult cuMemcpyAtoHAsync (void *dstHost, CUarray srcArray, size_t srcOffset, size_t ByteCount, CUstream hStream)

Copies memory from Array to Host.

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, CUDA_ERROR_INVALID_HANDLE

Description
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.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
- Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

See also:

CUresult cuMemcpyDtoA (CUarray dstArray, size_t dstOffset, CUdeviceptr srcDevice, size_t ByteCount)
Copies memory from Device to Array.

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

**Description**
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.

**Note:**
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.

**See also:**

**CUresult cuMemcpyDtoD (CUdeviceptr dstDevice, CUdeviceptr srcDevice, size_t ByteCount)**
Copies memory from Device to Device.

**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

Description
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:
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.

See also:
cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy,
cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D,
cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync,
cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHASync, cuMemcpyDtoA,
cuMemcpyDtoH, cuMemcpyDtoHASync, cuMemcpyHtoA, cuMemcpyHtoASync,
cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost,
cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer,
cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16,
cuMemsetD32, cudaMemcpy, cudaMemcpyToSymbol, cudaMemcpyFromSymbol

CUresult cuMemcpyDtoDAsync (CUdeviceptr
dstDevice, CUdeviceptr srcDevice, size_t ByteCount,
CUstream hStream)
Copies memory from Device to Device.

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, CUDA ERROR INVALID HANDLE

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

**See also:**
CUresult cuMemcpyDtoH (void *dstHost, CUdeviceptr srcDevice, size_t ByteCount)

Copies memory from Device to Host.

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

Description

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:
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.
- Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

See also:

CUresult cuMemcpyDtoHAsync (void *dstHost, CUdeviceptr srcDevice, size_t ByteCount, CUstream hStream)

Copies memory from Device to Host.

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, CUDA_ERROR_INVALID_HANDLE

Description

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:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
- Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync,

CUresult cuMemcpyHtoA (CUarray dstArray, size_t dstOffset, const void *srcHost, size_t ByteCount)
Copies memory from Host to Array.

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

Description
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. pSrc specifies the base address of the source. ByteCount specifies the number of bytes to copy.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.
- Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.
See also:
cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy,
cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpyp2D,
cuMemcpyp2DAsync, cuMemcpyp2DUnaligned, cuMemcpyp3D, cuMemcpyp3DAsync,
cuMemcpypAtoA, cuMemcpypAtoD, cuMemcpypAtoH, cuMemcpypAtoHAsync, cuMemcpypDtoA,
cuMemcpypDtoD, cuMemcpypDtoDAsync, cuMemcpypDtoH, cuMemcpypDtoHAsync,
cuMemcpypHtoAAsync, cuMemcpypHtoD, cuMemcpypHtoDAsync, cuMemFree, cuMemFreeHost,
cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer,
cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16,
cuMemsetD32, cudaMemcpypToAarray

CUresult cuMemcpypHtoAAsync (CUarray dstArray,
size_t dstOffset, const void *srcHost, size_t ByteCount, CUstream hStream)
Copies memory from Host to Array.

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,
CUDA_ERROR_INVALID_HANDLE

Description
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.
Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
- Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

See also:
cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyToA, cuMemcpyToD, cuMemcpyToH, cuMemcpyToHAsync, 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, cudaMemcpyToArrayListAsync

CUresult cuMemcpyHtoD (CUdeviceptr dstDevice, const void *srcHost, size_t ByteCount)

Copies memory from Host to Device.

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
Description

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:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.
- Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

See also:


CUresult cuMemcpyHtoDAsync (CUdeviceptr dstDevice, const void *srcHost, size_t ByteCount, CUstream hStream)

Copies memory from Host to Device.

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, CUDA_ERROR_INVALID_HANDLE

**Description**
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:**
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits asynchronous behavior for most use cases.
- This function uses standard default stream semantics.
- Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

**See also:**
CUresult cuMemcpyPeer (CUdeviceptr dstDevice, CUcontext dstContext, CUdeviceptr srcDevice, CUcontext srcContext, size_t ByteCount)

Copies device memory between two contexts.

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

Description

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:

- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.

See also:

cuMemcpyDtoD, cuMemcpy3DPeer, cuMemcpyDtoDAsync, cuMemcpyPeerAsync,
cuMemcpy3DPeerAsync, cudaMemcpyPeer
CUresult cuMemcpyPeerAsync (CUdeviceptr dstDevice, CUcontext dstContext, CUdeviceptr srcDevice, CUcontext srcContext, size_t ByteCount, CUstream hStream)

Copies device memory between two contexts asynchronously.

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
CU_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Description

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

See also:
cuMemcpyDtoD, cuMemcpyPeer, cuMemcpy3DPeer, cuMemcpyDtoDAsync, cuMemcpy3DPeerAsync, cudaMemcpyPeerAsync

CUresult cuMemFree (CUdeviceptr dptr)
Frees device memory.

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

Description
Frees the memory space pointed to by dptr, which must have been returned by a previous call to cuMemAlloc() or cuMemAllocPitch().

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, cudaFree

CUresult cuMemFreeHost (void *p)
Frees page-locked host memory.

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

Description

Frees the memory space pointed to by p, which must have been returned by a previous call to cuMemAllocHost().

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, cudaFreeHost

CUresult cuMemGetAddressRange (CUdeviceptr *pbase, size_t *psize, CUdeviceptr dptr)

Get information on memory allocations.

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_NOT_FOUND,
CUDA_ERROR_INVALID_VALUE
**Description**

Returns the base address in \*pbase and size in \*psize of the allocation by \texttt{cuMemAlloc()} or \texttt{cuMemAllocPitch()} that contains the input pointer \texttt{dptr}. Both parameters \texttt{pbase} and \texttt{psize} are optional. If one of them is \texttt{NULL}, it is ignored.

**Note:**

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

**See also:**


**CUresult cuMemGetInfo (size_t \*free, size_t \*total)**

Gets free and total memory.

**Parameters**

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

**Returns**

\texttt{CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE}

**Description**

Returns in \*\texttt{free} and \*\texttt{total} respectively, the free and total amount of memory available for allocation by the CUDA context, in bytes.
Note that this function may also return error codes from previous, asynchronous launches.

See also:

CUresult cuMemHostAlloc (void **pp, size_t bytesize, unsigned int Flags)
Allocates page-locked host memory.

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

Description
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()`. 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 **CU_MEMHOSTALLOC_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_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.

**Note:**

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

**See also:**

cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32, cudaHostAlloc

CUresult cuMemHostGetDevicePointer (CUdeviceptr *pdoptr, void *p, unsigned int Flags)
Passes back device pointer of mapped pinned memory.

Parameters

pdoptr
- 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

Description
Passes back the device pointer pdoptr corresponding to the mapped, pinned host buffer p
allocated by cuMemHostAlloc.

cuMemHostGetDevicePointer[] will fail if the CU_MEMHOSTALLOC_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.

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

Flags provides for future releases. For now, it must be set to 0.
Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy,
cuArrayGetSize, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D,
cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync,
cuMemcpyAsync, cuMemcpyAsyncD, cuMemcpyAsyncH, cuMemcpyAsyncHasync, cuMemcpyDtoA,
cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHasync,
cuMemcpyHtoA, cuMemcpyHtoAsync, cuMemcpyHtoD, cuMemcpyHtoDHasync,
cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc,
cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16,
cuMemsetD32, cudaHostGetDevicePointer

CUresult cuMemHostGetFlags (unsigned int *pFlags, void *p)
Passes back flags that were used for a pinned allocation.

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

Description
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
cuMemMallocHost[] or cuMemHostAlloc[].

Note:
Note that this function may also return error codes from previous, asynchronous launches.
CUresult cuMemHostRegister (void *p, size_t bytesize, unsigned int Flags)

Registers an existing host memory range for use by CUDA.

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, CUDA_ERROR_HOST_MEMORY_ALREADY_REGISTERED,
CUDA_ERROR_NOT_PERMITTED, CUDA_ERROR_NOT_SUPPORTED

Description

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.

This function has limited support on Mac OS X. OS 10.7 or higher is required.

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().
CU_MEMHOSTREGISTER_IOMEMORY: The pointer is treated as pointing to some I/O memory space, e.g. the PCI Express resource of a 3rd party device.

CU_MEMHOSTREGISTER_READ_ONLY: The pointer is treated as pointing to memory that is considered read-only by the device. On platforms without CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES, this flag is required in order to register memory mapped to the CPU as read-only. Support for the use of this flag can be queried from the device attribute CU_DEVICE_ATTRIBUTE_READ_ONLY_HOST_REGISTER_SUPPORTED. Using this flag with a current context associated with a device that does not have this attribute set will cause cuMemHostRegister to error with CUDA_ERROR_NOT_SUPPORTED.

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

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.

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

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

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

See also:
cuMemHostUnregister, cuMemHostGetFlags, cuMemHostGetDevicePointer, cudaHostRegister
CUresult cuMemHostUnregister (void *p)
Unregisters a memory range that was registered with 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, CUDA_ERROR_HOST_MEMORY_NOT_REGISTERED,

Description

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().

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

See also:
cuMemHostRegister, cudaHostUnregister

CUresult cuMemsetD16 (CUdeviceptr dstDevice, unsigned short us, size_t N)
Initializes device memory.

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

Description

Sets the memory range of $N$ 16-bit values to the specified value $us$. The dstDevice pointer must be two byte aligned.

Note:

‣ Note that this function may also return error codes from previous, asynchronous launches.
‣ See also memset synchronization details.

See also:
cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async, cudaMemset

CUresult cuMemsetD16Async (CUdeviceptr dstDevice, unsigned short us, size_t N, CUstream hStream)

Sets device memory.

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

Description
Sets the memory range of N 16-bit values to the specified value us. The dstDevice pointer must be two byte aligned.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- See also memset synchronization details.
- This function uses standard default stream semantics.

See also:

CUresult cuMemsetD2D16 (CUdeviceptr dstDevice, size_t dstPitch, unsigned short us, size_t Width, size_t Height)
Initializes device memory.

Parameters

- **dstDevice** - Destination device pointer
- **dstPitch** - Pitch of destination device pointer (Unused if Height is 1)
- **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

Description
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. The dstDevice pointer and dstPitch offset must be two byte aligned. This function
performs fastest when the pitch is one that has been passed back by cuMemAllocPitch[].

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- See also memset synchronization details.

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, cuMemcpyHtoHasync,
cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc,
cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16Async,
cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16,
cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async, cudaMemset2D
CUresult cuMemsetD2D16Async (CUdeviceptr dstDevice, size_t dstPitch, unsigned short us, size_t Width, size_t Height, CUstream hStream)

Sets device memory.

Parameters

**dstDevice**
- Destination device pointer

**dstPitch**
- Pitch of destination device pointer [Unused if Height is 1]

**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

Description

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. The `dstDevice` pointer and `dstPitch` offset must be two byte aligned. This function performs fastest when the pitch is one that has been passed back by `cuMemAllocPitch()`.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- See also memset synchronization details.
- This function uses standard default stream semantics.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync,
CUresult cuMemsetD2D32 (CUdeviceptr dstDevice, size_t dstPitch, unsigned int ui, size_t Width, size_t Height)

Initializes device memory.

Parameters

dstDevice
- Destination device pointer
dstPitch
- Pitch of destination device pointer [Unused if Height is 1]
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

Description

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. The dstDevice pointer and dstPitch offset must be four byte aligned. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- See also memset synchronization details.
See also:

CUresult cuMemsetD2D32Async (CUdeviceptr dstDevice, size_t dstPitch, unsigned int ui, size_t Width, size_t Height, CUstream hStream)

Sets device memory.

Parameters

dstDevice
- Destination device pointer
dstPitch
- Pitch of destination device pointer[Unused if Height is 1] 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

Description
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. The dstDevice pointer and dstPitch offset must be four byte aligned. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().
CUresult cuMemsetD2D8 (CUdeviceptr dstDevice, size_t dstPitch, unsigned char uc, size_t Width, size_t Height)

Initializes device memory.

Parameters

dstDevice
- Destination device pointer

dstPitch
- Pitch of destination device pointer (Unused if Height is 1)

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
Description
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().

Note:
› Note that this function may also return error codes from previous, asynchronous launches.
› See also memset synchronization details.

See also:
cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoHAsync, cuMemcpyDtoAAsync, cuMemcpyDtoDAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8Async, cuMemsetD2D16Async, cuMemsetD2D16Async, cuMemsetD2D16Async, cuMemsetD2D32Async, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async, cudaMemset2D

CUresult cuMemsetD2D8Async (CUdeviceptr dstDevice, size_t dstPitch, unsigned char uc, size_t Width, size_t Height, CUstream hStream)
Sets device memory.

Parameters

dstDevice
  - Destination device pointer
dstPitch
  - Pitch of destination device pointer[Unused if Height is 1]uc
  - Value to setWidth
  - 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

**Description**

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().

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- See also memset synchronization details.
- This function uses standard default stream semantics.

**See also:**


**CUresult cuMemsetD32 (CUdeviceptr dstDevice, unsigned int ui, size_t N)**

Initializes device memory.

**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

Description
Sets the memory range of N 32-bit values to the specified value ui. The dstDevice pointer must be four byte aligned.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- See also memset synchronization details.

See also:
cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy,
cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D,
cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync,
cuMemcpyToA, cuMemcpyToAAsync, cuMemcpyToHAsync, cuMemcpyDtoAAsync,
cuMemcpyDtoDAsync, cuMemcpyDtoHAsync, cuMemcpyToAAsync, cuMemcpyToHAsync,
cuMemcpyToAAsync, cuMemcpyToHAsync, cuMemcpyToAAsync, cuMemcpyToHAsync,
cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc,
cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16,
cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8,
cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32Async, cuMemset

CUresult cuMemsetD32Async (CUdeviceptr dstDevice, unsigned int ui, size_t N, CUstream hStream)
Sets device memory.

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

Description
Sets the memory range of N 32-bit values to the specified value ui. The dstDevice pointer
must be four byte aligned.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- See also memset synchronization details.
- This function uses standard default stream semantics.

See also:
cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy,
cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D,
cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync,
cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHASync, cuMemcpyDtoA,
cuMemcpyDtoD, cuMemcpyDtoHASync, cuMemcpyDtoH, cuMemcpyDtoHASync,
cuMemcpyHtoA, cuMemcpyHtoDAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync,
cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc,
cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16,
cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8,
cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cudaMemcpyAsync

CUresult cuMemsetD8 (CUdeviceptr dstDevice,
unsigned char uc, size_t N)
Initializes device memory.

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

Description
Sets the memory range of \( N \) 8-bit values to the specified value \( uc \).

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- See also memset synchronization details.

See also:
cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy,
cuArrayGetSize, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemset3DAsync,
cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHasync, cuMemcpyDtoA,
cuMemcpyDtoD, cuMemcpyDtoDasync, cuMemcpyDtoH, cuMemcpyDtoHasync,
cuMemcpyHtoA, cuMemcpyHtoDasync, cuMemcpyHtoDasync, cuMemFree, cuMemFreeHost,
cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc,
cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8async, cuMemsetD2D16,
cuMemsetD2D16async, cuMemsetD2D32, cuMemsetD2D32async, cuMemsetD8async,
cuMemsetD16, cuMemsetD16async, cuMemsetD32, cuMemsetD32async, cudaMemset

CUresult cuMemsetD8Async (CUdeviceptr dstDevice,
unsigned char uc, size_t N, CUstream hStream)
Sets device memory.

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

Description
Sets the memory range of N 8-bit values to the specified value uc.

Note:
› Note that this function may also return error codes from previous, asynchronous launches.
› See also memset synchronization details.
› This function uses standard default stream semantics.

See also:
cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy,
cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D,
cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync,
cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA,
cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync,
cuMemcpyHtoA, cuMemcpyHtoDAsync, cuMemcpyHtoDAsync,
cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc,
cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16,
cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8,
cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async, cudaMemsetAsync

CUresult cuMipmappedArrayCreate
(CUMipmappedArray *pHandle,
const CUDA_ARRAY3D_DESCRIPTOR *pMipmappedArrayDesc, unsigned int numMipmapLevels)

Creates a CUDA mipmapped array.

Parameters
pHandle
- Returned mipmapped array
pMipmappedArrayDesc
- mipmapped array descriptor
numMipmapLevels
- Number of mipmap levels

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

Description
Creates a CUDA mipmapped array according to the CUDA_ARRAY3D_DESCRIPTOR structure
pMipmappedArrayDesc and returns a handle to the new CUDA mipmapped array in
*pHandle. numMipmapLevels specifies the number of mipmap levels to be allocated. This
value is clamped to the range [1, 1 + floor{log2[max{width, height, depth}]}].

The CUDA_ARRAY3D_DESCRIPTOR is defined as:

```c
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 following types of CUDA arrays can be allocated:
  - A 1D mipmapped array is allocated if Height and Depth extents are both zero.
  - A 2D mipmapped array is allocated if only Depth extent is zero.
  - A 3D mipmapped array is allocated if all three extents are non-zero.
  - A 1D layered CUDA mipmapped array is allocated if only Height is zero and the
    CUDA_ARRAY3D_LAYERED flag is set. Each layer is a 1D array. The number of layers
    is determined by the depth extent.
  - A 2D layered CUDA mipmapped array is allocated if all three extents are non-zero and
    the CUDA_ARRAY3D_LAYERED flag is set. Each layer is a 2D array. The number of
    layers is determined by the depth extent.
  - A cubemap CUDA mipmapped array is allocated if all three extents are non-zero and
    the CUDA_ARRAY3D_CUBEMAP flag is set. Width must be equal to Height, and
    Depth must be six. A cubemap is a special type of 2D layered CUDA array, where the
    six layers represent the six faces of a cube. The order of the six layers in memory is the
    same as that listed in CUarray_cubemap_face.
  - A cubemap layered CUDA mipmapped array is allocated if all three extents are non-zero and
    both CUDA_ARRAY3D_CUBEMAP and CUDA_ARRAY3D_LAYERED flags are set. Width
    must be equal to Height, and Depth must be a multiple of six. A cubemap layered
    CUDA array is a special type of 2D layered CUDA array that consists
of a collection of cubemaps. The first six layers represent the first cubemap, the next six layers form the second cubemap, and so on.

- **Format** specifies the format of the elements; `CUarray_format` is defined as:

```c
typedef enum CUarray_format_enum {
    CU_AD_FORMAT_UNSIGNED_INT8 = 0x01,
    CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
    CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
    CU_AD_FORMAT_SIGNED_INT8 = 0x08,
    CU_AD_FORMAT_SIGNED_INT16 = 0x09,
    CU_AD_FORMAT_SIGNED_INT32 = 0x0a,
    CU_AD_FORMAT_HALF = 0x10,
    CU_AD_FORMAT_FLOAT = 0x20
} CUarray_format;
```

- **NumChannels** specifies the number of packed components per CUDA array element; it may be 1, 2, or 4;

- **Flags** may be set to
  - `CUDA_ARRAY3D_LAYERED` to enable creation of layered CUDA mipmapped 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 individual mipmap levels of the CUDA mipmapped array. If this flag is not set, `cuSurfRefSetArray` will fail when attempting to bind a mipmap level of the CUDA mipmapped array to a surface reference.
  - `CUDA_ARRAY3D_CUBEMAP` to enable creation of mipmapped cubemaps. If this flag is set, `Width` must be equal to `Height`, and `Depth` must be six. If the `CUDA_ARRAY3D_LAYERED` flag is also set, then `Depth` must be a multiple of six.
  - `CUDA_ARRAY3D_TEXTURE_GATHER` to indicate that the CUDA mipmapped array will be used for texture gather. Texture gather can only be performed on 2D CUDA mipmapped arrays.

`Width`, `Height` and `Depth` must meet certain size requirements as listed in the following table. All values are specified in elements. Note that for brevity’s sake, the full name of the device attribute is not specified. For ex., `TEXTURE1D_MIPMAPPED_WIDTH` refers to the device attribute `CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_MIPMAPPED_WIDTH`.

<table>
<thead>
<tr>
<th>CUDA array type</th>
<th>Valid extents that must always be met {(width range in elements), (height range), (depth range)}</th>
<th>Valid extents with <code>CUDA_ARRAY3D_SURFACE_LDST</code> set {(width range in elements), (height range), (depth range)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D</td>
<td>{1, <code>TEXTURE1D_MIPMAPPED_WIDTH</code>, <code>SURFACE1D_WIDTH</code>}, 0, 0</td>
<td>0, 0</td>
</tr>
<tr>
<td>2D</td>
<td>{1, <code>TEXTURE2D_MIPMAPPED_WIDTH</code>, <code>SURFACE2D_WIDTH</code>}, 1, <code>TEXTURE2D_MIPMAPPED_HEIGHT</code>, <code>SURFACE2D_HEIGHT</code>, 0</td>
<td>0</td>
</tr>
<tr>
<td>Mode</td>
<td>Format</td>
<td></td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>3D</td>
<td>{(1,\text{TEXTURE3D_WIDTH}), {1,\text{SURFACE3D_WIDTH}}, {1,\text{TEXTURE3D_HEIGHT}}, {1,\text{SURFACE3D_HEIGHT}}, {1,\text{TEXTURE3D_DEPTH}} OR {1,\text{TEXTURE3D_WIDTH_ALTERNATE}}, {1,\text{TEXTURE3D_HEIGHT_ALTERNATE}}, {1,\text{TEXTURE3D_DEPTH_ALTERNATE}} }</td>
<td></td>
</tr>
<tr>
<td>1D Layered</td>
<td>{(1,\text{TEXTURE1D_LAYERED_WIDTH}), {1,\text{SURFACE1D_LAYERED_WIDTH}}, 0, 0, {1,\text{TEXTURE1D_LAYERED_LAYERS}}, {1,\text{SURFACE1D_LAYERED_LAYERS}} }</td>
<td></td>
</tr>
<tr>
<td>2D Layered</td>
<td>{(1,\text{TEXTURE2D_LAYERED_WIDTH}), {1,\text{SURFACE2D_LAYERED_WIDTH}}, {1,\text{TEXTURE2D_LAYERED_HEIGHT}}, {1,\text{SURFACE2D_LAYERED_HEIGHT}}, {1,\text{TEXTURE2D_LAYERED_LAYERS}}, {1,\text{SURFACE2D_LAYERED_LAYERS}} }</td>
<td></td>
</tr>
<tr>
<td>Cubemap</td>
<td>{(1,\text{TEXTURECUBEMAP_WIDTH}), {1,\text{SURFACECUBEMAP_WIDTH}}, {1,\text{TEXTURECUBEMAP_WIDTH}}, {1,\text{SURFACECUBEMAP_WIDTH}}, 6 }</td>
<td></td>
</tr>
<tr>
<td>Cubemap Layered</td>
<td>{(1,\text{TEXTURECUBEMAP_LAYERED_WIDTH}), {1,\text{SURFACECUBEMAP_LAYERED_WIDTH}}, {1,\text{TEXTURECUBEMAP_LAYERED_DEPTH}}, {1,\text{SURFACECUBEMAP_LAYERED_DEPTH}}, {1,\text{TEXTURECUBEMAP_LAYERED_LAYERS}}, {1,\text{SURFACECUBEMAP_LAYERED_LAYERS}} }</td>
<td></td>
</tr>
</tbody>
</table>

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

**See also:**
- cuMipmappedArrayDestroy
- cuMipmappedArrayGetLevel
- cuArrayCreate
- cudaMallocMipmappedArray

**CUresult cuMipmappedArrayDestroy (CUmipmappedArray hMipmappedArray)**

Destroys a CUDA mipmapped array.

**Parameters**

- **hMipmappedArray**
  - Mipmapped 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, CUDA_ERROR_CONTEXT_IS_DESTROYED

Description

Destroys the CUDA mipmapped array hMipmappedArray.

Note:

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

See also:

cuMipmappedArrayCreate, cuMipmappedArrayGetLevel, cuArrayCreate, cudaFreeMipmappedArray

CUresult cuMipmappedArrayGetLevel (CUarray *pLevelArray, CUmipmappedArray hMipmappedArray, unsigned int level)

Gets a mipmap level of a CUDA mipmapped array.

Parameters

pLevelArray
- Returned mipmap level CUDA array
hMipmappedArray
- CUDA mipmapped array
level
- Mipmap level

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Description

Returns in *pLevelArray a CUDA array that represents a single mipmap level of the CUDA mipmapped array hMipmappedArray.
If `level` is greater than the maximum number of levels in this mipmapped array, `CUDA_ERROR_INVALID_VALUE` is returned.

**Note:**

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

See also:

- `cuMipmappedArrayCreate`, `cuMipmappedArrayDestroy`, `cuArrayCreate`,
- `cudaGetMipmappedArrayLevel`

**CUresult cuMipmappedArrayGetSparseProperties**

*(CUDA_ARRAY_SPARSE_PROPERTIES *sparseProperties, CUmipmappedArray mipmap)*

Returns the layout properties of a sparse CUDA mipmapped array.

**Parameters**

- **sparseProperties**
  - Pointer to `CUDA_ARRAY_SPARSE_PROPERTIES`
- **mipmap**
  - CUDA mipmapped array to get the sparse properties of

**Returns**

- `CUDA_SUCCESS`
- `CUDA_ERROR_INVALID_VALUE`

**Description**

Returns the sparse array layout properties in `sparseProperties`. If the CUDA mipmapped array is not allocated with flag `CUDA_ARRAY3D_SPARSE`, `CUDA_ERROR_INVALID_VALUE` will be returned.

For non-layered CUDA mipmapped arrays, `CUDA_ARRAY_SPARSE_PROPERTIES::miptailSize` returns the size of the mip tail region. The mip tail region includes all mip levels whose width, height or depth is less than that of the tile. For layered CUDA mipmapped arrays, if `CUDA_ARRAY_SPARSE_PROPERTIES::flags` contains `CU_ARRAY_SPARSE_PROPERTIES_SINGLE_MIPTAIL`, then `CUDA_ARRAY_SPARSE_PROPERTIES::miptailSize` specifies the size of the mip tail of all layers combined. Otherwise, `CUDA_ARRAY_SPARSE_PROPERTIES::miptailSize` specifies mip tail size per layer. The returned value of `CUDA_ARRAY_SPARSE_PROPERTIES::miptailFirstLevel` is valid only if `CUDA_ARRAY_SPARSE_PROPERTIES::miptailSize` is non-zero.
See also:
cuArrayGetSparseProperties, cuMemMapArrayAsync

5.12. Virtual Memory Management

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

CUresult cuMemAddressFree (CUdeviceptr ptr, size_t size)

Free an address range reservation.

Parameters

ptr
- Starting address of the virtual address range to free

size
- Size of the virtual address region to free

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED, CUDA_ERROR_NOT_SUPPORTED

Description

Frees a virtual address range reserved by cuMemAddressReserve. The size must match what was given to memAddressReserve and the ptr given must match what was returned from memAddressReserve.

See also:
cuMemAddressReserve
CUresult cuMemAddressReserve (CUdeviceptr *ptr, size_t size, size_t alignment, CUdeviceptr addr, unsigned long long flags)

Allocate an address range reservation.

Parameters

ptr
- Resulting pointer to start of virtual address range allocated

size
- Size of the reserved virtual address range requested

alignment
- Alignment of the reserved virtual address range requested

addr
- Fixed starting address range requested

flags
- Currently unused, must be zero

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED, CUDA_ERROR_NOT_SUPPORTED

Description

Reserves a virtual address range based on the given parameters, giving the starting address of the range in ptr. This API requires a system that supports UVA. The size and address parameters must be a multiple of the host page size and the alignment must be a power of two or zero for default alignment.

See also:

cuMemAddressFree
CUresult cuMemCreate
(CUmemGenericAllocationHandle *handle, size_t size,
const CUmemAllocationProp *prop, unsigned long long flags)

Create a CUDA memory handle representing a memory allocation of a given size described by the given properties.

Parameters
handle
- Value of handle returned. All operations on this allocation are to be performed using this handle.
size
- Size of the allocation requested
prop
- Properties of the allocation to create.
flags
- flags for future use, must be zero now.

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY,
CUDA_ERROR_INVALID_DEVICE, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED,
CUDA_ERROR_NOT_SUPPORTED

Description
This creates a memory allocation on the target device specified through the prop structure. The created allocation will not have any device or host mappings. The generic memory handle for the allocation can be mapped to the address space of calling process via cuMemMap. This handle cannot be transmitted directly to other processes (see cuMemExportToShareableHandle). On Windows, the caller must also pass an LPSECURITYATTRIBUTE in prop to be associated with this handle which limits or allows access to this handle for a recepient process (see CUmemAllocationProp::win32HandleMetaData for more). The size of this allocation must be a multiple of the the value given via cuMemGetAllocationGranularity with the CU_MEM_ALLOC_GRANULARITY_MINIMUM flag. If CUmemAllocationProp::allocFlags::usage contains CU_MEM_CREATE_USAGE TILE_POOL flag then the memory allocation is intended only to be used as backing tile pool for sparse CUDA arrays and sparse CUDA mipmapped arrays. (see cuMemMapArrayAsync).
Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuMemRelease, cuMemExportToShareableHandle, cuMemImportFromShareableHandle

CUresult cuMemExportToShareableHandle (void *shareableHandle, CUmemGenericAllocationHandle handle, CUmemAllocationHandleType handleType, unsigned long long flags)
Exports an allocation to a requested shareable handle type.

Parameters

shareableHandle
  - Pointer to the location in which to store the requested handle type

handle
  - CUDA handle for the memory allocation

handleType
  - Type of shareable handle requested (defines type and size of the shareableHandle output parameter)

flags
  - Reserved, must be zero

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED, CUDA_ERROR_NOT_SUPPORTED

Description

Given a CUDA memory handle, create a shareable memory allocation handle that can be used to share the memory with other processes. The recipient process can convert the shareable handle back into a CUDA memory handle using cuMemImportFromShareableHandle and map it with cuMemMap. The implementation of what this handle is and how it can be transferred is defined by the requested handle type in handleType

Once all shareable handles are closed and the allocation is released, the allocated memory referenced will be released back to the OS and uses of the CUDA handle afterward will lead to undefined behavior.
This API can also be used in conjunction with other APIs (e.g. Vulkan, OpenGL) that support importing memory from the shareable type.

See also:
cuMemImportFromShareableHandle

CUresult cuMemGetAccess (unsigned long long *flags, const CUmemLocation *location, CUdeviceptr ptr)
Get the access flags set for the given location and ptr.

Parameters

flags
- Flags set for this location

location
- Location in which to check the flags for

ptr
- Address in which to check the access flags for

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED, CUDA_ERROR_NOT_SUPPORTED

Description

See also:
cuMemSetAccess

CUresult cuMemGetAllocationGranularity (size_t *granularity, const CUmemAllocationProp *prop, CUmemAllocationGranularity_flags option)
Calculates either the minimal or recommended granularity.

Parameters

granularity
- Returned granularity.
prop
Property for which to determine the granularity for

option
Determines which granularity to return

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED,
CUDA_ERROR_NOT_SUPPORTED

Description
Calculates either the minimal or recommended granularity for a given allocation specification
and returns it in granularity. This granularity can be used as a multiple for alignment, size, or
address mapping.

See also:
cuMemCreate, cuMemMap

CUresult cuMemGetAllocationPropertiesFromHandle
(CUmemAllocationProp *prop,
CUmemGenericAllocationHandle handle)
Retrieve the contents of the property structure defining properties for this handle.

Parameters
prop
- Pointer to a properties structure which will hold the information about this handle
handle
- Handle which to perform the query on

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED,
CUDA_ERROR_NOT_SUPPORTED

Description

See also:
cuMemCreate, cuMemImportFromShareableHandle
CUresult cuMemImportFromShareableHandle(CUmemGenericAllocationHandle *handle, void *osHandle, CUmemAllocationHandleType shHandleType)
Imports an allocation from a requested shareable handle type.

Parameters

handle
- CUDA Memory handle for the memory allocation.

osHandle
- Shareable Handle representing the memory allocation that is to be imported.

shHandleType
- handle type of the exported handle CUmemAllocationHandleType.

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED, CUDA_ERROR_NOT_SUPPORTED

Description
If the current process cannot support the memory described by this shareable handle, this API will error as CUDA_ERROR_NOT_SUPPORTED.

Note:
Importing shareable handles exported from some graphics APIs (Vulkan, OpenGL, etc) created on devices under an SLI group may not be supported, and thus this API will return CUDA_ERROR_NOT_SUPPORTED. There is no guarantee that the contents of handle will be the same CUDA memory handle for the same given OS shareable handle, or the same underlying allocation.

See also:
cuMemExportToShareableHandle, cuMemMap, cuMemRelease
CUresult cuMemMap (CUdeviceptr ptr, size_t size, size_t offset, CUmemGenericAllocationHandle handle, unsigned long long flags)
Maps an allocation handle to a reserved virtual address range.

Parameters

ptr
- Address where memory will be mapped.

size
- Size of the memory mapping.

offset
handle from which to start mapping Note: currently must be zero.
- Offset into the memory represented by

handle
- Handle to a shareable memory

flags
- flags for future use, must be zero now.

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED, CUDA_ERROR_NOT_SUPPORTED

- handle from which to start mapping
- Note: currently must be zero.

Description
Maps bytes of memory represented by handle starting from byte offset to size to address range [addr, addr + size]. This range must be an address reservation previously reserved with cuMemAddressReserve, and offset + size must be less than the size of the memory allocation. Both ptr, size, and offset must be a multiple of the value given via cuMemGetAllocationGranularity with the CU_MEM_ALLOC_GRANULARITY_MINIMUM flag.

Please note calling cuMemMap does not make the address accessible, the caller needs to update accessibility of a contiguous mapped VA range by calling cuMemSetAccess.

Once a recipient process obtains a shareable memory handle from cuMemImportFromShareableHandle, the process must use cuMemMap to map the memory into its address ranges before setting accessibility with cuMemSetAccess.
cuMemMap can only create mappings on VA range reservations that are not currently mapped.

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

See also:
cuMemUnmap, cuMemSetAccess, cuMemCreate, cuMemAddressReserve, cuMemImportFromShareableHandle

CUresult cuMemMapArrayAsync (CUarrayMapInfo *mapInfoList, unsigned int count, CUstream hStream)
Maps or unmaps subregions of sparse CUDA arrays and sparse CUDA mipmapped arrays.

Parameters
mapInfoList
- List of CUarrayMapInfo

count
- Count of CUarrayMapInfo in mapInfoList

hStream
- Stream identifier for the stream to use for map or unmap operations

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Description
Performs map or unmap operations on subregions of sparse CUDA arrays and sparse CUDA mipmapped arrays. Each operation is specified by a CUarrayMapInfo entry in the mapInfoList array of size count. The structure CUarrayMapInfo is defined as follow:

```c
typedef struct CUarrayMapInfo_st {
    CResourcetype resourceType;
    union {
        CUmipmappedArray mipmap;
        CUarray array;
    } resource;

    CSparseSubresourceType subresourceType;
    union {
        struct {
            unsigned int level;
            unsigned int layer;
            unsigned int offsetX;
            unsigned int offsetY;
            unsigned int offsetZ;
            unsigned int extentWidth;
        } ;
    }
} CUarrayMapInfo;
```
unsigned int extentHeight;
unsigned int extentDepth;
} sparseLevel;
struct {
    unsigned int layer;
    unsigned long long offset;
    unsigned long long size;
} miptail;
} subresource;

CUmemOperationType memOperationType;
CUmemHandleType memHandleType;
union {
    CUmemGenericAllocationHandle memHandle;
} memHandle;

unsigned long long offset;
unsigned int deviceBitMask;
unsigned int flags;
unsigned int reserved[2];
} CUarrayMapInfo;

where `CUarrayMapInfo::resourceType` specifies the type of resource to be operated on. If `CUarrayMapInfo::resourceType` is set to `CUresourcetype::CU_RESOURCE_TYPE_ARRAY` then `CUarrayMapInfo::resource::array` must be set to a valid sparse CUDA array handle. The CUDA array must be either a 2D, 2D layered or 3D CUDA array and must have been allocated using `cuArrayCreate` or `cuArray3DCreate` with the flag `CUDA_ARRAY3D_SPARSE`. For CUDA arrays obtained using `cuMipmappedArrayGetLevel`, `CUDA_ERROR_INVALID_VALUE` will be returned. If `CUarrayMapInfo::resourceType` is set to `CUresourcetype::CU_RESOURCE_TYPE_MIPMAPPED_ARRAY` then `CUarrayMapInfo::resource::mipmap` must be set to a valid sparse CUDA mipmapped array handle. The CUDA mipmapped array must be either a 2D, 2D layered or 3D CUDA mipmapped array and must have been allocated using `cuMipmappedArrayCreate` with the flag `CUDA_ARRAY3D_SPARSE`.

`CUarrayMapInfo::subresourceType` specifies the type of subresource within the resource. `CUarraySparseSubresourceType_enum` is defined as:

```c
typedef enum CUarraySparseSubresourceType_enum {
    CU_ARRAY_SPARSE_SUBRESOURCE_TYPE_SPARSE_LEVEL = 0,
    CU_ARRAY_SPARSE_SUBRESOURCE_TYPE_MIPTAIL = 1
} CUarraySparseSubresourceType;
```

where `CUarraySparseSubresourceType::CU_ARRAY_SPARSE_SUBRESOURCE_TYPE_SPARSE_LEVEL` indicates a sparse-miplevel which spans at least one tile in every dimension. The remaining miplevels which are too small to span at least one tile in any dimension constitute the mip tail region as indicated by `CUarraySparseSubresourceType::CU_ARRAY_SPARSE_SUBRESOURCE_TYPE_MIPTAIL` subresource type.

If `CUarrayMapInfo::subresourceType` is set to `CUarraySparseSubresourceType::CU_ARRAY_SPARSE_SUBRESOURCE_TYPE_SPARSE_LEVEL` then `CUarrayMapInfo::subresource::sparseLevel` struct must contain valid array subregion offsets and extents. The `CUarrayMapInfo::subresource::sparseLevel::offsetX`,
CUarrayMapInfo::subresource::sparseLevel::offsetY and CUarrayMapInfo::subresource::sparseLevel::offsetZ must specify valid X, Y and Z offsets respectively. The CUarrayMapInfo::subresource::sparseLevel::extentWidth, CUarrayMapInfo::subresource::sparseLevel::extentHeight and CUarrayMapInfo::subresource::sparseLevel::extentDepth must specify valid width, height and depth extents respectively. These offsets and extents must be aligned to the corresponding tile dimension. For CUDA mipmapped arrays CUarrayMapInfo::subresource::sparseLevel::level must specify a valid mip level index. Otherwise, must be zero. For layered CUDA arrays and layered CUDA mipmapped arrays CUarrayMapInfo::subresource::sparseLevel::layer must specify a valid layer index. Otherwise, must be zero. CUarrayMapInfo::subresource::sparseLevel::offsetZ must be zero and CUarrayMapInfo::subresource::sparseLevel::extentDepth must be set to 1 for 2D and 2D layered CUDA arrays and CUDA mipmapped arrays. Tile extents can be obtained by calling cuArrayGetSparseProperties and cuMipmappedArrayGetSparseProperties.

If CUarrayMapInfo::subresourceType is set to CUArraySparseSubresourceType::CU_ARRAY_SPARSE_SUBRESOURCE_TYPE_MIPTAIL then CUarrayMapInfo::subresource::miptail struct must contain valid mip tail offset in CUarrayMapInfo::subresource::miptail::offset and size in CUarrayMapInfo::subresource::miptail::size. Both, mip tail offset and mip tail size must be aligned to the tile size. For layered CUDA mipmapped arrays which don’t have the flag CU_ARRAY_SPARSE_PROPERTIES_SINGLE_MIPTAIL set in CUDA_ARRAY_SPARSE_PROPERTIES::flags as returned by cuMipmappedArrayGetSparseProperties, CUarrayMapInfo::subresource::miptail::layer must specify a valid layer index. Otherwise, must be zero.

CUarrayMapInfo::memOperationType specifies the type of operation. CUmemOperationType is defined as:

```c
typedef enum CUmemOperationType_enum {
    CU_MEM_OPERATION_TYPE_MAP = 1,
    CU_MEM_OPERATION_TYPE_UNMAP = 2
} CUmemOperationType;
```

If CUarrayMapInfo::memOperationType is set to CUmemOperationType::CU_MEM_OPERATION_TYPE_MAP then the subresource will be mapped onto the tile pool memory specified by CUarrayMapInfo::memHandle at offset CUarrayMapInfo::offset. The tile pool allocation has to be created by specifying the CU_MEM_CREATE_USAGE_TILE_POOL flag when calling cuMemCreate. Also, CUarrayMapInfo::memHandleType must be set to CUmemHandleType::CU_MEM_HANDLE_TYPE_GENERIC.

If CUarrayMapInfo::memOperationType is set to CUmemOperationType::CU_MEM_OPERATION_TYPE_UNMAP then an unmapping operation is performed. CUarrayMapInfo::memHandle must be NULL.

CUarrayMapInfo::deviceBitMask specifies the list of devices that must map or unmap physical memory. Currently, this mask must have exactly one bit set, and the corresponding device must match the device associated with the stream. If CUarrayMapInfo::memOperationType
is set to CUmemOperationType::CU_MEM_OPERATION_TYPE_MAP, the device must also match the device associated with the tile pool memory allocation as specified by CUarrayMapInfo::memHandle.

CUarrayMapInfo::flags and CUarrayMapInfo::reserved[] are unused and must be set to zero.

See also:

cuMipmappedArrayCreate, cuArrayCreate, cuArray3DCreate, cuMemCreate, cuArrayGetSparseProperties, cuMipmappedArrayGetSparseProperties

CUresult cuMemRelease (CUmemGenericAllocationHandle handle)

Release a memory handle representing a memory allocation which was previously allocated through cuMemCreate.

Parameters

handle

Value of handle which was returned previously by cuMemCreate.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED, CUDA_ERROR_NOT_SUPPORTED

Description

Frees the memory that was allocated on a device through cuMemCreate.

The memory allocation will be freed when all outstanding mappings to the memory are unmapped and when all outstanding references to the handle (including it’s shareable counterparts) are also released. The generic memory handle can be freed when there are still outstanding mappings made with this handle. Each time a recipient process imports a shareable handle, it needs to pair it with cuMemRelease for the handle to be freed. If handle is not a valid handle the behavior is undefined.

Note:

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

See also:

cuMemCreate
CUresult cuMemRetainAllocationHandle (
CUmemGenericAllocationHandle *handle, void *addr)
Given an address addr, returns the allocation handle of the backing memory allocation.

Parameters

handle
- CUDA Memory handle for the backing memory allocation.

addr
- Memory address to query, that has been mapped previously.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_PERMITTED,
CUDA_ERROR_NOT_SUPPORTED

Description

The handle is guaranteed to be the same handle value used to map the memory. If the address requested is not mapped, the function will fail. The returned handle must be released with corresponding number of calls to cuMemRelease.

Note:
The address addr, can be any address in a range previously mapped by cuMemMap, and not necessarily the start address.

See also:
cuMemCreate, cuMemRelease, cuMemMap

CUresult cuMemSetAccess (CUdeviceptr ptr, size_t size, const CUmemAccessDesc *desc, size_t count)
Set the access flags for each location specified in desc for the given virtual address range.

Parameters

ptr
- Starting address for the virtual address range

size
- Length of the virtual address range

desc
mapping for each location specified
- Array of \texttt{CUmemAccessDesc} that describe how to change the count
- Number of \texttt{CUmemAccessDesc} in \texttt{desc}

\textbf{Returns}
\texttt{CUDA\ SUCCESS, CUDA\ ERROR\ INVALID\ VALUE, CUDA\ ERROR\ INVALID\ DEVICE, CUDA\ ERROR\ NOT\ SUPPORTED}

\textbullet\ mapping for each location specified

\textbf{Description}
Given the virtual address range via \texttt{ptr} and \texttt{size}, and the locations in the array given by \texttt{desc} and \texttt{count}, set the access flags for the target locations. The range must be a fully mapped address range containing all allocations created by \texttt{cuMemMap / cuMemCreate}.

\textbf{Note:}
\begin{itemize}
  \item Note that this function may also return error codes from previous, asynchronous launches.
  \item This function exhibits \textit{asynchronous} behavior for most use cases.
\end{itemize}

\textbf{See also:}
\texttt{cuMemSetAccess, \textcolor{blue}{cuMemCreate}}, \texttt{\textcolor{blue}{cuMemMap}}

\textbf{CUresult cuMemUnmap (CUdeviceptr ptr, size_t size)}
Unmap the backing memory of a given address range.

\textbf{Parameters}
\texttt{ptr}
- Starting address for the virtual address range to unmap
\texttt{size}
- Size of the virtual address range to unmap

\textbf{Returns}
\texttt{CUDA\ SUCCESS, CUDA\ ERROR\ INVALID\ VALUE, CUDA\ ERROR\ NOT\ INITIALIZED, CUDA\ ERROR\ DEINITIALIZED, CUDA\ ERROR\ NOT\ PERMITTED, CUDA\ ERROR\ NOT\ SUPPORTED}

\textbf{Description}
The range must be the entire contiguous address range that was mapped to. In other words, \texttt{cuMemUnmap} cannot unmap a sub-range of an address range mapped by \texttt{cuMemCreate} /
cuMemMap. Any backing memory allocations will be freed if there are no existing mappings and there are no unreleased memory handles.

When cuMemUnmap returns successfully the address range is converted to an address reservation and can be used for a future calls to cuMemMap. Any new mapping to this virtual address will need to have access granted through cuMemSetAccess, as all mappings start with no accessibility setup.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- This function exhibits synchronous behavior for most use cases.

See also:
cuMemCreate, cuMemAddressReserve

5.13. Unified Addressing

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

Overview

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

Supported Platforms

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

Unified addressing is automatically enabled in 64-bit processes

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.

Since 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 cudaMemcpy may be used to perform a copy between two pointers, ignoring whether they point to host or device memory.
memory (making \texttt{cuMemcpyHtoD()}, \texttt{cuMemcpyDtoD()}, and \texttt{cuMemcpyDtoH()} unnecessary for devices supporting unified addressing). For multidimensional copies, the memory type \texttt{CU\_MEMORYTYPE\_UNIFIED} may be used to specify that the CUDA driver should infer the location of the pointer from its value.

**Automatic Mapping of Host Allocated Host Memory**

All host memory allocated in all contexts using \texttt{cuMemHostAlloc()} and \texttt{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 \texttt{CU\_MEMHOSTALLOC\_PORTABLE} and \texttt{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 \texttt{cuMemHostGetDevicePointer()} to get the device pointer for these allocations.

Note that this is not the case for memory allocated using the flag \texttt{CU\_MEMHOSTALLOC\_WRITECOMBINED}, as discussed below.

**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 \texttt{cuCtxEnablePeerAccess()} all memory allocated in the peer context using \texttt{cuMemAlloc()} and \texttt{cuMemAllocPitch()} will immediately be accessible by the current context. 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.

**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 \texttt{cuMemHostRegister()} and host memory allocated using the flag \texttt{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 \texttt{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 \texttt{cuMemcpy()} and similar functions using the \texttt{CU\_MEMORYTYPE\_UNIFIED} memory type.
CUresult cuMemAdvise (CUdeviceptr devPtr, size_t count, CUmem_advise advice, CUdevice device)

Advise about the usage of a given memory range.

Parameters

devPtr
- Pointer to memory to set the advice for

count
- Size in bytes of the memory range

advice
- Advice to be applied for the specified memory range

device
- Device to apply the advice for

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Description

Advise the Unified Memory subsystem about the usage pattern for the memory range starting at devPtr with a size of count bytes. The start address and end address of the memory range will be rounded down and rounded up respectively to be aligned to CPU page size before the advice is applied. The memory range must refer to managed memory allocated via cuMemAllocManaged or declared via __managed__ variables. The memory range could also refer to system-allocated pageable memory provided it represents a valid, host-accessible region of memory and all additional constraints imposed by advice as outlined below are also satisfied. Specifying an invalid system-allocated pageable memory range results in an error being returned.

The advice parameter can take the following values:

- **CU_MEM_ADVISE_SET_READ_MOSTLY**: This implies that the data is mostly going to be read from and only occasionally written to. Any read accesses from any processor to this region will create a read-only copy of at least the accessed pages in that processor’s memory. Additionally, if cuMemPrefetchAsync is called on this region, it will create a read-only copy of the data on the destination processor. If any processor writes to this region, all copies of the corresponding page will be invalidated except for the one where the write occurred. The device argument is ignored for this advice. Note that for a page to be read-duplicated, the accessing processor must either be the CPU or a GPU that has a non-zero value for the device attribute **CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS**. Also, if a context is created on a device that does not have the device attribute **CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS** set, then read-duplication...
will not occur until all such contexts are destroyed. If the memory region refers to valid system-allocated pageable memory, then the accessing device must have a non-zero value for the device attribute `CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS` for a read-only copy to be created on that device. Note however that if the accessing device also has a non-zero value for the device attribute `CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES`, then setting this advice will not create a read-only copy when that device accesses this memory region.

‣ **CU_MEM_ADVISE_UNSET_READ_MOSTLY**: Undoes the effect of `CU_MEM_ADVISE_SET_READ_MOSTLY` and also prevents the Unified Memory driver from attempting heuristic read-duplication on the memory range. Any read-duplicated copies of the data will be collapsed into a single copy. The location for the collapsed copy will be the preferred location if the page has a preferred location and one of the read-duplicated copies was resident at that location. Otherwise, the location chosen is arbitrary.

‣ **CU_MEM_ADVISE_SET_PREFERRED_LOCATION**: This advice sets the preferred location for the data to be the memory belonging to `device`. Passing in `CU_DEVICE_CPU` for `device` sets the preferred location as host memory. If `device` is a GPU, then it must have a non-zero value for the device attribute `CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS`. Setting the preferred location does not cause data to migrate to that location immediately. Instead, it guides the migration policy when a fault occurs on that memory region. If the data is already in its preferred location and the faulting processor can establish a mapping without requiring the data to be migrated, then data migration will be avoided. On the other hand, if the data is not in its preferred location or if a direct mapping cannot be established, then it will be migrated to the processor accessing it. It is important to note that setting the preferred location does not prevent data prefetching done using `cuMemPrefetchAsync`. Having a preferred location can override the page thrash detection and resolution logic in the Unified Memory driver. Normally, if a page is detected to be constantly thrashing between for example host and device memory, the page may eventually be pinned to host memory by the Unified Memory driver. But if the preferred location is set as device memory, then the page will continue to thrash indefinitely. If `CU_MEM_ADVISE_SET_READ_MOSTLY` is also set on this memory region or any subset of it, then the policies associated with that advice will override the policies of this advice, unless read accesses from `device` will not result in a read-only copy being created on that device as outlined in description for the advice `CU_MEM_ADVISE_SET_READ_MOSTLY`. If the memory region refers to valid system-allocated pageable memory, then `device` must have a non-zero value for the device attribute `CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS`. Additionally, if `device` has a non-zero value for the device attribute `CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES`, then this call has no effect. Note however that this behavior may change in the future.
‣ **CU_MEM_ADVISE_UNSET_PREFERRED_LOCATION**: Undoes the effect of **CU_MEM_ADVISE_SET_PREFERRED_LOCATION** and changes the preferred location to none.

‣ **CU_MEM_ADVISE_SET_ACCESSED_BY**: This advice implies that the data will be accessed by device. Passing in **CU_DEVICE_CPU** for device will set the advice for the CPU. If device is a GPU, then the device attribute **CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS** must be non-zero. This advice does not cause data migration and has no impact on the location of the data per se. Instead, it causes the data to always be mapped in the specified processor’s page tables, as long as the location of the data permits a mapping to be established. If the data gets migrated for any reason, the mappings are updated accordingly. This advice is recommended in scenarios where data locality is not important, but avoiding faults is. Consider for example a system containing multiple GPUs with peer-to-peer access enabled, where the data located on one GPU is occasionally accessed by peer GPUs. In such scenarios, migrating data over to the other GPUs is not as important because the accesses are infrequent and the overhead of migration may be too high. But preventing faults can still help improve performance, and so having a mapping set up in advance is useful. Note that on CPU access of this data, the data may be migrated to host memory because the CPU typically cannot access device memory directly. Any GPU that had the **CU_MEM_ADVISE_SET_ACCESSED_BY** flag set for this data will now have its mapping updated to point to the page in host memory. If **CU_MEM_ADVISE_SET_READ_MOSTLY** is also set on this memory region or any subset of it, then the policies associated with that advice will override the policies of this advice. Additionally, if the preferred location of this memory region or any subset of it is also device, then the policies associated with **CU_MEM_ADVISE_SET_PREFERRED_LOCATION** will override the policies of this advice. If the memory region refers to valid system-allocated pageable memory, then device must have a non-zero value for the device attribute **CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS**. Additionally, if device has a non-zero value for the device attribute **CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES**, then this call has no effect.

‣ **CU_MEM_ADVISE_UNSET_ACCESSED_BY**: Undoes the effect of **CU_MEM_ADVISE_SET_ACCESSED_BY**. Any mappings to the data from device may be removed at any time causing accesses to result in non-fatal page faults. If the memory region refers to valid system-allocated pageable memory, then device must have a non-zero value for the device attribute **CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS**. Additionally, if device has a non-zero value for the device attribute **CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES**, then this call has no effect.
Note:

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

See also:
cuMemcpy, cuMemcpyPeer, cuMemcpyAsync, cuMemcpy3DPeerAsync, cuMemPrefetchAsync, cudaMemAdvise

CUresult cuMemPrefetchAsync (CUdeviceptr devPtr, size_t count, CUdevice dstDevice, CUstream hStream)
Prefetches memory to the specified destination device.

Parameters

devPtr
- Pointer to be prefetched

count
- Size in bytes

dstDevice
- Destination device to prefetch to

hStream
- Stream to enqueue prefetch operation

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Description
Prefetches memory to the specified destination device. devPtr is the base device pointer of the memory to be prefetched and dstDevice is the destination device. count specifies the number of bytes to copy. hStream is the stream in which the operation is enqueued. The memory range must refer to managed memory allocated via cuMemAllocManaged or declared via __managed__ variables.

Passing in CU_DEVICE_CPU for dstDevice will prefetch the data to host memory. If dstDevice is a GPU, then the device attribute CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS must be non-zero. Additionally, hStream must be associated with a device that has a non-zero value for the device attribute CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS.
The start address and end address of the memory range will be rounded down and rounded up respectively to be aligned to CPU page size before the prefetch operation is enqueued in the stream.

If no physical memory has been allocated for this region, then this memory region will be populated and mapped on the destination device. If there’s insufficient memory to prefetch the desired region, the Unified Memory driver may evict pages from other `cuMemAllocManaged` allocations to host memory in order to make room. Device memory allocated using `cuMemAlloc` or `cuArrayCreate` will not be evicted.

By default, any mappings to the previous location of the migrated pages are removed and mappings for the new location are only setup on `dstDevice`. The exact behavior however also depends on the settings applied to this memory range via `cuMemAdvise` as described below:

If `CU_MEM_ADVISE_SET_READ_MOSTLY` was set on any subset of this memory range, then that subset will create a read-only copy of the pages on `dstDevice`.

If `CU_MEM_ADVISE_SET_PREFERRED_LOCATION` was called on any subset of this memory range, then the pages will be migrated to `dstDevice` even if `dstDevice` is not the preferred location of any pages in the memory range.

If `CU_MEM_ADVISE_SET_ACCESSED_BY` was called on any subset of this memory range, then mappings to those pages from all the appropriate processors are updated to refer to the new location if establishing such a mapping is possible. Otherwise, those mappings are cleared.

Note that this API is not required for functionality and only serves to improve performance by allowing the application to migrate data to a suitable location before it is accessed. Memory accesses to this range are always coherent and are allowed even when the data is actively being migrated.

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

---

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

**See also:**
- `cuMemcpy`, `cuMemcpyPeer`, `cuMemcpyAsync`, `cuMemcpy3DPeerAsync`, `cuMemAdvise`, `cudaMemPrefetchAsync`
CUresult cuMemRangeGetAttribute (void *data, size_t dataSize, CUmem_range_attribute attribute, CUdeviceptr devPtr, size_t count)

Query an attribute of a given memory range.

Parameters

data
- A pointers to a memory location where the result of each attribute query will be written to.
dataSize
- Array containing the size of data
attribute
- The attribute to query
devPtr
- Start of the range to query
count
- Size of the range to query

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Description
Query an attribute about the memory range starting at devPtr with a size of count bytes. The memory range must refer to managed memory allocated via cuMemAllocManaged or declared via __managed__ variables.

The attribute parameter can take the following values:

- **CU_MEM_RANGE_ATTRIBUTE_READ_MOSTLY**: If this attribute is specified, data will be interpreted as a 32-bit integer, and dataSize must be 4. The result returned will be 1 if all pages in the given memory range have read-duplication enabled, or 0 otherwise.
- **CU_MEM_RANGE_ATTRIBUTE_PREFERRED_LOCATION**: If this attribute is specified, data will be interpreted as a 32-bit integer, and dataSize must be 4. The result returned will be a GPU device id if all pages in the memory range have that GPU as their preferred location, or it will be CU_DEVICE_CPU if all pages in the memory range have the CPU as their preferred location, or it will be CU_DEVICE_INVALID if either all the pages don’t have the same preferred location or some of the pages don’t have a preferred location at all. Note that the actual location of the pages in the memory range at the time of the query may be different from the preferred location.
- **CU_MEM_RANGE_ATTRIBUTE_ACCESSED_BY**: If this attribute is specified, data will be interpreted as an array of 32-bit integers, and dataSize must be a non-zero multiple of 4. The result returned will be a list of device ids that had
**CU_MEM_ADVISE_SET_ACCESSSED_BY** set for that entire memory range. If any device does not have that advice set for the entire memory range, that device will not be included. If `data` is larger than the number of devices that have that advice set for that memory range, `CU_DEVICE_INVALID` will be returned in all the extra space provided. For ex., if `dataSize` is 12 (i.e. `data` has 3 elements) and only device 0 has the advice set, then the result returned will be `{ 0, CU_DEVICE_INVALID, CU_DEVICE_INVALID }`. If `data` is smaller than the number of devices that have that advice set, then only as many devices will be returned as can fit in the array. There is no guarantee on which specific devices will be returned, however.

- **CU_MEM_RANGE_ATTRIBUTE_LAST_PREFETCH_LOCATION**: If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. The result returned will be the last location to which all pages in the memory range were prefetched explicitly via `cuMemPrefetchAsync`. This will either be a GPU id or `CU_DEVICE_CPU` depending on whether the last location for prefetch was a GPU or the CPU respectively. If any page in the memory range was never explicitly prefetched or if all pages were not prefetched to the same location, `CU_DEVICE_INVALID` will be returned. Note that this simply returns the last location that the application requested to prefetch the memory range to. It gives no indication as to whether the prefetch operation to that location has completed or even begun.

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

**See also:**
- `cuMemRangeGetAttributes`, `cuMemPrefetchAsync`, `cuMemAdvise`,
- `cudaMemRangeGetAttribute`
CUresult cuMemRangeGetAttributes (void **data, size_t *dataSizes, CUmem_range_attribute *attributes, size_t numAttributes, CUdeviceptr devPtr, size_t count)
Query attributes of a given memory range.

Parameters

**data**
- A two-dimensional array containing pointers to memory locations where the result of each attribute query will be written to.

**dataSizes**
- Array containing the sizes of each result

**attributes**
- An array of attributes to query [numAttributes and the number of attributes in this array should match]

**numAttributes**
- Number of attributes to query

**devPtr**
- Start of the range to query

**count**
- Size of the range to query

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Description

Query attributes of the memory range starting at devPtr with a size of count bytes. The memory range must refer to managed memory allocated via cuMemAllocManaged or declared via __managed__ variables. The attributes array will be interpreted to have numAttributes entries. The dataSizes array will also be interpreted to have numAttributes entries. The results of the query will be stored in data.

The list of supported attributes are given below. Please refer to cuMemRangeGetAttribute for attribute descriptions and restrictions.

- CU_MEM_RANGE_ATTRIBUTE_READ_MOSTLY
- CU_MEM_RANGE_ATTRIBUTE_PREFERRED_LOCATION
- CU_MEM_RANGE_ATTRIBUTE_ACCESSED_BY
- CU_MEM_RANGE_ATTRIBUTE_LAST_PREFETCH_LOCATION
Note:

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

See also:

cuMemRangeGetAttribute, cuMemAdvise cuMemPrefetchAsync, cudaMemRangeGetAttributes

CUresult cuPointerGetAttribute (void *data, CUpointer_attribute attribute, CUdeviceptr ptr)

Returns information about a pointer.

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

Description

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 \( \text{ptr} \) addresses host memory then \( \ast \text{data} \) is set to \texttt{CU_MEMORYTYPE_HOST}.

If \( \text{ptr} \) was not allocated by, mapped by, or registered with a \texttt{CUcontext} which uses unified virtual addressing then \texttt{CUDA_ERROR_INVALID_VALUE} is returned.

If the current \texttt{CUcontext} does not support unified virtual addressing then \texttt{CUDA_ERROR_INVALID_CONTEXT} is returned.

- \texttt{CU_POINTER_ATTRIBUTE_DEVICE_POINTER}:

  Returns in \( \ast \text{data} \) the device pointer value through which \( \text{ptr} \) may be accessed by kernels running in the current \texttt{CUcontext}. The type of \texttt{data} must be \texttt{CUdeviceptr \*}.

  If there exists no device pointer value through which kernels running in the current \texttt{CUcontext} may access \( \text{ptr} \) then \texttt{CUDA_ERROR_INVALID_VALUE} is returned.

  If there is no current \texttt{CUcontext} then \texttt{CUDA_ERROR_INVALID_CONTEXT} is returned.

  Except in the exceptional disjoint addressing cases discussed below, the value returned in \( \ast \text{data} \) will equal the input value \( \text{ptr} \).

- \texttt{CU_POINTER_ATTRIBUTE_HOST_POINTER}:

  Returns in \( \ast \text{data} \) the host pointer value through which \( \text{ptr} \) may be accessed by the host program. The type of \texttt{data} must be \texttt{void **}. If there exists no host pointer value through which the host program may directly access \( \text{ptr} \) then \texttt{CUDA_ERROR_INVALID_VALUE} is returned.

  Except in the exceptional disjoint addressing cases discussed below, the value returned in \( \ast \text{data} \) will equal the input value \( \text{ptr} \).

- \texttt{CU_POINTER_ATTRIBUTE_P2P_TOKENS}:

  Returns in \( \ast \text{data} \) two tokens for use with the \texttt{nv-p2p.h} Linux kernel interface. \texttt{data} must be a struct of type \texttt{CUDA_POINTER_ATTRIBUTE_P2P_TOKENS}.

  \( \text{ptr} \) must be a pointer to memory obtained from \texttt{cuMemAlloc()}. Note that p2pToken and vaSpaceToken are only valid for the lifetime of the source allocation. A subsequent allocation at the same address may return completely different tokens. Querying this attribute has a side effect of setting the attribute \texttt{CU_POINTER_ATTRIBUTE_SYNC_MEMOPS} for the region of memory that \( \text{ptr} \) points to.

- \texttt{CU_POINTER_ATTRIBUTE_SYNC_MEMOPS}:

  A boolean attribute which when set, ensures that synchronous memory operations initiated on the region of memory that \( \text{ptr} \) points to will always synchronize. See further documentation in the section titled “API synchronization behavior” to learn more about cases when synchronous memory operations can exhibit asynchronous behavior.

- \texttt{CU_POINTER_ATTRIBUTE_BUFFER_ID}:

  Returns in \( \ast \text{data} \) a buffer ID which is guaranteed to be unique within the process. \texttt{data} must point to an unsigned long long.
ptr must be a pointer to memory obtained from a CUDA memory allocation API. Every memory allocation from any of the CUDA memory allocation APIs will have a unique ID over a process lifetime. Subsequent allocations do not reuse IDs from previous freed allocations. IDs are only unique within a single process.

- **CU_POINTER_ATTRIBUTE_IS_MANAGED:**
  Returns in *data* a boolean that indicates whether the pointer points to managed memory or not.
  If ptr is not a valid CUDA pointer then CUDA_ERROR_INVALID_VALUE is returned.

- **CU_POINTER_ATTRIBUTE_DEVICE_ORDINAL:**
  Returns in *data* an integer representing a device ordinal of a device against which the memory was allocated or registered.

- **CU_POINTER_ATTRIBUTE_IS_LEGACY_CUDA_IPC_CAPABLE:**
  Returns in *data* a boolean that indicates if this pointer maps to an allocation that is suitable for cudaIpcGetMemHandle.

- **CU_POINTER_ATTRIBUTE_RANGE_START_ADDR:**
  Returns in *data* the starting address for the allocation referenced by the device pointer ptr. Note that this is not necessarily the address of the mapped region, but the address of the mappable address range ptr references (e.g. from cuMemAddressReserve).

- **CU_POINTER_ATTRIBUTE_RANGE_SIZE:**
  Returns in *data* the size for the allocation referenced by the device pointer ptr. Note that this is not necessarily the size of the mapped region, but the size of the mappable address range ptr references (e.g. from cuMemAddressReserve). To retrieve the size of the mapped region, see cuMemGetAddressRange.

- **CU_POINTER_ATTRIBUTE_MAPPED:**
  Returns in *data* a boolean that indicates if this pointer is in a valid address range that is mapped to a backing allocation.

- **CU_POINTER_ATTRIBUTE_ALLOWED_HANDLE_TYPES:**
  Returns a bitmask of the allowed handle types for an allocation that may be passed to cuMemExportToShareableHandle.

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.

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

See also:

- cuPointerSetAttribute, cuMemAlloc, cuMemFree, cuMemAllocHost, cuMemFreeHost, cuMemHostAlloc, cuMemHostRegister, cuMemHostUnregister, cudaPointerGetAttributes

```c
CUresult cuPointerGetAttributes (unsigned int numAttributes, CUpointer_attribute *attributes, void **data, CUdeviceptr ptr)
```

Returns information about a pointer.

**Parameters**

- **numAttributes**
  - Number of attributes to query

- **attributes**
  - An array of attributes to query (numAttributes and the number of attributes in this array should match)

- **data**
  - A two-dimensional array containing pointers to memory locations where the result of each attribute query will be written to.

- **ptr**
  - Pointer to query

**Returns**

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE
Description
The supported attributes are (refer to cuPointerGetAttribute for attribute descriptions and restrictions):

- `CU_POINTER_ATTRIBUTE_CONTEXT`
- `CU_POINTER_ATTRIBUTE_MEMORY_TYPE`
- `CU_POINTER_ATTRIBUTE_DEVICE_POINTER`
- `CU_POINTER_ATTRIBUTE_HOST_POINTER`
- `CU_POINTER_ATTRIBUTE_SYNC_MEMOPS`
- `CU_POINTER_ATTRIBUTE_BUFFER_ID`
- `CU_POINTER_ATTRIBUTE_IS_MANAGED`
- `CU_POINTER_ATTRIBUTE_DEVICE_ORDINAL`
- `CU_POINTER_ATTRIBUTE_RANGE_START_ADDR`
- `CU_POINTER_ATTRIBUTE_RANGE_SIZE`
- `CU_POINTER_ATTRIBUTE_MAPPED`
- `CU_POINTER_ATTRIBUTE_IS_LEGACY_CUDA_IPC_CAPABLE`
- `CU_POINTER_ATTRIBUTE_ALLOWED_HANDLE_TYPES`

Unlike cuPointerGetAttribute, this function will not return an error when the ptr encountered is not a valid CUDA pointer. Instead, the attributes are assigned default NULL values and CUDA_SUCCESS is returned.

If ptr was not allocated by, mapped by, or registered with a CUncontext which uses UVA (Unified Virtual Addressing), CUDA_ERROR_INVALID_CONTEXT is returned.

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

See also:
cuPointerGetAttribute, cuPointerSetAttribute, cudaPointerGetAttributes

CUresult cuPointerSetAttribute (const void *value, CUpointer_attribute attribute, CUdeviceptr ptr)
Set attributes on a previously allocated memory region.

Parameters

value
- Pointer to memory containing the value to be set
attribute
- Pointer attribute to set

ptr
- Pointer to a memory region allocated using CUDA memory allocation APIs

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE,
CUDA_ERROR_INVALID_DEVICE

Description
The supported attributes are:

- **CU_POINTER_ATTRIBUTE_SYNC_MEMOPS**: A boolean attribute that can either be set (1) or unset (0). When set, the region of memory that `ptr` points to is guaranteed to always synchronize memory operations that are synchronous. If there are some previously initiated synchronous memory operations that are pending when this attribute is set, the function does not return until those memory operations are complete. See further documentation in the section titled “API synchronization behavior” to learn more about cases when synchronous memory operations can exhibit asynchronous behavior. `value` will be considered as a pointer to an unsigned integer to which this attribute is to be set.

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

See also:
cuPointerGetAttribute, cuPointerGetAttributes, cuMemAlloc, cuMemFree, cuMemAllocHost,
cuMemFreeHost, cuMemHostAlloc, cuMemHostRegister, cuMemHostUnregister


This section describes the stream management functions of the low-level CUDA driver application programming interface.
CUresult cuStreamAddCallback (CUstream hStream, CUstreamCallback callback, void *userData, unsigned int flags)

Add a callback to a compute stream.

Parameters

**hStream**
- Stream to add callback to

**callback**
- The function to call once preceding stream operations are complete

**userData**
- User specified data to be passed to the callback function

**flags**
- Reserved for future use, must be 0

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_SUPPORTED

Description

**Note:**
This function is slated for eventual deprecation and removal. If you do not require the callback to execute in case of a device error, consider using `cuLaunchHostFunc`. Additionally, this function is not supported with `cuStreamBeginCapture` and `cuStreamEndCapture`, unlike `cuLaunchHostFunc`.

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

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

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

For the purposes of Unified Memory, callback execution makes a number of guarantees:
The callback stream is considered idle for the duration of the callback. Thus, for example, a callback may always use memory attached to the callback stream.

The start of execution of a callback has the same effect as synchronizing an event recorded in the same stream immediately prior to the callback. It thus synchronizes streams which have been “joined” prior to the callback.

Adding device work to any stream does not have the effect of making the stream active until all preceding host functions and stream callbacks have executed. Thus, for example, a callback might use global attached memory even if work has been added to another stream, if the work has been ordered behind the callback with an event.

Completion of a callback does not cause a stream to become active except as described above. The callback stream will remain idle if no device work follows the callback, and will remain idle across consecutive callbacks without device work in between. Thus, for example, stream synchronization can be done by signaling from a callback at the end of the stream.

Note:

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

See also:
cuStreamCreate, cuStreamQuery, cuStreamSynchronize, cuStreamWaitEvent, cuStreamDestroy, cuMemAllocManaged, cuStreamAttachMemAsync, cuStreamLaunchHostFunc, cudaStreamAddCallback

CUresult cuStreamAttachMemAsync (CUstream hStream, CUdeviceptr dptr, size_t length, unsigned int flags)

Attach memory to a stream asynchronously.

Parameters

hStream
  - Stream in which to enqueue the attach operation
dptr
  - Pointer to memory [must be a pointer to managed memory or to a valid host-accessible region of system-allocated pageable memory]
length
  - Length of memory
flags
- Must be one of `CUmemAttach_flags`

Returns
`CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_NOT_SUPPORTED`

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

dptr must point to one of the following types of memories:
- managed memory declared using the `__managed__` keyword or allocated with `cuMemAllocManaged`
- a valid host-accessible region of system-allocated pageable memory. This type of memory may only be specified if the device associated with the stream reports a non-zero value for the device attribute `CU_DEVICE_ATTRIBUTE_PAGEABLE_MEMORY_ACCESS`

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

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

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

When memory is associated with a single stream, the Unified Memory system will allow CPU access to this memory region so long as all operations in `hStream` have completed, regardless of whether other streams are active. In effect, this constrains exclusive ownership of the managed memory region by an active GPU to per-stream activity instead of whole-GPU activity.
Accessing memory on the device from streams that are not associated with it will produce undefined results. No error checking is performed by the Unified Memory system to ensure that kernels launched into other streams do not access this region.

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

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

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

See also:
- `cuStreamCreate`, `cuStreamQuery`, `cuStreamSynchronize`, `cuStreamWaitEvent`, `cuStreamDestroy`, `cuMemAllocManaged`, `cudaStreamAttachMemAsync`

```c
CUresult cuStreamBeginCapture (CUstream hStream, CUstreamCaptureMode mode)
```

Begins graph capture on a stream.

**Parameters**
- **hStream**
  - Stream in which to initiate capture
- **mode**
  - Controls the interaction of this capture sequence with other API calls that are potentially unsafe. For more details see `cuThreadExchangeStreamCaptureMode`.

**Returns**
- `CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_VALUE`
Description

Begin graph capture on hStream. When a stream is in capture mode, all operations pushed into the stream will not be executed, but will instead be captured into a graph, which will be returned via cuStreamEndCapture. Capture may not be initiated if stream is CU_STREAM_LEGACY. Capture must be ended on the same stream in which it was initiated, and it may only be initiated if the stream is not already in capture mode. The capture mode may be queried via cuStreamIsCapturing. A unique id representing the capture sequence may be queried via cuStreamGetCaptureInfo.

If mode is not CU_STREAM_CAPTURE_MODE_RELAXED, cuStreamEndCapture must be called on this stream from the same thread.

Note:

Kernels captured using this API must not use texture and surface references. Reading or writing through any texture or surface reference is undefined behavior. This restriction does not apply to texture and surface objects.

Note:

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

See also:

cuStreamCreate, cuStreamIsCapturing, cuStreamEndCapture, cuThreadExchangeStreamCaptureMode

CUresult cuStreamCopyAttributes (CUstream dst, CUstream src)

Copies attributes from source stream to destination stream.

Parameters

dst
  Destination stream

src
  Source stream For list of attributes see CUstreamAttrID

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE
Description
Copies attributes from source stream src to destination stream dst. Both streams must have the same context.

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

See also:
CUaccessPolicyWindow

CUresult cuStreamCreate (CUstream *phStream, unsigned int Flags)
Create a stream.

Parameters

phStream
- Returned newly created stream

Flags
- Parameters for stream creation

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE,
CUDA_ERROR_OUT_OF_MEMORY

Description
Creates a stream and returns a handle in phStream. The Flags argument determines behaviors of the stream.

Valid values for Flags are:

- **CU_STREAM_DEFAULT**: Default stream creation flag.
- **CU_STREAM_NON_BLOCKING**: Specifies that work running in the created stream may run concurrently with work in stream 0 (the NULL stream), and that the created stream should perform no implicit synchronization with stream 0.
Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuStreamDestroy
- cuStreamCreateWithPriority
- cuStreamGetPriority
- cuStreamGetFlags
- cuStreamWaitEvent
- cuStreamQuery
- cuStreamSynchronize
- cuStreamAddCallback
- cudaStreamCreate
- cudaStreamCreateWithFlags

**CUresult cuStreamCreateWithPriority (CUstream *phStream, unsigned int flags, int priority)**

Create a stream with the given priority.

**Parameters**

- **phStream**
  - Returned newly created stream
- **flags**
  - Flags for stream creation. See cuStreamCreate for a list of valid flags
- **priority**
  - Stream priority. Lower numbers represent higher priorities. See cuCtxGetStreamPriorityRange for more information about meaningful stream priorities that can be passed.

**Returns**

- CUDA_SUCCESS, CUDA_ERROR_DEVINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
- CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE,
- CUDA_ERROR_OUT_OF_MEMORY

**Description**

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

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

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Stream priorities are supported only on GPUs with compute capability 3.5 or higher.
In the current implementation, only compute kernels launched in priority streams are affected by the stream’s priority. Stream priorities have no effect on host-to-device and device-to-host memory operations.

See also:


**CUresult cuStreamDestroy (CUstream hStream)**

Destroys a stream.

**Parameters**

**hStream**
- Stream to destroy

**Returns**

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

**Description**

Destroys the stream specified by `hStream`.

In case 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`.

**Note:**

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

See also:

`cuStreamCreate`, `cuStreamWaitEvent`, `cuStreamQuery`, `cuStreamSynchronize`, `cuStreamAddCallback`, `cudaStreamDestroy`
CUresult cuStreamEndCapture (CUstream hStream, CUgraph *phGraph)

Ends capture on a stream, returning the captured graph.

Parameters

**hStream**
- Stream to query

**phGraph**
- The captured graph

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_STREAM_CAPTURE_WRONG_THREAD

Description

End capture on **hStream**, returning the captured graph via **phGraph**. Capture must have been initiated on **hStream** via a call to **cuStreamBeginCapture**. If capture was invalidated, due to a violation of the rules of stream capture, then a NULL graph will be returned.

If the **mode** argument to **cuStreamBeginCapture** was not **CU_STREAM_CAPTURE_MODE_RELAXED**, this call must be from the same thread as **cuStreamBeginCapture**.

**Note:**

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

See also:

**cuStreamCreate**, **cuStreamBeginCapture**, **cuStreamIsCapturing**

CUresult cuStreamGetAttribute (CUstream hStream, CUstreamAttrID attr, CUstreamAttrValue *value_out)

Queries stream attribute.

Parameters

**hStream**

**attr**

**value_out**
Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Description

Queries attribute `attr` from `hStream` and stores it in corresponding member of `value_out`.

Note:

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

See also:

CUaccessPolicyWindow

CUresult cuStreamGetCaptureInfo (CUstream hStream, CUstreamCaptureStatus *captureStatus, cuuint64_t *id)

Query capture status of a stream.

Returns

CUDA_SUCCESS, CUDA_ERROR_STREAM_CAPTURE_IMPLICIT

Description

Query the capture status of a stream and get an id for the capture sequence, which is unique over the lifetime of the process.

If called on CU_STREAM_LEGACY (the “null stream”) while a stream not created with CU_STREAM_NON_BLOCKING is capturing, returns CUDA_ERROR_STREAM_CAPTURE_IMPLICIT.

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

- the call returns CUDA_SUCCESS
- captureStatus is set to CU_STREAM_CAPTURE_STATUS_ACTIVE

Note:

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

See also:
cuStreamBeginCapture, cuStreamIsCapturing

CUresult cuStreamGetCtx (CUstream hStream, CUcontext *pctx)

Query the context associated with a stream.

Parameters

- **hStream**
  - Handle to the stream to be queried
- **pctx**
  - Returned context associated with the stream

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE.

Description

Returns the CUDA context that the stream is associated with.

The stream handle `hStream` can refer to any of the following:

- a stream created via any of the CUDA driver APIs such as `cuStreamCreate` and
  `cuStreamCreateWithPriority`, or their runtime API equivalents such as `cudaStreamCreate`,
  `cudaStreamCreateWithFlags` and `cudaStreamCreateWithPriority`. The returned context is
  the context that was active in the calling thread when the stream was created. Passing an
  invalid handle will result in undefined behavior.

- any of the special streams such as the NULL stream, `CU_STREAM_LEGACY` and
  `CU_STREAM_PER_THREAD`. The runtime API equivalents of these are also accepted,
  which are NULL, `cudaStreamLegacy` and `cudaStreamPerThread` respectively. Specifying
  any of the special handles will return the context current to the calling thread. If no context
  is current to the calling thread, `CUDA_ERROR_INVALID_CONTEXT` is returned.

Note:

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

See also:

cuStreamDestroy, cuStreamCreateWithPriority, cuStreamGetPriority, cuStreamGetFlags,
cuStreamWaitEvent, cuStreamQuery, cuStreamSynchronize, cuStreamAddCallback,
cudaStreamCreate, cudaStreamCreateWithFlags
CUresult cuStreamGetFlags (CUstream hStream, unsigned int *flags)
Query the flags of a given stream.

Parameters

hStream
- Handle to the stream to be queried

flags
- Pointer to an unsigned integer in which the stream’s flags are returned. The value returned in flags is a logical ‘OR’ of all flags that were used while creating this stream. See cuStreamCreate for the list of valid flags.

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

Description

Query the flags of a stream created using cuStreamCreate or cuStreamCreateWithPriority and return the flags in flags.

Note:

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

See also:

cuStreamDestroy, cuStreamCreate, cuStreamGetPriority, cudaStreamGetFlags

CUresult cuStreamGetPriority (CUstream hStream, int *priority)
Query the priority of a given stream.

Parameters

hStream
- Handle to the stream to be queried

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

Description
Query the priority of a stream created using cuStreamCreate or cuStreamCreateWithPriority
and return the priority in priority. Note that if the stream was created with a priority outside the numerical
range returned by cuCtxGetStreamPriorityRange, this function returns the clamped priority. See cuStreamCreateWithPriority
for details about priority clamping.

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

See also:
cuStreamDestroy, cuStreamCreate, cuStreamCreateWithPriority,
cuCtxGetStreamPriorityRange, cuStreamGetFlags, cudaStreamGetPriority

CUresult cuStreamIsCapturing (CUstream hStream,
cuStreamCaptureStatus *captureStatus)
Returns a stream’s capture status.

Parameters
hStream - Stream to query
captureStatus - Returns the stream’s capture status

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_STREAM_CAPTURE_IMPLICIT

Description
Return the capture status of hStream via captureStatus. After a successful call,
*captureStatus will contain one of the following:

- **CU_STREAM_CAPTURE_STATUS_NONE**: The stream is not capturing.
- **CU_STREAM_CAPTURE_STATUS_ACTIVE**: The stream is capturing.
CU_STREAM_CAPTURE_STATUS_INVALIDATED: The stream was capturing but an error has invalidated the capture sequence. The capture sequence must be terminated with cuStreamEndCapture on the stream where it was initiated in order to continue using hStream.

Note that, if this is called on CU_STREAM_LEGACY (the “null stream”) while a blocking stream in the same context is capturing, it will return CUDA_ERROR_STREAM_CAPTURE_IMPLICIT and *captureStatus is unspecified after the call. The blocking stream capture is not invalidated.

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

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

See also:
cuStreamCreate, cuStreamBeginCapture, cuStreamEndCapture

CUresult cuStreamQuery (CUstream hStream)
Determine status of a compute stream.

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

Description

Returns CUDA_SUCCESS if all operations in the stream specified by hStream have completed, or CUDA_ERROR_NOT_READY if not.

For the purposes of Unified Memory, a return value of CUDA_SUCCESS is equivalent to having called cuStreamSynchronize[].

Note:
This function uses standard default stream semantics.
This function may also return error codes from previous, asynchronous launches.

See also:
cuStreamCreate, cuStreamWaitEvent, cuStreamDestroy, cuStreamSynchronize,
cuStreamAddCallback, cudaStreamQuery

CUresult cuStreamSetAttribute (CUstream hStream, CUstreamAttrID attr, const CUstreamAttrValue *value)
Sets stream attribute.

Parameters
hStream
attr
value

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Description
Sets attribute attr on hStream from corresponding attribute of value. The updated attribute will be applied to subsequent work submitted to the stream. It will not affect previously submitted work.

Note:
This function may also return error codes from previous, asynchronous launches.

See also:
CUaccessPolicyWindow

CUresult cuStreamSynchronize (CUstream hStream)
Wait until a stream’s tasks are completed.

Parameters
hStream
- Stream to wait for
Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE

Description
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.

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

See also:
cuStreamCreate, cuStreamDestroy, cuStreamWaitEvent, cuStreamQuery,
cuStreamAddCallback, cudaStreamSynchronize

CUresult cuStreamWaitEvent (CUstream hStream, CUevent hEvent, unsigned int Flags)
Make a compute stream wait on an event.

Parameters
hStream
- Stream to wait
hEvent
- Event to wait on (may not be NULL)
Flags
- See CUevent_capture_flags

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE.

Description
Makes all future work submitted to hStream wait for all work captured in hEvent. See cuEventRecord[] for details on what is captured by an event. The synchronization will be
performed efficiently on the device when applicable. \texttt{hEvent} may be from a different context or device than \texttt{hStream}.

flags include:

- \texttt{CU\_EVENT\_WAIT\_DEFAULT}: Default event creation flag.
- \texttt{CU\_EVENT\_WAIT\_EXTERNAL}: Event is captured in the graph as an external event node when performing stream capture. This flag is invalid outside of stream capture.

**Note:**

- This function uses standard \texttt{default stream} semantics.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuStreamCreate, cuEventRecord, cuStreamQuery, cuStreamSynchronize, cuStreamAddCallback, cuStreamDestroy, cudaStreamWaitEvent

**CUresult cuThreadExchangeStreamCaptureMode (CUstreamCaptureMode *mode)**

Swaps the stream capture interaction mode for a thread.

**Parameters**

\texttt{mode}

- Pointer to mode value to swap with the current mode

**Returns**

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

**Description**

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

```c
CUstreamCaptureMode mode = desiredMode;
cuThreadExchangeStreamCaptureMode(&mode);
...;
cuThreadExchangeStreamCaptureMode(&mode); // restore previous mode
```

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

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

A thread’s mode is one of the following:

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

- **CU_STREAM_CAPTURE_MODE_THREAD_LOCAL**: If the local thread has an ongoing capture sequence not initiated with `CU_STREAM_CAPTURE_MODE_RELAXED`, it is prohibited from potentially unsafe API calls. Concurrent capture sequences in other threads are ignored.

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

**Note:**

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

**See also:**

`cuStreamBeginCapture`

### 5.15. Event Management

This section describes the event management functions of the low-level CUDA driver application programming interface.
CUresult cuEventCreate (CUevent *phEvent, unsigned int Flags)

Creates an event.

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

Description

Creates an event *phEvent for the current context 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.
- **CU_EVENT_INTERPROCESS**: Specifies that the created event may be used as an interprocess event by cuIpcGetEventHandle. **CU_EVENT_INTERPROCESS** must be specified along with **CU_EVENT_DISABLE_TIMING**.

Note:

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

See also:

cuEventRecord, cuEventQuery, cuEventSynchronize, cuEventDestroy, cuEventElapsedTime, cudaEventCreate, cudaEventCreateWithFlags
CUresult cuEventDestroy (CUevent hEvent)

Destroys an event.

Parameters

hEvent
- Event to destroy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE

Description

Destroys the event specified by hEvent.

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

Note:

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

See also:

cuEventCreate, cuEventRecord, cuEventQuery, cuEventSynchronize, cuEventElapsedTime, cudaEventDestroy

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

Computes the elapsed time between two events.

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

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

If either event was last recorded in a non-NULL stream, the resulting time may be greater
than expected (even if both used the same stream handle). This happens because the
`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.

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

See also:
cuEventCreate, cuEventRecord, cuEventQuery, cuEventSynchronize, cuEventDestroy,
cudaEventElapsedTime

CUresult cuEventQuery (CUevent hEvent)
Queries an event’s status.

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

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

Returns CUDA_SUCCESS if all captured work has been completed, or CUDA_ERROR_NOT_READY if any captured work is incomplete.

For the purposes of Unified Memory, a return value of CUDA_SUCCESS is equivalent to having called cuEventSynchronize().

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

See also:
cuEventCreate, cuEventRecord, cuEventSynchronize, cuEventDestroy, cuEventElapsedTime, cudaEventQuery

CUresult cuEventRecord (CUevent hEvent, CUstream hStream)
Records an event.

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

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

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

See also:
- `cuEventCreate`, `cuEventQuery`, `cuEventSynchronize`, `cuStreamWaitEvent`, `cuEventDestroy`, `cuEventElapsedTime`, `cudaEventRecord`, `cuEventRecordWithFlags`

```
CUresult cuEventRecordWithFlags (CUevent hEvent, CUstream hStream, unsigned int flags)
```
Records an event.

Parameters
- **hEvent**
  - Event to record
- **hStream**
  - Stream to record event for
- **flags**
  - See CUevent_capture_flags

Returns
- `CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_HANDLE`, `CUDA_ERROR_INVALID_VALUE`
**Description**

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

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

Flags include:

- **CU_EVENT_RECORD_DEFAULT**: Default event creation flag.
- **CU_EVENT_RECORD_EXTERNAL**: Event is captured in the graph as an external event node when performing stream capture. This flag is invalid outside of stream capture.

**Note:**

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

**See also:**

`cuEventCreate`, `cuEventQuery`, `cuEventSynchronize`, `cuStreamWaitEvent`, `cuEventDestroy`, `cuEventElapsedTime`, `cuEventRecord`, `cudaEventRecord`

**CResult cuEventSynchronize (CUevent hEvent)**

Waits for an event to complete.

**Parameters**

- **hEvent**
  - Event to wait for

**Returns**

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_HANDLE`
Description

Waits until the completion of all work currently captured in `hEvent`. See `cuEventRecord()` for details on what is captured by an event.

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.

Note:

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

See also:

`cuEventCreate`, `cuEventRecord`, `cuEventQuery`, `cuEventDestroy`, `cuEventElapsedTime`, `cudaEventSynchronize`

5.16.  External Resource Interoperability

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

`CUresult cuDestroyExternalMemory (CUexternalMemory extMem)`

Destroys an external memory object.

Parameters

`extMem`

- External memory object to be destroyed

Returns

`CUDA_SUCCESS`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_HANDLE`

Description

Destroys the specified external memory object. Any existing buffers and CUDA mipmapped arrays mapped onto this object must no longer be used and must be explicitly freed using `cuMemFree` and `cuMipmappedArrayDestroy` respectively.
CUresult cuDestroyExternalSemaphore (CUexternalSemaphore extSem)

Destroys an external semaphore.

Parameters

extSem
- External semaphore to be destroyed

Returns

CUDA_SUCCESS, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_HANDLE

Description

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

See also:

cuImportExternalSemaphore, cuSignalExternalSemaphoresAsync, cuWaitExternalSemaphoresAsync

Note:

Note that this function may also return error codes from previous, asynchronous launches.
CUresult cuExternalMemoryGetMappedBuffer
(CUdeviceptr *devPtr, CUexternalMemory extMem,
const CUDA_EXTERNAL_MEMORY_BUFFER_DESC *bufferDesc)

Maps a buffer onto an imported memory object.

Parameters

- *devPtr*
  - Returned device pointer to buffer

- *extMem*
  - Handle to external memory object

- *bufferDesc*
  - Buffer descriptor

Returns

- CUDA_SUCCESS, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_HANDLE

Description

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

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

```c
typedef struct CUDA_EXTERNAL_MEMORY_BUFFER_DESC_st {
    unsigned long long offset;
    unsigned long long size;
    unsigned int flags;
} CUDA_EXTERNAL_MEMORY_BUFFER_DESC;
```

where `CUDA_EXTERNAL_MEMORY_BUFFER_DESC.offset` is the offset in the memory object where the buffer’s base address is. `CUDA_EXTERNAL_MEMORY_BUFFER_DESC.size` is the size of the buffer. `CUDA_EXTERNAL_MEMORY_BUFFER_DESC.flags` must be zero.

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

The returned pointer *devPtr* must be freed using `cuMemFree`.
Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
- cuImportExternalMemory
- cuDestroyExternalMemory
- cuExternalMemoryGetMappedMipmappedArray

CUresult

**cuExternalMemoryGetMappedMipmappedArray**

(CUmipmappedArray *mipmap,
CUexternalMemory extMem, const
CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC *mipmapDesc)

Maps a CUDA mipmapped array onto an external memory object.

**Parameters**

- **mipmap**
  - Returned CUDA mipmapped array
- **extMem**
  - Handle to external memory object
- **mipmapDesc**
  - CUDA array descriptor

**Returns**

CUDA_SUCCESS, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_HANDLE

**Description**

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

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

```c
typedef struct CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC_st {
    unsigned long long offset;
    CUDA_ARRAY3D_DESCRIPTOR arrayDesc;
    unsigned int numLevels;
} CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC;
```
where `CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC::offset` is the offset in the memory object where the base level of the mipmap chain is. `CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC::arrayDesc` describes the format, dimensions and type of the base level of the mipmap chain. For further details on these parameters, please refer to the documentation for `cuMipmappedArrayCreate`. Note that if the mipmapped array is bound as a color target in the graphics API, then the flag `CUDA_ARRAY3D_COLOR_ATTACHMENT` must be specified in `CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC::arrayDesc::Flags`. `CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC::numLevels` specifies the total number of levels in the mipmap chain.

If `extMem` was imported from a handle of type `CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF`, then `CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC::numLevels` must be equal to 1.

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

**Note:**

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

**See also:**

`cuImportExternalMemory` `cuDestroyExternalMemory`, `cuExternalMemoryGetMappedBuffer`

```c
CUresult cuImportExternalMemory
(CUexternalMemory *extMem_out, const
CUDA_EXTERNAL_MEMORY_HANDLE_DESC *memHandleDesc)
```

Imports an external memory object.

**Parameters**

- `extMem_out`
  - Returned handle to an external memory object
- `memHandleDesc`
  - Memory import handle descriptor

**Returns**

`CUDA_SUCCESS`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_HANDLE`
Description

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

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

```c
typedef struct CUDA_EXTERNAL_MEMORY_HANDLE_DESC_st {
    CUexternalMemoryHandleType type;
    union {
        int fd;
        struct {
            void *handle;
            const void *name;
        } win32;
        const void *nvSciBufObject;
    } handle;
    unsigned long long size;
    unsigned int flags;
} CUDA_EXTERNAL_MEMORY_HANDLE_DESC;
```

where `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::type` specifies the type of handle being imported. `CUexternalMemoryHandleType` is defined as:

```c
typedef enum CUexternalMemoryHandleType_enum {
    CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD = 1,
    CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32 = 2,
    CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT = 3,
    CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP = 4,
    CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE = 5,
    CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE = 6,
    CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE_KMT = 7,
    CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF = 8
} CUexternalMemoryHandleType;
```

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

If `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::type` is `CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32`, then exactly one of `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::handle` and `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::name` must not be NULL. If `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::name` must not be NULL, then it must represent a valid shared NT handle that references a memory object. Ownership of this handle is not transferred to CUDA after the import operation, so the application must release the handle using the appropriate system call. If `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::name` is not NULL, then it must point to a NULL-terminated array of UTF-16 characters that refers to a memory object.

If `CUDA_EXTERNAL_MEMORY_HANDLE_DESC::type` is `CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT`, then
CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::handle must be non-NULL and CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::name must be NULL. The handle specified must be a globally shared KMT handle. This handle does not hold a reference to the underlying object, and thus will be invalid when all references to the memory object are destroyed.

If CUDA_EXTERNAL_MEMORY_HANDLE_DESC::type is
CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP, then exactly one of CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::handle and CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::name must not be NULL. If CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::handle is not NULL, then it must represent a valid shared NT handle that is returned by ID3D12Device::CreateSharedHandle when referring to a ID3D12Heap object. This handle holds a reference to the underlying object. If CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::name is not NULL, then it must point to a NULL-terminated array of UTF-16 characters that refers to a ID3D12Heap object.

If CUDA_EXTERNAL_MEMORY_HANDLE_DESC::type is
CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE, then exactly one of CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::handle and CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::name must not be NULL. If CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::handle is not NULL, then it must represent a valid shared NT handle that is returned by ID3D12Device::CreateSharedHandle when referring to a ID3D12Resource object. This handle holds a reference to the underlying object. If CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::name is not NULL, then it must point to a NULL-terminated array of UTF-16 characters that refers to a ID3D12Resource object.

If CUDA_EXTERNAL_MEMORY_HANDLE_DESC::type is
CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE, then CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::handle must represent a valid shared NT handle that is returned by IDXGIResource1::CreateSharedHandle when referring to a ID3D11Resource object. If CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::name is not NULL, then it must point to a NULL-terminated array of UTF-16 characters that refers to a ID3D11Resource object.

If CUDA_EXTERNAL_MEMORY_HANDLE_DESC::type is
CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE_KMT, then CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::handle must represent a valid shared KMT handle that is returned by IDXGIResource::GetSharedHandle when referring to a ID3D11Resource object and CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::win32::name must be NULL.

If CUDA_EXTERNAL_MEMORY_HANDLE_DESC::type is
CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF, then
CUDA_EXTERNAL_MEMORY_HANDLE_DESC::handle::nvSciBufObject must be non-NULL and reference a valid NvSciBuf object. If the NvSciBuf object imported into CUDA is also mapped by other drivers, then the application must use cuWaitExternalSemaphoresAsync or cuSignalExternalSemaphoresAsync as appropriate barriers to maintain coherence between CUDA and the other drivers.

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

Specifying the flag CUDA_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE in CUDA_EXTERNAL_MEMORY_HANDLE_DESC::flags indicates that the resource is a dedicated resource. The definition of what a dedicated resource is outside the scope of this extension. This flag must be set if CUDA_EXTERNAL_MEMORY_HANDLE_DESC::type is one of the following: CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE, CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE, CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE_KMT.

Note:
- Note that this function may also return error codes from previous, asynchronous launches.
- If the Vulkan memory imported into CUDA is mapped on the CPU then the application must use vkInvalidateMappedMemoryRanges/vkFlushMappedMemoryRanges as well as appropriate Vulkan pipeline barriers to maintain coherence between CPU and GPU. For more information on these APIs, please refer to “Synchronization and Cache Control” chapter from Vulkan specification.

See also:
cuDestroyExternalMemory, cuExternalMemoryGetMappedBuffer, cuExternalMemoryGetMappedMipmappedArray

CUresult cuImportExternalSemaphore (CUexternalSemaphore *extSem_out, const CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC *semHandleDesc)

Imports an external semaphore.

Parameters
extSem_out
- Returned handle to an external semaphore
semHandleDesc
- Semaphore import handle descriptor
Returns
CUDA_SUCCESS, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_NOT_SUPPORTED, CUDA_ERROR_INVALID_HANDLE

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

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

```
typedef struct CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC_st {
  CUexternalSemaphoreHandleType type;
  union {
    int fd;
    struct {
      void *handle;
      const void *name;
    } win32;
    const void* NvSciSyncObj;
  } handle;
  unsigned int flags;
} CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC;
```

where CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type specifies the type of handle being imported. CUexternalSemaphoreHandleType is defined as:

```
typedef enum CUexternalSemaphoreHandleType_enum {
  CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD = 1,
  CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32 = 2,
  CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT = 3,
  CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE = 4,
  CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE = 5,
  CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC = 6,
  CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX = 7,
  CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX_KMT = 8
} CUexternalSemaphoreHandleType;
```

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

If CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type is CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32, then exactly one of CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::handle and CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::name must not be NULL. If CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::handle is not NULL, then it must represent a valid shared NT handle that references a synchronization object. Ownership of this handle is not transferred to CUDA after the import operation, so the application must release the handle using the appropriate system call. If
CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::name is not NULL, then it must name a valid synchronization object.

If CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type is 
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT, then 
CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::handle must be non-NULL and CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::name must be NULL. The handle specified must be a globally shared KMT handle. This handle does not hold a reference to the underlying object, and thus will be invalid when all references to the synchronization object are destroyed.

If CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type is 
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE, then exactly one of CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::handle and CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::name must not be NULL. If CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::handle is not NULL, then it must represent a valid shared NT handle that is returned by ID3D12Device::CreateSharedHandle when referring to a ID3D12Fence object. This handle holds a reference to the underlying object. If CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::name is not NULL, then it must name a valid synchronization object that refers to a valid ID3D12Fence object.

If CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type is 
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE, then 
CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::handle represents a valid shared NT handle that is returned by ID3D11Device::CreateSharedHandle. If CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::name is not NULL, then it must name a valid synchronization object that refers to a valid ID3D11Fence object.

If CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type is 
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC, then 
CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::nvSciSyncObj represents a valid NvSciSyncObj.

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX, then 
CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::handle represents a valid shared NT handle that is returned by IDXGIResource1::CreateSharedHandle when referring to a IDXGIKeyedMutex object. If CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::name is not NULL, then it must name a valid synchronization object that refers to a valid IDXGIKeyedMutex object.

If CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type is 
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX_KMT, 
then CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::handle represents a valid shared KMT handle that is returned by IDXGIResource::GetSharedHandle when referring to a IDXGIKeyedMutex object and CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle::win32::name must be NULL.
CUresult cuSignalExternalSemaphoresAsync
(const CUexternalSemaphore *extSemArray, const
CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS
*paramsArray, unsigned int numExtSems, CUstream
stream)

Signals a set of external semaphore objects.

Parameters

extSemArray
- Set of external semaphores to be signaled

paramsArray
- Array of semaphore parameters

numExtSems
- Number of semaphores to signal

stream
- Stream to enqueue the signal operations in

Returns

CUDA_SUCCESS, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_NOT_SUPPORTED

Description

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

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

If the semaphore object is any one of the following types:
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD,
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32,
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT then signaling the
semaphore will set it to the signaled state.
If the semaphore object is any one of the following types:

```
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE,
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE
```
then the semaphore will be set to the value specified in

```
CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS::params::fence::value.
```

If the semaphore object is of the type

```
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC
```
this API sets

```
CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS::params::nvSciSync::fence
to a value that can be used by subsequent waiters of the same NvSciSync object to order
operations with those currently submitted in stream. Such an update will overwrite previous
contents of

```
CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS::params::nvSciSync::fence.
```

By default, signaling such an external semaphore object causes appropriate memory
synchronization operations to be performed over all external memory objects that
are imported as

```
CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF.
```
This ensures that any subsequent accesses made by other importers of the same set of NvSciBuf
memory object(s) are coherent. These operations can be skipped by specifying the flag

```
CUDA_EXTERNAL_SEMAPHORE_SIGNAL_SKIP_NVSCIBUF_MEMSYNC,
```
which can be used as a performance optimization when data coherency is not required. But specifying this
flag in scenarios where data coherency is required results in undefined behavior. Also, for
semaphore object of the type

```
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC,
```
if the NvSciSyncAttrList used to create the NvSciSyncObj had not set the flags in

```
cuDeviceGetNvSciSyncAttributes
to CUDA_NVSCISYNC_ATTR_SIGNAL,
```
this API will return

```
CUDA_ERROR_NOT_SUPPORTED.
```

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

```
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX,
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX_KMT
```
then the keyed mutex will be released with the key specified in

```
CUDA_EXTERNAL_SEMAPHORE_PARAMS::params::keyedmutex::key.
```

---

**Note:**

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

**See also:**

```
cuImportExternalSemaphore, cuDestroyExternalSemaphore,
cuWaitExternalSemaphoresAsync
```

---
CUresult cuWaitExternalSemaphoresAsync (const CUexternalSemaphore *extSemArray, const CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS *paramsArray, unsigned int numExtSems, CUstream stream)

Waits on a set of external semaphore objects.

Parameters

extSemArray
- External semaphores to be waited on

paramsArray
- Array of semaphore parameters

numExtSems
- Number of semaphores to wait on

stream
- Stream to enqueue the wait operations in

Returns

CUDA_SUCCESS, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_SUPPORTED, CUDA_ERROR_TIMEOUT

Description

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

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

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

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD,
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32,
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT

then waiting on the semaphore will wait until the semaphore reaches the signaled state. The semaphore will then be reset to the unsignaled state. Therefore for every signal operation, there can only be one wait operation.

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

CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE,
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE

then waiting on the semaphore will wait until the value of the semaphore is greater than or equal to CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS::params::fence::value.
If the semaphore object is of the type
`CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC` then, waiting on the semaphore
will wait until the
`CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS::params::nvSciSync::fence` is signaled
by the signaler of the `NvSciSyncObj` that was associated with this semaphore object.
By default, waiting on such an external semaphore object causes appropriate memory
synchronization operations to be performed over all external memory objects that
are imported as `CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF`. This ensures
that any subsequent accesses made by other importers of the same set of NvSciBuf
memory object(s) are coherent. These operations can be skipped by specifying the flag
`CUDA_EXTERNAL_SEMAPHORE_WAIT_SKIP_NVSCIBUF_MEMSYNC`, which can be used as
a performance optimization when data coherency is not required. But specifying this
flag in scenarios where data coherency is required results in undefined behavior. Also, for
semaphore object of the type `CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC`,
if the `NvSciSyncAttrList` used to create the `NvSciSyncObj` had not set the flags in
`cuDeviceGetNvSciSyncAttributes` to CUDA_NVSCISYNC_ATTR_WAIT, this API will return
CUDA_ERROR_NOT_SUPPORTED.

If the semaphore object is any one of the following types:
`CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX`,
`CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX_KMT` then
the keyed mutex will be acquired when it is released with the key specified in
`CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS::params::keyedmutex::key` or until the
timeout specified by
`CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS::params::keyedmutex::timeoutMs` has
lapsed. The timeout interval can either be a finite value specified in milliseconds or an infinite
value. In case an infinite value is specified the timeout never elapses. The windows INFINITE
macro must be used to specify infinite timeout.

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

**See also:**
cuImportExternalSemaphore, cuDestroyExternalSemaphore, 
cuSignalExternalSemaphoresAsync

### 5.17. Stream memory operations

This section describes the stream memory operations of the low-level CUDA driver application
programming interface.
The whole set of operations is disabled by default. Users are required to explicitly enable them, e.g. on Linux by passing the kernel module parameter shown below: modprobe nvidia NVreg_EnableStreamMemOPs=1. There is currently no way to enable these operations on other operating systems.

Users can programatically query whether the device supports these operations with `cuDeviceGetAttribute()` and `CU_DEVICE_ATTRIBUTE_CAN_USE_STREAM_MEM_OPS`.

Support for the `CU_STREAM_WAIT_VALUE_NOR` flag can be queried with `CU_DEVICE_ATTRIBUTE_CAN_USE_STREAM_WAIT_VALUE_NOR`.

Support for the `cuStreamWriteValue64()` and `cuStreamWaitValue64()` functions, as well as for the `CU_STREAM_MEM_OP_WAIT_VALUE_64` and `CU_STREAM_MEM_OP_WRITE_VALUE_64` flags, can be queried with `CU_DEVICE_ATTRIBUTE_CAN_USE_64_BIT_STREAM_MEM_OPS`.

Support for both `CU_STREAM_WAIT_VALUE_FLUSH` and `CU_STREAM_MEM_OP_FLUSH_REMOTE_WRITES` requires dedicated platform hardware features and can be queried with `cuDeviceGetAttribute()` and `CU_DEVICE_ATTRIBUTE_CAN_FLUSH_REMOTE_WRITES`.

Note that all memory pointers passed as parameters to these operations are device pointers. Where necessary a device pointer should be obtained, for example with `cuMemHostGetDevicePointer()`.

None of the operations accepts pointers to managed memory buffers (`cuMemAllocManaged`).

### CUresult cuStreamBatchMemOp (CUstream stream, unsigned int count, CUstreamBatchMemOpParams *paramArray, unsigned int flags)

Batch operations to synchronize the stream via memory operations.

#### Parameters

- **stream**
  - The stream to enqueue the operations in.

- **count**
  - The number of operations in the array. Must be less than 256.

- **paramArray**
  - The types and parameters of the individual operations.

- **flags**
  - Reserved for future expansion; must be 0.

#### Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_SUPPORTED
Description

This is a batch version of cuStreamWaitValue32() and cuStreamWriteValue32(). Batching operations may avoid some performance overhead in both the API call and the device execution versus adding them to the stream in separate API calls. The operations are enqueued in the order they appear in the array.

See CUstreamBatchMemOpType for the full set of supported operations, and cuStreamWaitValue32(), cuStreamWaitValue64(), cuStreamWriteValue32(), and cuStreamWriteValue64() for details of specific operations.

Basic support for this can be queried with cuDeviceGetAttribute() and CU_DEVICE_ATTRIBUTE_CAN_USE_STREAM_MEM_OPS. See related APIs for details on querying support for specific operations.

Note:

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

See also:

cuStreamWaitValue32, cuStreamWaitValue64, cuStreamWriteValue32, cuStreamWriteValue64, cuMemHostRegister

CUresult cuStreamWaitValue32 (CUstream stream, CUdeviceptr addr, cuuint32_t value, unsigned int flags)

Wait on a memory location.

Parameters

stream
The stream to synchronize on the memory location.

addr
The memory location to wait on.

value
The value to compare with the memory location.

flags
See CUstreamWaitValue_flags.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_SUPPORTED
Description
Enqueues a synchronization of the stream on the given memory location. Work ordered after the operation will block until the given condition on the memory is satisfied. By default, the condition is to wait for (int32_t)(*addr - value) >= 0, a cyclic greater-or-equal. Other condition types can be specified via flags.

If the memory was registered via cuMemHostRegister(), the device pointer should be obtained with cuMemHostGetDevicePointer(). This function cannot be used with managed memory (cuMemAllocManaged).

Support for this can be queried with cuDeviceGetAttribute() and CU_DEVICE_ATTRIBUTE_CAN_USE_STREAM_MEM_OPS.

Support for CU_STREAM_WAIT_VALUE_NOR can be queried with cuDeviceGetAttribute() and CU_DEVICE_ATTRIBUTE_CAN_USE_STREAM_WAIT_VALUE_NOR.

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

See also:
cuStreamWaitValue64, cuStreamWriteValue32, cuStreamWriteValue64
cuStreamBatchMemOp, cuMemHostRegister, cuStreamWaitEvent

CUresult cuStreamWaitValue64 (CUstream stream, CUdeviceptr addr, cuuint64_t value, unsigned int flags)
Wait on a memory location.

Parameters
stream
The stream to synchronize on the memory location.

addr
The memory location to wait on.

value
The value to compare with the memory location.

flags
See CUstreamWaitValue_flags.
Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_SUPPORTED

Description
Enqueues a synchronization of the stream on the given memory location. Work ordered after
the operation will block until the given condition on the memory is satisfied. By default, the
condition is to wait for \([\text{int64}_t(*addr - value)] \geq 0\), a cyclic greater-or-equal. Other condition
types can be specified via flags.

If the memory was registered via cuMemHostRegister(), the device pointer should be obtained
with cuMemHostGetDevicePointer().

Support for this can be queried with cuDeviceGetAttribute() and
CU_DEVICE_ATTRIBUTE_CAN_USE_64_BIT_STREAM_MEM_OPS.

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

See also:
cuStreamWaitValue32, cuStreamWriteValue32, cuStreamWriteValue64,
cuStreamBatchMemOp, cuMemHostRegister, cuStreamWaitEvent

CUresult cuStreamWriteValue32 (CUstream stream,
CUdeviceptr addr, cuuint32_t value, unsigned int flags)
Write a value to memory.

Parameters
stream
The stream to do the write in.
addr
The device address to write to.
value
The value to write.
flags
See CUstreamWriteValue_flags.

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_SUPPORTED
Description

Write a value to memory. Unless the `CUDA_STREAM_WRITE_VALUE_NO_MEMORY_BARRIER` flag is passed, the write is preceded by a system-wide memory fence, equivalent to a `__threadfence_system()` but scoped to the stream rather than a CUDA thread.

If the memory was registered via `cuMemHostRegister[ ]`, the device pointer should be obtained with `cuMemHostGetDevicePointer[ ]`. This function cannot be used with managed memory (`cuMemAllocManaged[ ]`).

Support for this can be queried with `cuDeviceGetAttribute[ ]` and `CU_DEVICE_ATTRIBUTE_CAN_USE_STREAM_MEM_OPS`.

Note:

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

See also:

`cuStreamWriteValue64`, `cuStreamWaitValue32`, `cuStreamWaitValue64`, `cuStreamBatchMemOp`, `cuMemHostRegister`, `cuEventRecord`

```c
CUresult cuStreamWriteValue64 (CUstream stream,
                                CUdeviceptr addr, cuuint64_t value, unsigned int flags)
```

Write a value to memory.

Parameters

- **stream**
  - The stream to do the write in.
- **addr**
  - The device address to write to.
- **value**
  - The value to write.
- **flags**
  - See `CUstreamWriteValue_flags`.

Returns

`CUDA_SUCCESS`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_NOT_SUPPORTED`
Description

Write a value to memory. Unless the `CU_STREAM_WRITE_VALUE_NO_MEMORY_BARRIER` flag is passed, the write is preceded by a system-wide memory fence, equivalent to a `__threadfence_system()` but scoped to the stream rather than a CUDA thread.

If the memory was registered via `cuMemHostRegister()`, the device pointer should be obtained with `cuMemHostGetDevicePointer()`.

Support for this can be queried with `cuDeviceGetAttribute()` and `CU_DEVICE_ATTRIBUTE_CAN_USE_64_BIT_STREAM_MEM_OPS`.

Note:

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

See also:

`cuStreamWriteValue32`, `cuStreamWaitValue32`, `cuStreamWaitValue64`, `cuStreamBatchMemOp`, `cuMemHostRegister`, `cuEventRecord`

5.18. Execution Control

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

`CUresult cuFuncGetAttribute (int *pi, CUfunction_attribute attrib, CUfunction hfunc)`

Returns information about a function.

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

Description

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.
- **CU_FUNC_CACHE_MODE_CA**: The attribute to indicate whether the function has been compiled with user specified option "-Xptxas --dlcm=ca" set.
- **CU_FUNC_ATTRIBUTE_MAX DYNAMIC SHARED SIZE BYTES**: The maximum size in bytes of dynamically-allocated shared memory.
- **CU_FUNC_ATTRIBUTE_PREFERRED_SHARED_MEMORY_CARVEOUT**: Preferred shared memory-L1 cache split ratio in percent of total shared memory.

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.
See also:
cuCtxGetCacheConfig, cuCtxSetCacheConfig, cuFuncGetCacheConfig, cuLaunchKernel,
cudaFuncGetAttributes cudaFuncSetAttribute

CUresult cuFuncSetAttribute (CUfunction hfunc, CUfunction_attribute attrib, int value)
Sets information about a function.

Parameters
hfunc
   - Function to query attribute of
attrib
   - Attribute requested
value
   - The value to set

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_INVALID_VALUE

Description
This call sets the value of a specified attribute attrib on the kernel given by hfunc to an
integer value specified by val. This function returns CUDA_SUCCESS if the new value of
the attribute could be successfully set. If the set fails, this call will return an error. Not all
attributes can have values set. Attempting to set a value on a read-only attribute will result in
an error (CUDA_ERROR_INVALID_VALUE).

Supported attributes for the cuFuncSetAttribute call are:

- **CU_FUNC_ATTRIBUTE_MAX_DYNAMIC_SHARED_SIZE_BYTES**: This maximum size in
  bytes of dynamically-allocated shared memory. The value should contain the requested
  maximum size of dynamically-allocated shared memory. The sum of this value and the
  function attribute **CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES** cannot exceed the device
  attribute **CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK_OPTIN**. The
  maximal size of requestable dynamic shared memory may differ by GPU architecture.

- **CU_FUNC_ATTRIBUTE_PREFERRED_SHARED_MEMORY_CARVEOUT**: On devices
  where the L1 cache and shared memory use the same hardware resources, this sets
  the shared memory carveout preference, in percent of the total shared memory. See
  **CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_MULTIPROCESSOR**. This is only a
  hint, and the driver can choose a different ratio if required to execute the function.
Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuCtxGetCacheConfig, cuCtxSetCacheConfig, cuFuncSetCacheConfig, cuLaunchKernel, cudaFuncGetAttributes, cudaFuncSetAttribute

CUresult cuFuncSetCacheConfig (CUfunction hfunc, CUfunc_cache config)
Sets the preferred cache configuration for a device function.

Parameters
hfunc
- Kernel to configure cache for
config
- Requested cache configuration

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Description
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
CU result `cuFuncSetSharedMemConfig` *(CUfunction hfunc, CUsharedconfig config)*

Sets the shared memory configuration for a device function.

### Parameters

- `hfunc` - kernel to be given a shared memory config
- `config` - requested shared memory configuration

### Returns

- `CUDA_SUCCESS`
- `CUDA_ERROR_INVALID_VALUE`
- `CUDA_ERROR_DEINITIALIZED`
- `CUDA_ERROR_NOT_INITIALIZED`
- `CUDA_ERROR_INVALID_CONTEXT`

### Description

On devices with configurable shared memory banks, this function will force all subsequent launches of the specified device function to have the given shared memory bank size configuration. On any given launch of the function, the shared memory configuration of the device will be temporarily changed if needed to suit the function’s preferred configuration. Changes in shared memory configuration between subsequent launches of functions, may introduce a device side synchronization point.

Any per-function setting of shared memory bank size set via `cuFuncSetSharedMemConfig` will override the context wide setting set with `cuCtxSetSharedMemConfig`.

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

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

The supported bank configurations are:

- **CU_FUNC_CACHE_PREFER_EQUAL**: prefer equal sized L1 cache and shared memory

---

**Note:**

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

See also:

- `cuCtxGetCacheConfig`
- `cuCtxSetCacheConfig`
- `cuFuncGetAttribute`
- `cuLaunchKernel`
- `cudaFuncSetCacheConfig`
- **CU_SHARED_MEM_CONFIG_DEFAULT_BANK_SIZE**: use the context’s shared memory configuration when launching this function.

- **CU_SHARED_MEM_CONFIG_FOUR_BYTE_BANK_SIZE**: set shared memory bank width to be natively four bytes when launching this function.

- **CU_SHARED_MEM_CONFIG_EIGHT_BYTE_BANK_SIZE**: set shared memory bank width to be natively eight bytes when launching this function.

**Note:**

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

See also:

cuCtxGetCacheConfig, cuCtxSetCacheConfig, cuCtxGetSharedMemConfig, cuCtxSetSharedMemConfig, cuFuncGetAttribute, cuLaunchKernel, cudaFuncSetSharedMemConfig

**CUresult cuLaunchCooperativeKernel (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)**

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

**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

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 COOPERATIVE_LAUNCH_TOO_LARGE,
CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Description
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.

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

The total number of blocks launched cannot exceed the maximum number of blocks
per multiprocessor as returned by cuOccupancyMaxActiveBlocksPerMultiprocessor
(or cuOccupancyMaxActiveBlocksPerMultiprocessorWithFlags) times
the number of multiprocessors as specified by the device attribute
CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT.

The kernel cannot make use of CUDA dynamic parallelism.

Kernel parameters must 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.

Calling cuLaunchCooperativeKernel() sets persistent function state that is the same as
function state set through cuLaunchKernel API

When the kernel $f$ is launched via cuLaunchCooperativeKernel(), the previous block shape,
shared size and parameter info associated with $f$ is overwritten.
Note that to use `cuLaunchCooperativeKernel()`, the kernel function 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 `cuLaunchCooperativeKernel()` will return `CUDA_ERROR_INVALID_IMAGE`.

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

See also:
- `cuCtxGetCacheConfig`, `cuCtxSetCacheConfig`, `cuFuncSetCacheConfig`, `cuFuncGetAttribute`, `cuLaunchCooperativeKernelMultiDevice`, `cudaLaunchCooperativeKernel`

```c
CUresult cuLaunchCooperativeKernelMultiDevice (CUDA_LAUNCH_PARAMS *launchParamsList, unsigned int numDevices, unsigned int flags)
```

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

**Parameters**

- `launchParamsList` - List of launch parameters, one per device
- `numDevices` - Size of the `launchParamsList` array
- `flags` - Flags to control launch behavior

**Returns**

- `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_COOPERATIVE_LAUNCH_TOO_LARGE`, `CUDA_ERROR_SHARED_OBJECT_INIT_FAILED`

**Description**

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

No two kernels can be launched on the same device. All the devices targeted by this multi-device launch must be identical. All devices must have a non-zero value for the device attribute `CU_DEVICE_ATTRIBUTE_COOPERATIVE_MULTI_DEVICE_LAUNCH`.

All kernels launched must be identical with respect to the compiled code. Note that any __device__, __constant__ or __managed__ variables present in the module that owns the kernel launched on each device, are independently instantiated on every device. It is the application’s responsibility to ensure these variables are initialized and used appropriately.

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

The streams used to launch these kernels must have been created via either `cuStreamCreate` or `cuStreamCreateWithPriority`. The NULL stream or `CU_STREAM_LEGACY` or `CU_STREAM_PER_THREAD` cannot be used.

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

The kernels cannot make use of CUDA dynamic parallelism.

The `CUDA_LAUNCH_PARAMS` structure is defined as:

```c
typedef struct CUDA_LAUNCH_PARAMS_st {
    CUfunction function;
    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;
} CUDA_LAUNCH_PARAMS;
```

where:

- **CUDA_LAUNCH_PARAMS::function** specifies the kernel to be launched. All functions must be identical with respect to the compiled code.
- **CUDA_LAUNCH_PARAMS::gridDimX** is the width of the grid in blocks. This must match across all kernels launched.
- **CUDA_LAUNCH_PARAMS::gridDimY** is the height of the grid in blocks. This must match across all kernels launched.
- **CUDA_LAUNCH_PARAMS::gridDimZ** is the depth of the grid in blocks. This must match across all kernels launched.

- **CUDA_LAUNCH_PARAMS::blockDimX** is the X dimension of each thread block. This must match across all kernels launched.

- **CUDA_LAUNCH_PARAMS::blockDimX** is the Y dimension of each thread block. This must match across all kernels launched.

- **CUDA_LAUNCH_PARAMS::blockDimZ** is the Z dimension of each thread block. This must match across all kernels launched.

- **CUDA_LAUNCH_PARAMS::sharedMemBytes** is the dynamic shared-memory size per thread block in bytes. This must match across all kernels launched.

- **CUDA_LAUNCH_PARAMS::hStream** is the handle to the stream to perform the launch in. This cannot be the NULL stream or CU_STREAM_LEGACY or CU_STREAM_PER_THREAD. The CUDA context associated with this stream must match that associated with **CUDA_LAUNCH_PARAMS::function**.

- **CUDA_LAUNCH_PARAMS::kernelParams** is an array of pointers to kernel parameters. If **CUDA_LAUNCH_PARAMS::function** has N parameters, then **CUDA_LAUNCH_PARAMS::kernelParams** needs to be an array of N pointers. Each of **CUDA_LAUNCH_PARAMS::kernelParams[0]** through **CUDA_LAUNCH_PARAMS::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.

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

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

Calling **cuLaunchCooperativeKernelMultiDevice[]** sets persistent function state that is the same as function state set through **cuLaunchKernel** API when called individually for each element in launchParamsList.

When kernels are launched via **cuLaunchCooperativeKernelMultiDevice[]**, the previous block shape, shared size and parameter info associated with each **CUDA_LAUNCH_PARAMS::function** in launchParamsList is overwritten.
Note that to use `cuLaunchCooperativeKernelMultiDevice()`, the kernels 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 `cuLaunchCooperativeKernelMultiDevice()` will return `CUDA_ERROR_INVALID_IMAGE`.

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

**See also:**
- `cuCtxGetCacheConfig`, `cuCtxSetCacheConfig`, `cuFuncSetCacheConfig`, `cuFuncGetAttribute`
- `cuLaunchCooperativeKernel`, `cudaLaunchCooperativeKernelMultiDevice`

### CUresult cuLaunchHostFunc (CUstream hStream, CUhostFn fn, void *userData)

Enqueues a host function call in a stream.

**Parameters**

- **hStream**
  - Stream to enqueue function call in
- **fn**
  - The function to call once preceding stream operations are complete
- **userData**
  - User-specified data to be passed to the function

**Returns**

- `CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_HANDLE`, `CUDA_ERROR_NOT_SUPPORTED`

**Description**

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

The host function must not make any CUDA API calls. Attempting to use a CUDA API may result in `CUDA_ERROR_NOT_PERMITTED`, but this is not required. The host function must not perform any synchronization that may depend on outstanding CUDA work not mandated to run earlier. Host functions without a mandated order (such as in independent streams) execute in undefined order and may be serialized.
For the purposes of Unified Memory, execution makes a number of guarantees:

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

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

**Note:**

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

See also:

- `cuStreamCreate`, `cuStreamQuery`, `cuStreamSynchronize`, `cuStreamWaitEvent`,
- `cuStreamDestroy`, `cuMemAllocManaged`, `cuStreamAttachMemAsync`, `cuStreamAddCallback`
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.

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
Description

Invokes the kernel \( f \) on a \( \text{gridDimX} \times \text{gridDimY} \times \text{gridDimZ} \) grid of blocks. Each block contains \( \text{blockDimX} \times \text{blockDimY} \times \text{blockDimZ} \) threads.

\( \text{sharedMemBytes} \) sets the amount of dynamic shared memory that will be available to each thread block.

Kernel parameters to \( f \) can be specified in one of two ways:

1) Kernel parameters can be specified via \( \text{kernelParams} \). If \( f \) has \( N \) parameters, then \( \text{kernelParams} \) needs to be an array of \( N \) pointers. Each of \( \text{kernelParams}[0] \) through \( \text{kernelParams}[N-1] \) 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 \( \text{extra} \) 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 \( \text{extra} \) parameter in this manner:

```c
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 \( \text{extra} \) parameter exists to allow \( \text{cuLaunchKernel} \) to take additional less commonly used arguments. \( \text{extra} \) 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 \( \text{NULL} \) or \( \text{CU_LAUNCH_PARAM_END} \).

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

The error \( \text{CUDA_ERROR_INVALID_VALUE} \) will be returned if kernel parameters are specified with both \( \text{kernelParams} \) and \( \text{extra} \) (i.e. both \( \text{kernelParams} \) and \( \text{extra} \) are non-\( \text{NULL} \)).

Calling \( \text{cuLaunchKernel}[] \) invalidates the persistent function state set through the following deprecated APIs: \( \text{cuFuncSetBlockShape}[] \), \( \text{cuFuncSetSharedSize}[] \), \( \text{cuParamSetSize}[] \), \( \text{cuParamSeti}[] \), \( \text{cuParamSetf}[] \), \( \text{cuParamSetv}[] \).
Note that to use `cudaLaunchKernel()`, the kernel 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 `cudaLaunchKernel()` will return `CUDA_ERROR_INVALID_IMAGE`.

**Note:**
- This function uses standard [default stream](#) semantics.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:
- `cuCtxGetCacheConfig`, `cuCtxSetCacheConfig`, `cuFuncSetCacheConfig`, `cuFuncGetAttribute`, `cudaLaunchKernel`

### 5.19. Execution Control [DEPRECATED]

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

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

Sets the block-dimensions for the function.

**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`
Description

Deprecated

Specifies the x, y, and z dimensions of the thread blocks that are created when the kernel given by hfunc is launched.

Note:

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

See also:


CUresult cuFuncSetSharedSize (CUfunction hfunc, unsigned int bytes)

Sets the dynamic shared-memory size for the function.

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

Description

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.

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

CUresult cuLaunch (CUfunction f)
Launches a CUDA function.

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

Description
Deprecated
Invokes the kernel f on a 1 x 1 x 1 grid of blocks. The block contains the number of threads specified by a previous call to cuFuncSetBlockShape().

The block shape, dynamic shared memory size, and parameter information must be set using cuFuncSetBlockShape(), cuFuncSetSharedSize(), cuParamSetSize(), cuParamSeti(), cuParamSetf(), and cuParamSetv() prior to calling this function.

Launching a function via cuLaunchKernel() invalidates the function’s block shape, dynamic shared memory size, and parameter information. After launching via cuLaunchKernel, this state must be re-initialized prior to calling this function. Failure to do so results in undefined behavior.

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

Launches a CUDA function.

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

Description

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().

The block shape, dynamic shared memory size, and parameter information must be set using cuFuncSetBlockShape(), cuFuncSetSharedSize(), cuParamSetSize(), cuParamSeti(), cuParamSetf(), and cuParamSetv() prior to calling this function.

Launching a function via cuLaunchKernel() invalidates the function’s block shape, dynamic shared memory size, and parameter information. After launching via cuLaunchKernel, this state must be re-initialized prior to calling this function. Failure to do so results in undefined behavior.

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
CUresult cuLaunchGridAsync (CUfunction f, int grid_width, int grid_height, CUstream hStream)
Launches a CUDA function.

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

Description

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().

The block shape, dynamic shared memory size, and parameter information must be set
using cuFuncSetBlockShape(), cuFuncSetSharedSize(), cuParamSetSize(), cuParamSeti(),
cuParamSetf(), and cuParamSetv() prior to calling this function.

Launching a function via cuLaunchKernel() invalidates the function’s block shape, dynamic
shared memory size, and parameter information. After launching via cuLaunchKernel, this
state must be re-initialized prior to calling this function. Failure to do so results in undefined
behavior.

Note:

In certain cases where cubins are created with no ABI (i.e., using ptxas --abi-
compile no), this function may serialize kernel launches. The CUDA driver retains
asynchronous behavior by growing the per-thread stack as needed per launch and not shrinking it afterwards.

- This function uses standard default stream semantics.
- 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

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

Adds a floating-point parameter to the function’s argument list.

**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

**Description**

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.

**Note:**

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

See also:
cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSeti, cuParamSetv, cuLaunch, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel
CUresult cuParamSeti (CUfunction hfunc, int offset, unsigned int value)

Adds an integer parameter to the function’s argument list.

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

Description

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.

**Note:**

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

See also:

cuParamSetSize, cuLaunchGridAsync, cuLaunchKernel, cuLaunchGrid, cuLaunch, cuLaunchKernel, cuFuncGetAttribute, cuFuncSetSharedSize, cuFuncSetBlockShape, cuParamSetf, cuParamSetv

CUresult cuParamSetSize (CUfunction hfunc, unsigned int numbytes)

Sets the parameter size for the function.

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

Description
Deprecated
Sets through numbytes the total size in bytes needed by the function parameters of the kernel corresponding to hfunc.

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

See also:
cuParamSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetf, cuParamSeti, cuParamSetv, cuLaunch, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel

CUresult cuParamSetTexRef (CUfunction hfunc, int texunit, CUtexref hTexRef)
Adds a texture-reference to the function’s argument list.

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

Description
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`.

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

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

Adds arbitrary data to the function’s argument list.

**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

**Description**

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.

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

**See also:**

cuFuncSetBlockSize, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSeti, cuLaunch, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel
5.20. Graph Management

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

CUresult cuGraphAddChildGraphNode (CUgraphNode *phGraphNode, CUgraph hGraph, const CUgraphNode *dependencies, size_t numDependencies, CUgraph childGraph)

Creates a child graph node and adds it to a graph.

Parameters

- **phGraphNode**: Returns newly created node
- **hGraph**: Graph to which to add the node
- **dependencies**: Dependencies of the node
- **numDependencies**: Number of dependencies
- **childGraph**: The graph to clone into this node

Returns

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE.

Description

Creates a new node which executes an embedded graph, and adds it to hGraph with numDependencies dependencies specified via dependencies. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. dependencies may not have any duplicate entries. A handle to the new node will be returned in phGraphNode.

The node executes an embedded child graph. The child graph is cloned in this call.

**Note:**

- Graph objects are not threadsafe. More here.
Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuGraphChildGraphNodeGetGraph
- cuGraphCreate
- cuGraphDestroyNode
- cuGraphAddEmptyNode
- cuGraphAddKernelNode
- cuGraphAddHostNode
- cuGraphAddMemcpyNode
- cuGraphAddMemsetNode
- cuGraphClone

## CUresult cuGraphAddDependencies (CUgraph hGraph, const CUgraphNode *from, const CUgraphNode *to, size_t numDependencies)

Adds dependency edges to a graph.

### Parameters

- **hGraph**:
  - Graph to which dependencies are added
- **from**:
  - Array of nodes that provide the dependencies
- **to**:
  - Array of dependent nodes
- **numDependencies**:
  - Number of dependencies to be added

### Returns

- CUDA_SUCCESS
- CUDA_ERROR_INVALID_VALUE

### Description

The number of dependencies to be added is defined by `numDependencies`. Elements in `from` and `to` at corresponding indices define a dependency. Each node in `from` and `to` must belong to `hGraph`.

If `numDependencies` is 0, elements in `from` and `to` will be ignored. Specifying an existing dependency will return an error.

### Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:
creating an empty node and adds it to a graph.

Parameters

- **phGraphNode**: Returns newly created node
- **g**: Graph to which to add the node
- **dependencies**: Dependencies of the node
- **numDependencies**: Number of dependencies

Returns

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
  CUDA_ERROR_INVALID_VALUE.

Description

Creates a new node which performs no operation, and adds it to g with
numDependencies dependencies specified via dependencies. It is possible for
numDependencies to be 0, in which case the node will be placed at the root of the graph.
dependencies may not have any duplicate entries. A handle to the new node will be returned
in phGraphNode.

An empty node performs no operation during execution, but can be used for transitive
ordering. For example, a phased execution graph with 2 groups of n nodes with a barrier
between them can be represented using an empty node and 2*n dependency edges, rather
than no empty node and n^2 dependency edges.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuGraphRemoveDependencies, cuGraphGetEdges, cuGraphNodeGetDependencies,
cuGraphNodeGetDependentNodes
CUresult cuGraphAddEventRecordNode
(CUgraphNode *phGraphNode, CUgraph hGraph, const CUgraphNode *dependencies, size_t numDependencies, CUevent event)

Creates an event record node and adds it to a graph.

Parameters

phGraphNode
- Returns newly created node

hGraph
- Graph to which to add the node

dependencies
- Dependencies of the node

numDependencies
- Number of dependencies

event
- Event for the node

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_NOT_SUPPORTED, CUDA_ERROR_INVALID_VALUE

Description

Creates a new event record node and adds it to hGraph with numDependencies dependencies specified via dependencies and arguments specified in params. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. dependencies may not have any duplicate entries. A handle to the new node will be returned in phGraphNode.

Each launch of the graph will record event to capture execution of the node’s dependencies.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:
CUresult cuGraphAddEventWaitNode (CUgraphNode *phGraphNode, CUgraph hGraph, const CUgraphNode *dependencies, size_t numDependencies, CUevent event)

Creates an event wait node and adds it to a graph.

Parameters

- **phGraphNode** - Returns newly created node
- **hGraph** - Graph to which to add the node
- **dependencies** - Dependencies of the node
- **numDependencies** - Number of dependencies
- **event** - Event for the node

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_NOT_SUPPORTED, CUDA_ERROR_INVALID_VALUE

Description

Creates a new event wait node and adds it to hGraph with numDependencies dependencies specified via dependencies and arguments specified in params. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. dependencies may not have any duplicate entries. A handle to the new node will be returned in phGraphNode.

The graph node will wait for all work captured in event. See cuEventRecord() for details on what is captured by an event. event may be from a different context or device than the launch stream.

Note:

- Graph objects are not threadsafe. More here.
CUresult cuGraphAddHostNode (CUgraphNode *phGraphNode, CUgraph hGraph, const CUgraphNode *dependencies, size_t numDependencies, const CUDA_HOST_NODE_PARAMS *nodeParams)

Creates a host execution node and adds it to a graph.

**Parameters**

- **phGraphNode**
  - Returns newly created node
- **hGraph**
  - Graph to which to add the node
- **dependencies**
  - Dependencies of the node
- **numDependencies**
  - Number of dependencies
- **nodeParams**
  - Parameters for the host node

**Returns**

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_NOT_SUPPORTED, CUDA_ERROR_INVALID_VALUE

**Description**

Creates a new CPU execution node and adds it to hGraph with numDependencies dependencies specified via dependencies and arguments specified in nodeParams. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. dependencies may not have any duplicate entries. A handle to the new node will be returned in phGraphNode.

When the graph is launched, the node will invoke the specified CPU function. Host nodes are not supported under MPS with pre-Volta GPUs.
Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuLaunchHostFunc, cuGraphHostNodeGetParams, cuGraphHostNodeSetParams, cuGraphCreate, cuGraphDestroyNode, cuGraphAddChildGraphNode, cuGraphAddEmptyNode, cuGraphAddKernelNode, cuGraphAddMemcpyNode, cuGraphAddMemsetNode

CUresult cuGraphAddKernelNode (CUgraphNode *phGraphNode, CUgraph hGraph, const CUgraphNode *dependencies, size_t numDependencies, const CUDA_KERNEL_NODE_PARAMS *nodeParams)

Creates a kernel execution node and adds it to a graph.

Parameters

phGraphNode
  - Returns newly created node

hGraph
  - Graph to which to add the node
dependencies
  - Dependencies of the node
numDependencies
  - Number of dependencies
nodeParams
  - Parameters for the GPU execution node

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE

Description

Creates a new kernel execution node and adds it to hGraph with numDependencies dependencies specified via dependencies and arguments specified in nodeParams. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. dependencies may not have any duplicate entries. A handle to the new node will be returned in phGraphNode.
The **CUDA_KERNEL_NODE_PARAMS** structure is defined as:

```c
typedef struct CUDA_KERNEL_NODE_PARAMS_st {
    CUfunction func;
    unsigned int gridDimX;
    unsigned int gridDimY;
    unsigned int gridDimZ;
    unsigned int blockDimX;
    unsigned int blockDimY;
    unsigned int blockDimZ;
    unsigned int sharedMemBytes;
    void **kernelParams;
    void **extra;
} CUDA_KERNEL_NODE_PARAMS;
```

When the graph is launched, the node will invoke kernel `func` 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.

Kernel parameters to `func` can be specified in one of two ways:

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

2) Kernel parameters for non-cooperative kernels can also be packaged by the application into a single buffer that is passed in via `extra`. This places the burden on the application of knowing each kernel parameter’s size and alignment/padding within the buffer. The `extra` parameter exists to allow this function to take additional less commonly used arguments. `extra` specifies a list of names of extra settings and their corresponding values. Each extra setting name is immediately followed by the corresponding value. The list must be terminated with either NULL or `CU_LAUNCH_PARAM_END`.

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

The error **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). **CUDA_ERROR_INVALID_VALUE** will be returned if `extra` is used for a cooperative kernel.
The `kernelParams` or extra array, as well as the argument values it points to, are copied during this call.

**Note:**
Kernels launched using graphs must not use texture and surface references. Reading or writing through any texture or surface reference is undefined behavior. This restriction does not apply to texture and surface objects.

**Note:**
- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.

See also:
- `cuLaunchKernel`, `cuLaunchCooperativeKernel`, `cuGraphKernelNodeGetParams`, `cuGraphKernelNodeSetParams`, `cuGraphCreate`, `cuGraphDestroyNode`, `cuGraphAddChildGraphNode`, `cuGraphAddEmptyNode`, `cuGraphAddHostNode`, `cuGraphAddMemcpyNode`, `cuGraphAddMemsetNode`

```c
CUresult cuGraphAddMemcpyNode (CUgraphNode *phGraphNode, CUgraph hGraph, const CUgraphNode *dependencies, size_t numDependencies, const CUDA_MEMCPY3D *copyParams, CUcontext ctx)
```

Creates a memcpy node and adds it to a graph.

**Parameters**
- `phGraphNode`
  - Returns newly created node
- `hGraph`
  - Graph to which to add the node
- `dependencies`
  - Dependencies of the node
- `numDependencies`
  - Number of dependencies
- `copyParams`
  - Parameters for the memory copy
- `ctx`
  - Context on which to run the node
Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE

Description
Creates a new memcpy node and adds it to hGraph with numDependencies dependencies specified via dependencies. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. dependencies may not have any duplicate entries. A handle to the new node will be returned in phGraphNode.

When the graph is launched, the node will perform the memcpy described by copyParams. See cuMemcpy3D() for a description of the structure and its restrictions.

Memcpy nodes have some additional restrictions with regards to managed memory, if the system contains at least one device which has a zero value for the device attribute CU_DEVICE_ATTRIBUTE_CONCURRENT_MANAGED_ACCESS. If one or more of the operands refer to managed memory, then using the memory type CU_MEMORYTYPE_UNIFIED is disallowed for those operand(s). The managed memory will be treated as residing on either the host or the device, depending on which memory type is specified.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuMemcpy3D, cuGraphMemcpyNodeGetParams, cuGraphMemcpyNodeSetParams, cuGraphCreate, cuGraphDestroyNode, cuGraphAddChildGraphNode, cuGraphAddEmptyNode, cuGraphAddKernelNode, cuGraphAddHostNode, cuGraphAddMemsetNode
CUresult cuGraphAddMemsetNode (CUgraphNode *phGraphNode, CUgraph hGraph, const CUgraphNode *dependencies, size_t numDependencies, const CUDA_MEMSET_NODE_PARAMS *memsetParams, CUcontext ctx)

Creates a memset node and adds it to a graph.

Parameters

- **phGraphNode**
  - Returns newly created node

- **hGraph**
  - Graph to which to add the node

- **dependencies**
  - Dependencies of the node

- **numDependencies**
  - Number of dependencies

- **memsetParams**
  - Parameters for the memory set

- **ctx**
  - Context on which to run the node

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_CONTEXT

Description

Creates a new memset node and adds it to hGraph with numDependencies dependencies specified via dependencies. It is possible for numDependencies to be 0, in which case the node will be placed at the root of the graph. dependencies may not have any duplicate entries. A handle to the new node will be returned in phGraphNode.

The element size must be 1, 2, or 4 bytes. When the graph is launched, the node will perform the memset described by memsetParams.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
See also:
cuMemsetD2D32, cuGraphMemsetNodeGetParams, cuGraphMemsetNodeSetParams, cuGraphCreate, cuGraphDestroyNode, cuGraphAddChildGraphNode, cuGraphAddEmptyNode, cuGraphAddKernelNode, cuGraphAddHostNode, cuGraphAddMemcpyNode

CUresult cuGraphChildGraphNodeGetGraph (CUgraphNode hNode, CUgraph *phGraph)
Gets a handle to the embedded graph of a child graph node.

Parameters
hNode
- Node to get the embedded graph for
phGraph
- Location to store a handle to the graph

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE.

Description
Gets a handle to the embedded graph in a child graph node. This call does not clone the graph. Changes to the graph will be reflected in the node, and the node retains ownership of the graph.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuGraphAddChildGraphNode, cuGraphNodeFindInClone
CUresult cuGraphClone (CUgraph *phGraphClone, 
CUgraph originalGraph)

Clones a graph.

Parameters

phGraphClone
- Returns newly created cloned graph

originalGraph
- Graph to clone

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Description

This function creates a copy of originalGraph and returns it in phGraphClone. All parameters are copied into the cloned graph. The original graph may be modified after this call without affecting the clone.

Child graph nodes in the original graph are recursively copied into the clone.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphCreate, cuGraphNodeFindInClone

CUresult cuGraphCreate (CUgraph *phGraph, 
unsigned int flags)

Creates a graph.

Parameters

phGraph
- Returns newly created graph

flags
- Graph creation flags, must be 0
Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Description
Creates an empty graph, which is returned via phGraph.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuGraphAddChildGraphNode, cuGraphAddEmptyNode, cuGraphAddKernelNode, cuGraphAddHostNode, cuGraphAddMemcpyNode, cuGraphAddMemsetNode, cuGraphInstantiate, cuGraphDestroy, cuGraphGetNodes, cuGraphGetRootNodes, cuGraphGetEdges, cuGraphClone

CUresult cuGraphDestroy (CUgraph hGraph)
Destroys a graph.

Parameters
hGraph
- Graph to destroy

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE

Description
Destroys the graph specified by hGraph, as well as all of its nodes.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuGraphCreate

CUresult cuGraphDestroyNode (CUgraphNode hNode)
Remove a node from the graph.

Parameters
hNode
- Node to remove

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE

Description
Removes hNode from its graph. This operation also severs any dependencies of other nodes on hNode and vice versa.

Note:
‣ Graph objects are not threadsafe. More here.
‣ Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuGraphAddChildGraphNode, cuGraphAddEmptyNode, cuGraphAddKernelNode, cuGraphAddHostNode, cuGraphAddMemcpyNode, cuGraphAddMemsetNode

CUresult cuGraphEventRecordNodeGetEvent (CUgraphNode hNode, CUevent *event_out)
Returns the event associated with an event record node.

Parameters
hNode
- Node to get the event for
event_out
- Pointer to return the event

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE
Description

Returns the event of event record node hNode in event_out.

Note:

- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuGraphAddEventRecordNode, cuGraphEventRecordNodeGetEvent, cuGraphEventWaitNodeGetEvent, cuEventRecord, cuStreamWaitEvent

CUresult cuGraphEventRecordNodeSetEvent (CUgraphNode hNode, CUevent event)

Sets an event record node’s event.

Parameters

- **hNode**
  - Node to set the event for
- **event**
  - Event to use

Returns

- CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY

Description

Sets the event of event record node hNode to event.

Note:

- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuGraphAddEventRecordNode, cuGraphEventRecordNodeGetEvent, cuGraphEventWaitNodeGetEvent, cuEventRecord, cuStreamWaitEvent
CUresult cuGraphEventWaitNodeGetEvent (CUgraphNode hNode, CUevent *event_out)
Returns the event associated with an event wait node.

Parameters

hNode
- Node to get the event for

event_out
- Pointer to return the event

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_VALUE

Description

Returns the event of event wait node hNode in event_out.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphAddEventWaitNode, cuGraphEventWaitNodeSetEvent,
cuGraphEventRecordNodeGetEvent, cuEventRecord, cuStreamWaitEvent

CUresult cuGraphEventWaitNodeSetEvent (CUgraphNode hNode, CUevent event)
Sets an event wait node’s event.

Parameters

hNode
- Node to set the event for

event
- Event to use
Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY

Description
Sets the event of event wait node hNode to event.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuGraphAddEventWaitNode, cuGraphEventWaitNodeGetEvent,
cuGraphEventRecordNodeSetEvent, cuEventRecord, cuStreamWaitEvent

CUresult cuGraphExecChildGraphNodeSetParams
(CUgraphExec hGraphExec, CUgraphNode hNode, CUgraph childGraph)
Updates node parameters in the child graph node in the given graphExec.

Parameters
hGraphExec
- The executable graph in which to set the specified node
hNode
- Host node from the graph which was used to instantiate graphExec
childGraph
- The graph supplying the updated parameters

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE.

Description
Updates the work represented by hNode in hGraphExec as though the nodes contained in hNode's graph had the parameters contained in childGraph's nodes at instantiation. hNode must remain in the graph which was used to instantiate hGraphExec. Changed edges to and from hNode are ignored.
The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `hNode` is also not modified by this call.

The topology of `childGraph`, as well as the node insertion order, must match that of the graph contained in `hNode`. See `cuGraphExecUpdate()` for a list of restrictions on what can be updated in an instantiated graph. The update is recursive, so child graph nodes contained within the top level child graph will also be updated.

### Note:
- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.

### See also:
- `cuGraphInstantiate`, `cuGraphExecUpdate`, `cuGraphExecKernelNodeSetParams`, `cuGraphExecMemcpyNodeSetParams`, `cuGraphExecMemsetNodeSetParams`

### CUresult cuGraphExecDestroy (CUgraphExec hGraphExec)

Destroys an executable graph.

#### Parameters

**hGraphExec**
- Executable graph to destroy

#### Returns

- `CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_VALUE`

#### Description

Destroys the executable graph specified by `hGraphExec`, as well as all of its executable nodes. If the executable graph is in-flight, it will not be terminated, but rather freed asynchronously on completion.

### Note:
- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.

### See also:
CUresult cuGraphExecEventRecordNodeSetEvent(CUgraphExec hGraphExec, CUgraphNode hNode, CUevent event)

Sets the event for an event record node in the given graphExec.

**Parameters**

- **hGraphExec**
  - The executable graph in which to set the specified node
- **hNode**
  - Event record node from the graph from which graphExec was instantiated
- **event**
  - Updated event to use

**Returns**

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE.

**Description**

Sets the event of an event record node in an executable graph hGraphExec. The node is identified by the corresponding node hNode in the non-executable graph, from which the executable graph was instantiated.

The modifications only affect future launches of hGraphExec. Already enqueued or running launches of hGraphExec are not affected by this call. hNode is also not modified by this call.

**Note:**

- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.

**See also:**
cuGraphAddEventRecordNode, cuGraphEventRecordNodeGetEvent, cuGraphEventWaitNodeSetEvent, cuEventRecord, cuStreamWaitEvent, cuGraphCreate, cuGraphDestroyNode, cuGraphInstantiate
CUresult cuGraphExecEventWaitNodeSetEvent
(CUgraphExec hGraphExec, CUgraphNode hNode, CUevent event)

Sets the event for an event record node in the given graphExec.

Parameters

hGraphExec
- The executable graph in which to set the specified node

hNode
- event wait node from the graph from which graphExec was instantiated

event
- Updated event to use

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE.

Description

Sets the event of an event record node in an executable graph hGraphExec. The node is identified by the corresponding node hNode in the non-executable graph, from which the executable graph was instantiated.

The modifications only affect future launches of hGraphExec. Already enqueued or running launches of hGraphExec are not affected by this call. hNode is also not modified by this call.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphAddEventWaitNode, cuGraphEventWaitNodeGetEvent, cuGraphEventRecordNodeSetEvent, cuEventRecord, cuStreamWaitEvent, cuGraphCreate, cuGraphDestroyNode, cuGraphInstantiate
CUresult cuGraphExecHostNodeSetParams (CUgraphExec hGraphExec, CUgraphNode hNode, const CUDA_HOST_NODE_PARAMS *nodeParams)
Sets the parameters for a host node in the given graphExec.

Parameters

hGraphExec
- The executable graph in which to set the specified node

hNode
- Host node from the graph which was used to instantiate graphExec

nodeParams
- The updated parameters to set

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE.

Description

Updates the work represented by hNode in hGraphExec as though hNode had contained nodeParams at instantiation. hNode must remain in the graph which was used to instantiate hGraphExec. Changed edges to and from hNode are ignored.

The modifications only affect future launches of hGraphExec. Already enqueued or running launches of hGraphExec are not affected by this call. hNode is also not modified by this call.

Note:

- Graph objects are not thread-safe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphInstantiate, cuGraphExecKernelNodeSetParams
cuGraphExecMemcpyNodeSetParams cuGraphExecMemsetNodeSetParams
CUresult cuGraphExecKernelNodeSetParams (CUgraphExec hGraphExec, CUgraphNode hNode, const CUDA_KERNEL_NODE_PARAMS *nodeParams)

Sets the parameters for a kernel node in the given graphExec.

Parameters

**hGraphExec**
- The executable graph in which to set the specified node

**hNode**
- Kernel node from the graph from which graphExec was instantiated

**nodeParams**
- Updated Parameters to set

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE.

Description

Sets the parameters of a kernel node in an executable graph hGraphExec. The node is identified by the corresponding node hNode in the non-executable graph, from which the executable graph was instantiated.

hNode must not have been removed from the original graph. The func field of nodeParams cannot be modified and must match the original value. All other values can be modified.

The modifications only affect future launches of hGraphExec. Already enqueued or running launches of hGraphExec are not affected by this call. hNode is also not modified by this call.

**Note:**

- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphAddKernelNode, cuGraphKernelNodeSetParams, cuGraphInstantiate
CUresult cuGraphExecMemcpyNodeSetParams (CUgraphExec hGraphExec, CUgraphNode hNode, const CUDA_MEMCPY3D *copyParams, CUcontext ctx)

Sets the parameters for a memcpy node in the given graphExec.

Parameters

- **hGraphExec**
  - The executable graph in which to set the specified node
- **hNode**
  - Memcpy node from the graph which was used to instantiate graphExec
- **copyParams**
  - The updated parameters to set
- **ctx**
  - Context on which to run the node

Returns

- **CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE**.

Description

Updates the work represented by hNode in hGraphExec as though hNode had contained copyParams at instantiation. hNode must remain in the graph which was used to instantiate hGraphExec. Changed edges to and from hNode are ignored.

The source and destination memory in copyParams must be allocated from the same contexts as the original source and destination memory. Both the instantiation-time memory operands and the memory operands in copyParams must be 1-dimensional. Zero-length operations are not supported.

The modifications only affect future launches of hGraphExec. Already enqueued or running launches of hGraphExec are not affected by this call. hNode is also not modified by this call.

Returns **CUDA_ERROR_INVALID_VALUE** if the memory operands’ mappings changed or either the original or new memory operands are multidimensional.

**Note:**
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.
CUresult cuGraphExecMemsetNodeSetParams
(CUgraphExec hGraphExec, CUgraphNode hNode, const CUDA_MEMSET_NODE_PARAMS *memsetParams, CUcontext ctx)

Sets the parameters for a memset node in the given graphExec.

Parameters

- **hGraphExec**
  - The executable graph in which to set the specified node
- **hNode**
  - Memset node from the graph which was used to instantiate graphExec
- **memsetParams**
  - The updated parameters to set
- **ctx**
  - Context on which to run the node

Returns

- **CUDA_SUCCESS**, **CUDA_ERROR_INVALID_VALUE**.

Description

Updates the work represented by `hNode` in `hGraphExec` as though `hNode` had contained `memsetParams` at instantiation. `hNode` must remain in the graph which was used to instantiate `hGraphExec`. Changed edges to and from `hNode` are ignored.

The destination memory in `memsetParams` must be allocated from the same contexts as the original destination memory. Both the instantiation-time memory operand and the memory operand in `memsetParams` must be 1-dimensional. Zero-length operations are not supported.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `hNode` is also not modified by this call.

Returns **CUDA_ERROR_INVALID_VALUE** if the memory operand’s mappings changed or either the original or new memory operand are multidimensional.

---

**Note:**
Graph objects are not threadsafe. More here.

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

See also:

cuGraphInstantiate, cuGraphExecKernelNodeSetParams
cuGraphExecMemcpyNodeSetParams cuGraphExecHostNodeSetParams

CUresult cuGraphExecUpdate (CUgraphExec hGraphExec, CUgraph hGraph, CUgraphNode *hErrorNode_out, CUgraphExecUpdateResult *updateResult_out)

Check whether an executable graph can be updated with a graph and perform the update if possible.

Parameters

hGraphExec
  The instantiated graph to be updated

hGraph
  The graph containing the updated parameters

hErrorNode_out
  The node which caused the permissibility check to forbid the update, if any

updateResult_out
  Whether the graph update was permitted. If was forbidden, the reason why

Returns

CUDA_SUCCESS, CUDA_ERROR_GRAPH_EXEC_UPDATE_FAILURE.

Description

Updates the node parameters in the instantiated graph specified by hGraphExec with the node parameters in a topologically identical graph specified by hGraph.

Limitations:

Kernel nodes:

  The function must not change (same restriction as cuGraphExecKernelNodeSetParams)

Memset and memcpy nodes:

  The CUDA device(s) to which the operand(s) was allocated/mapped cannot change.
- The source/destination memory must be allocated from the same contexts as the original source/destination memory.
- Only 1D memsets can be changed.
- Additional memcpy node restrictions:
  - Changing either the source or destination memory type (i.e. CU_MEMORYTYPE_DEVICE, CU_MEMORYTYPE_ARRAY, etc.) is not supported.

Note: The API may add further restrictions in future releases. The return code should always be checked.

Some node types are not currently supported:
- Empty graph nodes (CU_GRAPH_NODE_TYPE_EMPTY)
- Child graphs (CU_GRAPH_NODE_TYPE_GRAPH)

cuGraphExecUpdate sets `updateResult_out` to:
- CU_GRAPH_EXEC_UPDATE_ERROR_TOPOLOGY_CHANGED under the following conditions:
  - The count of nodes directly in `hGraphExec` and `hGraph` differ, in which case `hErrorNode_out` is NULL.
  - A node is deleted in `hGraph` but not not its pair from `hGraphExec`, in which case `hErrorNode_out` is NULL.
  - A node is deleted in `hGraphExec` but not its pair from `hGraph`, in which case `hErrorNode_out` is the pairless node from `hGraph`.
  - The dependent nodes of a pair differ, in which case `hErrorNode_out` is the node from `hGraph`.

cuGraphExecUpdate sets `updateResult_out` to:
- CU_GRAPH_EXEC_UPDATE_ERROR if passed an invalid value.
- CU_GRAPH_EXEC_UPDATE_ERROR_TOPOLOGY_CHANGED if the graph topology changed
- CU_GRAPH_EXEC_UPDATE_ERROR_NODE_TYPE_CHANGED if the type of a node changed, in which case `hErrorNode_out` is set to the node from `hGraph`.
- CU_GRAPH_EXEC_UPDATE_ERROR_FUNCTION_CHANGED if the func field of a kernel changed, in which case `hErrorNode_out` is set to the node from `hGraph`.
- CU_GRAPH_EXEC_UPDATE_ERROR_PARAMETERS_CHANGED if any parameters to a node changed in a way that is not supported, in which case `hErrorNode_out` is set to the node from `hGraph`.
- CU_GRAPH_EXEC_UPDATE_ERROR_NOT_SUPPORTED if something about a node is unsupported, like the node’s type or configuration, in which case `hErrorNode_out` is set to the node from `hGraph`.

If `updateResult_out` isn’t set in one of the situations described above, the update check passes and `cuGraphExecUpdate` updates `hGraphExec` to match the contents of `hGraph`. If an error happens during the update, `updateResult_out` will be set
to CU_GRAPH_EXEC_UPDATE_ERROR; otherwise, updateResult_out is set to
CU_GRAPH_EXEC_UPDATE_SUCCESS.

cuGraphExecUpdate returns CUDA_SUCCESS when the updated was performed successfully.
It returns CUDA_ERROR_GRAPH_EXEC_UPDATE_FAILURE if the graph update was not
performed because it included changes which violated constraints specific to instantiated
graph update.

Note:

‣ Graph objects are not threadsafe. More here.
‣ Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphInstantiate,

CUresult cuGraphGetEdges (CUgraph hGraph,
CUgraphNode *from, CUgraphNode *to, size_t
*numEdges)

Returns a graph's dependency edges.

Parameters

hGraph
  - Graph to get the edges from
from
  - Location to return edge endpoints
to
  - Location to return edge endpoints
numEdges
  - See description

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_VALUE

Description

Returns a list of hGraph's dependency edges. Edges are returned via corresponding indices
in from and to; that is, the node in to[i] has a dependency on the node in from[i]. from
and to may both be NULL, in which case this function only returns the number of edges in
numEdges. Otherwise, numEdges entries will be filled in. If numEdges is higher than the
actual number of edges, the remaining entries in `from` and `to` will be set to NULL, and the number of edges actually returned will be written to `numEdges`.

Note:
- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.

See also:
- `cuGraphGetNodes`, `cuGraphGetRootNodes`, `cuGraphAddDependencies`, `cuGraphRemoveDependencies`, `cuGraphNodeGetDependencies`, `cuGraphNodeGetDependentNodes`

**CUresult cuGraphGetNodes (CUgraph hGraph, CUgraphNode *nodes, size_t *numNodes)**

Returns a graph’s nodes.

**Parameters**
- **hGraph**
  - Graph to query
- **nodes**
  - Pointer to return the nodes
- **numNodes**
  - See description

**Returns**
- `CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_VALUE`

**Description**

Returns a list of `hGraph`’s nodes. `nodes` may be NULL, in which case this function will return the number of nodes in `numNodes`. Otherwise, `numNodes` entries will be filled in. If `numNodes` is higher than the actual number of nodes, the remaining entries in `nodes` will be set to NULL, and the number of nodes actually obtained will be returned in `numNodes`.

Note:
- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.
See also:

- cuGraphCreate, cuGraphGetRootNodes, cuGraphGetEdges, cuGraphNodeGetType,
cuGraphNodeGetDependencies, cuGraphNodeGetDependentNodes

**CUresult cuGraphGetRootNodes (CUgraph hGraph, CUGraphNode *rootNodes, size_t *numRootNodes)**

Returns a graph's root nodes.

**Parameters**

- **hGraph**
  - Graph to query
- **rootNodes**
  - Pointer to return the root nodes
- **numRootNodes**
  - See description

**Returns**

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_VALUE

**Description**

Returns a list of hGraph's root nodes. rootNodes may be NULL, in which case this
function will return the number of root nodes in numRootNodes. Otherwise, numRootNodes
entries will be filled in. If numRootNodes is higher than the actual number of root nodes,
the remaining entries in rootNodes will be set to NULL, and the number of nodes actually
obtained will be returned in numRootNodes.

**Note:**

- Graph objects are not threadsafe. [More here](#).
- Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

- cuGraphCreate, cuGraphGetNodes, cuGraphGetEdges, cuGraphNodeGetType,
cuGraphNodeGetDependencies, cuGraphNodeGetDependentNodes
CUresult cuGraphHostNodeGetParams
(CUgraphNode hNode, CUDA_HOST_NODE_PARAMS *nodeParams)

Returns a host node’s parameters.

Parameters

hNode
- Node to get the parameters for

nodeParams
- Pointer to return the parameters

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_VALUE

Description

Returns the parameters of host node hNode in nodeParams.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuLaunchHostFunc, cuGraphAddHostNode, cuGraphHostNodeSetParams

CUresult cuGraphHostNodeSetParams
(CUgraphNode hNode, const CUDA_HOST_NODE_PARAMS *nodeParams)

Sets a host node’s parameters.

Parameters

hNode
- Node to set the parameters for

nodeParams
- Parameters to copy
**Returns**

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE

**Description**

Sets the parameters of host node hNode to nodeParams.

**Note:**

‣ Graph objects are not thread safe. [More here](#).
‣ Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

cuLaunchHostFunc, cuGraphAddHostNode, cuGraphHostNodeGetParams

**CUresult cuGraphInstantiate (CUgraphExec *phGraphExec, CUgraph hGraph, CUgraphNode *phErrorNode, char *logBuffer, size_t bufferSize)**

Creates an executable graph from a graph.

**Parameters**

- **phGraphExec**
  - Returns instantiated graph
- **hGraph**
  - Graph to instantiate
- **phErrorNode**
  - In case of an instantiation error, this may be modified to indicate a node contributing to the error
- **logBuffer**
  - A character buffer to store diagnostic messages
- **bufferSize**
  - Size of the log buffer in bytes

**Returns**

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE
Description

Inherits `hGraph` as an executable graph. The graph is validated for any structural constraints or intra-node constraints which were not previously validated. If instantiation is successful, a handle to the instantiated graph is returned in `graphExec`.

If there are any errors, diagnostic information may be returned in `errorNode` and `logBuffer`. This is the primary way to inspect instantiation errors. The output will be null terminated unless the diagnostics overflow the buffer. In this case, they will be truncated, and the last byte can be inspected to determine if truncation occurred.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

`cuGraphCreate`, `cuGraphUpload`, `cuGraphLaunch`, `cuGraphExec_destroy`

**CUresult cuGraphKernelNodeCopyAttributes (CUgraphNode dst, CUgraphNode src)**

Copies attributes from source node to destination node.

**Parameters**

`dst`
- Destination node

`src`
- Source node For list of attributes see `CUkernelNodeAttrID`

**Returns**

`CUDA_SUCCESS`, `CUDA_ERROR_INVALID_VALUE`

**Description**

Copies attributes from source node `src` to destination node `dst`. Both node must have the same context.

Note:

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

CUresult cuGraphKernelNodeGetAttribute
(CUgraphNode hNode, CUkernelNodeAttrID attr,
CUkernelNodeAttrValue *value_out)
Queries node attribute.

Parameters
hNode
attr
value_out

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Description
Queries attribute attr from node hNode and stores it in corresponding member of value_out.

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

See also:
CUaccessPolicyWindow

CUresult cuGraphKernelNodeGetParams
(CUgraphNode hNode,
CUDA_KERNEL_NODE_PARAMS *nodeParams)
Returns a kernel node’s parameters.

Parameters
hNode
- Node to get the parameters for
nodeParams
- Pointer to return the parameters
Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE

Description
Returns the parameters of kernel node hNode in nodeParams. The kernelParams or extra array returned in nodeParams, as well as the argument values it points to, are owned by the node. This memory remains valid until the node is destroyed or its parameters are modified, and should not be modified directly. Use cuGraphKernelNodeSetParams to update the parameters of this node.

The params will contain either kernelParams or extra, according to which of these was most recently set on the node.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuLaunchKernel, cuGraphAddKernelNode, cuGraphKernelNodeSetParams

CUresult cuGraphKernelNodeSetAttribute
(CUgraphNode hNode, CUkernelNodeAttrID attr, const CUkernelNodeAttrValue *value)
Sets node attribute.

Parameters

hNode
attr
value

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Description
Sets attribute attr on node hNode from corresponding attribute of value.
Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
CUaccessPolicyWindow

CUresult cuGraphKernelNodeSetParams
(CUgraphNode hNode, const
CUDA_KERNEL_NODE_PARAMS *nodeParams)
Sets a kernel node's parameters.

Parameters
hNode
- Node to set the parameters for
nodeParams
- Parameters to copy

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY

Description
Sets the parameters of kernel node hNode to nodeParams.

Note:
- Graph objects are not threadsafe. More here.

See also:
cuLaunchKernel, cuGraphAddKernelNode, cuGraphKernelNodeGetParams
CUresult cuGraphLaunch (CUgraphExec hGraphExec, CUstream hStream)
Launches an executable graph in a stream.

Parameters
hGraphExec
- Executable graph to launch
hStream
- Stream in which to launch the graph

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_VALUE

Description
Executes hGraphExec in hStream. Only one instance of hGraphExec may be executing at
a time. Each launch is ordered behind both any previous work in hStream and any previous
launches of hGraphExec. To execute a graph concurrently, it must be instantiated multiple
times into multiple executable graphs.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuGraphInstantiate, cuGraphUpload, cuGraphExecDestroy

CUresult cuGraphMemcpyNodeGetParams
(CUgraphNode hNode, CUDA_MEMCPY3D *nodeParams)
Returns a memcpy node’s parameters.

Parameters
hNode
- Node to get the parameters for
nodeParams

- Pointer to return the parameters

**Returns**

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE

**Description**

Returns the parameters of memcpy node hNode in nodeParams.

**Note:**

- Graph objects are not threadsafe. [More here.](#)
- Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

cuMemcpy3D, cuGraphAddMemcpyNode, cuGraphMemcpyNodeSetParams

**CUresult cuGraphMemcpyNodeSetParams**

**(CUgraphNode hNode, const CUDA_MEMCPY3D *nodeParams)**

Sets a memcpy node’s parameters.

**Parameters**

**hNode**

- Node to set the parameters for

**nodeParams**

- Parameters to copy

**Returns**

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE.

**Description**

Sets the parameters of memcpy node hNode to nodeParams.

**Note:**
Graph objects are not threadsafe. More here.

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

See also:
cuMemcpy3D, cuGraphAddMemcpyNode, cuGraphMemcpyNodeGetParams

cuResult cuGraphMemsetNodeGetParams (CUGraphNode hNode, CUDA_MEMSET_NODE_PARAMS *nodeParams)

Returns a memset node’s parameters.

Parameters

hNode
  - Node to get the parameters for

nodeParams
  - Pointer to return the parameters

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_VALUE

Description

Returns the parameters of memset node hNode in nodeParams.

Note:

Graph objects are not threadsafe. More here.

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

See also:
cuMemsetD2D32, cuGraphAddMemsetNode, cuGraphMemsetNodeSetParams
CUresult cuGraphMemsetNodeSetParams
(CUgraphNode hNode, const
CUDA_MEMSET_NODE_PARAMS *nodeParams)

Sets a memset node’s parameters.

Parameters

- **hNode**
  - Node to set the parameters for

- **nodeParams**
  - Parameters to copy

Returns

- CUDA_SUCCESS
- CUDA_ERROR_DEINITIALIZED
- CUDA_ERROR_NOT_INITIALIZED
- CUDA_ERROR_INVALID_VALUE

Description

Sets the parameters of memset node hNode to nodeParams.

Note:

- Graph objects are not thread safe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuMemsetD2D32, cuGraphAddMemsetNode, cuGraphMemsetNodeGetParams

CUresult cuGraphNodeFindInClone (CUgraphNode
*phNode, CUgraphNode hOriginalNode, CUgraph
hClonedGraph)

Finds a cloned version of a node.

Parameters

- **phNode**
  - Returns handle to the cloned node

- **hOriginalNode**
  - Handle to the original node
**hClonedGraph**
- Cloned graph to query

**Returns**
CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE.

**Description**
This function returns the node in hClonedGraph corresponding to hOriginalNode in the original graph.

hClonedGraph must have been cloned from hOriginalGraph via cuGraphClone. hOriginalNode must have been in hOriginalGraph at the time of the call to cuGraphClone, and the corresponding cloned node in hClonedGraph must not have been removed. The cloned node is then returned via phClonedNode.

**Note:**
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

**See also:**
cuGraphClone

**CUresult cuGraphNodeGetDependencies**
(CUgraphNode hNode, CUgraphNode *dependencies, size_t *numDependencies)
Returns a node’s dependencies.

**Parameters**

- **hNode**
  - Node to query
- **dependencies**
  - Pointer to return the dependencies
- **numDependencies**
  - See description

**Returns**
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE
Description

Returns a list of node's dependencies. dependencies may be NULL, in which case this function will return the number of dependencies in numDependencies. Otherwise, numDependencies entries will be filled in. If numDependencies is higher than the actual number of dependencies, the remaining entries in dependencies will be set to NULL, and the number of nodes actually obtained will be returned in numDependencies.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphNodeGetDependentNodes, cuGraphGetNodes, cuGraphGetRootNodes, cuGraphGetEdges, cuGraphAddDependencies, cuGraphRemoveDependencies

CUresult cuGraphNodeGetDependentNodes (CUgraphNode hNode, CUgraphNode *dependentNodes, size_t *numDependentNodes)

Returns a node's dependent nodes.

Parameters

hNode
- Node to query

dependentNodes
- Pointer to return the dependent nodes

numDependentNodes
- See description

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE

Description

Returns a list of node's dependent nodes. dependentNodes may be NULL, in which case this function will return the number of dependent nodes in numDependentNodes. Otherwise, numDependentNodes entries will be filled in. If numDependentNodes is higher than the
actual number of dependent nodes, the remaining entries in dependentNodes will be set to NULL, and the number of nodes actually obtained will be returned in numDependentNodes.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphNodeGetDependencies, cuGraphGetNodes, cuGraphGetRootNodes, cuGraphGetEdges, cuGraphAddDependencies, cuGraphRemoveDependencies

CUresult cuGraphNodeGetType (CUgraphNode hNode, CUgraphNodeType *type)
Returns a node’s type.

Parameters
hNode
- Node to query
type
- Pointer to return the node type

Returns
CUDA_SUCCESS, CUDA_ERROR_DINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE

Description
Returns the node type of hNode in type.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphGetNodes, cuGraphGetRootNodes, cuGraphChildGraphNodeGetGraph, cuGraph KernelNodeGetParams, cuGraphKernelNodeSetParams, cuGraphHostNodeGetParams, cuGraphHostNodeSetParams.
CUresult cuGraphRemoveDependencies (CUgraph hGraph, const CUgraphNode *from, const CUgraphNode *to, size_t numDependencies)

Removes dependency edges from a graph.

Parameters

hGraph
- Graph from which to remove dependencies
from
- Array of nodes that provide the dependencies
to
- Array of dependent nodes
numDependencies
- Number of dependencies to be removed

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE

Description

The number of dependencies to be removed is defined by numDependencies. Elements in from and to at corresponding indices define a dependency. Each node in from and to must belong to hGraph.

If numDependencies is 0, elements in from and to will be ignored. Specifying a non-existing dependency will return an error.

Note:

- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphAddDependencies, cuGraphGetEdges, cuGraphNodeGetDependencies, cuGraphNodeGetDependentNodes
CUresult cuGraphUpload (CUgraphExec hGraphExec, CUstream hStream)

Uploads an executable graph in a stream.

Parameters

hGraphExec
- Executable graph to upload

hStream
- Stream in which to upload the graph

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_VALUE

Description

Uploads hGraphExec to the device in hStream without executing it. Uploads of the same
hGraphExec will be serialized. Each upload is ordered behind both any previous work in
hStream and any previous launches of hGraphExec.

Note:
- Graph objects are not threadsafe. More here.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphInstantiate, cuGraphLaunch, cuGraphExecDestroy

5.21. Occupancy

This section describes the occupancy calculation functions of the low-level CUDA driver
application programming interface.
CUresult

cuOccupancyAvailableDynamicSMemPerBlock (size_t *
dynamicSmemSize, CUfunction func, int numBlocks, int blockSize)

Returns dynamic shared memory available per block when launching numBlocks blocks on SM.

Parameters

dynamicSmemSize
  - Returned maximum dynamic shared memory
func
  - Kernel function for which occupancy is calculated
numBlocks
  - Number of blocks to fit on SM
blockSize
  - Size of the blocks

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE,
CUDA_ERROR_UNKNOWN

Description

Returns in *dynamicSmemSize the maximum size of dynamic shared memory to allow numBlocks blocks per SM.

Note:

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

See also:
CUresult

cuOccupancyMaxActiveBlocksPerMultiprocessor (int *numBlocks, CUfunction func, int blockSize, size_t dynamicSMemSize)

Returns occupancy of a function.

Parameters

numBlocks  
  - Returned occupancy

func  
  - Kernel for which occupancy is calculated

blockSize  
  - Block size the kernel is intended to be launched with

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

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_UNKNOWN

Description

Returns in *numBlocks the number of the maximum active blocks per streaming multiprocessor.

Note:

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

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessor
CUresult

cuOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
(int *numBlocks, CUfunction func, int blockSize, size_t
dynamicSMemSize, unsigned int flags)

Returns occupancy of a function.

Parameters

numBlocks
  - Returned occupancy
func
  - Kernel for which occupancy is calculated
blockSize
  - Block size the kernel is intended to be launched with
dynamicSMemSize
  - Per-block dynamic shared memory usage intended, in bytes
flags
  - Requested behavior for the occupancy calculator

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE,
CUDA_ERROR_UNKNOWN

Description

Returns in *numBlocks the number of the maximum active blocks per streaming
multiprocessor.

The Flags parameter controls how special cases are handled. The valid flags are:

- **CU_OCCUPANCY_DEFAULT**, which maintains the default behavior as
cuOccupancyMaxActiveBlocksPerMultiprocessor;

- **CU_OCCUPANCY_DISABLE_CACHING_OVERRIDE**, which suppresses the default
behavior on platform where global caching affects occupancy. On such platforms, if
caching is enabled, but per-block SM resource usage would result in zero occupancy,
the occupancy calculator will calculate the occupancy as if caching is disabled. Setting
**CU_OCCUPANCY_DISABLE_CACHING_OVERRIDE** makes the occupancy calculator to
return 0 in such cases. More information can be found about this feature in the “Unified
L1/Texture Cache” section of the Maxwell tuning guide.
Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags

cUresult cuOccupancyMaxPotentialBlockSize (int *minGridSize, int *blockSize, CUfunction func, CUoccupancyB2DSize blockSizeToDynamicSMemSize, size_t dynamicSMemSize, int blockSizeLimit)

Suggest a launch configuration with reasonable occupancy.

Parameters

minGridSize
- Returned minimum grid size needed to achieve the maximum occupancy

blockSize
- Returned maximum block size that can achieve the maximum occupancy

func
- Kernel for which launch configuration is calculated

blockSizeToDynamicSMemSize
- A function that calculates how much per-block dynamic shared memory func uses based on the block size

dynamicSMemSize
- Dynamic shared memory usage intended, in bytes

blockSizeLimit
- The maximum block size func is designed to handle

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_UNKNOWN

Description

Returns in *blockSize a reasonable block size that can achieve the maximum occupancy (or, the maximum number of active warps with the fewest blocks per multiprocessor), and in *minGridSize the minimum grid size to achieve the maximum occupancy.

If blockSizeLimit is 0, the configurator will use the maximum block size permitted by the device / function instead.
If per-block dynamic shared memory allocation is not needed, the user should leave both `blockSizeToDynamicSMemSize` and `dynamicSMemSize` as 0.

If per-block dynamic shared memory allocation is needed, then if the dynamic shared memory size is constant regardless of block size, the size should be passed through `dynamicSMemSize`, and `blockSizeToDynamicSMemSize` should be NULL.

Otherwise, if the per-block dynamic shared memory size varies with different block sizes, the user needs to provide a unary function through `blockSizeToDynamicSMemSize` that computes the dynamic shared memory needed by `func` for any given block size. `dynamicSMemSize` is ignored. An example signature is:

```c
    // Take block size, returns dynamic shared memory needed
    size_t blockToSmem(int blockSize);
```

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

See also:
`cudaOccupancyMaxPotentialBlockSize`

**CUresult**
`cuOccupancyMaxPotentialBlockSizeWithFlags`

```c
(int *minGridSize, int *blockSize, CUfunction func,
CUoccupancyB2DSize blockSizeToDynamicSMemSize,
size_t dynamicSMemSize, int blockSizeLimit,
unsigned int flags)
```

Suggest a launch configuration with reasonable occupancy.

**Parameters**
- `minGridSize` - Returned minimum grid size needed to achieve the maximum occupancy
- `blockSize` - Returned maximum block size that can achieve the maximum occupancy
- `func` - Kernel for which launch configuration is calculated
- `blockSizeToDynamicSMemSize` - A function that calculates how much per-block dynamic shared memory `func` uses based on the block size
**dynamicSMemSize**
- Dynamic shared memory usage intended, in bytes

**blockSizeLimit**
- The maximum block size `func` is designed to handle

**flags**
- Options

**Returns**
CUDA SUCCESS, CUDA ERROR DEINITIALIZED, CUDA ERROR NOT INITIALIZED,
CUDA ERROR INVALID CONTEXT, CUDA ERROR INVALID VALUE,
CUDA ERROR UNKNOWN

**Description**
An extended version of `cuOccupancyMaxPotentialBlockSize`. In addition to arguments passed to `cuOccupancyMaxPotentialBlockSize`, `cuOccupancyMaxPotentialBlockSizeWithFlags` also takes a `Flags` parameter.

The `Flags` parameter controls how special cases are handled. The valid flags are:

- **CU_OCCUPANCY_DEFAULT**, which maintains the default behavior as `cuOccupancyMaxPotentialBlockSize`;
- **CU_OCCUPANCY_DISABLE_CACHING_OVERRIDE**, which suppresses the default behavior on platform where global caching affects occupancy. On such platforms, the launch configurations that produces maximal occupancy might not support global caching. Setting `CU_OCCUPANCY_DISABLE_CACHING_OVERRIDE` guarantees that the the produced launch configuration is global caching compatible at a potential cost of occupancy. More information can be found about this feature in the “Unified L1/Texture Cache” section of the Maxwell tuning guide.

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

**See also:**
cuOccupancyMaxPotentialBlockSizeWithFlags
5.22. Texture Reference Management [DEPRECATED]

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

CUresult cuTexRefCreate (CUtexref *pTexRef)

Creates a 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

Description

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.

See also:

cuTexRefDestroy

CUresult cuTexRefDestroy (CUtexref hTexRef)

Destroys a texture reference.

Parameters

hTexRef
- Texture reference to destroy
Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description
Deprecated
Destroys the texture reference specified by hTexRef.

See also:
cuTexRefCreate

CUresult cuTexRefGetAddress (CUdeviceptr *pdptr, CUtexref hTexRef)
Gets the address associated with a texture reference.

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

Description
Deprecated
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.

See also:
cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray,
cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddressMode,
cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat
CUresult cuTexRefGetAddressMode (CUaddress_mode *pam, CUtexref hTexRef, int dim)

Gets the addressing mode used by a texture reference.

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

Description

Deprecated

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.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray,
cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress,
cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefGetArray (CUarray *phArray, CUtexref hTexRef)

Gets the array bound to a texture reference.

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

Description

Deprecated

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.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefGetBorderColor (float *pBorderColor, CUtexref hTexRef)

Gets the border color used by a texture reference.

Parameters

pBorderColor
  - Returned Type and Value of RGBA color
hTexRef
  - Texture reference

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Deprecated


See also:

cuTexRefSetAddressMode, cuTexRefSetAddressMode, cuTexRefSetBorderColor
CUresult cuTexRefGetFilterMode (CUfilter_mode *pfm, CUtexref hTexRef)

Gets the filter-mode used by a texture reference.

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`

Description

Deprecated

Returns in `*pfm` the filtering mode of the texture reference `hTexRef`.

See also:

- `cuTexRefSetAddress`, `cuTexRefSetAddress2D`, `cuTexRefSetAddressMode`, `cuTexRefSetArray`, `cuTexRefSetFilterMode`, `cuTexRefSetFlags`, `cuTexRefSetFormat`, `cuTexRefGetAddress`, `cuTexRefGetAddressMode`, `cuTexRefGetArray`, `cuTexRefGetFlags`, `cuTexRefGetFormat`

CUresult cuTexRefGetFlags (unsigned int *pFlags, CUtexref hTexRef)

Gets the flags used by a texture reference.

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`
Description

Deprecated

Returns in *pFlags the flags of the texture reference hTexRef.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray,
cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress,
cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags

CUresult cuTexRefGetFormat (CUarray_format *pFormat, int *pNumChannels, CUtexref hTexRef)

Gets the format used by a texture reference.

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

Description

Deprecated

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.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray,
cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress,
cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags
CUresult cuTexRefGetMaxAnisotropy (int *pmaxAniso, CUtexref hTexRef)

Gets the maximum anisotropy for a texture reference.

Parameters

pmaxAniso
- Returned maximum anisotropy

hTexRef
- Texture reference

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Deprecated

Returns the maximum anisotropy in pmaxAniso that’s used when reading memory through
the texture reference hTexRef.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray,
cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode,
cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefGetMipmapFilterMode
(CUfilter_mode *pfm, CUtexref hTexRef)

Gets the mipmap filtering mode for a texture reference.

Parameters

pfm
- Returned mipmap filtering mode

hTexRef
- Texture reference

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE
Description

Deprecated

Returns the mipmap filtering mode in pfm that’s used when reading memory through the texture reference hTexRef.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefGetMipmapLevelBias (float *pbias, CUtexref hTexRef)

Gets the mipmap level bias for a texture reference.

Parameters

pbias
  - Returned mipmap level bias

hTexRef
  - Texture reference

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Deprecated

Returns the mipmap level bias in pbias that’s added to the specified mipmap level when reading memory through the texture reference hTexRef.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat
CUresult cuTexRefGetMipmapLevelClamp
(float *pminMipmapLevelClamp, float *
*pmaxMipmapLevelClamp, CUtexref hTexRef)

Gets the min/max mipmap level clamps for a texture reference.

Parameters
- **pminMipmapLevelClamp**
  - Returned mipmap min level clamp
- **pmaxMipmapLevelClamp**
  - Returned mipmap max level clamp
- **hTexRef**
  - Texture reference

Returns
- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
  CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description
- **Deprecated**

Returns the min/max mipmap level clamps in `pminMipmapLevelClamp` and
`pmaxMipmapLevelClamp` that's used when reading memory through the texture reference
`hTexRef`.

See also:
- `cuTexRefSetAddress`, `cuTexRefSetAddress2D`, `cuTexRefSetAddressMode`,
  `cuTexRefSetArray`, `cuTexRefSetFlags`, `cuTexRefSetFormat`, `cuTexRefGetAddress`,
  `cuTexRefGetAddressMode`, `cuTexRefGetArray`, `cuTexRefGetFilterMode`,
  `cuTexRefGetFlags`, `cuTexRefGetFormat`

CUresult cuTexRefGetMipmappedArray
(CUmipmappedArray *phMipmappedArray, CUtexref
hTexRef)

Gets the mipmapped array bound to a texture reference.

Parameters
- **phMipmappedArray**
  - Returned mipmapped array
**hTexRef**
- Texture reference

**Returns**
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

**Description**
Deprecated

Returns in `*phMipmappedArray` the CUDA mipmapped array bound to the texture reference `hTexRef`, or returns CUDA_ERROR_INVALID_VALUE if the texture reference is not bound to any CUDA mipmapped array.

**See also:**
cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

**CUresult cuTexRefSetAddress (size_t *ByteOffset, CUtexref hTexRef, CUdeviceptr dptr, size_t bytes)**
Binds an address as a texture reference.

**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

**Description**
Deprecated
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.

The total number of elements (or texels) in the linear address range cannot exceed CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LINEAR_WIDTH. The number of elements is computed as (bytes / bytesPerElement), where bytesPerElement is determined from the data format and number of components set using cuTexRefSetFormat().

See also:
cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat, cudaBindTexture

CUresult cuTexRefSetAddress2D (CUtexref hTexRef, const CUDA_ARRAY_DESCRIPTOR *desc, CUdeviceptr dptr, size_t Pitch)
Binds an address as a 2D texture reference.

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
Description

Deprecated

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.

Pitch has to be aligned to the hardware-specific texture pitch alignment. This value can be queried using the device attribute CU_DEVICE_ATTRIBUTE_TEXTURE_PITCH_ALIGNMENT. If an unaligned Pitch is supplied, CUDA_ERROR_INVALID_VALUE is returned.

Width and Height, which are specified in elements [or texels], cannot exceed CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_WIDTH and CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_HEIGHT respectively. Pitch, which is specified in bytes, cannot exceed CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_PITCH.

See also:

cuTexRefSetAddress, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat, cudaBindTexture2D

CUresult cuTexRefSetAddressMode (CUtexref hTexRef, int dim, CUaddress_mode am)

Sets the addressing mode for a texture reference.

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

Description
Deprecated
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:

```c
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.

See also:
cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetArray,
cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress,
cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags,
cuTexRefGetFormat, cudaBindTexture, cudaBindTexture2D, cudaBindTextureToArray,
cudaBindTextureToMipmappedArray

CUresult cuTexRefSetArray (CUtexref hTexRef,
CUarray hArray, unsigned int Flags)
Binds an array as a texture reference.

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

Description
Deprecated
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.

See also:
cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat, cudaBindTextureToArray

CUresult cuTexRefSetBorderColor (CUtexref hTexRef, float *pBorderColor)
Sets the border color for a texture reference.

Parameters
hTexRef
   - Texture reference
pBorderColor
   - RGBA color

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description
Deprecated
Note that the color values can be set only when the Address mode is set to CU_TR_ADDRESS_MODE_BORDER using cuTexRefSetAddressMode. Applications using integer border color values have to “reinterpret_cast” their values to float.

See also:
cuTexRefSetAddressMode, cuTexRefGetAddressMode, cuTexRefGetBorderColor, cudaBindTexture, cudaBindTexture2D, cudaBindTextureToArray, cudaBindTextureToMipmappedArray

CUresult cuTexRefSetFilterMode (CUtexref hTexRef, CUfilter_mode fm)
Sets the filtering mode for a texture reference.

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

Description
Deprecated
Specifies the filtering mode fm to be used when reading memory through the texture reference hTexRef. CUfilter_mode_enum is defined as:

```c
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.

See also:
cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat, cudaBindTextureToArray
CUresult cuTexRefSetFlags (CUtexref hTexRef, unsigned int Flags)

Sets the flags for a texture reference.

Parameters

- **hTexRef**: Texture reference
- **Flags**: Optional flags to set

Returns

CU_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Deprecated

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]. Note that texture with 32-bit integer format would not be promoted, regardless of whether or not this flag is specified;
- **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;
- **CU_TRSF_DISABLE_TRILINEAR_OPTIMIZATION**, which disables any trilinear filtering optimizations. Trilinear optimizations improve texture filtering performance by allowing bilinear filtering on textures in scenarios where it can closely approximate the expected results.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat, cudaBindTexture, cudaBindTexture2D, cudaBindTextureToArray, cudaBindTextureToMipmappedArray
CUresult cuTexRefSetFormat (CUtexref hTexRef, CUarray_format fmt, int NumPackedComponents)

Sets the format for a texture reference.

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

Description

Deprecated

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.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray,
cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefGetAddress, cuTexRefGetAddressMode,
cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat,
cudaCreateChannelDesc, cudaBindTexture, cudaBindTexture2D, cudaBindTextureToArray,
cudaBindTextureToMipmappedArray

CUresult cuTexRefSetMaxAnisotropy (CUtexref hTexRef, unsigned int maxAniso)

Sets the maximum anisotropy for a texture reference.

Parameters

hTexRef
- Texture reference
maxAniso
- Maximum anisotropy

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description
Deprecated
Specifies the maximum anisotropy maxAniso to be used when reading memory through the
texture reference hTexRef.
Note that this call has no effect if hTexRef is bound to linear memory.

See also:
cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray,
cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode,
cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat,
cuBindTextureToArray, cuBindTextureToMipmappedArray

CUresult cuTexRefSetMipmapFilterMode (CUtexref hTexRef, CUfilter_mode fm)
Sets the mipmap filtering mode for a texture reference.

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

Description
Deprecated
Specifies the mipmap filtering mode fm to be used when reading memory through the texture
reference hTexRef. CUfilter_mode_enum is defined as:

```c
typedef enum CUfilter_mode_enum {
    CU_TR_FILTER_MODE_POINT = 0,
    CU_TR_FILTER_MODE_LINEAR = 1
};
```
Note that this call has no effect if \texttt{hTexRef} is not bound to a mipmapped array.

See also:
\texttt{cuTexRefSetAddress}, \texttt{cuTexRefSetAddress2D}, \texttt{cuTexRefSetAddressMode}, \texttt{cuTexRefSetArray}, \texttt{cuTexRefSetFlags}, \texttt{cuTexRefSetFormat}, \texttt{cuTexRefGetAddress}, \texttt{cuTexRefGetAddressMode}, \texttt{cuTexRefGetArray}, \texttt{cuTexRefGetFilterMode}, \texttt{cuTexRefGetFlags}, \texttt{cuTexRefGetFormat}, \texttt{cudaBindTextureToMipmappedArray}

**CUresult cuTexRefSetMipmapLevelBias (CUtexref hTexRef, float bias)**
Sets the mipmap level bias for a texture reference.

**Parameters**

<table>
<thead>
<tr>
<th>hTexRef</th>
<th>Texture reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>bias</td>
<td>Mipmap level bias</td>
</tr>
</tbody>
</table>

**Returns**

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

**Description**

Deprecated

Specifies the mipmap level bias \texttt{bias} to be added to the specified mipmap level when reading memory through the texture reference \texttt{hTexRef}.

Note that this call has no effect if \texttt{hTexRef} is not bound to a mipmapped array.

See also:
\texttt{cuTexRefSetAddress}, \texttt{cuTexRefSetAddress2D}, \texttt{cuTexRefSetAddressMode}, \texttt{cuTexRefSetArray}, \texttt{cuTexRefSetFlags}, \texttt{cuTexRefSetFormat}, \texttt{cuTexRefGetAddress}, \texttt{cuTexRefGetAddressMode}, \texttt{cuTexRefGetArray}, \texttt{cuTexRefGetFilterMode}, \texttt{cuTexRefGetFlags}, \texttt{cuTexRefGetFormat}, \texttt{cudaBindTextureToMipmappedArray}
CUresult cuTexRefSetMipmapLevelClamp (CUtexref hTexRef, float minMipmapLevelClamp, float maxMipmapLevelClamp)

Sets the mipmap min/max mipmap level clamps for a texture reference.

Parameters

- **hTexRef**: Texture reference
- **minMipmapLevelClamp**: Mipmap min level clamp
- **maxMipmapLevelClamp**: Mipmap max level clamp

Returns

- CUDA_SUCCESS
- CUDA_ERROR_DEINITIALIZED
- CUDA_ERROR_NOT_INITIALIZED
- CUDA_ERROR_INVALID_CONTEXT
- CUDA_ERROR_INVALID_VALUE

Description

**Deprecated**

Specifies the min/max mipmap level clamps, minMipmapLevelClamp and maxMipmapLevelClamp respectively, to be used when reading memory through the texture reference hTexRef.

Note that this call has no effect if hTexRef is not bound to a mipmapped array.

See also:

- cuTexRefSetAddress
- cuTexRefSetAddress2D
- cuTexRefSetAddressMode
- cuTexRefSetArray
- cuTexRefSetFlags
- cuTexRefSetFormat
- cuTexRefGetAddress
- cuTexRefGetAddressMode
- cuTexRefGetArray
- cuTexRefGetFilterMode
- cuTexRefGetFlags
- cuTexRefGetFormat
- cudaBindTextureToMipmappedArray
CUresult cuTexRefSetMipmappedArray (CUtexref hTexRef, CUmipmappedArray hMipmappedArray, unsigned int Flags)
Binds a mipmapped array to a texture reference.

Parameters

hTexRef
- Texture reference to bind

hMipmappedArray
- Mipmapped 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

Description
Deprecated
Binds the CUDA mipmapped array hMipmappedArray 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.

See also:
cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode,
cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress,
cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags,
cuTexRefGetFormat, cudaBindTextureToMipmappedArray

5.23. Surface Reference Management [DEPRECATED]
This section describes the surface reference management functions of the low-level CUDA driver application programming interface.
CUresult cuSurfRefGetArray (CUarray *phArray, CUsurfref hSurfRef)

Passes back the CUDA array bound to a surface reference.

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

Description

Deprecated

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.

See also:

- cuModuleGetSurfRef, cuSurfRefSetArray

CUresult cuSurfRefSetArray (CUsurfref hSurfRef, CUarray hArray, unsigned int Flags)

Sets the CUDA array for a surface reference.

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
Description

Deprecated

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.

See also:
cuModuleGetSurfRef, cuSurfRefGetArray, cudaBindSurfaceToArray

5.24. Texture Object Management

This section describes the texture object management functions of the low-level CUDA driver application programming interface. The texture object API is only supported on devices of compute capability 3.0 or higher.

CUresult cuTexObjectCreate (CUtexObject *pTexObject, const CUDA_RESOURCE_DESC *pResDesc, const CUDA_TEXTURE_DESC *pTexDesc, const CUDA_RESOURCE_VIEW_DESC *pResViewDesc)

Creates a texture object.

Parameters

pTexObject
- Texture object to create

pResDesc
- Resource descriptor

pTexDesc
- Texture descriptor

pResViewDesc
- Resource view descriptor

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE
Description

Creates a texture object and returns it in pTexObject. pResDesc describes the data to texture from. pTexDesc describes how the data should be sampled. pResViewDesc is an optional argument that specifies an alternate format for the data described by pResDesc, and also describes the subresource region to restrict access to when texturing. pResViewDesc can only be specified if the type of resource is a CUDA array or a CUDA mipmapped array.

Texture objects are only supported on devices of compute capability 3.0 or higher. Additionally, a texture object is an opaque value, and, as such, should only be accessed through CUDA API calls.

The **CUDA_RESOURCE_DESC** structure is defined as:

```c
typedef struct CUDA_RESOURCE_DESC_st {
    CUresourcetype resType;
    union {
        struct {
            CUarray hArray;
        } array;
        struct {
            CUmipmappedArray hMipmappedArray;
        } mipmap;
        struct {
            CUdeviceptr devPtr;
            CUarray_format format;
            unsigned int numChannels;
            size_t sizeInBytes;
        } linear;
        struct {
            CUdeviceptr devPtr;
            CUarray_format format;
            unsigned int numChannels;
            size_t width;
            size_t height;
            size_t pitchInBytes;
        } pitch2D;
    } res;
    unsigned int flags;
} CUDA_RESOURCE_DESC;
```

where:

- **CUDA_RESOURCE_DESC::resType** specifies the type of resource to texture from. CUresourceType is defined as:

```c
typedef enum CUresourcetype_enum {
    CU_RESOURCE_TYPE_ARRAY = 0x00,
    CU_RESOURCE_TYPE_MIPMAPPED_ARRAY = 0x01,
    CU_RESOURCE_TYPE_LINEAR = 0x02,
    CU_RESOURCE_TYPE_PITCH2D = 0x03
} CUresourcetype;
```

If **CUDA_RESOURCE_DESC::resType** is set to **CU_RESOURCE_TYPE_ARRAY**, **CUDA_RESOURCE_DESC::res::array::hArray** must be set to a valid CUDA array handle.
If `CUDA_RESOURCE_DESC::resType` is set to `CURESOURCE_TYPE_MIPMAPPED_ARRAY`, `CUDA_RESOURCE_DESC::res::mipmap::hMipmappedArray` must be set to a valid CUDA mipmapped array handle.

If `CUDA_RESOURCE_DESC::resType` is set to `CURESOURCE_TYPE_LINEAR`, `CUDA_RESOURCE_DESC::res::linear::devPtr` must be set to a valid device pointer, that is aligned to `CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT`. `CUDA_RESOURCE_DESC::res::linear::devPtr`, `CUDA_RESOURCE_DESC::res::linear::format` and `CUDA_RESOURCE_DESC::res::linear::numChannels` describe the format of each component and the number of components per array element. `CUDA_RESOURCE_DESC::res::linear::sizeInBytes` specifies the size of the array in bytes. The total number of elements in the linear address range cannot exceed `CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LINEAR_WIDTH`. The number of elements is computed as `(sizeInBytes / (sizeof(format) * numChannels))`.

If `CUDA_RESOURCE_DESC::resType` is set to `CURESOURCE_TYPE_PITCH2D`, `CUDA_RESOURCE_DESC::res::pitch2D::devPtr` must be set to a valid device pointer, that is aligned to `CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT`. `CUDA_RESOURCE_DESC::res::pitch2D::format` and `CUDA_RESOURCE_DESC::res::pitch2D::numChannels` describe the format of each component and the number of components per array element. `CUDA_RESOURCE_DESC::res::pitch2D::width` and `CUDA_RESOURCE_DESC::res::pitch2D::height` specify the width and height of the array in elements, and cannot exceed `CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_WIDTH` and `CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_HEIGHT` respectively. `CUDA_RESOURCE_DESC::res::pitch2D::pitchInBytes` specifies the pitch between two rows in bytes and has to be aligned to `CU_DEVICE_ATTRIBUTE_TEXTURE_PITCH_ALIGNMENT`. Pitch cannot exceed `CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_PITCH`.

- flags must be set to zero.

The `CUDA_TEXTURE_DESC` struct is defined as

```c
typedef struct CUDA_TEXTURE_DESC_st {
    CUaddress_mode addressMode[3];
    CUfilter_mode filterMode;
    unsigned int flags;
    unsigned int maxAnisotropy;
    CUfilter_mode mipmapFilterMode;
    float mipmapLevelBias;
    float minMipmapLevelClamp;
    float maxMipmapLevelClamp;
} CUDA_TEXTURE_DESC;
```

where

- `CUDA_TEXTURE_DESC::addressMode` specifies the addressing mode for each dimension of the texture data. `CUaddress_mode` is defined as:

```c
typedef enum CUaddress_mode_enum {
    CU_TR_ADDRESS_MODE_WRAP = 0,
    CU_TR_ADDRESS_MODE_CLAMP = 1,
    ...} CUaddress_mode;
```
CU_TR_ADDRESS_MODE_MIRROR = 2,
CU_TR_ADDRESS_MODE_BORDER = 3
} CUaddress_mode;

This is ignored if CUDA_RESOURCE_DESC::resType is CU_RESOURCE_TYPE_LINEAR.
Also, if the flag, CU_TRSF_NORMALIZED_COORDINATES is not set, the only supported address mode is CU_TR_ADDRESS_MODE_CLAMP.

- **CUDA_TEXTURE_DESC::filterMode** specifies the filtering mode to be used when fetching from the texture. CUFILTER_MODE is defined as:

  ```c
  typedef enum CUFILTER_MODE_ENUM {
    CU_TRFILTER_MODE_POINT = 0,
    CU_TRFILTER_MODE_LINEAR = 1
  } CUFILTER_MODE;
  ```

  This is ignored if CUDA_RESOURCE_DESC::resType is CU_RESOURCE_TYPE_LINEAR.

- **CUDA_TEXTURE_DESC::flags** can be any combination of the following:
  - **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]. Note that texture with 32-bit integer format would not be promoted, regardless of whether or not this flag is specified.
  - **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; Note that for CUDA mipmapmed arrays, this flag has to be set.
  - **CU_TRSF_DISABLE_TRILINEAR_OPTIMIZATION**, which disables any trilinear filtering optimizations. Trilinear optimizations improve texture filtering performance by allowing bilinear filtering on textures in scenarios where it can closely approximate the expected results.
  - **CUDA_TEXTURE_DESC::maxAnisotropy** specifies the maximum anisotropy ratio to be used when doing anisotropic filtering. This value will be clamped to the range [1,16].
  - **CUDA_TEXTURE_DESC::mipmapFilterMode** specifies the filter mode when the calculated mipmap level lies between two defined mipmap levels.
  - **CUDA_TEXTURE_DESC::mipmapLevelBias** specifies the offset to be applied to the calculated mipmap level.
  - **CUDA_TEXTURE_DESC::minMipmapLevelClamp** specifies the lower end of the mipmap level range to clamp access to.
  - **CUDA_TEXTURE_DESC::maxMipmapLevelClamp** specifies the upper end of the mipmap level range to clamp access to.

The **CUDA_RESOURCE_VIEW_DESC** struct is defined as

```c
typedef struct CUDA_RESOURCE_VIEW_DESC_st {
  CUREsourceViewFormat format;
  size_t width;
  size_t height;
  size_t depth;
} CUDA_RESOURCE_VIEW_DESC;
```
where:

- **CUDA_RESOURCE_VIEW_DESC::format** specifies how the data contained in the CUDA array or CUDA mipmapped array should be interpreted. Note that this can incur a change in size of the texture data. If the resource view format is a block compressed format, then the underlying CUDA array or CUDA mipmapped array has to have a base of format **CU_AD_FORMAT_UNSIGNED_INT32**, with 2 or 4 channels, depending on the block compressed format. For ex., BC1 and BC4 require the underlying CUDA array to have a format of **CU_AD_FORMAT_UNSIGNED_INT32** with 2 channels. The other BC formats require the underlying resource to have the same base format but with 4 channels.

- **CUDA_RESOURCE_VIEW_DESC::width** specifies the new width of the texture data. If the resource view format is a block compressed format, this value has to be 4 times the original width of the resource. For non block compressed formats, this value has to be equal to that of the original resource.

- **CUDA_RESOURCE_VIEW_DESC::height** specifies the new height of the texture data. If the resource view format is a block compressed format, this value has to be 4 times the original height of the resource. For non block compressed formats, this value has to be equal to that of the original resource.

- **CUDA_RESOURCE_VIEW_DESC::depth** specifies the new depth of the texture data. This value has to be equal to that of the original resource.

- **CUDA_RESOURCE_VIEW_DESC::firstMipmapLevel** specifies the most detailed mipmap level. This will be the new mipmap level zero. For non-mipmapped resources, this value has to be zero. **CUDA_TEXTURE_DESC::minMipmapLevelClamp** and **CUDA_TEXTURE_DESC::maxMipmapLevelClamp** will be relative to this value. For ex., if the firstMipmapLevel is set to 2, and a minMipmapLevelClamp of 1.2 is specified, then the actual minimum mipmap level clamp will be 3.2.

- **CUDA_RESOURCE_VIEW_DESC::lastMipmapLevel** specifies the least detailed mipmap level. For non-mipmapped resources, this value has to be zero.

- **CUDA_RESOURCE_VIEW_DESC::firstLayer** specifies the first layer index for layered textures. This will be the new layer zero. For non-layered resources, this value has to be zero.

- **CUDARESOURCEVIEWDESC::lastLayer** specifies the last layer index for layered textures. For non-layered resources, this value has to be zero.

See also:

- [cuTexObjectDestroy](#)
- [cudaCreateTextureObject](#)
CUresult cuTexObjectDestroy (CUtexObject texObject)
Destroys a texture object.

Parameters

texObject
- Texture object to destroy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Destroys the texture object specified by texObject.

See also:
cuTexObjectCreate, cudaDestroyTextureObject

CUresult cuTexObjectGetResourceDesc
(CUDA_RESOURCE_DESC *pResDesc, CUtexObject texObject)
Returns a texture object’s resource descriptor.

Parameters

pResDesc
- Resource descriptor
texObject
- Texture object

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Returns the resource descriptor for the texture object specified bytexObject.

See also:
cuTexObjectCreate, cudaGetTextureObjectResourceDesc,
CUresult cuTexObjectGetResourceViewDesc
(CUDA_RESOURCE_VIEW_DESC *pResViewDesc, CUtexObject texObject)
Returns a texture object’s resource view descriptor.

Parameters
pResViewDesc
- Resource view descriptor
texObject
- Texture object

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description
Returns the resource view descriptor for the texture object specified by texObject. If no resource view was set for texObject, the CUDA_ERROR_INVALID_VALUE is returned.

See also:
cuTexObjectCreate, cudaGetTextureObjectResourceViewDesc

CUresult cuTexObjectGetTextureDesc
(CUDA_TEXTURE_DESC *pTexDesc, CUtexObject texObject)
Returns a texture object’s texture descriptor.

Parameters
pTexDesc
- Texture descriptor
texObject
- Texture object

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE
Description

Returns the texture descriptor for the texture object specified by `texObject`.

See also:

`cuTexObjectCreate`, `cudaGetTextureObjectTextureDesc`

5.25. Surface Object Management

This section describes the surface object management functions of the low-level CUDA driver application programming interface. The surface object API is only supported on devices of compute capability 3.0 or higher.

`CUresult cuSurfObjectCreate (CUsurfObject *pSurfObject, const CUDA_RESOURCE_DESC *pResDesc)`
Creates a surface object.

Parameters

- `pSurfObject` - Surface object to create
- `pResDesc` - Resource descriptor

Returns

- `CUDA_SUCCESS`
- `CUDA_ERROR_DEINITIALIZED`
- `CUDA_ERROR_NOT_INITIALIZED`
- `CUDA_ERROR_INVALID_CONTEXT`
- `CUDA_ERROR_INVALID_VALUE`

Description

Creates a surface object and returns it in `pSurfObject`. `pResDesc` describes the data to perform surface load/stores on. `CUDA_RESOURCE_DESC::resType` must be `CU_RESOURCE_TYPE_ARRAY` and `CUDA_RESOURCE_DESC::res::array::hArray` must be set to a valid CUDA array handle. `CUDA_RESOURCE_DESC::flags` must be set to zero.

Surface objects are only supported on devices of compute capability 3.0 or higher. Additionally, a surface object is an opaque value, and, as such, should only be accessed through CUDA API calls.

See also:
CUresult cuSurfObjectDestroy (CUsurfObject surfObject)
Destroys a surface object.

Parameters
surfObject
   - Surface object to destroy

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description
Destroys the surface object specified by surfObject.

See also:
cuSurfObjectCreate, cudaDestroySurfaceObject

CUresult cuSurfObjectGetResourceDesc
(CUDA_RESOURCE_DESC *pResDesc, CUsurfObject surfObject)
Returns a surface object’s resource descriptor.

Parameters
pResDesc
   - Resource descriptor
surfObject
   - Surface object

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description
Returns the resource descriptor for the surface object specified by surfObject.
See also:
cuSurfObjectCreate, cudaGetSurfaceObjectResourceDesc

5.26. Peer Context Memory Access

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

CUresult cuCtxDisablePeerAccess (CUcontext peerContext)

Disables direct access to memory allocations in a peer context and unregisters any registered allocations.

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.

Description

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.

Note:

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

See also:
cuDeviceCanAccessPeer, cuCtxEnablePeerAccess, cudaDeviceDisablePeerAccess
CUresult cuCtxEnablePeerAccess (CUcontext peerContext, unsigned int Flags)

Enables direct access to memory allocations in a peer context.

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_PEER_ACCESS_ALREADY_ENABLED, CUDA_ERROR_TOO_MANY_PEERS, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_PEER_ACCESS_UNSUPPORTED, CUDA_ERROR_INVALID_VALUE

Description

If both the current context and peerContext are on devices which support unified addressing (as may be queried using CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING) and same major compute capability, then on success all allocations from peerContext will immediately be accessible by the current context. 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.

Note that there are both device-wide and system-wide limitations per system configuration, as noted in the CUDA Programming Guide under the section “Peer-to-Peer Memory Access”.

Returns CUDA_ERROR_PEER_ACCESS_UNSUPPORTED 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_TOO_MANY_PEERS if direct peer access is not possible because hardware resources required for peer access have been exhausted.

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

See also:
cuDeviceCanAccessPeer, cuCtxDisablePeerAccess, cudaDeviceEnablePeerAccess

CUresult cuDeviceCanAccessPeer (int *canAccessPeer, CUdevice dev, CUdevice peerDev)
Queries if a device may directly access a peer device’s memory.

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

Description
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().
CUresult cuDeviceGetP2PAttribute (int *value, CUdevice_P2PAttribute attrib, CUdevice srcDevice, CUdevice dstDevice)

Queries attributes of the link between two devices.

Parameters

value
  - Returned value of the requested attribute
attrib
  - The requested attribute of the link between srcDevice and dstDevice.
srcDevice
  - The source device of the target link.
dstDevice
  - The destination device of the target link.

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_INVALID_DEVICE, CUDA_ERROR_INVALID_VALUE

Description

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

- **CU_DEVICE_P2P_ATTRIBUTE_PERFORMANCE_RANK**: A relative value indicating the performance of the link between two devices.
- **CU_DEVICE_P2P_ATTRIBUTE_ACCESS_SUPPORTED P2P**: 1 if P2P Access is enable.
- **CU_DEVICE_P2P_ATTRIBUTE_NATIVE_ATOMIC_SUPPORTED**: 1 if Atomic operations over the link are supported.
- **CU_DEVICE_P2P_ATTRIBUTE_CUDA_ARRAY_ACCESS_SUPPORTED**: 1 if cudaArray can be accessed over the link.

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

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

Note:

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

See also:
cuCtxEnablePeerAccess, cuCtxDisablePeerAccess, cuDeviceCanAccessPeer, cudaDeviceGetP2PAttribute

5.27. Graphics Interoperability

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

CUresult cuGraphicsMapResources (unsigned int count, CUgraphicsResource *resources, CUstream hStream)

Map graphics resources for access by CUDA.

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

Description

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

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

This function provides the synchronization guarantee that any graphics calls issued before 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.
Note:

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

See also:


CUresult

cuGraphicsResourceGetMappedMipmappedArray

(CUmipmappedArray *pMipmappedArray, CUgraphicsResource resource)

Get a mipmapped array through which to access a mapped graphics resource.

Parameters

pMipmappedArray
- Returned mipmapped array through which resource may be accessed

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_ARRAY

Description

Returns in *pMipmappedArray a mipmapped array through which the mapped graphics resource resource. The value set in *pMipmappedArray may change every time that resource is mapped.

If resource is not a texture then it cannot be accessed via a mipmapped array and CUDA_ERROR_NOT_MAPPED_AS_ARRAY is returned. If resource is not mapped then CUDA_ERROR_NOT_MAPPED is returned.

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

See also:

CUresult cuGraphicsResourceGetMappedPointer(CUdeviceptr *pDevPtr, size_t *pSize, CUgraphicsResource resource)

Get a device pointer through which to access a mapped graphics resource.

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

Description

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.

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

See also:
CUresult cuGraphicsResourceSetMapFlags (CUgraphicsResource resource, unsigned int flags)
Set usage flags for mapping a graphics resource.

Parameters

resource
- Registered resource to set flags for

flags
- Parameters for resource mapping

Returns

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

Description

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

- 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.

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

See also:
cuGraphicsMapResources, cudaGraphicsResourceSetMapFlags
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.

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

Description

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.

Note:

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

See also:

CUresult cuGraphicsUnmapResources (unsigned int count, CUgraphicsResource *resources, CUstream hStream)

Unmap graphics resources.

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

Description

Unmaps the count graphics resources in resources.

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

This function provides the synchronization guarantee that any CUDA work issued in stream before 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.

Note:

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

See also:

- cuGraphicsMapResources
- cudaGraphicsUnmapResources
CUresult cuGraphicsUnregisterResource (CUgraphicsResource resource)
Unregisters a graphics resource for access by CUDA.

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

Description
Unregisters the graphics resource resource so it is not accessible by CUDA unless registered again.
If resource is invalid then CUDA_ERROR_INVALID_HANDLE is returned.

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

See also:

5.28. Profiler Control [DEPRECATED]
This section describes the profiler control functions of the low-level CUDA driver application programming interface.
CUresult cuProfilerInitialize (const char *configFile, const char *outputFile, CUoutput_mode outputMode)

Initialize the profiling.

Parameters

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

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

- **outputMode**
  - outputMode, can be CU_OUT_KEY_VALUE_PAIR or CU_OUT_CSV.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_PROFILER_DISABLED

Description

Deprecated

Using this API user can initialize the CUDA profiler by specifying the configuration file, output file and output file format. This API is generally used to profile different set of counters by looping the kernel launch. The configFile parameter can be used to select profiling options including profiler counters. Refer to the "Compute Command Line Profiler User Guide" for supported profiler options and counters.

Limitation: The CUDA profiler cannot be initialized with this API if another profiling tool is already active, as indicated by the CUDA_ERROR_PROFILER_DISABLED return code.

Typical usage of the profiling APIs is as follows:

for each set of counters/options { cuProfilerInitialize(); //Initialize profiling, set the counters or options in the config file ... cuProfilerStart(); // code to be profiled cuProfilerStop(); ... }

Note:

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

See also:

cuProfilerStart, cuProfilerStop, cudaProfilerInitialize
5.29. Profiler Control

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

CUresult cuProfilerStart (void)

Enable profiling.

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT

Description
Enables profile collection by the active profiling tool for the current context. If profiling is already enabled, then cuProfilerStart() has no effect.

cuProfilerStart and cuProfilerStop APIs are used to programmatically control the profiling granularity by allowing profiling to be done only on selective pieces of code.

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

See also:
cuProfilerInitialize, cuProfilerStop, cudaProfilerStart

CUresult cuProfilerStop (void)

Disable profiling.

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT

Description
Disables profile collection by the active profiling tool for the current context. If profiling is already disabled, then cuProfilerStop() has no effect.

cuProfilerStart and cuProfilerStop APIs are used to programmatically control the profiling granularity by allowing profiling to be done only on selective pieces of code.
5.30. OpenGL Interoperability

This section describes the OpenGL interoperability functions of the low-level CUDA driver application programming interface. Note that mapping of OpenGL resources is performed with the graphics API agnostic, resource mapping interface described in Graphics Interoperability.

OpenGL Interoperability [DEPRECATED]

enum CUGLDeviceList

CUDA devices corresponding to an OpenGL device

Values

CU_GL_DEVICE_LIST_ALL = 0x01
The CUDA devices for all GPUs used by the current OpenGL context

CU_GL_DEVICE_LIST_CURRENT_FRAME = 0x02
The CUDA devices for the GPUs used by the current OpenGL context in its currently rendering frame

CU_GL_DEVICE_LIST_NEXT_FRAME = 0x03
The CUDA devices for the GPUs to be used by the current OpenGL context in the next frame

CUresult cuGLGetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, CUGLDeviceList deviceList)

Gets the CUDA devices associated with the current OpenGL context.

Parameters

pCudaDeviceCount
- Returned number of CUDA devices.
pCudaDevices
- Returned CUDA devices.
cudaDeviceCount
- The size of the output device array pCudaDevices.
deviceList
- The set of devices to return.

Returns
CUDA_SUCCESS, CUDA_ERROR_NO_DEVICE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_GRAPHICS_CONTEXT

Description
Returns in *pCudaDeviceCount the number of CUDA-compatible devices corresponding to the current OpenGL context. Also returns in *pCudaDevices at most cudaDeviceCount of the CUDA-compatible devices corresponding to the current OpenGL context. If any of the GPUs being used by the current OpenGL context are not CUDA capable then the call will return CUDA_ERROR_NO_DEVICE.

The deviceList argument may be any of the following:

- CU_GL_DEVICE_LIST_ALL: Query all devices used by the current OpenGL context.
- CU_GL_DEVICE_LIST_CURRENT_FRAME: Query the devices used by the current OpenGL context to render the current frame (in SLI).
- CU_GL_DEVICE_LIST_NEXT_FRAME: Query the devices used by the current OpenGL context to render the next frame (in SLI). Note that this is a prediction, it can’t be guaranteed that this is correct in all cases.

Note:
- This function is not supported on Mac OS X.
- Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuWGLGetDevice, cudaGLGetDevices
CUresult cuGraphicsGLRegisterBuffer (CUgraphicsResource *pCudaResource, GLuint buffer, unsigned int Flags)

Registers an OpenGL buffer object.

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.

Description

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.

**Note:**

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

See also:

**CUresult cuGraphicsGLRegisterImage**
(CUgraphicsResource *pCudaResource, GLuint image, GLenum target, unsigned int Flags)

Register an OpenGL texture or renderbuffer object.

**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
- **Flags**
  - Register flags

**Returns**

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_INVALID_CONTEXT.

**Description**

Registers the texture or renderbuffer object specified by image for access by CUDA. A handle to the registered object is returned as pCudaResource.

Target must match the type of the object, and must be one of GL_TEXTURE_2D, GL_TEXTURE_RECTANGLE, GL_TEXTURE_CUBE_MAP, GL_TEXTURE_3D, GL_TEXTURE_2D_ARRAY, or GL_RENDERBUFFER.

The register flags Flags specify the intended usage, as follows:

- 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.
- CU_GRAPHICS_REGISTER_FLAGS_TEXTURE_GATHER: Specifies that CUDA will perform texture gather operations on this resource.
The following image formats are supported. For brevity's sake, the list is abbreviated. For ex., 
\{GL_R, GL_RG\} X \{8, 16\} would expand to the following 4 formats \{GL_R8, GL_R16, GL_RG8, GL_RG16\}:

- GL_RED, GL_RG, GL_RGBA, GL_LUMINANCE, GL_ALPHA, GL_LUMINANCE_ALPHA, GL_INTENSITY
- \{GL_R, GL_RG, GL_RGBA\} X \{8, 16, 16F, 32F, 8UI, 16UI, 32UI, 8I, 16I, 32I\}
- \{GL_LUMINANCE, GL_ALPHA, GL_LUMINANCE_ALPHA, GL_INTENSITY\} X \{8, 16, 16F_ARB, 32F_ARB, 8UI_EXT, 16UI_EXT, 32UI_EXT, 8I_EXT, 16I_EXT, 32I_EXT\}

The following image classes are currently disallowed:

- Textures with borders
- Multisampled renderbuffers

**Note:**

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

**See also:**

**CUresult cuWGLGetDevice (CUdevice *pDevice, HGPUNV hGpu)**

Gets the CUDA device associated with hGpu.

**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

**Description**

Returns in *pDevice the CUDA device associated with a hGpu, if applicable.
Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuGLMapBufferObject, cuGLRegisterBufferObject, cuGLUnmapBufferObject,
cuGLUnregisterBufferObject, cuGLUnmapBufferObjectAsync, cuGLSetBufferObjectMapFlags,
cudaWGLGetDevice

5.30.1. OpenGL Interoperability [DEPRECATED]
OpenGL Interoperability

This section describes deprecated OpenGL interoperability functionality.

enum CUGLmap_flags
Flags to map or unmap a resource

Values
CU_GL_MAP_RESOURCE_FLAGS_NONE = 0x00
CU_GL_MAP_RESOURCE_FLAGS_READ_ONLY = 0x01
CU_GL_MAP_RESOURCE_FLAGS_WRITE_DISCARD = 0x02

CUresult cuGLCtxCreate (CUcontext *pCtx, unsigned int Flags, CUdevice device)
Create a CUDA context for interoperability with OpenGL.

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
Description

Deprecated This function is deprecated as of Cuda 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA context with an OpenGL context in order to achieve maximum interoperability performance.

Note:

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

See also:

cuCtxCreate, cuGLInit, cuGLMapBufferObject, cuGLRegisterBufferObject, cuGLUnmapBufferObject, cuGLUnregisterBufferObject, cuGLMapBufferObjectAsync, cuGLUnregisterBufferObjectAsync, cuGLMapBufferObjectAsync, cuGLSetBufferObjectMapFlags, cuWGLGetDevice

CUresult cuGLInit (void)
Initializes OpenGL interoperability.

Returns
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_UNKNOWN

Description

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.

Note:

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

See also:

cuGLMapBufferObject, cuGLRegisterBufferObject, cuGLUnmapBufferObject, cuGLUnregisterBufferObject, cuGLSetBufferObjectMapFlags, cuWGLGetDevice
CUresult cuGLMapBufferObject (CUdeviceptr *dptr, size_t *size, GLuint buffer)
Maps an OpenGL buffer object.

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

Description

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.

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

See also:
cuGraphicsMapResources
CUresult cuGLMapBufferObjectAsync (CUdeviceptr *dptr, size_t *size, GLuint buffer, CUstream hStream)
Maps an OpenGL buffer object.

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

Description
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.

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

See also:
cuGraphicsMapResources
CUresult cuGLRegisterBufferObject (GLuint buffer)
Registers an OpenGL buffer object.

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

Description
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.

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

See also:
cuGraphicsGLRegisterBuffer

CUresult cuGLSetBufferObjectMapFlags (GLuint buffer, unsigned int Flags)
Set the map flags for an OpenGL buffer object.

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.
Description

**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_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_GL_MAPRESOURCE_FLAGS_READ_ONLY**: Specifies that CUDA kernels which access this resource will not write to this resource.
- **CU_GL_MAPRESOURCE_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.

**Note:**

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

See also:

*cuGraphicsResourceSetMapFlags*

**CUresult cuGLUnmapBufferObject (GLuint buffer)**

Unmaps an OpenGL buffer object.

**Parameters**

**buffer**
- Buffer object to unmmap

**Returns**

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE
Description

**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.

**Note:**

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

See also:

`cuGraphicsUnmapResources`

**CUresult cuGLUnmapBufferObjectAsync (GLuint buffer, CUstream hStream)**

Unmaps an OpenGL buffer object.

**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`

**Description**

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

See also:

cuGraphicsUnmapResources

CUresult cuGLUnregisterBufferObject (GLuint buffer)
Unregister an OpenGL buffer object.

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

Description

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.

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

See also:

cuGraphicsUnregisterResource

5.31. VDPAU Interoperability

This section describes the VDPAU interoperability functions of the low-level CUDA driver application programming interface.
CUresult cuGraphicsVDPAURegisterOutputSurface (CUgraphicsResource *pCudaResource, VdpOutputSurface vdpSurface, unsigned int flags)

Registers a VDPAU VdpOutputSurface object.

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.

Description

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.

Note:

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


CUresult cuGraphicsVDPAURegisterVideoSurface (CUgraphicsResource *pCudaResource, VdpVideoSurface vdpSurface, unsigned int flags)

Registers a VDPAU VdpVideoSurface object.

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.

Description

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.

Note:

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

See also:

CUresult cuVDPAUCtxCreate (CUcontext *pCtx, unsigned int flags, CUdevice device, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)

Create a CUDA context for interoperability with VDPAU.

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

Description

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()`.

Note:

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

See also:


**CUresult cuVDPAUGetDevice (CUdevice *pDevice, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)**

Gets the CUDA device associated with a VDPAU device.

**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`

**Description**

Returns in *pDevice the CUDA device associated with a vdpDevice, if applicable.

Note:

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

See also:
**5.32. EGL Interoperability**

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

```c
CUresult cuEGLStreamConsumerAcquireFrame
(CUEglStreamConnection *conn, CUGraphicsResource *pCudaResource, CUstream *pStream, unsigned int timeout)
```

Acquire an image frame from the EGLStream with CUDA as a consumer.

**Parameters**

- **conn**
  - Connection on which to acquire

- **pCudaResource**
  - CUDA resource on which the stream frame will be mapped for use.

- **pStream**
  - CUDA stream for synchronization and any data migrations implied by `CUeglResourceLocationFlags`.

- **timeout**
  - Desired timeout in usec for a new frame to be acquired. If set as `CUDA_EGL_INFINITE_TIMEOUT`, acquire waits infinitely. After timeout occurs CUDA consumer tries to acquire an old frame if available and EGL_SUPPORT_REUSE_NV flag is set.

**Returns**

`CUDA_SUCCESS`, `CUDA_ERROR_INVALID_HANDLE`, `CUDA_ERROR_LAUNCH_TIMEOUT`.

**Description**

Acquire an image frame from EGLStreamKHR. This API can also acquire an old frame presented by the producer unless explicitly disabled by setting EGL_SUPPORT_REUSE_NV flag to EGL_FALSE during stream initialization. By default, EGLStream is created with this flag set to EGL_TRUE. `cuGraphicsResourceGetMappedEglFrame` can be called on `pCudaResource` to get `CUeglFrame`.
CUresult cuEGLStreamConsumerConnect(CUeglStreamConnection *conn, EGLStreamKHR stream)

Connect CUDA to EGLStream as a consumer.

Parameters

conn
- Pointer to the returned connection handle

stream
- EGLStreamKHR handle

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_CONTEXT.

Description

Connect CUDA as a consumer to EGLStreamKHR specified by stream.

The EGLStreamKHR is an EGL object that transfers a sequence of image frames from one API to another.

See also:

cuEGLStreamConsumerConnect, cuEGLStreamConsumerDisconnect, cuEGLStreamConsumerAcquireFrame, cuEGLStreamConsumerReleaseFrame, cudaEGLStreamConsumerAcquireFrame

CUresult cuEGLStreamConsumerConnectWithFlags(CUeglStreamConnection *conn, EGLStreamKHR stream, unsigned int flags)

Connect CUDA to EGLStream as a consumer with given flags.

Parameters

conn
- Pointer to the returned connection handle
stream
  - EGLStreamKHR handle
flags
  - Flags denote intended location - system or video.

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_CONTEXT.

Description
Connect CUDA as a consumer to EGLStreamKHR specified by stream with specified flags defined by CUeglResourceLocationFlags.

The flags specify whether the consumer wants to access frames from system memory or video memory. Default is CU_EGL_Resource_Location_VIDMEM.

See also:
cuEGLStreamConsumerConnect, cuEGLStreamConsumerDisconnect,
cuEGLStreamConsumerAcquireFrame, cuEGLStreamConsumerReleaseFrame,
cudaEGLStreamConsumerConnectWithFlags

CUresult cuEGLStreamConsumerDisconnect(CUEglStreamConnection *conn)
Disconnect CUDA as a consumer to EGLStream.

Parameters
conn
  - Connection to disconnect.

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_CONTEXT.

Description
Disconnect CUDA as a consumer to EGLStreamKHR.

See also:
cuEGLStreamConsumerConnect, cuEGLStreamConsumerDisconnect,
cuEGLStreamConsumerAcquireFrame, cuEGLStreamConsumerReleaseFrame,
cudaEGLStreamConsumerDisconnect
CUresult cuEGLStreamConsumerReleaseFrame
(CUEglStreamConnection *conn, CUgraphicsResource pCudaResource, CUstream *pStream)

Releases the last frame acquired from the EGLStream.

Parameters

conn
- Connection on which to release

pCudaResource
- CUDA resource whose corresponding frame is to be released

pStream
- CUDA stream on which release will be done.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE.

Description

Release the acquired image frame specified by pCudaResource to EGLStreamKHR. If
EGL_SUPPORT_REUSE_NV flag is set to EGL_TRUE, at the time of EGL creation this API
doesn’t release the last frame acquired on the EGLStream. By default, EGLStream is created
with this flag set to EGL_TRUE.

See also:
cuEGLStreamConsumerConnect, cuEGLStreamConsumerDisconnect,
cuEGLStreamConsumerAcquireFrame, cuEGLStreamConsumerReleaseFrame,
cudaEGLStreamConsumerReleaseFrame

CUresult cuEGLStreamProducerConnect
(CUEglStreamConnection *conn, EGLStreamKHR stream, EGLint width, EGLint height)

Connect CUDA to EGLStream as a producer.

Parameters

conn
- Pointer to the returned connection handle

stream
- EGLStreamKHR handle
width
- width of the image to be submitted to the stream

height
- height of the image to be submitted to the stream

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_CONTEXT.

Description
Connect CUDA as a producer to EGLStreamKHR specified by stream.
The EGLStreamKHR is an EGL object that transfers a sequence of image frames from one API to another.

See also:
cuEGLStreamProducerConnect, cuEGLStreamProducerDisconnect, cuEGLStreamProducerPresentFrame, cudaEGLStreamProducerConnect

CUresult cuEGLStreamProducerDisconnect (CUeglStreamConnection *conn)
Disconnect CUDA as a producer to EGLStream.

Parameters
conn
- Connection to disconnect.

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_CONTEXT.

Description
Disconnect CUDA as a producer to EGLStreamKHR.

See also:
cuEGLStreamProducerConnect, cuEGLStreamProducerDisconnect, cuEGLStreamProducerPresentFrame, cudaEGLStreamProducerDisconnect
CUresult cuEGLStreamProducerPresentFrame
(CUeglStreamConnection *conn, CUeglFrame eglframe, CUstream *pStream)

Present a CUDA eglFrame to the EGLStream with CUDA as a producer.

Parameters

conn
- Connection on which to present the CUDA array

eglframe
- CUDA Eglstream Producer Frame handle to be sent to the consumer over EglStream.

pStream
- CUDA stream on which to present the frame.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE.

Description

When a frame is presented by the producer, it gets associated with the EGLStream and thus it
is illegal to free the frame before the producer is disconnected. If a frame is freed and reused
it may lead to undefined behavior.

If producer and consumer are on different GPUs (iGPU and dGPU) then frametype
CU_EGL_FRAME_TYPE_ARRAY is not supported. CU_EGL_FRAME_TYPE_PITCH can be used
for such cross-device applications.

The CUeglFrame is defined as:

```c
typedef struct CUeglFrame_st {
    union {
        CUarray pArray[MAX_PLANES];
        void*   pPitch[MAX_PLANES];
    } frame;
    unsigned int width;
    unsigned int height;
    unsigned int depth;
    unsigned int pitch;
    unsigned int planeCount;
    unsigned int numChannels;
    CUeglFrameType frameType;
    CUeglColorFormat eglColorFormat;
    CUarray_format cuFormat;
} CUeglFrame;
```

For CUeglFrame of type CU_EGL_FRAME_TYPE_PITCH, the application may present sub-
region of a memory allocation. In that case, the pitched pointer will specify the start address
of the sub-region in the allocation and corresponding CUeglFrame fields will specify the
dimensions of the sub-region.
See also:

cuEGLStreamProducerConnect, cuEGLStreamProducerDisconnect, 
cuEGLStreamProducerReturnFrame, cudaEGLStreamProducerPresentFrame

CUresult cuEGLStreamProducerReturnFrame (CUeglStreamConnection *conn, CUeglFrame *eglframe, CUstream *pStream)
Return the CUDA eglFrame to the EGLStream released by the consumer.

Parameters
conn
- Connection on which to return
eglframe
- CUDA Eglstream Proucer Frame handle returned from the consumer over EglStream.
pStream
- CUDA stream on which to return the frame.

Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_LAUNCH_TIMEOUT

Description
This API can potentially return CUDA_ERROR_LAUNCH_TIMEOUT if the consumer has not returned a frame to EGL stream. If timeout is returned the application can retry.

See also:

cuEGLStreamProducerConnect, cuEGLStreamProducerDisconnect, 
cuEGLStreamProducerPresentFrame, cudaEGLStreamProducerReturnFrame

CUresult cuEventCreateFromEGLSync (CUevent *phEvent, EGLSyncKHR eglSync, unsigned int flags)
Creates an event from EGLSync object.

Parameters
phEvent
- Returns newly created event
eglSync
- Opaque handle to EGLSync object
**flags**

- Event creation flags

**Returns**

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

**Description**

Creates an event *phEvent from an EGLSyncKHR eglSync 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 completed.

Once the eglSync gets destroyed, cuEventDestroy is the only API that can be invoked on the event.

cuEventRecord and TimingData are not supported for events created from EGLSync.
The EGLSyncKHR is an opaque handle to an EGL sync object. typedef void* EGLSyncKHR

**See also:**

cuEventQuery, cuEventSynchronize, cuEventDestroy

**CResult cuGraphicsEGLRegisterImage**

(CUgraphicsResource *pCudaResource, EGLImageKHR image, unsigned int flags)

Registers an EGL image.

**Parameters**

- **pCudaResource**
  - Pointer to the returned object handle
- **image**
  - An EGLImageKHR image which can be used to create target resource.
- **flags**
  - Map flags
Returns
CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_INVALID_CONTEXT.

Description
Registers the EGLImageKHR specified by image for access by CUDA. A handle to the registered object is returned as pCudaResource. Additional Mapping/Unmapping is not required for the registered resource and cuGraphicsResourceGetMappedEglFrame can be directly called on the pCudaResource.

The application will be responsible for synchronizing access to shared objects. The application must ensure that any pending operation which access the objects have completed before passing control to CUDA. This may be accomplished by issuing and waiting for glFinish command on all GLcontexts (for OpenGL and likewise for other APIs). The application will be also responsible for ensuring that any pending operation on the registered CUDA resource has completed prior to executing subsequent commands in other APIs accessing the same memory objects. This can be accomplished by calling cuCtxSynchronize or cuEventSynchronize (preferably).

The surface’s intended usage is specified using flags, as follows:

- 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 EGLImageKHR is an object which can be used to create EGLImage target resource. It is defined as a void pointer. typedef void* EGLImageKHR

See also:
CUresult cuGraphicsResourceGetMappedEglFrame(CUeglFrame *eglFrame, CUgraphicsResource resource, unsigned int index, unsigned int mipLevel)

Get an eglFrame through which to access a registered EGL graphics resource.

Parameters

eglFrame
- Returned eglFrame.

resource
- Registered resource to access.

index
- Index for cubemap surfaces.

mipLevel
- Mipmap level for the subresource to access.

Returns

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

Description

Returns in *eglFrame an eglFrame pointer through which the registered graphics resource may be accessed. This API can only be called for registered EGL graphics resources.

The CUeglFrame is defined as:

```c
typedef struct CUeglFrame_st {
    union {
        CUarray pArray[MAX_PLANES];
        void*   pPitch[MAX_PLANES];
    } frame;
    unsigned int width;
    unsigned int height;
    unsigned int depth;
    unsigned int pitch;
    unsigned int planeCount;
    unsigned int numChannels;
    CUeglFrameType frameType;
    CUeglColorFormat eglColorFormat;
    CUarray_format cuFormat;
} CUeglFrame;
```

If resource is not registered then CUDA_ERROR_NOT_MAPPED is returned.

See also:
cuGraphicsMapResources, cuGraphicsSubResourceGetMappedArray, 
Chapter 6. Data Structures

Here are the data structures with brief descriptions:

- **CUaccessPolicyWindow**
- **CUarrayMapInfo**
- **CUDA_ARRAY3D_DESCRIPTOR**
- **CUDA_ARRAY_DESCRIPTOR**
- **CUDA_ARRAY_SPARSE_PROPERTIES**
- **CUDA_EXTERNAL_MEMORY_BUFFER_DESC**
- **CUDA_EXTERNAL_MEMORY_HANDLE_DESC**
- **CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC**
- **CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC**
- **CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS**
- **CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS**
- **CUDA_HOST_NODE_PARAMS**
- **CUDA_KERNEL_NODE_PARAMS**
- **CUDA_LAUNCH_PARAMS**
- **CUDA_MEMCPY2D**
- **CUDA_MEMCPY3D**
- **CUDA_MEMCPY3D_PEER**
- **CUDA_MEMSET_NODE_PARAMS**
- **CUDA_POINTER_ATTRIBUTE_P2P_TOKENS**
- **CUDA_RESOURCE_DESC**
- **CUDA_RESOURCE_VIEW_DESC**
- **CUDA_TEXTURE_DESC**
- **CUdevprop**
- **CUEglFrame**
- **CUipcEventHandle**
- **CUipcMemHandle**
- **CUkernelNodeAttrValue**
- **CUMemAccessDesc**
- **CUMemAllocationProp**
- **CUMemLocation**
- **CUSTreamAttrValue**
- **CUSTreamBatchMemOpParams**
6.1. **CUaccessPolicyWindow Struct Reference**

Specifies an access policy for a window, a contiguous extent of memory beginning at base_ptr and ending at base_ptr + num_bytes. num_bytes is limited by

CU_DEVICE_ATTRIBUTE_MAX_ACCESS_POLICY_WINDOW_SIZE. Partition into many segments and assign segments such that: sum of "hit segments" / window == approx. ratio. sum of "miss segments" / window == approx 1-ratio. Segments and ratio specifications are fitted to the capabilities of the architecture. Accesses in a hit segment apply the hitProp access policy. Accesses in a miss segment apply the missProp access policy.

```c
void *CUaccessPolicyWindow::base_ptr
```

Starting address of the access policy window. CUDA driver may align it.

```c
CUaccessProperty CUaccessPolicyWindow::hitProp
```

CUaccessProperty set for hit.

```c
float CUaccessPolicyWindow::hitRatio
```

hitRatio specifies percentage of lines assigned hitProp, rest are assigned missProp.

```c
CUaccessProperty CUaccessPolicyWindow::missProp
```

CUaccessProperty set for miss. Must be either NORMAL or STREAMING

```c
size_t CUaccessPolicyWindow::num_bytes
```

Size in bytes of the window policy. CUDA driver may restrict the maximum size and alignment.

6.2. **CUarrayMapInfo Struct Reference**

Specifies the CUDA array or CUDA mipmapped array memory mapping information

```c
unsigned int CUarrayMapInfo::deviceBitMask
```

Device ordinal bit mask
**unsigned int CUarrayMapInfo::extentDepth**
Depth in elements

**unsigned int CUarrayMapInfo::extentHeight**
Height in elements

**unsigned int CUarrayMapInfo::extentWidth**
Width in elements

**unsigned int CUarrayMapInfo::flags**
flags for future use, must be zero now.

**unsigned int CUarrayMapInfo::layer**
For CUDA layered arrays must be a valid layer index. Otherwise, must be zero

**unsigned int CUarrayMapInfo::level**
For CUDA mipmapped arrays must a valid mipmap level. For CUDA arrays must be zero

**CUmemHandleType**
**CUarrayMapInfo::memHandleType**
Memory handle type

**CUmemOperationType**
**CUarrayMapInfo::memOperationType**
Memory operation type

**unsigned long long CUarrayMapInfo::offset**
Offset within mip tail
Offset within the memory

**unsigned int CUarrayMapInfo::offsetX**
Starting X offset in elements
unsigned int CUarrayMapInfo::offsetY
Starting Y offset in elements

unsigned int CUarrayMapInfo::offsetZ
Starting Z offset in elements

unsigned int CUarrayMapInfo::reserved
Reserved for future use, must be zero now.

CUresourcetype CUarrayMapInfo::resourceType
Resource type

unsigned long long CUarrayMapInfo::size
Extent in bytes

CUArraySparseSubresourceType
CUarrayMapInfo::subresourceType
Sparse subresource type

6.3. CUDA_ARRAY3D_DESCRIPTOR
Struct Reference

3D array descriptor

size_t CUDA_ARRAY3D_DESCRIPTOR::Depth
Depth of 3D array

unsigned int CUDA_ARRAY3D_DESCRIPTOR::Flags
Flags
CUarray_format
CUDA_ARRAY3D_DESCRIPTOR::Format
Array format

size_t CUDA_ARRAY3D_DESCRIPTOR::Height
Height of 3D array

unsigned int
CUDA_ARRAY3D_DESCRIPTOR::NumChannels
Channels per array element

size_t CUDA_ARRAY3D_DESCRIPTOR::Width
Width of 3D array

6.4. CUDA_ARRAY_DESCRIPTOR Struct
Reference

Array descriptor

CUarray_format CUDA_ARRAY_DESCRIPTOR::Format
Array format

size_t CUDA_ARRAY_DESCRIPTOR::Height
Height of array

unsigned int
CUDA_ARRAY_DESCRIPTOR::NumChannels
Channels per array element

size_t CUDA_ARRAY_DESCRIPTOR::Width
Width of array
6.5. CUDA_ARRAY_SPARSE_PROPERTIES Struct Reference

CUDA array sparse properties

unsigned int
CUDA_ARRAY_SPARSE_PROPERTIES::depth
Depth of sparse tile in elements

unsigned int
CUDA_ARRAY_SPARSE_PROPERTIES::flags
Flags will either be zero or CU_ARRAY_SPARSE_PROPERTIES_SINGLE_MIPTAIL

unsigned int
CUDA_ARRAY_SPARSE_PROPERTIES::height
Height of sparse tile in elements

unsigned int
CUDA_ARRAY_SPARSE_PROPERTIES::miptailFirstLevel
First mip level at which the mip tail begins.

unsigned long long
CUDA_ARRAY_SPARSE_PROPERTIES::miptailSize
Total size of the mip tail.

unsigned int
CUDA_ARRAY_SPARSE_PROPERTIES::width
Width of sparse tile in elements
6.6. CUDA_EXTERNAL_MEMORY_BUFFER_DESC Struct Reference

External memory buffer descriptor

unsigned int
CUDA_EXTERNAL_MEMORY_BUFFER_DESC::flags

Flags reserved for future use. Must be zero.

unsigned long long
CUDA_EXTERNAL_MEMORY_BUFFER_DESC::offset

Offset into the memory object where the buffer’s base is.

unsigned long long
CUDA_EXTERNAL_MEMORY_BUFFER_DESC::size

Size of the buffer.

6.7. CUDA_EXTERNAL_MEMORY_HANDLE_DESC Struct Reference

External memory handle descriptor

int CUDA_EXTERNAL_MEMORY_HANDLE_DESC::fd

File descriptor referencing the memory object. Valid when type is
CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD

unsigned int
CUDA_EXTERNAL_MEMORY_HANDLE_DESC::flags

Flags must either be zero or CUDA_EXTERNAL_MEMORY_DEDICATED
void
*CUDA_EXTERNAL_MEMORY_HANDLE_DESC::*handle

Valid NT handle. Must be NULL if 'name' is non-NULL.

const void
*CUDA_EXTERNAL_MEMORY_HANDLE_DESC::*name

Name of a valid memory object. Must be NULL if 'handle' is non-NULL.

const void
*CUDA_EXTERNAL_MEMORY_HANDLE_DESC::*nvSciBufObject

A handle representing an NvSciBuf Object. Valid when type is
CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF

unsigned long long
CUDA_EXTERNAL_MEMORY_HANDLE_DESC::*size

Size of the memory allocation

CUexternalMemoryHandleType
CUDA_EXTERNAL_MEMORY_HANDLE_DESC::*type

Type of the handle

CUDA_EXTERNAL_MEMORY_HANDLE_DESC::*@11::*@12
CUDA_EXTERNAL_MEMORY_HANDLE_DESC::*win32

Win32 handle referencing the semaphore object. Valid when type is one of the following:

- CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32
- CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT
- CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP
- CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE
- CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE
- CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE_KMT

Exactly one of 'handle' and 'name' must be non-NULL. If type is one of the following: CU_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT
6.8. **CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC**

Struct Reference

External memory mipmap descriptor

```c
struct CUDA_ARRAY3D_DESCRIPTOR
CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC::arrayDesc;
```

Format, dimension and type of base level of the mipmap chain

```c
unsigned int
CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC::numLevels;
```

Total number of levels in the mipmap chain

```c
unsigned long long
CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC::offset;
```

Offset into the memory object where the base level of the mipmap chain is.

6.9. **CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC**

Struct Reference

External semaphore handle descriptor

```c
int
CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::fd;
```

File descriptor referencing the semaphore object. Valid when type is

```c
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD
```

CU_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_RESOURCE_KMT then 'name' must be NULL.
unsigned int
CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::flags

Flags reserved for the future. Must be zero.

void
*CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::handle

Valid NT handle. Must be NULL if ‘name’ is non-NULL

const void
*CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::name

Name of a valid synchronization primitive. Must be NULL if ‘handle’ is non-NULL.

const void
*CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::nvSciSyncObj

Valid NvSciSyncObj. Must be non NULL

CUexternalSemaphoreHandleType
CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::type

Type of the handle

CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::@13::@14
CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC::win32

Win32 handle referencing the semaphore object. Valid when type is one of the following:

- CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32
- CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT
- CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE
- CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE
- CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX

Exactly one of ‘handle’ and ‘name’ must be non-NULL. If type is one of the following: CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT
CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_KEYED_MUTEX_KMT then ‘name’ must be NULL.
6.10. CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS Struct Reference

External semaphore signal parameters

**void**

*CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS::*fence

Pointer to NvSciSyncFence. Valid if **CUDA_EXTERNAL_SEMAPHORE_HANDLE_TYPE** is of type **CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC**.

CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS::*@15::*@16

CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS::*fence

Parameters for fence objects

**unsigned int**

CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS::*flags

Only when **CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS** is used to signal a **CUDA_EXTERNAL_SEMAPHORE** of type **CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC**, the valid flag is **CUDA_EXTERNAL_SEMAPHORE_SIGNAL_SKIP_NVSCIBUF_MEMSYNC** which indicates that while signaling the **CUDA_EXTERNAL_SEMAPHORE**, no memory synchronization operations should be performed for any external memory object imported as **CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF**. For all other types of **CUDA_EXTERNAL_SEMAPHORE**, flags must be zero.

**unsigned long long**

CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS::*key

Value of key to release the mutex with

CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS::*@15::*@18

CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS::*keyedMutex

Parameters for keyed mutex objects
unsigned long long
CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS::value

Value of fence to be signaled

6.11. CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS

Struct Reference

External semaphore wait parameters

CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS::@19::@20
CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS::fence

Parameters for fence objects

unsigned int
CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS::flags

Only when CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS is used to wait on a
CUexternalSemaphore of type CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC,
the valid flag is CUDA_EXTERNAL_SEMAPHORE_WAIT_SKIP_NVSCIBUF_MEMSYNC
which indicates that while waiting for the CUexternalSemaphore, no memory
synchronization operations should be performed for any external memory object
imported as CU_EXTERNAL_MEMORY_HANDLE_TYPE_NVSCIBUF. For all other types of
CUexternalSemaphore, flags must be zero.

unsigned long long
CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS::key

Value of key to acquire the mutex with

CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS::@19::@22
CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS::keyedMutex

Parameters for keyed mutex objects
CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS::@19::@21
CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS::nvSciSync

Pointer to NvSciSyncFence. Valid if CUexternalSemaphoreHandleType is of type CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC.

unsigned int
CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS::timeoutMs

Timeout in milliseconds to wait to acquire the mutex

unsigned long long
CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS::value

Value of fence to be waited on

6.12. CUDA_HOST_NODE_PARAMS Struct Reference

Host node parameters

CUhostFn CUDA_HOST_NODE_PARAMS::fn

The function to call when the node executes

void *CUDA_HOST_NODE_PARAMS::userData

Argument to pass to the function

6.13. CUDAKERNEL_NODE_PARAMS Struct Reference

GPU kernel node parameters
unsigned int
CUDA_KERNEL_NODE_PARAMS::blockDimX
X dimension of each thread block

unsigned int
CUDA_KERNEL_NODE_PARAMS::blockDimY
Y dimension of each thread block

unsigned int
CUDA_KERNEL_NODE_PARAMS::blockDimZ
Z dimension of each thread block

**CUDA_KERNEL_NODE_PARAMS::extra
Extra options

CUfunction CUDA_KERNEL_NODE_PARAMS::func
Kernel to launch

unsigned int
CUDA_KERNEL_NODE_PARAMS::gridDimX
Width of grid in blocks

unsigned int
CUDA_KERNEL_NODE_PARAMS::gridDimY
Height of grid in blocks

unsigned int
CUDA_KERNEL_NODE_PARAMS::gridDimZ
Depth of grid in blocks
**CUDA_KERNEL_NODE_PARAMS::kernelParams**

Array of pointers to kernel parameters

`unsigned int CUDA_KERNEL_NODE_PARAMS::sharedMemBytes`

Dynamic shared-memory size per thread block in bytes

### 6.14. `CUDA_LAUNCH_PARAMS` Struct

**Reference**

Kernel launch parameters

`unsigned int CUDA_LAUNCH_PARAMS::blockDimX`

X dimension of each thread block

`unsigned int CUDA_LAUNCH_PARAMS::blockDimY`

Y dimension of each thread block

`unsigned int CUDA_LAUNCH_PARAMS::blockDimZ`

Z dimension of each thread block

`CUfunction CUDA_LAUNCH_PARAMS::function`

Kernel to launch

`unsigned int CUDA_LAUNCH_PARAMS::gridDimX`

Width of grid in blocks

`unsigned int CUDA_LAUNCH_PARAMS::gridDimY`

Height of grid in blocks

`unsigned int CUDA_LAUNCH_PARAMS::gridDimZ`

Depth of grid in blocks
CUstream CUDA_LAUNCH_PARAMS::hStream
Stream identifier

**CUDA_LAUNCH_PARAMS::kernelParams
Array of pointers to kernel parameters

unsigned int
CUDA_LAUNCH_PARAMS::sharedMemBytes
Dynamic shared-memory size per thread block in bytes

### 6.15. CUDA_MEMCPY2D Struct Reference

2D memory copy parameters

CUarray CUDA_MEMCPY2D::dstArray
Destination array reference

CUdeviceptr CUDA_MEMCPY2D::dstDevice
Destination device pointer

void *CUDA_MEMCPY2D::dstHost
Destination host pointer

CUmemorytype CUDA_MEMCPY2D::dstMemoryType
Destination memory type (host, device, array)

size_t CUDA_MEMCPY2D::dstPitch
Destination pitch (ignored when dst is array)

size_t CUDA_MEMCPY2D::dstXInBytes
Destination X in bytes
size_t CUDA_MEMCPY2D::dstY
Destination Y

size_t CUDA_MEMCPY2D::Height
Height of 2D memory copy

CUarray CUDA_MEMCPY2D::srcArray
Source array reference

CUdeviceptr CUDA_MEMCPY2D::srcDevice
Source device pointer

const void *CUDA_MEMCPY2D::srcHost
Source host pointer

CUmemorytype CUDA_MEMCPY2D::srcMemoryType
Source memory type [host, device, array]

size_t CUDA_MEMCPY2D::srcPitch
Source pitch [ignored when src is array]

size_t CUDA_MEMCPY2D::srcXInBytes
Source X in bytes

size_t CUDA_MEMCPY2D::srcY
Source Y

size_t CUDA_MEMCPY2D::WidthInBytes
Width of 2D memory copy in bytes
6.16. CUDA_MEMCPY3D Struct Reference

3D memory copy parameters

size_t CUDA_MEMCPY3D::Depth
Depth of 3D memory copy

CUarray CUDA_MEMCPY3D::dstArray
Destination array reference

CUdeviceptr CUDA_MEMCPY3D::dstDevice
Destination device pointer

size_t CUDA_MEMCPY3D::dstHeight
Destination height (ignored when dst is array; may be 0 if Depth==1)

void *CUDA_MEMCPY3D::dstHost
Destination host pointer

size_t CUDA_MEMCPY3D::dstLOD
Destination LOD

CUmemorytype CUDA_MEMCPY3D::dstMemoryType
Destination memory type (host, device, array)

size_t CUDA_MEMCPY3D::dstPitch
Destination pitch (ignored when dst is array)

size_t CUDA_MEMCPY3D::dstXInBytes
Destination X in bytes
size_t CUDA_MEMCPY3D::dstY
Destination Y

size_t CUDA_MEMCPY3D::dstZ
Destination Z

size_t CUDA_MEMCPY3D::Height
Height of 3D memory copy

void *CUDA_MEMCPY3D::reserved0
Must be NULL

void *CUDA_MEMCPY3D::reserved1
Must be NULL

CUarray CUDA_MEMCPY3D::srcArray
Source array reference

CUdeviceptr CUDA_MEMCPY3D::srcDevice
Source device pointer

size_t CUDA_MEMCPY3D::srcHeight
Source height (ignored when src is array; may be 0 if Depth==1)

const void *CUDA_MEMCPY3D::srcHost
Source host pointer

size_t CUDA_MEMCPY3D::srcLOD
Source LOD

CUmemorytype CUDA_MEMCPY3D::srcMemoryType
Source memory type (host, device, array)
size_t CUDA_MEMCPY3D::srcPitch
Source pitch (ignored when src is array)

size_t CUDA_MEMCPY3D::srcXInBytes
Source X in bytes

size_t CUDA_MEMCPY3D::srcY
Source Y

size_t CUDA_MEMCPY3D::srcZ
Source Z

size_t CUDA_MEMCPY3D::WidthInBytes
Width of 3D memory copy in bytes

6.17. CUDA_MEMCPY3D_PEER Struct
Reference

3D memory cross-context copy parameters

size_t CUDA_MEMCPY3D_PEER::Depth
Depth of 3D memory copy

CUarray CUDA_MEMCPY3D_PEER::dstArray
Destination array reference

CUcontext CUDA_MEMCPY3D_PEER::dstContext
Destination context (ignored with dstMemoryType is CU_MEMORYTYPE_ARRAY)

CUdeviceptr CUDA_MEMCPY3D_PEER::dstDevice
Destination device pointer
size_t CUDA_MEMCPY3D_PEER::dstHeight
Destination height (ignored when dst is array; may be 0 if Depth==1)

void *CUDA_MEMCPY3D_PEER::dstHost
Destination host pointer

size_t CUDA_MEMCPY3D_PEER::dstLOD
Destination LOD

CUmemorytype CUDA_MEMCPY3D_PEER::dstMemoryType
Destination memory type (host, device, array)

size_t CUDA_MEMCPY3D_PEER::dstPitch
Destination pitch (ignored when dst is array)

size_t CUDA_MEMCPY3D_PEER::dstXInBytes
Destination X in bytes

size_t CUDA_MEMCPY3D_PEER::dstY
Destination Y

size_t CUDA_MEMCPY3D_PEER::dstZ
Destination Z

size_t CUDA_MEMCPY3D_PEER::Height
Height of 3D memory copy

CUarray CUDA_MEMCPY3D_PEER::srcArray
Source array reference
CUcontext CUDA_MEMCPY3D_PEER::srcContext
Source context (ignored with srcMemoryType is CU_MEMORYTYPE_ARRAY)

CUdeviceptr CUDA_MEMCPY3D_PEER::srcDevice
Source device pointer

size_t CUDA_MEMCPY3D_PEER::srcHeight
Source height (ignored when src is array; may be 0 if Depth==1)

const void *CUDA_MEMCPY3D_PEER::srcHost
Source host pointer

size_t CUDA_MEMCPY3D_PEER::srcLOD
Source LOD

CUmemorytype CUDA_MEMCPY3D_PEER::srcMemoryType
Source memory type (host, device, array)

size_t CUDA_MEMCPY3D_PEER::srcPitch
Source pitch (ignored when src is array)

size_t CUDA_MEMCPY3D_PEER::srcXInBytes
Source X in bytes

size_t CUDA_MEMCPY3D_PEER::srcY
Source Y

size_t CUDA_MEMCPY3D_PEER::srcZ
Source Z
size_t CUDA_MEMCPY3D_PEER::WidthInBytes
Width of 3D memory copy in bytes

6.18.  CUDA_MEMSET_NODE_PARAMS
Struct Reference
Memset node parameters

CUdeviceptr CUDA_MEMSET_NODE_PARAMS::dst
Destination device pointer

unsigned int
CUDA_MEMSET_NODE_PARAMS::elementSize
Size of each element in bytes. Must be 1, 2, or 4.

size_t CUDA_MEMSET_NODE_PARAMS::height
Number of rows

size_t CUDA_MEMSET_NODE_PARAMS::pitch
Pitch of destination device pointer. Unused if height is 1

unsigned int CUDA_MEMSET_NODE_PARAMS::value
Value to be set

size_t CUDA_MEMSET_NODE_PARAMS::width
Width of the row in elements

6.19.  CUDA_POINTER_ATTRIBUTE_P2P_TOKENS
Struct Reference
GPU Direct v3 tokens
6.20. CUDA_RESOURCE_DESC Struct Reference

CUDA Resource descriptor

CUdeviceptr CUDARESOURCE_DESC::devPtr
Device pointer

unsigned int CUDARESOURCE_DESC::flags
Flags (must be zero)

CUarray_format CUDARESOURCE_DESC::format
Array format

CUarray CUDARESOURCE_DESC::hArray
CUDA array

size_t CUDARESOURCE_DESC::height
Height of the array in elements

C UMipmappedArray
CUDARESOURCE_DESC::hMipmappedArray
CUDA mipmapped array

unsigned int CUDARESOURCE_DESC::numChannels
Channels per array element

size_t CUDARESOURCE_DESC::pitchInBytes
Pitch between two rows in bytes

C Uresourcetype CUDARESOURCE_DESC::resType
Resource type
size_t CUDARESOURCEDESC::sizeInBytes
Size in bytes

size_t CUDARESOURCEDESC::width
Width of the array in elements

6.21. CUDARESOURCEVIEWDESC
Struct Reference

Resource view descriptor

size_t CUDARESOURCEVIEWDESC::depth
Depth of the resource view

unsigned int
CUDARESOURCEVIEWDESC::firstLayer
First layer index

unsigned int
CUDARESOURCEVIEWDESC::firstMipmapLevel
First defined mipmap level

CUresourceViewFormat
CUDARESOURCEVIEWDESC::format
Resource view format

size_t CUDARESOURCEVIEWDESC::height
Height of the resource view
unsigned int
CUDA_RESOURCE_VIEW_DESC::lastLayer
Last layer index

unsigned int
CUDA_RESOURCE_VIEW_DESC::lastMipmapLevel
Last defined mipmap level

size_t CUDA_RESOURCE_VIEW_DESC::width
Width of the resource view

6.22. CUDA_TEXTURE_DESC Struct
Reference

Texture descriptor

CUaddress_mode
CUDA_TEXTURE_DESC::addressMode
Address modes

float CUDA_TEXTURE_DESC::borderColor
Border Color

CUfilter_mode CUDA_TEXTURE_DESC::filterMode
Filter mode

unsigned int CUDA_TEXTURE_DESC::flags
Flags

unsigned int CUDA_TEXTURE_DESC::maxAnisotropy
Maximum anisotropy ratio
float CUDA_TEXTURE_DESC::maxMipmapLevelClamp
Mipmap maximum level clamp

float CUDA_TEXTURE_DESC::minMipmapLevelClamp
Mipmap minimum level clamp

CUfilter_mode
CUDA_TEXTURE_DESC::mipmapFilterMode
Mipmap filter mode

float CUDA_TEXTURE_DESC::mipmapLevelBias
Mipmap level bias

6.23.  CUdevprop Struct Reference
Legacy device properties

int CUdevprop::clockRate
Clock frequency in kilohertz

int CUdevprop::maxGridSize
Maximum size of each dimension of a grid

int CUdevprop::maxThreadsDim
Maximum size of each dimension of a block

int CUdevprop::maxThreadsPerBlock
Maximum number of threads per block

int CUdevprop::memPitch
Maximum pitch in bytes allowed by memory copies
int CUdevprop::regsPerBlock
32-bit registers available per block

int CUdevprop::sharedMemPerBlock
Shared memory available per block in bytes

int CUdevprop::SIMDWidth
Warp size in threads

int CUdevprop::textureAlign
Alignment requirement for textures

int CUdevprop::totalConstantMemory
Constant memory available on device in bytes

6.24. CUeglFrame Struct Reference
CUDA EGLFrame structure Descriptor - structure defining one frame of EGL.
Each frame may contain one or more planes depending on whether the surface is Multiplanar or not.

CUarray_format CUeglFrame::cuFormat
CUDA Array Format

unsigned int CUeglFrame::depth
Depth of first plane

CUeglColorFormat CUeglFrame::eglColorFormat
CUDA EGL Color Format

CUeglFrameType CUeglFrame::frameType
Array or Pitch
unsigned int CUeglFrame::height
Height of first plane

unsigned int CUeglFrame::numChannels
Number of channels for the plane

CUArray CUeglFrame::pArray
Array of CUArray corresponding to each plane

unsigned int CUeglFrame::pitch
Pitch of first plane

unsigned int CUeglFrame::planeCount
Number of planes

void *CUeglFrame::pPitch
Array of Pointers corresponding to each plane

unsigned int CUeglFrame::width
Width of first plane

6.25. CUipcEventHandle Struct Reference
CUDA IPC event handle

6.26. CUipcMemHandle Struct Reference
CUDA IPC mem handle
6.27. **CUkernelNodeAttrValue Union Reference**

Graph kernel node attributes union, used with cuKernelNodeSetAttribute/cuKernelNodeGetAttribute

```c
struct CUaccessPolicyWindow

CUkernelNodeAttrValue::accessPolicyWindow
```

Attribute `CUaccessPolicyWindow`.

```c
int CUkernelNodeAttrValue::cooperative
```

Nonzero indicates a cooperative kernel (see `cuLaunchCooperativeKernel`).

6.28. **CUmemAccessDesc Struct Reference**

Memory access descriptor

```c
CUmemAccess_flags CUmemAccessDesc::flags
```

`CUmemProt` accessibility flags to set on the request

```c
struct CUmemLocation CUmemAccessDesc::location
```

Location on which the request is to change it’s accessibility

6.29. **CUmemAllocationProp Struct Reference**

Specifies the allocation properties for a allocation.

```c
unsigned char
CUmemAllocationProp::compressionType
```

Allocation hint for requesting compressible memory. On devices that support Compute Data Compression, compressible memory can be used to accelerate accesses to data.
with unstructured sparsity and other compressible data patterns. Applications are expected to query allocation property of the handle obtained with cuMemCreate using cuMemGetAllocationPropertiesFromHandle to validate if the obtained allocation is compressible or not. Note that compressed memory may not be mappable on all devices.

```c
struct CUmemLocation
CUmemAllocationProp::location
```
Location of allocation

```c
CUmemAllocationHandleType
CUmemAllocationProp::requestedHandleTypes
```
requested CUmemAllocationHandleType

```c
CUmemAllocationType CUmemAllocationProp::type
```
Allocation type

```c
unsigned short CUmemAllocationProp::usage
```
Bitmask indicating intended usage for this allocation

```c
void *CUmemAllocationProp::win32HandleMetaData
```
Windows-specific LPSECURITYATTRIBUTES required when CU_MEM_HANDLE_TYPE_WIN32 is specified. This security attribute defines the scope of which exported allocations may be transferred to other processes. In all other cases, this field is required to be zero.

### 6.30. CUmemLocation Struct Reference

Specifies a location for an allocation.

```c
int CUmemLocation::id
```
identifier for a given this location’s CUmemLocationType.

```c
CUmemLocationType CUmemLocation::type
```
Specifies the location type, which modifies the meaning of id.
6.31. **CUstreamAttrValue Union Reference**

Stream attributes union, used with cuStreamSetAttribute/cuStreamGetAttribute

```c
struct CUaccessPolicyWindow
CUstreamAttrValue::accessPolicyWindow
```

Attribute **CUaccessPolicyWindow**.

**CUsynchronizationPolicy**

```c
CUstreamAttrValue::syncPolicy
```

Value for **CU_STREAM_ATTRIBUTE_SYNCHRONIZATION_POLICY**.

6.32. **CUstreamBatchMemOpParams Union Reference**

Per-operation parameters for cuStreamBatchMemOp
Chapter 7.  Data Fields

Here is a list of all documented struct and union fields with links to the struct/union documentation for each field:

A

accessPolicyWindow
  CUkernelNodeAttrValue
  CUstreamAttrValue
addressMode
  CUDA_TEXTURE_DESC
arrayDesc
  CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC

B

base_ptr
  CUaccessPolicyWindow
blockDimX
  CUDA_KERNEL_NODE_PARAMS
  CUDA_LAUNCH_PARAMS
blockDimY
  CUDA_KERNEL_NODE_PARAMS
  CUDA_LAUNCH_PARAMS
blockDimZ
  CUDA_KERNEL_NODE_PARAMS
  CUDA_LAUNCH_PARAMS
borderColor
  CUDA_TEXTURE_DESC

C

clockRate
  CUdevprop
compressionType
  CUmemp AllocationProp
cooperative
  CUKernelNodeAttrValue

cuFormat
  CUeglFrame

D

depth
  CUDA ARRAY SPARSE_PROPERTIES
  CUDA RESOURCE VIEW_DESC

Depth
  CUDA_MEMCPY3D
  CUDA_MEMCPY3D_PEER

depth
  CUeglFrame

Depth
  CUDA ARRAY3D_DESCRIPTOR

deviceBitMask
  CUarrayMapInfo
devPtr
  CUDA RESOURCE_DESC
dst
  CUDA_MEMSET_NODE_PARAMS
dstArray
  CUDA_MEMCPY2D
  CUDA_MEMCPY3D
  CUDA_MEMCPY3D_PEER
dstContext
  CUDA_MEMCPY3D_PEER
dstDevice
  CUDA_MEMCPY3D_PEER
  CUDA_MEMCPY2D
  CUDA_MEMCPY3D
dstHeight
  CUDA_MEMCPY3D
  CUDA_MEMCPY3D_PEER
dstHost
  CUDA_MEMCPY2D
  CUDA_MEMCPY3D
  CUDA_MEMCPY3D_PEER
dstLOD
  CUDA_MEMCPY3D
  CUDA_MEMCPY3D_PEER
dstMemoryType
   CUDA_MEMCPY2D
   CUDA_MEMCPY3D
   CUDA_MEMCPY3D_PEER
dstPitch
   CUDA_MEMCPY3D_PEER
   CUDA_MEMCPY2D
   CUDA_MEMCPY3D
dstXInBytes
   CUDA_MEMCPY3D_PEER
   CUDA_MEMCPY2D
   CUDA_MEMCPY3D
dstY
   CUDA_MEMCPY3D_PEER
   CUDA_MEMCPY2D
   CUDA_MEMCPY3D
dstZ
   CUDA_MEMCPY3D_PEER
   CUDA_MEMCPY3D

eglColorFormat
   CUeglFrame
elementSize
   CUDA_MEMSET_NODE_PARAMS
extentDepth
   CUarrayMapInfo
extentHeight
   CUarrayMapInfo
extentWidth
   CUarrayMapInfo
extra
   CUDA_KERNEL_NODE_PARAMS

fd
   CUDA_EXTERNAL_MEMORY_HANDLE_DESC
   CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC
fence
   CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS
   CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS
   CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS
Data Fields

**filterMode**
CUDA_TEXTURE_DESC

**firstLayer**
CUDARESOURCEVIEWDESC

**firstMipmapLevel**
CUDARESOURCEVIEWDESC

**flags**
CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS

**fn**
CUDA_HOST_NODE_PARAMS

**format**
CUDARESOURCEDESC
CUDARESOURCEVIEWDESC

**func**
CUDA_KERNEL_NODE_PARAMS

**function**
CUDA_LAUNCH_PARAMS

**G**

**gridDimX**
CUDA_KERNEL_NODE_PARAMS
CUDA_LAUNCH_PARAMS

**gridDimY**
CUDA_LAUNCH_PARAMS
CUDA_KERNEL_NODE_PARAMS
gridDimZ
    CUDA_LAUNCH_PARAMS
    CUDA_KERNEL_NODE_PARAMS

H
handle
    CUDA_EXTERNAL_MEMORY_HANDLE_DESC
    CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC
hArray
    CUDA_RESOURCE_DESC
height
    CUDA_MEMSET_NODE_PARAMS
    CUeqlFrame
Height
    CUDA_ARRAY3D_DESCRIPTOR
    CUDA_MEMCPY2D
    CUDA_MEMCPY3D_PEER
height
    CUDA_ARRAY_SPARSE_PROPERTIES
Height
    CUDA_MEMCPY3D
height
    CUDA_RESOURCE_DESC
    CUDA_RESOURCE_VIEW_DESC
Height
    CUDA_ARRAY_DESCRIPTOR
hitProp
    CUaccessPolicyWindow
hitRatio
    CUaccessPolicyWindow
hMipmappedArray
    CUDA_RESOURCE_DESC
hStream
    CUDA_LAUNCH_PARAMS

I
id
    CUmemLocation

K
kernelParams
    CUDA_KERNEL_NODE_PARAMS
    CUDA_LAUNCH_PARAMS
Data Fields

key
  CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS
  CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS
keyedMutex
  CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS
  CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS

L
lastLayer
  CUDA_RESOURCE_VIEW_DESC
lastMipmapLevel
  CUDA_RESOURCE_VIEW_DESC
layer
  CUarrayMapInfo
level
  CUarrayMapInfo
location
  CUmemAllocationProp
  CUmemAccessDesc

M
maxAnisotropy
  CUDA_TEXTURE_DESC
maxGridSize
  CUdevprop
maxMipmapLevelClamp
  CUDA_TEXTURE_DESC
maxThreadsDim
  CUdevprop
maxThreadsPerBlock
  CUdevprop
memHandleType
  CUarrayMapInfo
memOperationType
  CUarrayMapInfo
memPitch
  CUdevprop
minMipmapLevelClamp
  CUDA_TEXTURE_DESC
mipmapFilterMode
  CUDA_TEXTURE_DESC
mipmapLevelBias
  CUDA_TEXTURE_DESC
miptailFirstLevel
  CUDA_ARRAY_SPARSE_PROPERTIES
miptailSize
  CUDA_ARRAY_SPARSE_PROPERTIES
missProp
  CUaccessPolicyWindow

N
name
  CUDA_EXTERNAL_MEMORY_HANDLE_DESC
  CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC
num_bytes
  CUaccessPolicyWindow
numChannels
  CUDARESOURCE_DESC
  CUEglFrame
NumChannels
  CUDA_ARRAY3D_DESCRIPTOR
  CUDA_ARRAY_DESCRIPTOR
numLevels
  CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC
nvSciBufObject
  CUDA_EXTERNAL_MEMORY_HANDLE_DESC
nvSciSync
  CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS
nvSciSyncObj
  CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC

O
offset
  CUDA_EXTERNAL_MEMORY_BUFFER_DESC
  CUDA_EXTERNAL_MEMORY_MIPMAPPED_ARRAY_DESC
  CUarrayMapInfo
offsetX
  CUarrayMapInfo
offsetY
  CUarrayMapInfo
offsetZ
  CUarrayMapInfo

P
pArray
  CUEglFrame
Data Fields

pitch
  CUDA_MEMSET_NODE_PARAMS
  CUeglFrame
pitchInBytes
  CUDA_RESOURCE_DESC
planeCount
  CUeglFrame
pPitch
  CUeglFrame

R
regsPerBlock
  CUdevprop
requestedHandleTypes
  CUmemAllocationProp
reserved
  CUarrayMapInfo
reserved0
  CUDA_MEMCPY3D
reserved1
  CUDA_MEMCPY3D
resourceType
  CUarrayMapInfo
resType
  CUDA_RESOURCE_DESC

S
sharedMemBytes
  CUDA_KERNEL_NODE_PARAMS
  CUDA_LAUNCH_PARAMS
sharedMemPerBlock
  CUdevprop
SIMDWidth
  CUdevprop
size
  CUDA_EXTERNAL_MEMORY_HANDLE_DESC
  CUDA_EXTERNAL_MEMORY_BUFFER_DESC
  CUarrayMapInfo
sizeInBytes
  CUDA_RESOURCE_DESC
srcArray
  CUDA_MEMCPY2D
  CUDA_MEMCPY3D
CUDA_MEMCPY3D_PEER

srcContext
CUDA_MEMCPY3D_PEER

srcDevice
CUDA_MEMCPY2D
CUDA_MEMCPY3D
CUDA_MEMCPY3D_PEER

srcHeight
CUDA_MEMCPY3D
CUDA_MEMCPY3D_PEER

srcHost
CUDA_MEMCPY2D
CUDA_MEMCPY3D
CUDA_MEMCPY3D_PEER

srcLOD
CUDA_MEMCPY3D
CUDA_MEMCPY3D_PEER

srcMemoryType
CUDA_MEMCPY2D
CUDA_MEMCPY3D
CUDA_MEMCPY3D_PEER

srcPitch
CUDA_MEMCPY3D
CUDA_MEMCPY2D
CUDA_MEMCPY3D_PEER

srcXInBytes
CUDA_MEMCPY2D
CUDA_MEMCPY3D_PEER
CUDA_MEMCPY3D

srcY
CUDA_MEMCPY2D
CUDA_MEMCPY3D_PEER
CUDA_MEMCPY3D

srcZ
CUDA_MEMCPY3D_PEER
CUDA_MEMCPY3D

subresourceType
CUarrayMapInfo

syncPolicy
CUstreamAttrValue
Data Fields

T

tex
ureAlign
CUdevprop
timeoutMs
CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS
totalConstantMemory
CUdevprop
type
CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC
CUDA_EXTERNAL_MEMORY_HANDLE_DESC
CUmemLocation
CUmemAllocationProp

U

usage
CUmemAllocationProp
userData
CUDA_HOST_NODE_PARAMS

V

value
CUDA_MEMSET_NODE_PARAMS
CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS
CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS

W

Width
CUDA_ARRAY_DESCRIPTOR
CUDA_ARRAY3D_DESCRIPTOR
width
CUDA_ARRAY_sparse_PROPERTIES
CUDA_RESOURCEDESC
CUDA_MEMSET_NODE_PARAMS
CUDA_RESOURCEVIEW_DESC
CUeglFrame

WidthInBytes
CUDA_MEMCPY2D
CUDA_MEMCPY3D
CUDA_MEMCPY3D_PEER

win32
CUDA_EXTERNAL_MEMORY_HANDLE_DESC
CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC
win32HandleMetaData
CUmemAllocationProp
Chapter 8. Deprecated List

**Global CU_CTX_BLOCKING_SYNC**
This flag was deprecated as of CUDA 4.0 and was replaced with CU_CTX_SCHED_BLOCKING_SYNC.

**Global CU_CTX_MAP_HOST**
This flag was deprecated as of CUDA 11.0 and it no longer has any effect. All contexts as of CUDA 3.2 behave as though the flag is enabled.

**Global CU_DEVICE_P2P_ATTRIBUTE_ACCESS_ACCESS_SUPPORTED**
use CU_DEVICE_P2P_ATTRIBUTE_CUDA_ARRAY_ACCESS_SUPPORTED instead

**Global CUDA_ERROR_PROFILER_NOT_INITIALIZED**
This error return is deprecated as of CUDA 5.0. It is no longer an error to attempt to enable/disable the profiling via cuProfilerStart or cuProfilerStop without initialization.

**Global CUDA_ERROR_PROFILER_ALREADY_STARTED**
This error return is deprecated as of CUDA 5.0. It is no longer an error to call cuProfilerStart() when profiling is already enabled.

**Global CUDA_ERROR_PROFILER_ALREADY_STOPPED**
This error return is deprecated as of CUDA 5.0. It is no longer an error to call cuProfilerStop() when profiling is already disabled.

**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 cuDeviceComputeCapability

Global cuDeviceGetProperties

Global cuCtxAttach

Global cuCtxDetach

Global cuFuncSetBlockShape

Global cuFuncSetSharedSize

Global cuLaunch

Global cuLaunchGrid

Global cuLaunchGridAsync

Global cuParamSetf

Global cuParamSeti

Global cuParamsetSize

Global cuParamSetTexRef

Global cuParamSetv
Deprecated List

- Global cuTexRefCreate
- Global cuTexRefDestroy
- Global cuTexRefGetAddress
- Global cuTexRefGetAddressMode
- Global cuTexRefGetArray
- Global cuTexRefGetBorderColor
- Global cuTexRefGetFilterMode
- Global cuTexRefGetFlags
- Global cuTexRefGetFormat
- Global cuTexRefGetMaxAnisotropy
- Global cuTexRefGetMipmapFilterMode
- Global cuTexRefGetMipmapLevelBias
- Global cuTexRefGetMipmapLevelClamp
Global cuTexRefGetMipmappedArray

Global cuTexRefSetAddress

Global cuTexRefSetAddress2D

Global cuTexRefSetAddressMode

Global cuTexRefSetArray

Global cuTexRefSetBorderColor

Global cuTexRefSetFilterMode

Global cuTexRefSetFlags

Global cuTexRefSetFormat

Global cuTexRefSetMaxAnisotropy

Global cuTexRefSetMipmapFilterMode

Global cuTexRefSetMipmapLevelBias

Global cuTexRefSetMipmapLevelClamp

Global cuTexRefSetMipmappedArray
Global cuSurfRefGetArray

Global cuSurfRefSetArray

Global cuProfilerInitialize

Global cuGLCtxCreate
   This function is deprecated as of Cuda 5.0.

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.
Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation (“NVIDIA”) makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

VESA DisplayPort

DisplayPort and DisplayPort Compliance Logo, DisplayPort Compliance Logo for Dual-mode Sources, and DisplayPort Compliance Logo for Active Cables are trademarks owned by the Video Electronics Standards Association in the United States and other countries.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2007-2019 NVIDIA Corporation. All rights reserved.