# Contents

1 Deprecated List ................................................................. 1

2 Module Index ................................................................. 7
  2.1 Modules ...................................................................... 7

3 Data Structure Index .......................................................... 9
  3.1 Data Structures ........................................................... 9

4 Module Documentation ......................................................... 11
  4.1 CUDA Runtime API ...................................................... 11
    4.1.1 Detailed Description ................................................. 12
    4.1.2 Define Documentation .............................................. 12
      4.1.2.1 CUDART_VERSION ........................................... 12
  4.2 Device Management ...................................................... 13
    4.2.1 Detailed Description ................................................. 13
    4.2.2 Function Documentation ............................................ 14
      4.2.2.1 cudaChooseDevice ............................................. 14
      4.2.2.2 cudaDeviceGetCacheConfig ................................ 14
      4.2.2.3 cudaDeviceGetLimit ........................................... 15
      4.2.2.4 cudaDeviceReset ............................................... 15
      4.2.2.5 cudaDeviceSetCacheConfig ................................ 15
      4.2.2.6 cudaDeviceSetLimit ............................................. 16
      4.2.2.7 cudaDeviceSynchronize ...................................... 17
      4.2.2.8 cudaGetDevice ................................................ 17
      4.2.2.9 cudaGetDeviceCount .......................................... 18
      4.2.2.10 cudaGetDeviceProperties ................................... 18
      4.2.2.11 cudaSetDevice ................................................ 21
      4.2.2.12 cudaSetDeviceFlags ......................................... 21
      4.2.2.13 cudaSetValidDevices ...................................... 22
4.3 Thread Management [DEPRECATED] ........................................ 23
  4.3.1 Detailed Description ................................................. 23
  4.3.2 Function Documentation ............................................ 23
    4.3.2.1 cudaThreadExit ............................................. 23
    4.3.2.2 cudaThreadGetCacheConfig ................................ 24
    4.3.2.3 cudaThreadGetLimit ......................................... 24
    4.3.2.4 cudaThreadSetCacheConfig ................................ 25
    4.3.2.5 cudaThreadSetLimit ......................................... 26
    4.3.2.6 cudaThreadSynchronize ...................................... 27

4.4 Error Handling .......................................................... 28
  4.4.1 Detailed Description ................................................. 28
  4.4.2 Function Documentation ............................................ 28
    4.4.2.1 cudaGetErrorString ....................................... 28
    4.4.2.2 cudaGetLastError ......................................... 28
    4.4.2.3 cudaPeekAtLastError ....................................... 29

4.5 Stream Management .................................................... 30
  4.5.1 Detailed Description ................................................. 30
  4.5.2 Function Documentation ............................................ 30
    4.5.2.1 cudaStreamCreate ......................................... 30
    4.5.2.2 cudaStreamDestroy ......................................... 30
    4.5.2.3 cudaStreamQuery ........................................... 31
    4.5.2.4 cudaStreamSynchronize ..................................... 31
    4.5.2.5 cudaStreamWaitEvent ...................................... 32

4.6 Event Management ........................................................ 33
  4.6.1 Detailed Description ................................................. 33
  4.6.2 Function Documentation ............................................ 33
    4.6.2.1 cudaEventCreate ......................................... 33
    4.6.2.2 cudaEventCreateWithFlags ................................ 34
    4.6.2.3 cudaEventDestroy ......................................... 34
    4.6.2.4 cudaEventElapsedTime ..................................... 35
    4.6.2.5 cudaEventQuery ........................................... 35
    4.6.2.6 cudaEventRecord ........................................... 36
    4.6.2.7 cudaEventSynchronize ..................................... 36

4.7 Execution Control ..................................................... 37
  4.7.1 Detailed Description ................................................. 37
  4.7.2 Function Documentation ............................................ 37
    4.7.2.1 cudaConfigureCall ......................................... 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.7.2.2 cudaFuncGetAttributes</td>
<td>38</td>
</tr>
<tr>
<td>4.7.2.3 cudaFuncSetCacheConfig</td>
<td>38</td>
</tr>
<tr>
<td>4.7.2.4 cudaLaunch</td>
<td>39</td>
</tr>
<tr>
<td>4.7.2.5 cudaSetDoubleForDevice</td>
<td>39</td>
</tr>
<tr>
<td>4.7.2.6 cudaSetDoubleForHost</td>
<td>40</td>
</tr>
<tr>
<td>4.7.2.7 cudaSetupArgument</td>
<td>40</td>
</tr>
<tr>
<td>4.8 Memory Management</td>
<td>41</td>
</tr>
<tr>
<td>4.8.1 Detailed Description</td>
<td>44</td>
</tr>
<tr>
<td>4.8.2 Function Documentation</td>
<td>44</td>
</tr>
<tr>
<td>4.8.2.1 cudaMemcpy</td>
<td>44</td>
</tr>
<tr>
<td>4.8.2.2 cudaMemcpy2D</td>
<td>45</td>
</tr>
<tr>
<td>4.8.2.3 cudaMemcpy2DFromArray</td>
<td>45</td>
</tr>
<tr>
<td>4.8.2.4 cudaMemcpy2DToArray</td>
<td>46</td>
</tr>
<tr>
<td>4.8.2.5 cudaMemcpy2DToArrayAsync</td>
<td>46</td>
</tr>
<tr>
<td>4.8.2.6 cudaMemcpy2DToArrayAsync2D</td>
<td>47</td>
</tr>
<tr>
<td>4.8.2.7 cudaMemcpy3D</td>
<td>48</td>
</tr>
<tr>
<td>4.8.2.8 cudaMemcpy3DPeer</td>
<td>48</td>
</tr>
<tr>
<td>4.8.2.9 cudaMemcpy3DPeerAsync</td>
<td>48</td>
</tr>
<tr>
<td>4.8.2.10 cudaMemcpy3DPeerAsync2D</td>
<td>49</td>
</tr>
<tr>
<td>4.8.2.11 cudaMemcpy</td>
<td>50</td>
</tr>
<tr>
<td>4.8.2.12 cudaMemcpy2D</td>
<td>50</td>
</tr>
<tr>
<td>4.8.2.13 cudaMemcpy2DArray</td>
<td>51</td>
</tr>
<tr>
<td>4.8.2.14 cudaMemcpy2DArray2D</td>
<td>52</td>
</tr>
<tr>
<td>4.8.2.15 cudaMemcpy2DArrayHost</td>
<td>53</td>
</tr>
<tr>
<td>4.8.2.16 cudaMemcpy2DArrayPitch</td>
<td>53</td>
</tr>
<tr>
<td>4.8.2.17 cudaMemcpy</td>
<td>54</td>
</tr>
<tr>
<td>4.8.2.18 cudaMemcpy2D</td>
<td>54</td>
</tr>
<tr>
<td>4.8.2.19 cudaMemcpy2DArrayToArray</td>
<td>55</td>
</tr>
<tr>
<td>4.8.2.20 cudaMemcpy2DAsync</td>
<td>56</td>
</tr>
<tr>
<td>4.8.2.21 cudaMemcpy2DFromArray</td>
<td>57</td>
</tr>
<tr>
<td>4.8.2.22 cudaMemcpy2DFromArrayAsync</td>
<td>57</td>
</tr>
<tr>
<td>4.8.2.23 cudaMemcpy2DToArray</td>
<td>57</td>
</tr>
<tr>
<td>4.8.2.24 cudaMemcpy2DToArrayAsync</td>
<td>58</td>
</tr>
<tr>
<td>4.8.2.25 cudaMemcpy3D</td>
<td>60</td>
</tr>
<tr>
<td>4.8.2.26 cudaMemcpy3DAsync</td>
<td>61</td>
</tr>
<tr>
<td>4.8.2.27 cudaMemcpy3DPeer</td>
<td>63</td>
</tr>
<tr>
<td>4.8.2.28 cudaMemcpy3DPeerAsync</td>
<td>63</td>
</tr>
</tbody>
</table>
4.8.2.29 cudaMemcpyArrayToArray ................................. 63
4.8.2.30 cudaMemcpyAsync ........................................ 64
4.8.2.31 cudaMemcpyFromArray ................................... 65
4.8.2.32 cudaMemcpyFromArrayAsync ................................ 65
4.8.2.33 cudaMemcpyFromSymbol .................................. 66
4.8.2.34 cudaMemcpyFromSymbolAsync ......................... 67
4.8.2.35 cudaMemcpyPeer ............................................ 67
4.8.2.36 cudaMemcpyPeerAsync ..................................... 68
4.8.2.37 cudaMemcpyToArray ....................................... 69
4.8.2.38 cudaMemcpyToArrayAsync ................................ 69
4.8.2.39 cudaMemcpyToSymbol ..................................... 70
4.8.2.40 cudaMemcpyToSymbolAsync ................................ 71
4.8.2.41 cudaMemcpy ............................................... 71
4.8.2.42 cudaMemcpy ............................................... 72
4.8.2.43 cudaMemcpy2D ............................................. 72
4.8.2.44 cudaMemcpy2DAasync ..................................... 73
4.8.2.45 cudaMemcpy3D ............................................. 73
4.8.2.46 cudaMemcpy3DAasync ..................................... 74
4.8.2.47 cudaMemcpyAsync ......................................... 75
4.8.2.48 make_cudaExtent ......................................... 75
4.8.2.49 make_cudaPitchedPtr .................................... 75
4.8.2.50 make_cudaPos ............................................. 76

4.9 Unified Addressing .................................................. 77
  4.9.1 Detailed Description ......................................... 77
  4.9.2 Overview ..................................................... 77
  4.9.3 Supported Platforms ......................................... 77
  4.9.4 Looking Up Information from Pointer Values ............ 77
  4.9.5 Automatic Mapping of Host Allocated Host Memory ...... 77
  4.9.6 Direct Access ................................................ 78
  4.9.7 Exceptions, Disjoint Addressing ......................... 78
  4.9.8 Function Documentation ...................................... 78
    4.9.8.1 cudaMemcpyAttributes .................................. 78

4.10 Peer Device Memory Access ........................................ 80
  4.10.1 Detailed Description ....................................... 80
  4.10.2 Function Documentation ..................................... 80
    4.10.2.1 cudaMemcpyCanAccessPeer .............................. 80
    4.10.2.2 cudaMemcpyDisablePeerAccess ......................... 80
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.15</td>
<td>VDPAU Interoperability</td>
<td>100</td>
</tr>
<tr>
<td>4.15.1</td>
<td>Detailed Description</td>
<td>100</td>
</tr>
<tr>
<td>4.15.2</td>
<td>Function Documentation</td>
<td>100</td>
</tr>
<tr>
<td>4.15.2.1</td>
<td>cudaGraphicsVDPAURegisterOutputSurface</td>
<td>100</td>
</tr>
<tr>
<td>4.15.2.2</td>
<td>cudaGraphicsVDPAURegisterVideoSurface</td>
<td>101</td>
</tr>
<tr>
<td>4.15.2.3</td>
<td>cudaVDPAUGetDevice</td>
<td>101</td>
</tr>
<tr>
<td>4.15.2.4</td>
<td>cudaVDPAUSetVDPAUDevice</td>
<td>102</td>
</tr>
<tr>
<td>4.16</td>
<td>Graphics Interoperability</td>
<td>103</td>
</tr>
<tr>
<td>4.16.1</td>
<td>Detailed Description</td>
<td>103</td>
</tr>
<tr>
<td>4.16.2</td>
<td>Function Documentation</td>
<td>103</td>
</tr>
<tr>
<td>4.16.2.1</td>
<td>cudaGraphicsMapResources</td>
<td>103</td>
</tr>
<tr>
<td>4.16.2.2</td>
<td>cudaGraphicsResourceGetMappedPointer</td>
<td>104</td>
</tr>
<tr>
<td>4.16.2.3</td>
<td>cudaGraphicsResourceSetMapFlags</td>
<td>104</td>
</tr>
<tr>
<td>4.16.2.4</td>
<td>cudaGraphicsSubResourceGetMappedArray</td>
<td>105</td>
</tr>
<tr>
<td>4.16.2.5</td>
<td>cudaGraphicsUnmapResources</td>
<td>106</td>
</tr>
<tr>
<td>4.16.2.6</td>
<td>cudaGraphicsUnregisterResource</td>
<td>106</td>
</tr>
<tr>
<td>4.17</td>
<td>Texture Reference Management</td>
<td>107</td>
</tr>
<tr>
<td>4.17.1</td>
<td>Detailed Description</td>
<td>107</td>
</tr>
<tr>
<td>4.17.2</td>
<td>Function Documentation</td>
<td>107</td>
</tr>
<tr>
<td>4.17.2.1</td>
<td>cudaBindTexture</td>
<td>107</td>
</tr>
<tr>
<td>4.17.2.2</td>
<td>cudaBindTexture2D</td>
<td>108</td>
</tr>
<tr>
<td>4.17.2.3</td>
<td>cudaBindTextureToArray</td>
<td>109</td>
</tr>
<tr>
<td>4.17.2.4</td>
<td>cudaCreateChannelDesc</td>
<td>109</td>
</tr>
<tr>
<td>4.17.2.5</td>
<td>cudaGetChannelDesc</td>
<td>110</td>
</tr>
<tr>
<td>4.17.2.6</td>
<td>cudaGetTextureAlignmentOffset</td>
<td>110</td>
</tr>
<tr>
<td>4.17.2.7</td>
<td>cudaGetTextureReference</td>
<td>111</td>
</tr>
<tr>
<td>4.17.2.8</td>
<td>cudaUnbindTexture</td>
<td>111</td>
</tr>
<tr>
<td>4.18</td>
<td>Surface Reference Management</td>
<td>112</td>
</tr>
<tr>
<td>4.18.1</td>
<td>Detailed Description</td>
<td>112</td>
</tr>
<tr>
<td>4.18.2</td>
<td>Function Documentation</td>
<td>112</td>
</tr>
<tr>
<td>4.18.2.1</td>
<td>cudaBindSurfaceToArray</td>
<td>112</td>
</tr>
<tr>
<td>4.18.2.2</td>
<td>cudaGetSurfaceReference</td>
<td>112</td>
</tr>
<tr>
<td>4.19</td>
<td>Version Management</td>
<td>114</td>
</tr>
<tr>
<td>4.19.1</td>
<td>Function Documentation</td>
<td>114</td>
</tr>
<tr>
<td>4.19.1.1</td>
<td>cudaDriverGetVersion</td>
<td>114</td>
</tr>
<tr>
<td>4.19.1.2</td>
<td>cudaRuntimeGetVersion</td>
<td>114</td>
</tr>
<tr>
<td>4.20</td>
<td>C++ API Routines</td>
<td>115</td>
</tr>
</tbody>
</table>
4.20.1 Detailed Description .................................................. 116
4.20.2 Function Documentation ............................................. 116
  4.20.2.1 cudaBindSurfaceToArray ........................................ 116
  4.20.2.2 cudaBindSurfaceToArray ........................................ 117
  4.20.2.3 cudaBindTexture ................................................ 117
  4.20.2.4 cudaBindTexture ................................................ 118
  4.20.2.5 cudaBindTexture2D .............................................. 119
  4.20.2.6 cudaBindTexture2D .............................................. 119
  4.20.2.7 cudaBindTextureToArray ....................................... 120
  4.20.2.8 cudaBindTextureToArray ....................................... 121
  4.20.2.9 cudaCreateChannelDesc ....................................... 121
  4.20.2.10 cudaEventCreate .............................................. 122
  4.20.2.11 cudaFuncGetAttributes ....................................... 122
  4.20.2.12 cudaFuncSetCacheConfig ..................................... 123
  4.20.2.13 cudaGetSymbolAddress ....................................... 123
  4.20.2.14 cudaGetSymbolSize .......................................... 124
  4.20.2.15 cudaGetTextureAlignmentOffset ............................. 124
  4.20.2.16 cudaLaunch ................................................... 125
  4.20.2.17 cudaMallocHost ............................................. 125
  4.20.2.18 cudaSetupArgument ......................................... 126
  4.20.2.19 cudaUnbindTexture ......................................... 127

4.21 Interactions with the CUDA Driver API .............................. 128
  4.21.1 Primary Contexts ................................................ 128
  4.21.2 Initialization and Tear-Down ................................... 128
  4.21.3 Context Interoperability ....................................... 128
  4.21.4 Interactions between CUstream and cudaStream_t ............ 129
  4.21.5 Interactions between CUevent and cudaEvent_t ............... 129
  4.21.6 Interactions between CUarray and struct cudaArray * ....... 129
  4.21.7 Interactions between CUgraphicsResource and cudaGraphicsResource_t .... 129

4.22 Direct3D 9 Interoperability [DEPRECATED] ......................... 130
  4.22.1 Detailed Description ............................................ 131
  4.22.2 Enumeration Type Documentation ............................... 131
    4.22.2.1 cudaD3D9MapFlags ......................................... 131
    4.22.2.2 cudaD3D9RegisterFlags .................................... 131
  4.22.3 Function Documentation ........................................ 131
    4.22.3.1 cudaD3D9MapResources ..................................... 131
    4.22.3.2 cudaD3D9RegisterResource ................................ 132
4.22.3.3 cudaD3D9ResourceGetMappedArray ............................. 133
4.22.3.4 cudaD3D9ResourceGetMappedPitch .......................... 134
4.22.3.5 cudaD3D9ResourceGetMappedPointer ........................ 134
4.22.3.6 cudaD3D9ResourceGetMappedSize .......................... 135
4.22.3.7 cudaD3D9ResourceGetSurfaceDimensions .................... 136
4.22.3.8 cudaD3D9ResourceSetMapFlags .............................. 137
4.22.3.9 cudaD3D9UnmapResources .................................. 137
4.22.3.10 cudaD3D9UnregisterResource ............................... 138

4.23 Direct3D 10 Interoperability [DEPRECATED] .......................... 139
  4.23.1 Detailed Description ......................................... 140
  4.23.2 Enumeration Type Documentation ............................. 140
    4.23.2.1 cudaD3D10MapFlags ..................................... 140
    4.23.2.2 cudaD3D10RegisterFlags ................................ 140
  4.23.3 Function Documentation ....................................... 140
    4.23.3.1 cudaD3D10MapResources .................................. 140
    4.23.3.2 cudaD3D10RegisterResource ................................ 141
    4.23.3.3 cudaD3D10ResourceGetMappedArray ......................... 142
    4.23.3.4 cudaD3D10ResourceGetMappedPitch ......................... 143
    4.23.3.5 cudaD3D10ResourceGetMappedPointer ....................... 143
    4.23.3.6 cudaD3D10ResourceGetMappedSize ........................ 144
    4.23.3.7 cudaD3D10ResourceGetSurfaceDimensions .................. 145
    4.23.3.8 cudaD3D10ResourceSetMapFlags ............................ 145
    4.23.3.9 cudaD3D10UnmapResources .................................. 146
    4.23.3.10 cudaD3D10UnregisterResource ............................ 147

4.24 OpenGL Interoperability [DEPRECATED] ................................ 148
  4.24.1 Detailed Description ......................................... 148
  4.24.2 Enumeration Type Documentation ............................. 148
    4.24.2.1 cudaGLMapFlags ......................................... 148
  4.24.3 Function Documentation ....................................... 149
    4.24.3.1 cudaGLMapBufferObject ................................. 149
    4.24.3.2 cudaGLMapBufferObjectAsync ........................... 149
    4.24.3.3 cudaGLRegisterBufferObject ............................ 150
    4.24.3.4 cudaGLSetBufferObjectMapFlags ......................... 150
    4.24.3.5 cudaGLUnmapBufferObject ............................... 151
    4.24.3.6 cudaGLUnmapBufferObjectAsync .......................... 151
    4.24.3.7 cudaGLUnregisterBufferObject .......................... 152

4.25 Data types used by CUDA Runtime ................................ 153
<table>
<thead>
<tr>
<th>Section</th>
<th>Documentation</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.25.1</td>
<td>Define Documentation</td>
<td>157</td>
</tr>
<tr>
<td>4.25.1.1</td>
<td>cudaArrayDefault</td>
<td>157</td>
</tr>
<tr>
<td>4.25.1.2</td>
<td>cudaArrayLayered</td>
<td>157</td>
</tr>
<tr>
<td>4.25.1.3</td>
<td>cudaArraySurfaceLoadStore</td>
<td>157</td>
</tr>
<tr>
<td>4.25.1.4</td>
<td>cudaDeviceBlockingSync</td>
<td>157</td>
</tr>
<tr>
<td>4.25.1.5</td>
<td>cudaDeviceLmemResizeToMax</td>
<td>157</td>
</tr>
<tr>
<td>4.25.1.6</td>
<td>cudaDeviceMapHost</td>
<td>157</td>
</tr>
<tr>
<td>4.25.1.7</td>
<td>cudaDeviceMask</td>
<td>157</td>
</tr>
<tr>
<td>4.25.1.8</td>
<td>cudaDevicePropDontCare</td>
<td>157</td>
</tr>
<tr>
<td>4.25.1.9</td>
<td>cudaDeviceScheduleAuto</td>
<td>157</td>
</tr>
<tr>
<td>4.25.1.10</td>
<td>cudaDeviceScheduleBlockingSync</td>
<td>158</td>
</tr>
<tr>
<td>4.25.1.11</td>
<td>cudaDeviceScheduleSpin</td>
<td>158</td>
</tr>
<tr>
<td>4.25.1.12</td>
<td>cudaDeviceScheduleYield</td>
<td>158</td>
</tr>
<tr>
<td>4.25.1.13</td>
<td>cudaEventBlockingSync</td>
<td>158</td>
</tr>
<tr>
<td>4.25.1.14</td>
<td>cudaEventDefault</td>
<td>158</td>
</tr>
<tr>
<td>4.25.1.15</td>
<td>cudaEventDisableTiming</td>
<td>158</td>
</tr>
<tr>
<td>4.25.1.16</td>
<td>cudaHostAllocDefault</td>
<td>158</td>
</tr>
<tr>
<td>4.25.1.17</td>
<td>cudaHostAllocMapped</td>
<td>158</td>
</tr>
<tr>
<td>4.25.1.18</td>
<td>cudaHostAllocPortable</td>
<td>158</td>
</tr>
<tr>
<td>4.25.1.19</td>
<td>cudaHostAllocWriteCombined</td>
<td>158</td>
</tr>
<tr>
<td>4.25.1.20</td>
<td>cudaHostRegisterDefault</td>
<td>158</td>
</tr>
<tr>
<td>4.25.1.21</td>
<td>cudaHostRegisterMapped</td>
<td>158</td>
</tr>
<tr>
<td>4.25.1.22</td>
<td>cudaHostRegisterPortable</td>
<td>159</td>
</tr>
<tr>
<td>4.25.1.23</td>
<td>cudaPeerAccessDefault</td>
<td>159</td>
</tr>
<tr>
<td>4.25.2</td>
<td>Typedef Documentation</td>
<td>159</td>
</tr>
<tr>
<td>4.25.2.1</td>
<td>cudaError_t</td>
<td>159</td>
</tr>
<tr>
<td>4.25.2.2</td>
<td>cudaEvent_t</td>
<td>159</td>
</tr>
<tr>
<td>4.25.2.3</td>
<td>cudaGraphicsResource_t</td>
<td>159</td>
</tr>
<tr>
<td>4.25.2.4</td>
<td>cudaOutputMode_t</td>
<td>159</td>
</tr>
<tr>
<td>4.25.2.5</td>
<td>cudaStream_t</td>
<td>159</td>
</tr>
<tr>
<td>4.25.2.6</td>
<td>cudaUUID_t</td>
<td>159</td>
</tr>
<tr>
<td>4.25.3</td>
<td>Enumeration Type Documentation</td>
<td>159</td>
</tr>
<tr>
<td>4.25.3.1</td>
<td>cudaChannelFormatKind</td>
<td>159</td>
</tr>
<tr>
<td>4.25.3.2</td>
<td>cudaComputeMode</td>
<td>160</td>
</tr>
<tr>
<td>4.25.3.3</td>
<td>cudaError</td>
<td>160</td>
</tr>
<tr>
<td>4.25.3.4</td>
<td>cudaFuncCache</td>
<td>163</td>
</tr>
<tr>
<td>4.25.3.5</td>
<td>cudaGraphicsCubeFace</td>
<td>164</td>
</tr>
</tbody>
</table>
4.27.3.5 CUctx_flags_enum .................................................. 179
4.27.3.6 cudaError_enum .................................................... 180
4.27.3.7 C Ud evice_attribute_enum ....................................... 182
4.27.3.8 CUevent_flags_enum ............................................... 184
4.27.3.9 CUfilter_mode_enum ................................................. 184
4.27.3.10 CUfunc_cache_enum ............................................... 184
4.27.3.11 CUfunction_attribute_enum .................................... 184
4.27.3.12 CUgraphicsMapResourceFlags_enum .......................... 185
4.27.3.13 CUgraphicsRegisterFlags_enum ................................. 185
4.27.3.14 CUjit_fallback_enum ............................................. 185
4.27.3.15 CUjit_option_enum ................................................. 185
4.27.3.16 CUjit_target_enum ................................................ 186
4.27.3.17 CUlimit_enum ..................................................... 186
4.27.3.18 CUmemorytype_enum ............................................. 187
4.27.3.19 CUpointer_attribute_enum ...................................... 187

4.28 Initialization .......................................................... 188
  4.28.1 Detailed Description .............................................. 188
  4.28.2 Function Documentation ....................................... 188
  4.28.2.1 cuInit ......................................................... 188

4.29 Version Management .................................................. 189
  4.29.1 Detailed Description .............................................. 189
  4.29.2 Function Documentation ....................................... 189
  4.29.2.1 cuDriverGetVersion ........................................... 189

4.30 Device Management ................................................... 190
  4.30.1 Detailed Description .............................................. 190
  4.30.2 Function Documentation ....................................... 190
  4.30.2.1 cuDeviceComputeCapability .................................. 190
  4.30.2.2 cuDeviceGet .................................................... 191
  4.30.2.3 cuDeviceGetAttribute ........................................ 191
  4.30.2.4 cuDeviceGetCount ............................................. 193
  4.30.2.5 cuDeviceGetName .............................................. 194
  4.30.2.6 cuDeviceGetProperties ....................................... 194
  4.30.2.7 cuDeviceTotalMem ............................................. 195

4.31 Context Management .................................................. 197
  4.31.1 Detailed Description .............................................. 198
  4.31.2 Function Documentation ....................................... 198
  4.31.2.1 cuCtxCreate ................................................... 198
4.34.2.6 cuMemAlloc ................................. 223
4.34.2.7 cuMemAllocHost ......................... 224
4.34.2.8 cuMemAllocPitch ....................... 224
4.34.2.9 cuMemcpy .................................. 225
4.34.2.10 cuMemcpy2D ............................ 226
4.34.2.11 cuMemcpy2DAsync ....................... 228
4.34.2.12 cuMemcpy2DUnaligned .................. 231
4.34.2.13 cuMemcpy3D ............................ 233
4.34.2.14 cuMemcpy3DAsync ....................... 236
4.34.2.15 cuMemcpy3DPeer ....................... 238
4.34.2.16 cuMemcpy3DPeerAsync ................. 239
4.34.2.17 cuMemcpyAsync ......................... 239
4.34.2.18 cuMemcpyAtoA .......................... 240
4.34.2.19 cuMemcpyAtoD .......................... 240
4.34.2.20 cuMemcpyAtoH .......................... 241
4.34.2.21 cuMemcpyAtoHAsync .................... 242
4.34.2.22 cuMemcpyDtoA .......................... 242
4.34.2.23 cuMemcpyDtoD .......................... 243
4.34.2.24 cuMemcpyDtoDAsync .................... 243
4.34.2.25 cuMemcpyDtoH .......................... 244
4.34.2.26 cuMemcpyDtoHAsync .................... 245
4.34.2.27 cuMemcpyHtoA .......................... 245
4.34.2.28 cuMemcpyHtoAAsync .................... 246
4.34.2.29 cuMemcpyHtoD .......................... 247
4.34.2.30 cuMemcpyHtoDAsync .................... 247
4.34.2.31 cuMemcpyPeer ......................... 248
4.34.2.32 cuMemcpyPeerAsync .................... 248
4.34.2.33 cuMemFree ............................ 249
4.34.2.34 cuMemFreeHost ....................... 250
4.34.2.35 cuMemGetAddressRange ............... 250
4.34.2.36 cuMemGetInfo ......................... 251
4.34.2.37 cuMemHostAlloc ...................... 251
4.34.2.38 cuMemHostGetDevicePointer .......... 252
4.34.2.39 cuMemHostGetFlags ................... 253
4.34.2.40 cuMemHostRegister ................... 253
4.34.2.41 cuMemHostUnregister .................. 254
4.34.2.42 cuMemsetD16 .......................... 255

Generated for NVIDIA CUDA Library by Doxygen
4.34.2.43 cuMemsetD16Async ................................. 255
4.34.2.44 cuMemsetD2D16 ................................. 256
4.34.2.45 cuMemsetD2D16Async .......................... 257
4.34.2.46 cuMemsetD2D32 ................................. 257
4.34.2.47 cuMemsetD2D32Async .......................... 258
4.34.2.48 cuMemsetD2D8 ................................. 259
4.34.2.49 cuMemsetD2D8Async ........................... 259
4.34.2.50 cuMemsetD32 ................................. 260
4.34.2.51 cuMemsetD32Async ............................ 261
4.34.2.52 cuMemsetD8 ................................. 261
4.34.2.53 cuMemsetD8Async .............................. 262

4.35 Unified Addressing .................................. 263
  4.35.1 Detailed Description ............................ 263
  4.35.2 Overview ....................................... 263
  4.35.3 Supported Platforms ............................ 263
  4.35.4 Looking Up Information from Pointer Values .. 263
  4.35.5 Automatic Mapping of Host Allocated Host Memory . 263
  4.35.6 Automatic Registration of Peer Memory .......... 264
  4.35.7 Exceptions, Disjoint Addressing ................. 264
  4.35.8 Function Documentation .......................... 264
    4.35.8.1 cuPointerGetAttribute ...................... 264

4.36 Stream Management ................................ 267
  4.36.1 Detailed Description ............................ 267
  4.36.2 Function Documentation .......................... 267
    4.36.2.1 cuStreamCreate ............................. 267
    4.36.2.2 cuStreamDestroy ............................ 268
    4.36.2.3 cuStreamQuery .............................. 268
    4.36.2.4 cuStreamSynchronize ......................... 268
    4.36.2.5 cuStreamWaitEvent .......................... 269

4.37 Event Management .................................. 270
  4.37.1 Detailed Description ............................ 270
  4.37.2 Function Documentation .......................... 270
    4.37.2.1 cuEventCreate ............................. 270
    4.37.2.2 cuEventDestroy ............................ 271
    4.37.2.3 cuEventElapsedTime ......................... 271
    4.37.2.4 cuEventQuery .............................. 272
    4.37.2.5 cuEventRecord ............................. 272
CONTENTS 

4.41 Detailed Description ............................................................ 293

4.41.1 Function Documentation ...................................................... 293

4.41.2.1 cuTexRefCreate .......................................................... 293

4.41.2.2 cuTexRefDestroy .......................................................... 293

4.42 Surface Reference Management ................................................. 295

4.42.1 Detailed Description .......................................................... 295

4.42.2 Function Documentation ...................................................... 295

4.42.2.1 cuSurfRefGetArray ....................................................... 295

4.42.2.2 cuSurfRefSetArray ....................................................... 295

4.43 Peer Context Memory Access ................................................... 297

4.43.1 Detailed Description .......................................................... 297

4.43.2 Function Documentation ...................................................... 297

4.43.2.1 cuCtxDisablePeerAccess ............................................... 297

4.43.2.2 cuCtxEnablePeerAccess ............................................... 297

4.43.2.3 cuDeviceCanAccessPeer ............................................... 298

4.44 Graphics Interoperability ...................................................... 299

4.44.1 Detailed Description .......................................................... 299

4.44.2 Function Documentation ...................................................... 299

4.44.2.1 cuGraphicsMapResources ............................................. 299

4.44.2.2 cuGraphicsResourceGetMappedPointer .................................. 300

4.44.2.3 cuGraphicsResourceSetMapFlags ....................................... 300

4.44.2.4 cuGraphicsSubResourceGetMappedArray .................................. 301

4.44.2.5 cuGraphicsUnmapResources ........................................... 302

4.44.2.6 cuGraphicsUnregisterResource .......................................... 302

4.45 OpenGL Interoperability ...................................................... 304

4.45.1 Detailed Description .......................................................... 304

4.45.2 Function Documentation ...................................................... 304

4.45.2.1 cuGLCtxCreate .......................................................... 304

4.45.2.2 cuGraphicsGLRegisterBuffer ......................................... 305

4.45.2.3 cuGraphicsGLRegisterImage ........................................... 305

4.45.2.4 cuWGLGetDevice .......................................................... 306

4.46 OpenGL Interoperability [DEPRECATED] .................................... 307

4.46.1 Detailed Description .......................................................... 307

4.46.2 Typedef Documentation ...................................................... 307

4.46.2.1 CUGLmap_flags .......................................................... 307

4.46.3 Enumeration Type Documentation ......................................... 308

4.46.3.1 CUGLmap_flags_enum .................................................. 308

Generated for NVIDIA CUDA Library by Doxygen
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.46.4</td>
<td>Function Documentation</td>
<td>308</td>
</tr>
<tr>
<td>4.46.4.1</td>
<td>cuGLInit</td>
<td>308</td>
</tr>
<tr>
<td>4.46.4.2</td>
<td>cuGLMapBufferObject</td>
<td>308</td>
</tr>
<tr>
<td>4.46.4.3</td>
<td>cuGLMapBufferObjectAsync</td>
<td>309</td>
</tr>
<tr>
<td>4.46.4.4</td>
<td>cuGLRegisterBufferObject</td>
<td>309</td>
</tr>
<tr>
<td>4.46.4.5</td>
<td>cuGLSetBufferObjectMapFlags</td>
<td>310</td>
</tr>
<tr>
<td>4.46.4.6</td>
<td>cuGLUnmapBufferObject</td>
<td>311</td>
</tr>
<tr>
<td>4.46.4.7</td>
<td>cuGLUnmapBufferObjectAsync</td>
<td>311</td>
</tr>
<tr>
<td>4.46.4.8</td>
<td>cuGLUnregisterBufferObject</td>
<td>312</td>
</tr>
<tr>
<td>4.47</td>
<td>Direct3D 9 Interoperability</td>
<td>313</td>
</tr>
<tr>
<td>4.47.1</td>
<td>Detailed Description</td>
<td>313</td>
</tr>
<tr>
<td>4.47.2</td>
<td>Typedef Documentation</td>
<td>314</td>
</tr>
<tr>
<td>4.47.2.1</td>
<td>CUd3d9DeviceList</td>
<td>314</td>
</tr>
<tr>
<td>4.47.3</td>
<td>Enumeration Type Documentation</td>
<td>314</td>
</tr>
<tr>
<td>4.47.3.1</td>
<td>CUd3d9DeviceList_enum</td>
<td>314</td>
</tr>
<tr>
<td>4.47.4</td>
<td>Function Documentation</td>
<td>314</td>
</tr>
<tr>
<td>4.47.4.1</td>
<td>cuD3D9CtxCreate</td>
<td>314</td>
</tr>
<tr>
<td>4.47.4.2</td>
<td>cuD3D9CtxCreateOnDevice</td>
<td>315</td>
</tr>
<tr>
<td>4.47.4.3</td>
<td>cuD3D9GetDevice</td>
<td>315</td>
</tr>
<tr>
<td>4.47.4.4</td>
<td>cuD3D9GetDevices</td>
<td>316</td>
</tr>
<tr>
<td>4.47.4.5</td>
<td>cuD3D9GetDirect3DDevice</td>
<td>316</td>
</tr>
<tr>
<td>4.47.4.6</td>
<td>cuGraphicsD3D9RegisterResource</td>
<td>317</td>
</tr>
<tr>
<td>4.48</td>
<td>Direct3D 9 Interoperability [DEPRECATED]</td>
<td>319</td>
</tr>
<tr>
<td>4.48.1</td>
<td>Detailed Description</td>
<td>320</td>
</tr>
<tr>
<td>4.48.2</td>
<td>Typedef Documentation</td>
<td>320</td>
</tr>
<tr>
<td>4.48.2.1</td>
<td>CUd3d9map_flags</td>
<td>320</td>
</tr>
<tr>
<td>4.48.2.2</td>
<td>CUd3d9register_flags</td>
<td>320</td>
</tr>
<tr>
<td>4.48.3</td>
<td>Enumeration Type Documentation</td>
<td>320</td>
</tr>
<tr>
<td>4.48.3.1</td>
<td>CUd3d9map_flags_enum</td>
<td>320</td>
</tr>
<tr>
<td>4.48.3.2</td>
<td>CUd3d9register_flags_enum</td>
<td>320</td>
</tr>
<tr>
<td>4.48.4</td>
<td>Function Documentation</td>
<td>320</td>
</tr>
<tr>
<td>4.48.4.1</td>
<td>cuD3D9MapResources</td>
<td>320</td>
</tr>
<tr>
<td>4.48.4.2</td>
<td>cuD3D9RegisterResource</td>
<td>321</td>
</tr>
<tr>
<td>4.48.4.3</td>
<td>cuD3D9ResourceGetMappedArray</td>
<td>322</td>
</tr>
<tr>
<td>4.48.4.4</td>
<td>cuD3D9ResourceGetMappedPitch</td>
<td>323</td>
</tr>
<tr>
<td>4.48.4.5</td>
<td>cuD3D9ResourceGetMappedPointer</td>
<td>324</td>
</tr>
<tr>
<td>4.48.4.6</td>
<td>cuD3D9ResourceGetMappedSize</td>
<td>324</td>
</tr>
</tbody>
</table>
4.49 Direct3D 10 Interoperability .................................................. 328
4.49.1 Detailed Description .......................................................... 328
4.49.2 Typedef Documentation ....................................................... 329
4.49.2.1 C Ud3d10DeviceList .................................................. 329
4.49.3 Enumeration Type Documentation ........................................ 329
4.49.3.1 C Ud3d10DeviceList_enum ............................................ 329
4.49.4 Function Documentation ..................................................... 329
4.49.4.1 cuD3D10CtxCreate ...................................................... 329
4.49.4.2 cuD3D10CtxCreateOnDevice ........................................... 330
4.49.4.3 cuD3D10GetDevice ...................................................... 330
4.49.4.4 cuD3D10GetDevices ..................................................... 331
4.49.4.5 cuD3D10GetDirect3DDevice ............................................ 331
4.49.4.6 cuGraphicsD3D10RegisterResource ................................... 332
4.50 Direct3D 10 Interoperability [DEPRECATED] ............................... 334
4.50.1 Detailed Description .......................................................... 335
4.50.2 Typedef Documentation ....................................................... 335
4.50.2.1 C Ud3d10map_flags .................................................... 335
4.50.2.2 C Ud3d10register_flags ................................................ 335
4.50.3 Enumeration Type Documentation ........................................ 335
4.50.3.1 C Ud3d10map_flags_enum ............................................. 335
4.50.3.2 C Ud3d10register_flags_enum ....................................... 335
4.50.4 Function Documentation ..................................................... 335
4.50.4.1 cuD3D10MapResources .................................................. 335
4.50.4.2 cuD3D10RegisterResource ............................................. 336
4.50.4.3 cuD3D10ResourceGetMappedArray .................................. 337
4.50.4.4 cuD3D10ResourceGetMappedPitch .................................. 338
4.50.4.5 cuD3D10ResourceGetMappedPointer ................................ 339
4.50.4.6 cuD3D10ResourceGetMappedSize ................................... 339
4.50.4.7 cuD3D10ResourceGetSurfaceDimensions .............................. 340
4.50.4.8 cuD3D10ResourceSetMapFlags ....................................... 341
4.50.4.9 cuD3D10UnmapResources ............................................. 341
4.50.4.10 cuD3D10UnregisterResource ......................................... 342
4.51 Direct3D 11 Interoperability ................................................ 343
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3 CUDA_MEMCPY2D_st Struct Reference</td>
<td>356</td>
</tr>
<tr>
<td>5.3.1 Detailed Description</td>
<td>356</td>
</tr>
<tr>
<td>5.3.2 Field Documentation</td>
<td>356</td>
</tr>
<tr>
<td>5.3.2.1 dstArray</td>
<td>356</td>
</tr>
<tr>
<td>5.3.2.2 dstDevice</td>
<td>356</td>
</tr>
<tr>
<td>5.3.2.3 dstHost</td>
<td>356</td>
</tr>
<tr>
<td>5.3.2.4 dstMemoryType</td>
<td>356</td>
</tr>
<tr>
<td>5.3.2.5 dstPitch</td>
<td>356</td>
</tr>
<tr>
<td>5.3.2.6 dstXInBytes</td>
<td>357</td>
</tr>
<tr>
<td>5.3.2.7 dstY</td>
<td>357</td>
</tr>
<tr>
<td>5.3.2.8 Height</td>
<td>357</td>
</tr>
<tr>
<td>5.3.2.9 srcArray</td>
<td>357</td>
</tr>
<tr>
<td>5.3.2.10 srcDevice</td>
<td>357</td>
</tr>
<tr>
<td>5.3.2.11 srcHost</td>
<td>357</td>
</tr>
<tr>
<td>5.3.2.12 srcMemoryType</td>
<td>357</td>
</tr>
<tr>
<td>5.3.2.13 srcPitch</td>
<td>357</td>
</tr>
<tr>
<td>5.3.2.14 srcXInBytes</td>
<td>357</td>
</tr>
<tr>
<td>5.3.2.15 srcY</td>
<td>357</td>
</tr>
<tr>
<td>5.3.2.16 WidthInBytes</td>
<td>357</td>
</tr>
<tr>
<td>5.4 CUDA_MEMCPY3D_PEER_st Struct Reference</td>
<td>358</td>
</tr>
<tr>
<td>5.4.1 Detailed Description</td>
<td>358</td>
</tr>
<tr>
<td>5.4.2 Field Documentation</td>
<td>358</td>
</tr>
<tr>
<td>5.4.2.1 Depth</td>
<td>358</td>
</tr>
<tr>
<td>5.4.2.2 dstArray</td>
<td>358</td>
</tr>
<tr>
<td>5.4.2.3 dstContext</td>
<td>358</td>
</tr>
<tr>
<td>5.4.2.4 dstDevice</td>
<td>359</td>
</tr>
<tr>
<td>5.4.2.5 dstHeight</td>
<td>359</td>
</tr>
<tr>
<td>5.4.2.6 dstHost</td>
<td>359</td>
</tr>
<tr>
<td>5.4.2.7 dstLOD</td>
<td>359</td>
</tr>
<tr>
<td>5.4.2.8 dstMemoryType</td>
<td>359</td>
</tr>
<tr>
<td>5.4.2.9 dstPitch</td>
<td>359</td>
</tr>
<tr>
<td>5.4.2.10 dstXInBytes</td>
<td>359</td>
</tr>
<tr>
<td>5.4.2.11 dstY</td>
<td>359</td>
</tr>
<tr>
<td>5.4.2.12 dstZ</td>
<td>359</td>
</tr>
<tr>
<td>5.4.2.13 Height</td>
<td>359</td>
</tr>
<tr>
<td>5.4.2.14 srcArray</td>
<td>359</td>
</tr>
<tr>
<td>5.4.2.15 srcContext</td>
<td>359</td>
</tr>
</tbody>
</table>
5.4.2.16 srcDevice  ......................................................... 360
5.4.2.17 srcHeight .......................................................... 360
5.4.2.18 srcHost ............................................................ 360
5.4.2.19 srcLOD ............................................................. 360
5.4.2.20 srcMemoryType .................................................. 360
5.4.2.21 srcPitch ........................................................... 360
5.4.2.22 srcXInBytes ...................................................... 360
5.4.2.23 srcY ................................................................. 360
5.4.2.24 srcZ ................................................................. 360
5.4.2.25 WidthInBytes .................................................... 360

5.5 CUDA_MEMCPY3D_st Struct Reference .......................... 361
  5.5.1 Detailed Description ............................................. 361
  5.5.2 Field Documentation ............................................. 361
    5.5.2.1 Depth ......................................................... 361
    5.5.2.2 dstArray ...................................................... 361
    5.5.2.3 dstDevice ..................................................... 361
    5.5.2.4 dstHeight ..................................................... 362
    5.5.2.5 dstHost ....................................................... 362
    5.5.2.6 dstLOD ......................................................... 362
    5.5.2.7 dstMemoryType ............................................... 362
    5.5.2.8 dstPitch ...................................................... 362
    5.5.2.9 dstXInBytes .................................................. 362
    5.5.2.10 dstY .......................................................... 362
    5.5.2.11 dstZ ........................................................ 362
    5.5.2.12 Height ......................................................... 362
    5.5.2.13 reserved0 .................................................... 362
    5.5.2.14 reserved1 .................................................... 362
    5.5.2.15 srcArray ...................................................... 362
    5.5.2.16 srcDevice ..................................................... 362
    5.5.2.17 srcHeight ..................................................... 363
    5.5.2.18 srcHost ....................................................... 363
    5.5.2.19 srcLOD ........................................................ 363
    5.5.2.20 srcMemoryType .............................................. 363
    5.5.2.21 srcPitch ...................................................... 363
    5.5.2.22 srcXInBytes .................................................. 363
    5.5.2.23 srcY .......................................................... 363
    5.5.2.24 srcZ .......................................................... 363
5.5.2.25 WidthInBytes ........................................ 363

5.6 cudaChannelFormatDesc Struct Reference .......................... 364
  5.6.1 Detailed Description ..................................... 364
  5.6.2 Field Documentation ..................................... 364
    5.6.2.1 f ....................................................... 364
    5.6.2.2 w ....................................................... 364
    5.6.2.3 x ....................................................... 364
    5.6.2.4 y ....................................................... 364
    5.6.2.5 z ....................................................... 364

5.7 cudaDeviceProp Struct Reference ................................ 365
  5.7.1 Detailed Description ..................................... 365
  5.7.2 Field Documentation ..................................... 366
    5.7.2.1 asyncEngineCount ................................... 366
    5.7.2.2 canMapHostMemory ................................... 366
    5.7.2.3 clockRate ........................................... 366
    5.7.2.4 computeMode ......................................... 366
    5.7.2.5 concurrentKernels ................................... 366
    5.7.2.6 deviceOverlap ....................................... 366
    5.7.2.7 ECCEnabled .......................................... 366
    5.7.2.8 integrated .......................................... 366
    5.7.2.9 kernelExecTimeoutEnabled ........................... 366
    5.7.2.10 l2CacheSize ........................................ 366
    5.7.2.11 major .............................................. 366
    5.7.2.12 maxGridSize ......................................... 367
    5.7.2.13 maxTexture1D ....................................... 367
    5.7.2.14 maxTexture1DLayered ................................ 367
    5.7.2.15 maxTexture2D ....................................... 367
    5.7.2.16 maxTexture2DLayered ................................ 367
    5.7.2.17 maxTexture3D ....................................... 367
    5.7.2.18 maxThreadsDim ...................................... 367
    5.7.2.19 maxThreadsPerBlock ................................ 367
    5.7.2.20 maxThreadsPerMultiProcessor ........................ 367
    5.7.2.21 memoryBusWidth ..................................... 367
    5.7.2.22 memoryClockRate ................................... 367
    5.7.2.23 memPitch .......................................... 367
    5.7.2.24 minor ............................................. 368
    5.7.2.25 multiProcessorCount ............................... 368
5.10.2.6 srcArray .................................................. 373
5.10.2.7 srcPos .................................................. 373
5.10.2.8 srcPtr ............................................... 374

5.11 cudaMemcpy3DPeerParms Struct Reference ........................................ 375
  5.11.1 Detailed Description .......................................... 375
  5.11.2 Field Documentation .......................................... 375
    5.11.2.1 dstArray ........................................... 375
    5.11.2.2 dstDevice .......................................... 375
    5.11.2.3 dstPos ............................................. 375
    5.11.2.4 dstPtr ............................................. 375
    5.11.2.5 extent ............................................. 375
    5.11.2.6 srcArray ........................................... 375
    5.11.2.7 srcDevice .......................................... 375
    5.11.2.8 srcPos ............................................. 376
    5.11.2.9 srcPtr ............................................. 376

5.12 cudaPitchedPtr Struct Reference ................................................. 377
  5.12.1 Detailed Description .......................................... 377
  5.12.2 Field Documentation .......................................... 377
    5.12.2.1 pitch .............................................. 377
    5.12.2.2 ptr ................................................. 377
    5.12.2.3 xsize .............................................. 377
    5.12.2.4 ysize .............................................. 377

5.13 cudaPointerAttributes Struct Reference ........................................... 378
  5.13.1 Detailed Description .......................................... 378
  5.13.2 Field Documentation .......................................... 378
    5.13.2.1 device ............................................ 378
    5.13.2.2 devicePointer .................................... 378
    5.13.2.3 hostPointer ...................................... 378
    5.13.2.4 memoryType ...................................... 378

5.14 cudaPos Struct Reference .................................................. 379
  5.14.1 Detailed Description .......................................... 379
  5.14.2 Field Documentation .......................................... 379
    5.14.2.1 x .................................................. 379
    5.14.2.2 y .................................................. 379
    5.14.2.3 z .................................................. 379

5.15 CUdevprop_st Struct Reference ................................................. 380
  5.15.1 Detailed Description .......................................... 380
Chapter 1

Deprecated List

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

- `cudaDeviceMapHost`: This flag must be set in order to allocate pinned host memory that is accessible to the device. If this flag is not set, `cudaHostGetDevicePointer()` will always return a failure code.
- `cudaDeviceLmemResizeToMax`: Instruct CUDA to not reduce local memory after resizing local memory for a kernel. This can prevent thrashing by local memory allocations when launching many kernels with high local memory usage at the cost of potentially increased memory usage.

Global `cudaThreadExit`

Global `cudaThreadGetCacheConfig`

Global `cudaThreadGetLimit`

Global `cudaThreadSetCacheConfig`

Global `cudaThreadSetLimit`

Global `cudaThreadSynchronize`

Global `cudaD3D9MapResources`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D9RegisterResource`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D9ResourceGetMappedArray`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D9ResourceGetMappedPitch`  This function is deprecated as of Cuda 3.0.
Global `cudaD3D9ResourceGetMappedPointer`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D9ResourceGetMappedSize`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D9ResourceGetSurfaceDimensions`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D9ResourceSetMapFlags`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D9UnmapResources`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D9UnregisterResource`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D10MapResources`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D10RegisterResource`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D10ResourceGetMappedArray`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D10ResourceGetMappedPitch`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D10ResourceGetMappedPointer`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D10ResourceGetMappedSize`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D10ResourceGetSurfaceDimensions`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D10ResourceSetMapFlags`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D10UnmapResources`  This function is deprecated as of Cuda 3.0.

Global `cudaD3D10UnregisterResource`  This function is deprecated as of Cuda 3.0.

Global `cudaGLMapBufferObject`  This function is deprecated as of Cuda 3.0.

Global `cudaGLMapBufferObjectAsync`  This function is deprecated as of Cuda 3.0.
Global `cudaGLRegisterBufferObject`  This function is deprecated as of Cuda 3.0.

Global `cudaGLSetBufferObjectMapFlags`  This function is deprecated as of Cuda 3.0.

Global `cudaGLUnmapBufferObject`  This function is deprecated as of Cuda 3.0.

Global `cudaGLUnmapBufferObjectAsync`  This function is deprecated as of Cuda 3.0.

Global `cudaGLUnregisterBufferObject`  This function is deprecated as of Cuda 3.0.

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

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

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

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

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

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

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

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

Global `cudaDeviceBlockingSync`  

Global `CU_CTX_BLOCKING_SYNC`  

Global `CUDA_ERROR_CONTEXT_ALREADY_CURRENT`  This error return is deprecated as of CUDA 3.2. It is no longer an error to attempt to push the active context via `cuCtxPushCurrent()`.
Global `cuCtxCreate`  This flag was deprecated as of CUDA 4.0 and was replaced with `CU_CTX_SCHED_BLOCKING_SYNC`.

Global `cuCtxAttach`

Global `cuCtxDetach`

Global `cuFuncSetBlockShape`

Global `cuFuncSetSharedSize`

Global `cuLaunch`

Global `cuLaunchGrid`

Global `cuLaunchGridAsync`

Global `cuParamSetf`

Global `cuParamSeti`

Global `cuParamSetSize`

Global `cuParamSetTexRef`

Global `cuParamSetv`

Global `cuTexRefCreate`

Global `cuTexRefDestroy`

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

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

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

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

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

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

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

Global `cuD3D9MapResources`  This function is deprecated as of Cuda 3.0.

Global `cuD3D9RegisterResource`  This function is deprecated as of Cuda 3.0.

Global `cuD3D9ResourceGetMappedArray`  This function is deprecated as of Cuda 3.0.

Global `cuD3D9ResourceGetMappedPitch`  This function is deprecated as of Cuda 3.0.

Global `cuD3D9ResourceGetMappedPointer`  This function is deprecated as of Cuda 3.0.

Global `cuD3D9ResourceGetMappedSize`  This function is deprecated as of Cuda 3.0.

Global `cuD3D9ResourceGetSurfaceDimensions`  This function is deprecated as of Cuda 3.0.

Global `cuD3D9ResourceSetMapFlags`  This function is deprecated as of Cuda 3.0.

Global `cuD3D9UnmapResources`  This function is deprecated as of Cuda 3.0.

Global `cuD3D9UnregisterResource`  This function is deprecated as of Cuda 3.0.

Global `cuD3D10MapResources`  This function is deprecated as of Cuda 3.0.

Global `cuD3D10RegisterResource`  This function is deprecated as of Cuda 3.0.

Global `cuD3D10ResourceGetMappedArray`  This function is deprecated as of Cuda 3.0.
Global `cuD3D10ResourceGetMappedPitch`  This function is deprecated as of Cuda 3.0.

Global `cuD3D10ResourceGetMappedPointer`  This function is deprecated as of Cuda 3.0.

Global `cuD3D10ResourceGetMappedSize`  This function is deprecated as of Cuda 3.0.

Global `cuD3D10ResourceGetSurfaceDimensions`  This function is deprecated as of Cuda 3.0.

Global `cuD3D10ResourceSetMapFlags`  This function is deprecated as of Cuda 3.0.

Global `cuD3D10UnmapResources`  This function is deprecated as of Cuda 3.0.

Global `cuD3D10UnregisterResource`  This function is deprecated as of Cuda 3.0.
Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

CUDA Runtime API .......................................................... 11
   Device Management ..................................................... 13
      Thread Management [DEPRECATED] .......................... 23
   Error Handling ......................................................... 28
   Stream Management ................................................... 30
   Event Management ..................................................... 33
   Execution Control ....................................................... 37
   Memory Management .................................................. 41
   Unified Addressing ..................................................... 77
   Peer Device Memory Access ......................................... 80
   OpenGL Interoperability ............................................. 82
      OpenGL Interoperability [DEPRECATED] ............... 148
   Direct3D 9 Interoperability ........................................ 85
      Direct3D 9 Interoperability [DEPRECATED] ............ 130
   Direct3D 10 Interoperability ....................................... 90
      Direct3D 10 Interoperability [DEPRECATED] .......... 139
   Direct3D 11 Interoperability ....................................... 95
   VDPAU Interoperability .............................................. 100
   Graphics Interoperability ........................................... 103
   Texture Reference Management .................................. 107
   Surface Reference Management ................................... 112
   Version Management .................................................. 114
   C++ API Routines ...................................................... 115
   Interactions with the CUDA Driver API ....................... 128
   Data types used by CUDA Runtime ............................... 153
CUDA Driver API ................................................................ 167
   Data types used by CUDA driver ................................... 168
   Initialization .............................................................. 188
   Version Management ................................................... 189
   Device Management ..................................................... 190
   Context Management ................................................... 197
      Context Management [DEPRECATED] ........................ 206
   Module Management ................................................... 208
<table>
<thead>
<tr>
<th>Memory Management</th>
<th>215</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unified Addressing</td>
<td>263</td>
</tr>
<tr>
<td>Stream Management</td>
<td>267</td>
</tr>
<tr>
<td>Event Management</td>
<td>270</td>
</tr>
<tr>
<td>Execution Control</td>
<td>274</td>
</tr>
<tr>
<td>Execution Control [DEPRECATED]</td>
<td>278</td>
</tr>
<tr>
<td>Texture Reference Management</td>
<td>285</td>
</tr>
<tr>
<td>Texture Reference Management [DEPRECATED]</td>
<td>293</td>
</tr>
<tr>
<td>Surface Reference Management</td>
<td>295</td>
</tr>
<tr>
<td>Peer Context Memory Access</td>
<td>297</td>
</tr>
<tr>
<td>Graphics Interoperability</td>
<td>299</td>
</tr>
<tr>
<td>OpenGL Interoperability</td>
<td>304</td>
</tr>
<tr>
<td>OpenGL Interoperability [DEPRECATED]</td>
<td>307</td>
</tr>
<tr>
<td>Direct3D 9 Interoperability</td>
<td>313</td>
</tr>
<tr>
<td>Direct3D 9 Interoperability [DEPRECATED]</td>
<td>319</td>
</tr>
<tr>
<td>Direct3D 10 Interoperability</td>
<td>328</td>
</tr>
<tr>
<td>Direct3D 10 Interoperability [DEPRECATED]</td>
<td>334</td>
</tr>
<tr>
<td>Direct3D 11 Interoperability</td>
<td>343</td>
</tr>
<tr>
<td>VDPAU Interoperability</td>
<td>349</td>
</tr>
</tbody>
</table>
Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

- CUDA ARRAY3D_DESCRIPTOR_st .............................................. 353
- CUDA ARRAY_DESCRIPTOR_st .................................................. 355
- CUDA_MEMCPY2D_st .............................................................. 356
- CUDA_MEMCPY3D_PEER_st ......................................................... 358
- CUDA_MEMCPY3D_st ............................................................... 361
- cudaChannelFormatDesc .......................................................... 364
- cudaMemcpy3DParms .............................................................. 373
- cudaMemcpy3DPeerParms ......................................................... 375
- cudaPitchedPtr ................................................................. 377
- cudaPointerAttributes .......................................................... 378
- cudaPos ............................................................................. 379
- CUDevprop_st ................................................................. 380
- surfaceReference .............................................................. 382
- textureReference .............................................................. 383
Chapter 4

Module Documentation

4.1 CUDA Runtime API

Modules

- Device Management
- Error Handling
- Stream Management
- Event Management
- Execution Control
- Memory Management
- Unified Addressing
- Peer Device Memory Access
- OpenGL Interoperability
- Direct3D 9 Interoperability
- Direct3D 10 Interoperability
- Direct3D 11 Interoperability
- VDPAU Interoperability
- Graphics Interoperability
- Texture Reference Management
- Surface Reference Management
- Version Management
- C++ API Routines

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

- Interactions with the CUDA Driver API

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

- Data types used by CUDA Runtime

Defines

- #define CUDART_VERSION 4000
4.1.1 Detailed Description

There are two levels for the runtime API.

The C API (cuda_runtime_api.h) is a C-style interface that does not require compiling with nvcc.

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

4.1.2 Define Documentation

4.1.2.1 #define CUDART_VERSION 4000

CUDA Runtime API Version 4.0
4.2 Device Management

Modules

- Thread Management [DEPRECATED]

Functions

- `cudaError_t cudaChooseDevice (int *device, const struct cudaDeviceProp *prop)`
  Select compute-device which best matches criteria.

- `cudaError_t cudaDeviceGetCacheConfig (enum cudaFuncCache *pCacheConfig)`
  Returns the preferred cache configuration for the current device.

- `cudaError_t cudaDeviceGetLimit (size_t *pValue, enum cudaLimit limit)`
  Returns resource limits.

- `cudaError_t cudaDeviceReset (void)`
  Destroy all allocations and reset all state on the current device in the current process.

- `cudaError_t cudaDeviceSetCacheConfig (enum cudaFuncCache cacheConfig)`
  Sets the preferred cache configuration for the current device.

- `cudaError_t cudaDeviceSetLimit (enum cudaLimit limit, size_t value)`
  Set resource limits.

- `cudaError_t cudaDeviceSynchronize (void)`
  Wait for compute device to finish.

- `cudaError_t cudaGetDevice (int *device)`
  Returns which device is currently being used.

- `cudaError_t cudaGetDeviceCount (int *count)`
  Returns the number of compute-capable devices.

- `cudaError_t cudaGetDeviceProperties (struct cudaDeviceProp *prop, int device)`
  Returns information about the compute-device.

- `cudaError_t cudaSetDevice (int device)`
  Set device to be used for GPU executions.

- `cudaError_t cudaSetDeviceFlags (unsigned int flags)`
  Sets flags to be used for device executions.

- `cudaError_t cudaSetValidDevices (int *device_arr, int len)`
  Set a list of devices that can be used for CUDA.
4.2.1 Detailed Description

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

4.2.2 Function Documentation

4.2.2.1 cudaError_t cudaChooseDevice (int *device, const struct cudaDeviceProp *prop)

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

Parameters:

    device - Device with best match
    prop   - Desired device properties

Returns:

    cudaSuccess, cudaErrorInvalidValue

Note:

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

See also:

    cudaGetDeviceCount, cudaGetDevice, cudaSetDevice, cudaGetDeviceProperties

4.2.2.2 cudaError_t cudaDeviceGetCacheConfig (enum cudaFuncCache *pCacheConfig)

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

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

The supported cache configurations are:

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

Parameters:

    pCacheConfig - Returned cache configuration

Returns:

    cudaSuccess, cudaErrorInitializationError

Note:

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

See also:

    cudaDeviceSetCacheConfig, cudaFuncSetCacheConfig (C API), cudaFuncSetCacheConfig (C++ API)
4.2.2.3  cudaError_t cudaDeviceGetLimit (size_t ∗pValue, enum cudaLimit limit)

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

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

Parameters:

  limit  - Limit to query
  pValue  - Returned size in bytes of limit

Returns:

cudaSuccess, cudaErrorUnsupportedLimit, cudaErrorInvalidValue

Note:

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

See also:

cudaDeviceSetLimit

4.2.2.4  cudaError_t cudaDeviceReset (void)

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

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

Returns:

cudaSuccess

Note:

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

See also:

cudaDeviceSynchronize

4.2.2.5  cudaError_t cudaDeviceSetCacheConfig (enum cudaFuncCache cacheConfig)

On devices where the L1 cache and shared memory use the same hardware resources, this sets through cacheConfig the preferred cache configuration for the current device. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute the function. Any function preference set via cudaFuncSetCacheConfig (C API) or cudaFuncSetCacheConfig (C++ API) will be preferred over this device-wide setting. Setting the device-wide cache configuration to cudaFuncCachePreferNone will cause subsequent kernel launches to prefer to not change the cache configuration unless required to launch the kernel.
This setting does nothing on devices where the size of the L1 cache and shared memory are fixed.

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

The supported cache configurations are:

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

**Parameters:**

- `cacheConfig` - Requested cache configuration

**Returns:**

- `cudaSuccess`, `cudaErrorInitializationError`

**Note:**

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

**See also:**

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

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

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

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

- `cudaLimitStackSize` controls the stack size of each GPU thread. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error `cudaErrorUnsupportedLimit` being returned.

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

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

**Parameters:**

- `limit` - Limit to set
4.2 Device Management

value - Size in bytes of limit

Returns:

cudaSuccess, cudaErrorUnsupportedLimit, cudaErrorInvalidValue

Note:

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

See also:

cudaDeviceGetLimit

4.2.2.7 cudaError_t cudaDeviceSynchronize (void)

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

Returns:

cudaSuccess

Note:

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

See also:

cudaDeviceSynchronize

4.2.2.8 cudaError_t cudaGetDevice (int * device)

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

Parameters:

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

Returns:

cudaSuccess

Note:

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

See also:

cudaGetDeviceCount, cudaSetDevice, cudaGetDeviceProperties, cudaChooseDevice
4.2.2.9 cudaError_t cudaGetDeviceCount (int * count)

Returns in *count the number of devices with compute capability greater or equal to 1.0 that are available for execution. If there is no such device then cudaGetDeviceCount() will return cudaErrorNoDevice. If no driver can be loaded to determine if any such devices exist then cudaGetDeviceCount() will return cudaErrorInsufficientDriver.

Parameters:

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

Returns:

- cudaSuccess, cudaErrorNoDevice, cudaErrorInsufficientDriver

Note:

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

See also:

cudaGetDevice, cudaSetDevice, cudaGetDeviceProperties, cudaChooseDevice

4.2.2.10 cudaError_t cudaGetDeviceProperties (struct cudaDeviceProp * prop, int device)

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

```c
struct cudaDeviceProp {
    char name[256];
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int regsPerBlock;
    int warpSize;
    size_t memPitch;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    int clockRate;
    size_t totalConstMem;
    int major;
    int minor;
    size_t textureAlignment;
    int deviceOverlap;
    int multiProcessorCount;
    int kernelExecTimeoutEnabled;
    int integrated;
    int canMapHostMemory;
    int computeMode;
    int maxTexture1D;
    int maxTexture2D[2];
    int maxTexture3D[3];
    int maxTexture1DLayered[2];
    int maxTexture2DLayered[3];
    size_t surfaceAlignment;
    int concurrentKernels;
    int ECCEnabled;
    int pciBusID;
    int pciDeviceID;
    int pciDomainID;
    int tccDriver;
    int asyncEngineCount;
    int unifiedAddressing;
};
```
int memoryClockRate;
int memoryBusWidth;
int l2CacheSize;
int maxThreadsPerMultiProcessor;
}

where:

- **name[256]** is an ASCII string identifying the device;
- **totalGlobalMem** is the total amount of global memory available on the device in bytes;
- **sharedMemPerBlock** is the maximum amount of shared memory available to a thread block in bytes; this amount is shared by all thread blocks simultaneously resident on a multiprocessor;
- **regsPerBlock** is the maximum number of 32-bit registers available to a thread block; this number is shared by all thread blocks simultaneously resident on a multiprocessor;
- **warpSize** is the warp size in threads;
- **memPitch** is the maximum pitch in bytes allowed by the memory copy functions that involve memory regions allocated through `cudaMallocPitch()`;
- **maxThreadsPerBlock** is the maximum number of threads per block;
- **maxThreadsDim[3]** contains the maximum size of each dimension of a block;
- **maxGridSize[3]** contains the maximum size of each dimension of a grid;
- **clockRate** is the clock frequency in kilohertz;
- **totalConstMem** is the total amount of constant memory available on the device in bytes;
- **major**, **minor** are the major and minor revision numbers defining the device’s compute capability;
- **textureAlignment** is the alignment requirement; texture base addresses that are aligned to `textureAlignment` bytes do not need an offset applied to texture fetches;
- **deviceOverlap** is 1 if the device can concurrently copy memory between host and device while executing a kernel, or 0 if not. Deprecated, use instead `asyncEngineCount`.
- **multiProcessorCount** is the number of multiprocessors on the device;
- **kernelExecTimeoutEnabled** is 1 if there is a run time limit for kernels executed on the device, or 0 if not.
- **integrated** is 1 if the device is an integrated (motherboard) GPU and 0 if it is a discrete (card) component.
- **canMapHostMemory** is 1 if the device can map host memory into the CUDA address space for use with `cudaHostAlloc()`/`cudaHostGetDevicePointer()`, or 0 if not;
- **computeMode** is the compute mode that the device is currently in. Available modes are as follows:
  - **cudaComputeModeDefault**: Default mode - Device is not restricted and multiple threads can use `cudaSetDevice()` with this device.
  - **cudaComputeModeExclusive**: Compute-exclusive mode - Only one thread will be able to use `cudaSetDevice()` with this device.
  - **cudaComputeModeProhibited**: Compute-prohibited mode - No threads can use `cudaSetDevice()` with this device.
cudaComputeModeExclusiveProcess: Compute-exclusive-process mode - Many threads in one process will be able to use cudaSetDevice() with this device. Any errors from calling cudaSetDevice() with an exclusive (and occupied) or prohibited device will only show up after a non-device management runtime function is called. At that time, cudaErrorNoDevice will be returned.

- `maxTexture1D` is the maximum 1D texture size.
- `surfaceAlignment` specifies the alignment requirements for surfaces.
- `concurrentKernels` is 1 if the device supports executing multiple kernels within the same context simultaneously, or 0 if not. It is not guaranteed that multiple kernels will be resident on the device concurrently so this feature should not be relied upon for correctness;
- `ECCEnabled` is 1 if the device has ECC support turned on, or 0 if not.
- `pciBusID` is the PCI bus identifier of the device.
- `pciDeviceID` is the PCI device (sometimes called slot) identifier of the device.
- `pciDomainID` is the PCI domain identifier of the device.
- `tccDriver` is 1 if the device is using a TCC driver or 0 if not.
- `asyncEngineCount` is 1 when the device can concurrently copy memory between host and device while executing a kernel. It is 2 when the device can concurrently copy memory between host and device in both directions and execute a kernel at the same time. It is 0 if neither of these is supported.
- `unifiedAddressing` is 1 if the device shares a unified address space with the host and 0 otherwise.
- `memoryClockRate` is the peak memory clock frequency in kilohertz.
- `memoryBusWidth` is the memory bus width in bits.
- `l2CacheSize` is L2 cache size in bytes.
- `maxThreadsPerMultiProcessor` is the number of maximum resident threads per multiprocessor.

Parameters:

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

Returns:

- `cudaSuccess`, `cudaErrorInvalidDevice`

See also:

- `cudaGetDeviceCount`, `cudaGetDevice`, `cudaSetDevice`, `cudaChooseDevice`
4.2.2.11 cudaError_t cudaSetDevice (int device)

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

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

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

Parameters:

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

Returns:

cudaSuccess, cudaErrorInvalidDevice, cudaErrorSetOnActiveProcess

Note:

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

See also:

cudaGetDeviceCount, cudaGetDevice, cudaGetDeviceProperties, cudaChooseDevice

4.2.2.12 cudaError_t cudaSetDeviceFlags (unsigned int flags)

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

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

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

- cudaDeviceScheduleAuto: The default value if the flags parameter is zero, uses a heuristic based on the number of active CUDA contexts in the process $C$ and the number of logical processors in the system $P$. If $C > P$, then CUDA will yield to other OS threads when waiting for the device, otherwise CUDA will not yield while waiting for results and actively spin on the processor.

- cudaDeviceScheduleSpin: Instruct CUDA to actively spin when waiting for results from the device. This can decrease latency when waiting for the device, but may lower the performance of CPU threads if they are performing work in parallel with the CUDA thread.

- cudaDeviceScheduleYield: Instruct CUDA to yield its thread when waiting for results from the device. This can increase latency when waiting for the device, but can increase the performance of CPU threads performing work in parallel with the device.

- cudaDeviceScheduleBlockingSync: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the device to finish work.
• **cudaDeviceBlockingSync**: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the device to finish work.

  **Deprecated**

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

  **Parameters:**

  `flags` - Parameters for device operation

  **Returns:**

  `cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorSetOnActiveProcess`

  **See also:**

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

4.2.2.13 **cudaError_t cudaSetValidDevices (int *device_arr, int len)**

Sets a list of devices for CUDA execution in priority order using `device_arr`. The parameter `len` specifies the number of elements in the list. CUDA will try devices from the list sequentially until it finds one that works. If this function is not called, or if it is called with a `len` of 0, then CUDA will go back to its default behavior of trying devices sequentially from a default list containing all of the available CUDA devices in the system. If a specified device ID in the list does not exist, this function will return `cudaErrorInvalidDevice`. If `len` is not 0 and `device_arr` is NULL or if `len` exceeds the number of devices in the system, then `cudaErrorInvalidValue` is returned.

**Parameters:**

`device_arr` - List of devices to try

`len` - Number of devices in specified list

**Returns:**

`cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`

**Note:**

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

**See also:**

`cudaGetDeviceCount`, `cudaSetDevice`, `cudaGetDeviceProperties`, `cudaSetDeviceFlags`, `cudaChooseDevice`
4.3 Thread Management [DEPRECATED]

Functions

- `cudaError_t cudaThreadExit (void)`
  Exit and clean up from CUDA launches.

- `cudaError_t cudaThreadGetCacheConfig (enum cudaFuncCache *pCacheConfig)`
  Returns the preferred cache configuration for the current device.

- `cudaError_t cudaThreadGetLimit (size_t *pValue, enum cudaLimit limit)`
  Returns resource limits.

- `cudaError_t cudaThreadSetCacheConfig (enum cudaFuncCache cacheConfig)`
  Sets the preferred cache configuration for the current device.

- `cudaError_t cudaThreadSetLimit (enum cudaLimit limit, size_t value)`
  Set resource limits.

- `cudaError_t cudaThreadSynchronize (void)`
  Wait for compute device to finish.

4.3.1 Detailed Description

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

4.3.2 Function Documentation

4.3.2.1 `cudaError_t cudaThreadExit (void)`

Deprecated

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

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

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

Returns:

- `cudaSuccess`

Note:

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

    cudaDeviceReset

4.3.2.2  cudaError_t cudaThreadGetCacheConfig (enum cudaFuncCache * pCacheConfig)

Deprecated

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

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

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

The supported cache configurations are:

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

Parameters:

    pCacheConfig - Returned cache configuration

Returns:

    cudaSuccess, cudaErrorInitializationError

Note:

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

See also:

    cudaDeviceGetCacheConfig

4.3.2.3  cudaError_t cudaThreadGetLimit (size_t * pValue, enum cudaLimit limit)

Deprecated

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

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

- cudaLimitStackSize: stack size of each GPU thread;
4.3 Thread Management [DEPRECATED]

- **cudaLimitPrintFifoSize**: size of the FIFO used by the printf() device system call.
- **cudaLimitMallocHeapSize**: size of the heap used by the malloc() and free() device system calls;

**Parameters:**

- \textit{limit} - Limit to query
- \textit{pValue} - Returned size in bytes of limit

**Returns:**

- cudaSuccess, cudaErrorUnsupportedLimit, cudaErrorInvalidValue

**Note:**

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

**See also:**

cudaDeviceGetLimit

4.3.2.4 \texttt{cudaError_t cudaThreadSetCacheConfig (enum cudaFuncCache \textit{cacheConfig})}

**Deprecation Note:**

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

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

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

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

The supported cache configurations are:

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

**Parameters:**

- \textit{cacheConfig} - Requested cache configuration

**Returns:**

- cudaSuccess, cudaErrorInitializationError

---

Generated for NVIDIA CUDA Library by Doxygen
Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cudaDeviceSetCacheConfig

4.3.2.5 cudaError_t cudaThreadSetLimit (enum cudaLimit limit, size_t value)

Deprecated

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

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

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

- cudaLimitStackSize controls the stack size of each GPU thread. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error cudaErrorUnsupportedLimit being returned.

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

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

Parameters:

- limit - Limit to set
- value - Size in bytes of limit

Returns:
cudaSuccess, cudaErrorUnsupportedLimit, cudaErrorInvalidValue

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

See also:
cudaDeviceSetLimit
4.3.2.6  cudaError_t cudaThreadSynchronize (void)

Deprecated

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

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

Returns:

cudaSuccess

Note:

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

See also:

cudaDeviceSynchronize
4.4 Error Handling

Functions

- const char ∗ cudaGetErrorString (cudaError_t error)
  Returns the message string from an error code.

- cudaError_t cudaGetLastError (void)
  Returns the last error from a runtime call.

- cudaError_t cudaPeekAtLastError (void)
  Returns the last error from a runtime call.

4.4.1 Detailed Description

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

4.4.2 Function Documentation

4.4.2.1 const char ∗ cudaGetErrorString (cudaError_t error)

Returns the message string from an error code.

Parameters:

- error - Error code to convert to string

Returns:

- char ∗ pointer to a NULL-terminated string

See also:

cudaGetLastError, cudaPeekAtLastError, cudaError

4.4.2.2 cudaError_t cudaGetLastError (void)

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

Returns:

4.4 Error Handling

Note:

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

See also:

cudaPeekAtLastError, cudaGetErrorString, cudaError

4.4.2.3 cudaError_t cudaPeekAtLastError (void)

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

Returns:


Note:

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

See also:

cudaGetLastError, cudaGetErrorString, cudaError
4.5 Stream Management

Functions

- `cudaError_t cudaStreamCreate (cudaStream_t *pStream)`
  Create an asynchronous stream.

- `cudaError_t cudaStreamDestroy (cudaStream_t stream)`
  Destroys and cleans up an asynchronous stream.

- `cudaError_t cudaStreamQuery (cudaStream_t stream)`
  Queries an asynchronous stream for completion status.

- `cudaError_t cudaStreamSynchronize (cudaStream_t stream)`
  Waits for stream tasks to complete.

- `cudaError_t cudaStreamWaitEvent (cudaStream_t stream, cudaEvent_t event, unsigned int flags)`
  Make a compute stream wait on an event.

4.5.1 Detailed Description

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

4.5.2 Function Documentation

4.5.2.1 `cudaError_t cudaStreamCreate (cudaStream_t * pStream)`

Creates a new asynchronous stream.

Parameters:

- `pStream` - Pointer to new stream identifier

Returns:

- cudaSuccess, cudaErrorInvalidValue

Note:

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

See also:

- cudaStreamQuery, cudaStreamSynchronize, cudaStreamWaitEvent, cudaStreamDestroy

4.5.2.2 `cudaError_t cudaStreamDestroy (cudaStream_t stream)`

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

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

Parameters:

- \textit{stream} - Stream identifier

Returns:

- cudaSuccess, cudaErrorInvalidResourceHandle

Note:

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

See also:

cudaStreamCreate, cudaStreamQuery, cudaStreamWaitEvent, cudaStreamSynchronize

4.5.2.3 \texttt{cudaError_t cudaStreamQuery (cudaStream_t \textit{stream})}

Returns \texttt{cudaSuccess} if all operations in \textit{stream} have completed, or \texttt{cudaErrorNotReady} if not.

Parameters:

- \textit{stream} - Stream identifier

Returns:

- cudaSuccess, cudaErrorNotReady, cudaErrorInvalidResourceHandle

Note:

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

See also:

cudaStreamCreate, cudaStreamWaitEvent, cudaStreamSynchronize, cudaStreamDestroy

4.5.2.4 \texttt{cudaError_t cudaStreamSynchronize (cudaStream_t \textit{stream})}

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

Parameters:

- \textit{stream} - Stream identifier

Returns:

- cudaSuccess, cudaErrorInvalidResourceHandle

Note:

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

See also:

cudaStreamCreate, cudaStreamQuery, cudaStreamWaitEvent, cudaStreamDestroy
4.5.2.5 `cudaError_t cudaStreamWaitEvent (cudaStream_t stream, cudaEvent_t event, unsigned int flags)`

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

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

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

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

**Parameters:**

- `stream` - Stream to wait
- `event` - Event to wait on
- `flags` - Parameters for the operation (must be 0)

**Returns:**

- `cudaSuccess`, `cudaErrorInvalidResourceHandle`

**Note:**

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

**See also:**

- `cudaStreamCreate`, `cudaStreamQuery`, `cudaStreamSynchronize`, `cudaStreamDestroy`
4.6 Event Management

Functions

- cudaError_t cudaEventCreate (cudaEvent_t *event)
  Creates an event object.

- cudaError_t cudaEventCreateWithFlags (cudaEvent_t *event, unsigned int flags)
  Creates an event object with the specified flags.

- cudaError_t cudaEventDestroy (cudaEvent_t event)
  Destroys an event object.

- cudaError_t cudaEventElapsedTime (float *ms, cudaEvent_t start, cudaEvent_t end)
  Computes the elapsed time between events.

- cudaError_t cudaEventQuery (cudaEvent_t event)
  Queries an event’s status.

- cudaError_t cudaEventRecord (cudaEvent_t event, cudaStream_t stream=0)
  Records an event.

- cudaError_t cudaEventSynchronize (cudaEvent_t event)
  Waits for an event to complete.

4.6.1 Detailed Description

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

4.6.2 Function Documentation

4.6.2.1 cudaError_t cudaEventCreate (cudaEvent_t * event)

Creates an event object using cudaEventDefault.

Parameters:

  event - Newly created event

Returns:

  cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorLaunchFailure, cudaErrorMemoryAllocation

Note:

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

See also:

  cudaEventCreate (C++ API), cudaEventCreateWithFlags, cudaEventRecord, cudaEventQuery, cudaEventSynchronize, cudaEventDestroy, cudaEventElapsedTime, cudaStreamWaitEvent
4.6.2.2 \texttt{cudaError_t cudaEventCreateWithFlags (cudaEvent_t * event, unsigned int flags)}

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

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

\textbf{Parameters:}

- \texttt{event} - Newly created event
- \texttt{flags} - Flags for new event

\textbf{Returns:}

- \texttt{cudaSuccess}, \texttt{cudaErrorInitializationError}, \texttt{cudaErrorInvalidValue}, \texttt{cudaErrorLaunchFailure}, \texttt{cudaErrorMemoryAllocation}

\textbf{Note:}

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

\textbf{See also:}

- \texttt{cudaEventCreate (C API)}, \texttt{cudaEventSynchronize}, \texttt{cudaEventDestroy}, \texttt{cudaEventElapsedTime}, \texttt{cudaStreamWaitEvent}

4.6.2.3 \texttt{cudaError_t cudaEventDestroy (cudaEvent_t event)}

Destroys the event specified by \texttt{event}.

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

\textbf{Parameters:}

- \texttt{event} - Event to destroy

\textbf{Returns:}

- \texttt{cudaSuccess}, \texttt{cudaErrorInitializationError}, \texttt{cudaErrorInvalidValue}, \texttt{cudaErrorLaunchFailure}

\textbf{Note:}

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

\textbf{See also:}

- \texttt{cudaEventCreate (C API)}, \texttt{cudaEventCreateWithFlags}, \texttt{cudaEventQuery}, \texttt{cudaEventSynchronize}, \texttt{cudaEventRecord}, \texttt{cudaEventElapsedTime}
4.6 Event Management

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

Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds).

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

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

**Parameters:**

- `ms` - Time between `start` and `end` in ms
- `start` - Starting event
- `end` - Ending event

**Returns:**

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

**Note:**

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

**See also:**

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

### 4.6.2.5 cudaError_t cudaEventQuery (cudaEvent_t event)

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

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

**Parameters:**

- `event` - Event to query

**Returns:**

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

**Note:**

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

**See also:**

- `cudaEventCreate (C API)`, `cudaEventCreateWithFlags`, `cudaEventRecord`, `cudaEventSynchronize`, `cudaEventDestroy`, `cudaEventElapsedTime`
4.6.2.6  cudaError_t cudaEventRecord (cudaEvent_t event, cudaStream_t stream = 0)

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

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

Parameters:

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

Returns:

  cudaSuccess, cudaErrorInvalidValue, cudaErrorInitializationError, cudaErrorInvalidResourceHandle, cudaErrorLaunchFailure

Note:

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

See also:

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

4.6.2.7  cudaError_t cudaEventSynchronize (cudaEvent_t event)

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

If cudaEventRecord() has not been called on event, cudaSuccess is returned immediately.

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

Parameters:

  event - Event to wait for

Returns:

  cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorLaunchFailure

Note:

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

See also:

  cudaEventCreate (C API), cudaEventCreateWithFlags, cudaEventRecord, cudaEventQuery, cudaEventDestroy, cudaEventElapsedTime
4.7 Execution Control

Functions

- cudaError_t cudaConfigureCall (dim3 gridDim, dim3 blockDim, size_t sharedMem=0, cudaStream_t stream=0)
  
  Configure a device-launch.

- cudaError_t cudaFuncGetAttributes (struct cudaFuncAttributes *attr, const char *func)
  
  Find out attributes for a given function.

- cudaError_t cudaFuncSetCacheConfig (const char *func, enum cudaFuncCache cacheConfig)
  
  Sets the preferred cache configuration for a device function.

- cudaError_t cudaLaunch (const char *entry)
  
  Launches a device function.

- cudaError_t cudaSetDoubleForDevice (double *d)
  
  Converts a double argument to be executed on a device.

- cudaError_t cudaSetDoubleForHost (double *d)
  
  Converts a double argument after execution on a device.

- cudaError_t cudaSetupArgument (const void *arg, size_t size, size_t offset)
  
  Configure a device launch.

4.7.1 Detailed Description

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

4.7.2 Function Documentation

4.7.2.1 cudaError_t cudaConfigureCall (dim3 gridDim, dim3 blockDim, size_t sharedMem = 0, cudaStream_t stream = 0)

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

Parameters:

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

Returns:

- cudaSuccess, cudaErrorInvalidConfiguration
Note:

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

See also:

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

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

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

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

Parameters:

attr - Return pointer to function’s attributes

func - Function to get attributes of

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidDeviceFunction

Note:

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

See also:

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

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

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

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

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

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

The supported cache configurations are:

- cudaFuncCachePreferNone: no preference for shared memory or L1 (default)
- cudaFuncCachePreferShared: prefer larger shared memory and smaller L1 cache
4.7 Execution Control

- cudaFuncCachePreferL1: prefer larger L1 cache and smaller shared memory

Parameters:

`func` - Char string naming device function
`cacheConfig` - Requested cache configuration

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidDeviceFunction

Note:

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

See also:

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

4.7.2.4 cudaError_t cudaLaunch (const char * entry)

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

Parameters:

`entry` - Device char string naming device function to execute

Returns:

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

Note:

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

See also:

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

4.7.2.5 cudaError_t cudaSetDoubleForDevice (double * d)

Parameters:

`d` - Double to convert

Converts the double value of `d` to an internal float representation if the device does not support double arithmetic. If the device does natively support doubles, then this function does nothing.
Returns:

cudaSuccess

Note:

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

See also:

cudaConfigureCall, cudaFuncSetCacheConfig (C API), cudaFuncGetAttributes (C API), cudaLaunch (C API), cudaSetDoubleForHost, cudaSetupArgument (C API)

4.7.2.6 cudaError_t cudaSetDoubleForHost (double *d)

Converts the double value of \(d\) from a potentially internal float representation if the device does not support double arithmetic. If the device does natively support doubles, then this function does nothing.

Parameters:

\(d\) - Double to convert

Returns:

cudaSuccess

Note:

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

See also:

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

4.7.2.7 cudaError_t cudaSetupArgument (const void *arg, size_t size, size_t offset)

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

Parameters:

\(arg\) - Argument to push for a kernel launch

\(size\) - Size of argument

\(offset\) - Offset in argument stack to push new arg

Returns:

cudaSuccess

Note:

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

See also:

cudaConfigureCall, cudaFuncSetCacheConfig (C API), cudaFuncGetAttributes (C API), cudaLaunch (C API), cudaSetDoubleForDevice, cudaSetDoubleForHost, cudaSetupArgument (C++ API)
4.8 Memory Management

Functions

- **cudaError_t cudaFree** (void *devPtr)
  Frees memory on the device.

- **cudaError_t cudaFreeArray** (struct cudaMemcpy *array)
  Frees an array on the device.

- **cudaError_t cudaFreeHost** (void *ptr)
  Frees page-locked memory.

- **cudaError_t cudaGetSymbolAddress** (void **devPtr, const char *symbol)
  Finds the address associated with a CUDA symbol.

- **cudaError_t cudaGetSymbolSize** (size_t *size, const char *symbol)
  Finds the size of the object associated with a CUDA symbol.

- **cudaError_t cudaHostAlloc** (void **pHost, size_t size, unsigned int flags)
  Allocates page-locked memory on the host.

- **cudaError_t cudaHostGetDevicePointer** (void **pDevice, void **pHost, unsigned int flags)
  Passes back device pointer of mapped host memory allocated by **cudaHostAlloc** or registered by **cudaHostRegister**.

- **cudaError_t cudaHostGetFlags** (unsigned int *pFlags, void **pHost)
  Passes back flags used to allocate pinned host memory allocated by **cudaHostAlloc**.

- **cudaError_t cudaHostRegister** (void *ptr, size_t size, unsigned int flags)
  Registers an existing host memory range for use by CUDA.

- **cudaError_t cudaHostUnregister** (void *ptr)
  Unregisters a memory range that was registered with **cudaHostRegister**.

- **cudaError_t cudaMalloc** (void **devPtr, size_t size)
  Allocate memory on the device.

- **cudaError_t cudaMalloc3D** (struct cudaMemcpy *pitchedDevPtr, struct cudaMemcpy *devPtr, struct cudaMemcpy *extent)
  Allocates logical 3D memory objects on the device.

- **cudaError_t cudaMalloc3DArray** (struct cudaMemcpy **array, const struct cudaMemcpy *desc, struct cudaMemcpy *extent, unsigned int flags=0)
  Allocate an array on the device.

- **cudaError_t cudaMallocArray** (struct cudaMemcpy **array, const struct cudaMemcpy *desc, size_t width, size_t height=0, unsigned int flags=0)
  Allocate an array on the device.

- **cudaError_t cudaMallocHost** (void **ptr, size_t size)
  Allocates page-locked memory on the host.
• cudaError_t cudaMallocPitch (void **devPtr, size_t *pitch, size_t width, size_t height)
  Allocates pitched memory on the device.

• cudaError_t cudaMemcpy (void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
  Copies data between host and device.

• cudaError_t cudaMemcpy2D (void *dst, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind)
  Copies data between host and device.

• cudaError_t cudaMemcpy2DToArrayToarray (struct cudaArray *dst, size_t wOffsetDst, size_t hOffsetDst, const struct cudaArray *src, size_t wOffsetSrc, size_t hOffsetSrc, size_t width, size_t height, enum cudaMemcpyKind kind=cudaMemcpyDeviceToDevice)
  Copies data between host and device.

• cudaError_t cudaMemcpy2DAsync (void *dst, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind, cudaStream_t stream=0)
  Copies data between host and device.

• cudaError_t cudaMemcpy3D (const struct cudaMemcpy3DParms *p)
  Copies data between 3D objects.

• cudaError_t cudaMemcpy3DAsync (const struct cudaMemcpy3DParms *p, cudaStream_t stream=0)
  Copies data between 3D objects.

• cudaError_t cudaMemcpy3DPeer (const struct cudaMemcpy3DPeerParms *p)
  Copies memory between devices.

• cudaError_t cudaMemcpy3DPeerAsync (const struct cudaMemcpy3DPeerParms *p, cudaStream_t stream=0)
  Copies memory between devices asynchronously.

• cudaError_t cudaMemcpyArrayToArray (struct cudaArray *dst, size_t wOffsetDst, size_t hOffsetDst, const struct cudaArray *src, size_t wOffsetSrc, size_t hOffsetSrc, size_t count, enum cudaMemcpyKind kind=cudaMemcpyDeviceToDevice)
Copies data between host and device.

- **cudaError_t cudaMemcpyAsync** (void *dst, const void *src, size_t count, enum cudaMemcpyKind kind, cudaMemcpyStream_t stream=0)
  Copies data between host and device.

- **cudaError_t cudaMemcpyFromArray** (void *dst, const struct cudaArray *src, size_t wOffset, size_t hOffset, size_t count, enum cudaMemcpyKind kind)
  Copies data between host and device.

- **cudaError_t cudaMemcpyFromArrayAsync** (void *dst, const struct cudaArray *src, size_t wOffset, size_t hOffset, size_t count, enum cudaMemcpyKind kind, cudaMemcpyStream_t stream=0)
  Copies data between host and device.

- **cudaError_t cudaMemcpyFromArray** (void *dst, const char *symbol, size_t offset=0, enum cudaMemcpyKind kind=cudaMemcpyDeviceToHost)
  Copies data from the given symbol on the device.

- **cudaError_t cudaMemcpyFromArrayAsync** (void *dst, const char *symbol, size_t offset=0, enum cudaMemcpyKind kind, cudaMemcpyStream_t stream=0)
  Copies data from the given symbol on the device.

- **cudaError_t cudaMemcpyPeer** (void *dst, int dstDevice, const void *src, int srcDevice, size_t count)
  Copies memory between two devices.

- **cudaError_t cudaMemcpyPeerAsync** (void *dst, int dstDevice, const void *src, int srcDevice, size_t count, cudaMemcpyStream_t stream=0)
  Copies memory between two devices asynchronously.

- **cudaError_t cudaMemcpyToArray** (struct cudaArray *dst, size_t wOffset, size_t hOffset, const void *src, size_t count, enum cudaMemcpyKind kind)
  Copies data between host and device.

- **cudaError_t cudaMemcpyToArrayAsync** (struct cudaArray *dst, size_t wOffset, size_t hOffset, const void *src, size_t count, enum cudaMemcpyKind kind, cudaMemcpyStream_t stream=0)
  Copies data between host and device.

- **cudaError_t cudaMemcpyToSymbol** (const char *symbol, const void *src, size_t count, size_t offset=0, enum cudaMemcpyKind kind=cudaMemcpyHostToDevice)
  Copies data to the given symbol on the device.

- **cudaError_t cudaMemcpyToSymbolAsync** (const char *symbol, const void *src, size_t count, size_t offset, enum cudaMemcpyKind kind, cudaMemcpyStream_t stream=0)
  Copies data to the given symbol on the device.

- **cudaError_t cudaMemcpyGetInfo** (size_t *free, size_t *total)
  Gets free and total device memory.

- **cudaError_t cudaMemcpySet** (void *devPtr, int value, size_t count)
  Initializes or sets device memory to a value.
• cudaError_t cudaMemset2D (void *devPtr, size_t pitch, int value, size_t width, size_t height)

  Initializes or sets device memory to a value.

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

  Initializes or sets device memory to a value.

• cudaError_t cudaMemset3D (struct cudaPitchedPtr pitchedDevPtr, int value, struct cudaExtent extent)

  Initializes or sets device memory to a value.

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

  Initializes or sets device memory to a value.

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

  Initializes or sets device memory to a value.

• struct cudaExtent make_cudaExtent (size_t w, size_t h, size_t d)

  Returns a cudaExtent based on input parameters.

• struct cudaPitchedPtr make_cudaPitchedPtr (void *d, size_t p, size_t xsz, size_t ysz)

  Returns a cudaPitchedPtr based on input parameters.

• struct cudaPos make_cudaPos (size_t x, size_t y, size_t z)

  Returns a cudaPos based on input parameters.

### 4.8.1 Detailed Description

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

### 4.8.2 Function Documentation

#### 4.8.2.1 cudaError_t cudaFree (void *devPtr)

Frees the memory space pointed to by devPtr, which must have been returned by a previous call to cudaMalloc() or cudaMallocPitch(). Otherwise, or if cudaFree(devPtr) has already been called before, an error is returned. If devPtr is 0, no operation is performed. cudaFree() returns cudaErrorInvalidDevicePointer in case of failure.

**Parameters:**

- devPtr - Device pointer to memory to free

**Returns:**

cudaSuccess, cudaErrorInvalidDevicePointer, cudaErrorInitializationError

**Note:**

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

See also:

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

4.8.2.2 cudaError_t cudaFreeArray (struct cudaArray *array)

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

Parameters:

array - Pointer to array to free

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInitializationError

Note:

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

See also:

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

4.8.2.3 cudaError_t cudaFreeHost (void *ptr)

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

Parameters:

ptr - Pointer to memory to free

Returns:

cudaSuccess, cudaErrorInitializationError

Note:

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

See also:

cudaMalloc, cudaMallocPitch, cudaFree, cudaMallocArray, cudaFreeArray, cudaMallocHost (C API), cudaMalloc3D, cudaMalloc3DArray, cudaHostAlloc
4.8.2.4  cudaError_t cudaGetSymbolAddress (void **devPtr, const char *symbol)

Returns in *devPtr the address of symbol symbol on the device. symbol can either be a variable that resides in global or constant memory space, or it can be a character string, naming a variable that resides in global or constant memory space. If symbol cannot be found, or if symbol is not declared in the global or constant memory space, *devPtr is unchanged and the error cudaErrorInvalidSymbol is returned. If there are multiple global or constant variables with the same string name (from separate files) and the lookup is done via character string, cudaErrorDuplicateVariableName is returned.

Parameters:

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

Returns:

  cudaSuccess, cudaErrorInvalidSymbol, cudaErrorDuplicateVariableName

Note:

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

See also:

  cudaMemcpy (C++ API) cudaMemcpyHost (C API)

4.8.2.5  cudaError_t cudaGetSymbolSize (size_t *size, const char *symbol)

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

Parameters:

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

Returns:

  cudaSuccess, cudaErrorInvalidSymbol

Note:

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

See also:

  cudaMemcpy (C API) cudaMemcpyHost (C++ API)

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

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

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

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

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

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

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

Memory allocated by this function must be freed with cudaFreeHost.

**Parameters:**

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

**Returns:**

cudaSuccess, cudaErrorMemoryAllocation

**Note:**

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

**See also:**

cudaSetDeviceFlags, cudaMallocHost (C API), cudaFreeHost

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

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

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

flags provides for future releases. For now, it must be set to 0.
4.8.2.8  cudaError_t cudaHostGetFlags (unsigned int *pFlags, void *pHost)

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

Parameters:

- pFlags - Returned flags word
- pHost - Host pointer

Returns:

cudaSuccess, cudaErrorInvalidValue

Note:

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

See also:

cudaHostAlloc

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

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

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

- cudaHostRegisterPortable: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
4.8 Memory Management

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

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

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

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

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

The memory page-locked by this function must be unregistered with `cudaHostUnregister()`.

**Parameters:**

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

**Returns:**

- `cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorMemoryAllocation`

**Note:**

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

**See also:**

- `cudaHostUnregister`, `cudaHostGetFlags`, `cudaHostGetDevicePointer`

### 4.8.2.10 `cudaError_t cudaHostUnregister (void *ptr)`

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

The base address must be the same one specified to `cudaHostRegister()`.

**Parameters:**

- `ptr` - Host pointer to memory to unregister

**Returns:**

- `cudaSuccess`, `cudaErrorInvalidValue`

**Note:**

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

**See also:**

- `cudaHostUnregister`
4.8.2.11  cudaError_t cudaMalloc (void **devPtr, size_t size)

Allocates size bytes of linear memory on the device and returns in *devPtr a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. cudaMalloc() returns cudaErrorMemoryAllocation in case of failure.

Parameters:

devPtr - Pointer to allocated device memory

size - Requested allocation size in bytes

Returns:

cudaSuccess, cudaErrorMemoryAllocation

See also:

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

4.8.2.12  cudaError_t cudaMalloc3D (struct cudaPitchedPtr *pitchedDevPtr, struct cudaExtent extent)

Allocates at least width * height * depth bytes of linear memory on the device and returns a cudaPitchedPtr in which ptr is a pointer to the allocated memory. The function may pad the allocation to ensure hardware alignment requirements are met. The pitch returned in the pitch field of pitchedDevPtr is the width in bytes of the allocation.

The returned cudaPitchedPtr contains additional fields xsize and ysize, the logical width and height of the allocation, which are equivalent to the width and height extent parameters provided by the programmer during allocation.

For allocations of 2D and 3D objects, it is highly recommended that programmers perform allocations using cudaMemcpy3D() or cudaMemcpyPitch(). Due to alignment restrictions in the hardware, this is especially true if the application will be performing memory copies involving 2D or 3D objects (whether linear memory or CUDA arrays).

Parameters:

pitchedDevPtr - Pointer to allocated pitched device memory

extent - Requested allocation size (width field in bytes)

Returns:

cudaSuccess, cudaErrorMemoryAllocation

Note:

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

See also:

cudaMallocPitch, cudaFree, cudaMemcpy3D, cudaMemcpy3DAligned, cudaMemcpy3DArray, cudaMemcpy3DArrayBase, cudaMemcpyArray, cudaMemcpyHost, cudaMemcpyHostAsync, cudaMemcpyHostToHost, cudaMemcpyHostToHostAsync, make_cudaPitchedPtr, make_cudaExtent
4.8 Memory Management

4.8.2.13  

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

Allocates a CUDA array according to the cudaChannelFormatDesc structure desc and returns a handle to the new CUDA array in *array.

The cudaChannelFormatDesc is defined as:

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

where cudaChannelFormatKind is one of cudaChannelFormatKindSigned, cudaChannelFormatKindUnsigned, or cudaChannelFormatKindFloat.

cudaMalloc3DArray() can allocate the following:

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

Note:

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

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

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

Parameters:

- array - Pointer to allocated array in device memory
- desc - Requested channel format
- extent - Requested allocation size (width field in elements)
- flags - Flags for extensions (must be 0 for now)

Returns:

cudaSuccess, cudaErrorMemoryAllocation

Generated for NVIDIA CUDA Library by Doxygen
Note:

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

See also:

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

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

Allocates a CUDA array according to the cudaChannelFormatDesc structure desc and returns a handle to the new CUDA array in *array.

The cudaChannelFormatDesc is defined as:

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

where cudaChannelFormatKind is one of cudaChannelFormatKindSigned, cudaChannelFormatKindUnsigned, or cudaChannelFormatKindFloat.

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

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

Parameters:

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

Returns:

cudaSuccess, cudaErrorMemoryAllocation

Note:

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

See also:

cudaMalloc, cudaMallocPitch, cudaFree, cudaFreeArray, cudaMallocHost (C API), cudaFreeHost, cudaMalloc3D, cudaMalloc3DArray, cudaHostAlloc
4.8 Memory Management

4.8.2.15 cudaError_t cudaMallocHost (void **ptr, size_t size)

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

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

Returns:
- cudaSuccess, cudaErrorMemoryAllocation

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

See also:
- cudaMalloc, cudaMallocPitch, cudaMallocArray, cudaMalloc3D, cudaMalloc3DArray, cudaHostAlloc, cudaFree, cudaFreeArray, cudaMallocHost (C++ API), cudaFreeHost, cudaHostAlloc

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

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

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

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

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

Returns:
- cudaSuccess, cudaErrorMemoryAllocation
Note:

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

See also:

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

4.8.2.17 cudaMemcpy (void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)

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

Parameters:

dst - Destination memory address
src - Source memory address
count - Size in bytes to copy
kind - Type of transfer

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

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

See also:

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

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

Copies a matrix (height rows of width bytes each) from the memory area pointed to by src to the memory area pointed to by dst, where kind is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy. dpitch and spitch are the widths in memory in bytes of the 2D arrays pointed to by dst and src, including any padding added to the end of each row. The memory areas may not overlap. width must not exceed either dpitch or spitch. Calling cudaMemcpy2D() with dst and src pointers that do not match the direction of the copy results in an undefined behavior. cudaMemcpy2D() returns an error if dpitch or spitch exceeds the maximum allowed.

Parameters:

dst - Destination memory address
4.8 Memory Management

\[ dpitch \] - Pitch of destination memory
\[ src \] - Source memory address
\[ spitch \] - Pitch of source memory
\[ width \] - Width of matrix transfer (columns in bytes)
\[ height \] - Height of matrix transfer (rows)
\[ kind \] - Type of transfer

Returns:

\texttt{cudaSuccess}, \texttt{cudaErrorInvalidValue}, \texttt{cudaErrorInvalidPitchValue}, \texttt{cudaErrorInvalidDevicePointer}, \texttt{cudaErrorInvalidMemcpyDirection}

Note:

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

See also:

\texttt{cudaMemcpy}, \texttt{cudaMemcpyToArray}, \texttt{cudaMemcpy2DToArray}, \texttt{cudaMemcpyFromArray}, \texttt{cudaMemcpy2DFromArray}, \texttt{cudaMemcpyArrayToArray}, \texttt{cudaMemcpy2DArrayToArray}, \texttt{cudaMemcpyToSymbol}, \texttt{cudaMemcpyFromSymbol}, \texttt{cudaMemcpyAsync}, \texttt{cudaMemcpy2DAsync}, \texttt{cudaMemcpyToArrayAsync}, \texttt{cudaMemcpy2DToArrayAsync}, \texttt{cudaMemcpyFromArrayAsync}, \texttt{cudaMemcpy2DFromArrayAsync}, \texttt{cudaMemcpyToSymbolAsync}, \texttt{cudaMemcpyFromSymbolAsync}

4.8.2.19 \texttt{cudaError_t cudaMemcpy2DArrayToArray (struct cudaArray * dst, size_t wOffsetDst, size_t hOffsetDst, const struct cudaArray * src, size_t wOffsetSrc, size_t hOffsetSrc, size_t width, size_t height, enum cudaMemcpyKind kind = cudaMemcpyDeviceToDevice)}

Copies a matrix (\texttt{height} rows of \texttt{width} bytes each) from the CUDA array \texttt{srcArray} starting at the upper left corner (\texttt{wOffsetSrc, hOffsetSrc}) to the CUDA array \texttt{dst} starting at the upper left corner (\texttt{wOffsetDst, hOffsetDst}), where \texttt{kind} is one of \texttt{cudaMemcpyHostToHost}, \texttt{cudaMemcpyHostToDevice}, \texttt{cudaMemcpyDeviceToHost}, or \texttt{cudaMemcpyDeviceToDevice}, and specifies the direction of the copy. \texttt{wOffsetDst + width} must not exceed the width of the CUDA array \texttt{dst}. \texttt{wOffsetSrc + width} must not exceed the width of the CUDA array \texttt{src}.

Parameters:

\[ dst \] - Destination memory address
\[ wOffsetDst \] - Destination starting X offset
\[ hOffsetDst \] - Destination starting Y offset
\[ src \] - Source memory address
\[ wOffsetSrc \] - Source starting X offset
\[ hOffsetSrc \] - Source starting Y offset
\[ width \] - Width of matrix transfer (columns in bytes)
\[ height \] - Height of matrix transfer (rows)
\[ kind \] - Type of transfer

Returns:

\texttt{cudaSuccess}, \texttt{cudaErrorInvalidValue}, \texttt{cudaErrorInvalidMemcpyDirection}
Note:

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

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArrays, cudaMemcpy2DToArrays, cudaMemcpyFromArrays, cudaMemcpy2DFromArrays, cudaMemcpyArrayToArrays, cudaMemcpyToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToArrayAsync, cudaMemcpy2DArrayToArrayAsync, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync

cudaError_t cudaMemcpy2DAsync (void *dst, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind, cudaStream_t stream = 0)

Copies a matrix (height rows of width bytes each) from the memory area pointed to by src to the memory area pointed to by dst, where kind is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy. dpitch and spitch are the widths in memory in bytes of the 2D arrays pointed to by dst and src, including any padding added to the end of each row. The memory areas may not overlap. width must not exceed either dpitch or spitch. Calling cudaMemcpy2DAsync() with dst and src pointers that do not match the direction of the copy results in an undefined behavior. cudaMemcpy2DAsync() returns an error if dpitch or spitch is greater than the maximum allowed.

cudaMemcpy2DAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and stream is non-zero, the copy may overlap with operations in other streams.

Parameters:

dst - Destination memory address
dpitch - Pitch of destination memory
dsrc - Source memory address
dspitch - Pitch of source memory
width - Width of matrix transfer (columns in bytes)
height - Height of matrix transfer (rows)
kind - Type of transfer
stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidPitchValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

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

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArrays, cudaMemcpy2DToArrays, cudaMemcpyFromArrays, cudaMemcpy2DFromArrays, cudaMemcpyArrayToArrays, cudaMemcpyToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToArrayAsync, cudaMemcpy2DArrayToArrayAsync, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync

Generated for NVIDIA CUDA Library by Doxygen
4.8.2.21 cudaError_t cudaMemcpy2DFromArray (void *dst, size_t dpitch, const struct cudaArray *src, size_t wOffset, size_t hOffset, size_t width, size_t height, enum cudaMemcpyKind kind)

Copies a matrix (height rows of width bytes each) from the CUDA array srcArray starting at the upper left corner (wOffset, hOffset) to the memory area pointed to by dst, where kind is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy. dpitch is the width in memory in bytes of the 2D array pointed to by dst, including any padding added to the end of each row. wOffset + width must not exceed the width of the CUDA array src. width must not exceed dpitch. cudaMemcpy2DFromArray() returns an error if dpitch exceeds the maximum allowed.

Parameters:

- dst - Destination memory address
- dpitch - Pitch of destination memory
- src - Source memory address
- wOffset - Source starting X offset
- hOffset - Source starting Y offset
- width - Width of matrix transfer (columns in bytes)
- height - Height of matrix transfer (rows)
- kind - Type of transfer

Returns:

cudaSuccess, cudaMemcpyInvalidValue, cudaMemcpyInvalidDevicePointer, cudaMemcpyInvalidPitchValue, cudaMemcpyInvalidMemcpyDirection

Note:

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

See also:

ccudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromArraySymbol, cudaMemcpy2DFromArraySymbol, cudaMemcpyToArraySymbol, cudaMemcpy2DToArraySymbol, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromSymbolAsync, cudaMemcpy2DFromSymbolAsync

4.8.2.22 cudaError_t cudaMemcpy2DFromArrayAsync (void *dst, size_t dpitch, const struct cudaArray *src, size_t wOffset, size_t hOffset, size_t width, size_t height, enum cudaMemcpyKind kind, cudaStream_t stream = 0)

Copies a matrix (height rows of width bytes each) from the CUDA array srcArray starting at the upper left corner (wOffset, hOffset) to the memory area pointed to by dst, where kind is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy. dpitch is the width in memory in bytes of the 2D array pointed to by dst, including any padding added to the end of each row. wOffset + width must not exceed the width of the CUDA array src. width must not exceed dpitch. cudaMemcpy2DFromArrayAsync() returns an error if dpitch exceeds the maximum allowed.

ccudaMemcpy2DFromArrayAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and stream is non-zero, the copy may overlap with operations in other streams.
Parameters:

- `dst` - Destination memory address
- `dpitch` - Pitch of destination memory
- `src` - Source memory address
- `wOffset` - Source starting X offset
- `hOffset` - Source starting Y offset
- `width` - Width of matrix transfer (columns in bytes)
- `height` - Height of matrix transfer (rows)
- `kind` - Type of transfer
- `stream` - Stream identifier

Returns:

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

Note:

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

See also:

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

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

Copies a matrix (`height` rows of `width` bytes each) from the memory area pointed to by `src` to the CUDA array `dst` starting at the upper left corner (`wOffset`, `hOffset`) where `kind` is one of `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyHostToDevice`, or `cudaMemcpyDeviceToDevice`, and specifies the direction of the copy. `spitch` is the width in memory in bytes of the 2D array pointed to by `src`, including any padding added to the end of each row. `wOffset + width` must not exceed the width of the CUDA array `dst`. `width` must not exceed `spitch`. `cudaMemcpy2DToArray()` returns an error if `spitch` exceeds the maximum allowed.

Parameters:

- `dst` - Destination memory address
- `wOffset` - Destination starting X offset
- `hOffset` - Destination starting Y offset
- `src` - Source memory address
- `spitch` - Pitch of source memory
- `width` - Width of matrix transfer (columns in bytes)
- `height` - Height of matrix transfer (rows)
- `kind` - Type of transfer
Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidPitchValue, cudaErrorInvalidMemcpyDirection

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

See also:
cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromArray, cudaMemcpy2DToArrayAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

4.8.2.24 cudaMemcpy2DToArrayAsync (struct cudaArray * dst, size_t wOffset, size_t hOffset, const void * src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind, cudaStream_t stream = 0)

Copies a matrix (height rows of width bytes each) from the memory area pointed to by src to the CUDA array dst starting at the upper left corner (wOffset, hOffset) where kind is one of cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, and specifies the direction of the copy. spitch is the width in memory in bytes of the 2D array pointed to by src, including any padding added to the end of each row. wOffset + width must not exceed the width of the CUDA array dst. width must not exceed spitch. cudaMemcpy2DToArrayAsync() returns an error if spitch exceeds the maximum allowed.

cudaMemcpy2DToArrayAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and stream is non-zero, the copy may overlap with operations in other streams.

Parameters:

dst - Destination memory address
wOffset - Destination starting X offset
hOffset - Destination starting Y offset
src - Source memory address
spitch - Pitch of source memory
width - Width of matrix transfer (columns in bytes)
height - Height of matrix transfer (rows)
kind - Type of transfer
stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidPitchValue, cudaErrorInvalidMemcpyDirection

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

cudamempy, cudamempy2D, cudamempyToArray, cudamempy2DToArray, cudamempyFromArray, cudamempy2DFromArray, cudamempyArrayToArray, cudamempy2DArrayToArray, cudamempyToDevice, cudamempyFromDevice, cudamempyAsync, cudamempy2DAsync, cudamempyToArrayAsync, cudamempy2DToArrayAsync, cudamempyFromArrayAsync, cudamempyToDeviceAsync, cudamempyFromDeviceAsync, cudamempyFromSymbolAsync

4.8.2.25  cudaError_t cudaMemcpy3D (const struct cudaMemcpy3DParms * p)

struct cudaExtent {  
  size_t width;  
  size_t height;  
  size_t depth;  
};  
struct cudaExtent make_cudaExtent(size_t w, size_t h, size_t d);

struct cudaPos {  
  size_t x;  
  size_t y;  
  size_t z;  
};  
struct cudaPos make_cudaPos(size_t x, size_t y, size_t z);

struct cudaMemcpy3DParms {  
  struct cudaArray * srcArray;  
  struct cudaPos srcPos;  
  struct cudaPitchedPtr srcPtr;  
  struct cudaArray * dstArray;  
  struct cudaPos dstPos;  
  struct cudaPitchedPtr dstPtr;  
  struct cudaExtent extent;  
  enum cudaMemcpyKind kind;  
};

cudaMemcpy3D() copies data between two 3D objects. The source and destination objects may be in either host memory, device memory, or a CUDA array. The source, destination, extent, and kind of copy performed is specified by the cudaMemcpy3DParms struct which should be initialized to zero before use:

cudaMemcpy3DParms myParms = {0};

The struct passed to cudaMemcpy3D() must specify one of srcArray or srcPtr and one of dstArray or dstPtr. Passing more than one non-zero source or destination will cause cudaMemcpy3D() to return an error.

The srcPos and dstPos fields are optional offsets into the source and destination objects and are defined in units of each object’s elements. The element for a host or device pointer is assumed to be unsigned char. For CUDA arrays, positions must be in the range [0, 2048) for any dimension.

The extent field defines the dimensions of the transferred area in elements. If a CUDA array is participating in the copy, the extent is defined in terms of that array’s elements. If no CUDA array is participating in the copy then the extents are defined in elements of unsigned char.

The kind field defines the direction of the copy. It must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice.

If the source and destination are both arrays, cudaMemcpy3D() will return an error if they do not have the same element size.

The source and destination object may not overlap. If overlapping source and destination objects are specified, undefined behavior will result.
4.8 Memory Management

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

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

Parameters:

- `p` - 3D memory copy parameters

Returns:

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

Note:

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

See also:

- `cudaMalloc3D`, `cudaMalloc3DArray`, `cudaMemset3D`, `cudaMemcpy3DAsync`, `cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`, `cudaMemcpyFromSymbol`, `cudaMemcpyFromSymbolAsync`, `cudaMemcpyToSymbol`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`, `cudaMemcpy2DFromSymbolAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyFromArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbolAsync`, `cudaMemcpyFromSymbolAsync`, `make_cudaExtent`, `make_cudaPos`

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

```c
struct cudaExtent {
    size_t width;
    size_t height;
    size_t depth;
};
struct cudaExtent make_cudaExtent(size_t w, size_t h, size_t d);

struct cudaPos {
    size_t x;
    size_t y;
    size_t z;
};
struct cudaPos make_cudaPos(size_t x, size_t y, size_t z);

struct cudaMemcpy3DParms {
    struct cudaArray *srcArray;
    struct cudaPos srcPos;
    struct cudaPitchedPtr srcPtr;
    struct cudaArray *dstArray;
    struct cudaPos dstPos;
    struct cudaMemcpy2DRetryStatus dstStatus;
    struct cudaMemcpy2DAligned dstAligned;
    struct cudaExtent extent;
    enum cudaMemcpyKind kind;
};
```

cudaMemcpy3DAsync() copies data between two 3D objects. The source and destination objects may be in either host memory, device memory, or a CUDA array. The source, destination, extent, and kind of copy performed is specified by the `cudaMemcpy3DParms` struct which should be initialized to zero before use.
CUDA memcpy3D parameters

```c
cudaMemcpy3DParms myParms = {0};
```

The struct passed to `cudaMemcpy3DAsync()` must specify one of `srcArray` or `srcPtr` and one of `dstArray` or `dstPtr`. Passing more than one non-zero source or destination will cause `cudaMemcpy3DAsync()` to return an error.

The `srcPos` and `dstPos` fields are optional offsets into the source and destination objects and are defined in units of each object’s elements. The element for a host or device pointer is assumed to be `unsigned char`. For CUDA arrays, positions must be in the range [0, 2048) for any dimension.

The `extent` field defines the dimensions of the transferred area in elements. If a CUDA array is participating in the copy, the extent is defined in terms of that array’s elements. If no CUDA array is participating in the copy then the extents are defined in elements of `unsigned char`.

The `kind` field defines the direction of the copy. It must be one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, or `cudaMemcpyDeviceToDevice`.

If the source and destination are both arrays, `cudaMemcpy3DAsync()` will return an error if they do not have the same element size.

The source and destination object may not overlap. If overlapping source and destination objects are specified, undefined behavior will result.

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

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

`cudaMemcpy3DAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost` and `stream` is non-zero, the copy may overlap with operations in other streams.

**Parameters:**

- `p` - 3D memory copy parameters
  - `stream` - Stream identifier

**Returns:**

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

**Note:**

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

**See also:**

- `cudaMalloc3D`, `cudaMalloc3DArray`, `cudaMemset3D`, `cudaMemcpy3D`, `cudaMemcpy2D`, `cudaMemcpyToArray`, `cudaMemcpy2DToArray`, `cudaMemcpyFromArray`, `cudaMemcpy2DFromArray`, `cudaMemcpyToArrayAsync`, `cudaMemcpy2DToArrayAsync`, `cudaMemcpyFromArrayAsync`, `cudaMemcpy2DFromArrayAsync`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, `cudaMemcpyAsync`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyHostToDeviceAsync`, `cudaMemcpyDeviceToDeviceAsync`, `cudaMemcpyFromSymbolAsync`, `make_cudaExtent`, `make_cudaPos`
4.8 Memory Management

4.8.2.27 cudaMemcpy3DPeer

*cudaMemcpy3DPeer (const struct cudaMemcpy3DPeerParms * p)*

Perform a 3D memory copy according to the parameters specified in `p`. See the definition of the `cudaMemcpy3DPeerParms` structure for documentation of its parameters.

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

**Parameters:**

- `p` - Parameters for the memory copy

**Returns:**

- `cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`

**Note:**

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

**See also:**

- `cudaMemcpy`, `cudaMemcpyPeer`, `cudaMemcpyAsync`, `cudaMemcpyPeerAsync`, `cudaMemcpy3DPeerAsync`

4.8.2.28 cudaMemcpy3DPeerAsync

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

Perform a 3D memory copy according to the parameters specified in `p`. See the definition of the `cudaMemcpy3DPeerParms` structure for documentation of its parameters.

**Parameters:**

- `p` - Parameters for the memory copy
- `stream` - Stream identifier

**Returns:**

- `cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`

**Note:**

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

**See also:**

- `cudaMemcpy`, `cudaMemcpyPeer`, `cudaMemcpyAsync`, `cudaMemcpyPeerAsync`, `cudaMemcpy3DPeerAsync`

4.8.2.29 cudaMemcpyArrayOfArray

*cudaMemcpyArrayOfArray (struct cudaArray * dst, size_t wOffsetDst, size_t hOffsetDst, const struct cudaArray * src, size_t wOffsetSrc, size_t hOffsetSrc, size_t count, enum cudaMemcpyKind kind = cudaMemcpyDeviceToDevice)*

Copies `count` bytes from the CUDA array `src` starting at the upper left corner `(wOffsetSrc, hOffsetSrc)` to the CUDA array `dst` starting at the upper left corner `(wOffsetDst, hOffsetDst)` where `kind` is one of `cudaMemcpyHostToDevice`, `cudaMemcpyHostToHost`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDeviceToHost`, and specifies the direction of the copy.
Parameters:

- `dst` - Destination memory address
- `wOffsetDst` - Destination starting X offset
- `hOffsetDst` - Destination starting Y offset
- `src` - Source memory address
- `wOffsetSrc` - Source starting X offset
- `hOffsetSrc` - Source starting Y offset
- `count` - Size in bytes to copy
- `kind` - Type of transfer

Returns:

- `cudaSuccess`
- `cudaErrorInvalidValue`
- `cudaErrorInvalidMemcpyDirection`

Note:

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

See also:

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

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

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

`cudaMemcpyAsync()` is asynchronous with respect to the host, so the call may return before the copy is complete. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost` and the `stream` is non-zero, the copy may overlap with operations in other streams.

Parameters:

- `dst` - Destination memory address
- `src` - Source memory address
- `count` - Size in bytes to copy
- `kind` - Type of transfer
- `stream` - Stream identifier

Returns:

- `cudaSuccess`
- `cudaErrorInvalidValue`
- `cudaErrorInvalidDevicePointer`
- `cudaErrorInvalidMemcpyDirection`
4.8 Memory Management

Note:

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

See also:

cudamemcpy, cudamemcpy2D, cudamemcpyToArray, cudamemcpy2DToArray, cudamemcpyFromArray, cudamemcpy2DFromArray, cudamemcpyToArrayAsync, cudamemcpy2DToArrayAsync, cudamemcpyToSymbol, cudamemcpyFromSymbol, cudamemcpy2DAsync, cudamemcpyToArrayAsync, cudamemcpyFromArrayAsync, cudamemcpy2DFromArrayAsync, cudamemcpyToSymbolAsync, cudamemcpyFromSymbolAsync

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

Copies count bytes from the CUDA array src starting at the upper left corner (wOffset, hOffset) to the memory area pointed to by dst, where kind is one of cudaMemcpyHostToDevice, cudaMemcpyHostToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDeviceToHost, and specifies the direction of the copy.

Parameters:

dst - Destination memory address
src - Source memory address
wOffset - Source starting X offset
hOffset - Source starting Y offset
count - Size in bytes to copy
kind - Type of transfer

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

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

See also:

cudamemcpy, cudamemcpy2D, cudamemcpyToArray, cudamemcpy2DToArray, cudamemcpy2DFromArray, cudamemcpyFromArray, cudamemcpy2DFromArray, cudamemcpyToSymbol, cudamemcpyFromSymbol, cudamemcpy2DAsync, cudamemcpyToArrayAsync, cudamemcpyFromArrayAsync, cudamemcpy2DFromArrayAsync, cudamemcpyToSymbolAsync, cudamemcpyFromSymbolAsync

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

Copies count bytes from the CUDA array src starting at the upper left corner (wOffset, hOffset) to the memory area pointed to by dst, where kind is one of cudaMemcpyHostToDevice, cudaMemcpyHostToHost, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy. cudaMemcpyFromArrayAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed
as input. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and stream is non-zero, the copy may overlap with operations in other streams.

Parameters:
- \texttt{dst} - Destination memory address
- \texttt{src} - Source memory address
- \texttt{wOffset} - Source starting X offset
- \texttt{hOffset} - Source starting Y offset
- \texttt{count} - Size in bytes to copy
- \texttt{kind} - Type of transfer
- \texttt{stream} - Stream identifier

Returns:
- cudaSuccess
- cudaErrorInvalidValue
- cudaErrorInvalidDevicePointer
- cudaErrorInvalidMemcpyDirection

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

See also:
- cudaMemcpy
- cudaMemcpy2D
- cudaMemcpyToArray
- cudaMemcpy2DToArray
- cudaMemcpyFromArray
- cudaMemcpy2DFromArray
- cudaMemcpyArrayToarray
- cudaMemcpy2DArrayToarray
- cudaMemcpyToArrayAsync
- cudaMemcpy2DToArrayAsync
- cudaMemcpy2DFromArrayAsync
- cudaMemcpyToSymbol
- cudaMemcpyFromSymbol
- cudaMemcpyAsync
- cudaMemcpy2DAsync
- cudaMemcpyToArrayAsync
- cudaMemcpy2DToArrayAsync
- cudaMemcpyFromSymbolAsync
- cudaMemcpy2Symbol
- cudaMemcpyFromSymbol

4.8.2.33 cudaError_t cudaMemcpyFromSymbol (void * \texttt{dst}, const char * \texttt{symbol}, size_t \texttt{count}, size_t \texttt{offset} = 0, enum cudaMemcpyKind \texttt{kind} = cudaMemcpyDeviceToHost)

Copies \texttt{count} bytes from the memory area pointed to by \texttt{offset} bytes from the start of \texttt{symbol} to the memory area pointed to by \texttt{dst}. The memory areas may not overlap. \texttt{symbol} can either be a variable that resides in global or constant memory space, or it can be a character string, naming a variable that resides in global or constant memory space. \texttt{kind} can be either cudaMemcpyDeviceToHost or cudaMemcpyDeviceToDevice.

Parameters:
- \texttt{dst} - Destination memory address
- \texttt{symbol} - Symbol source from device
- \texttt{count} - Size in bytes to copy
- \texttt{offset} - Offset from start of symbol in bytes
- \texttt{kind} - Type of transfer

Returns:
- cudaSuccess
- cudaErrorInvalidValue
- cudaErrorInvalidSymbol
- cudaErrorInvalidDevicePointer
- cudaErrorInvalidMemcpyDirection

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

See also:

cudamemcpy, cudaMemcpy2D, cudaMemcpyToArrays, cudaMemcpy2DToArrays, cudaMemcpyFromArrays, cudaMemcpy2DFromArrayArrays, cudaMemcpyToArrayArrays, cudaMemcpy2DToArrayArrays, cudaMemcpyFromArrayArrays, cudaMemcpy2DFromArrayArrays, cudaMemcpyToArrayArrays, cudaMemcpy2DToArrayArrays, cudaMemcpyFromArrayArrays, cudaMemcpyToArrayArrays

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

Copies count bytes from the memory area pointed to by offset bytes from the start of symbol symbol to the memory area pointed to by dst. The memory areas may not overlap. symbol can either be a variable that resides in global or constant memory space, or it can be a character string, naming a variable that resides in global or constant memory space. kind can be either cudaMemcpyDeviceToHost or cudaMemcpyDeviceToDevice.

cudaMemcpyFromSymbolAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is cudaMemcpyDeviceToHost and stream is non-zero, the copy may overlap with operations in other streams.

Parameters:

dst - Destination memory address
symbol - Symbol source from device
count - Size in bytes to copy
offset - Offset from start of symbol in bytes
kind - Type of transfer
stream - Stream identifier

Returns:
cudaSuccess, cudaMemcpyErrorInvalidValue, cudaMemcpyErrorInvalidSymbol, cudaMemcpyErrorInvalidDevicePointer, cudaMemcpyErrorInvalidMemcpyDirection

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

See also:

cudamemcpy, cudaMemcpy2D, cudaMemcpyToArrays, cudaMemcpy2DToArrays, cudaMemcpyFromArrays, cudaMemcpy2DFromArrayArrays, cudaMemcpyToArrayArrays, cudaMemcpy2DToArrayArrays, cudaMemcpyFromArrayArrays, cudaMemcpy2DFromArrayArrays, cudaMemcpyToArrayArrays, cudaMemcpy2DToArrayArrays, cudaMemcpyFromArrayArrays, cudaMemcpyToArrayArrays

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

Copies memory from one device to memory on another device. dst is the base device pointer of the destination memory and dstDevice is the destination device. src is the base device pointer of the source memory and srcDevice is the source device. count specifies the number of bytes to copy.
Note that this function is asynchronous with respect to the host, but serialized with respect all pending and future asynchronous work in to the current device, srcDevice, and dstDevice (use cudaMemcpyPeerAsync to avoid this synchronization).

Parameters:

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

Returns:

- `cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`

Note:

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

See also:

- cudaMemcpy, cudaMemcpyPeer3D, cudaMemcpyAsync, cudaMemcpyPeerAsync, cudaMemcpy3DPeerAsync

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

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

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

Parameters:

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

Returns:

- `cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevice`

Note:

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

See also:

- cudaMemcpy, cudaMemcpyPeer, cudaMemcpyPeer3D, cudaMemcpyAsync, cudaMemcpy3DPeerAsync
4.8 Memory Management

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

Copies count bytes from the memory area pointed to by src to the CUDA array dst starting at the upper left corner (wOffset, hOffset), where kind is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy.

Parameters:

- dst - Destination memory address
- wOffset - Destination starting X offset
- hOffset - Destination starting Y offset
- src - Source memory address
- count - Size in bytes to copy
- kind - Type of transfer

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

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

See also:

ccudaMemcpy, cudaMemcpy2D, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpy2DToSymbol, cudaMemcpy2DFromArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync

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

Copies count bytes from the memory area pointed to by src to the CUDA array dst starting at the upper left corner (wOffset, hOffset), where kind is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy.

ccudaMemcpyToArrayAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and stream is non-zero, the copy may overlap with operations in other streams.

Parameters:

- dst - Destination memory address
- wOffset - Destination starting X offset
- hOffset - Destination starting Y offset
- src - Source memory address
- count - Size in bytes to copy
- kind - Type of transfer
- stream - CUDA stream
4.8.2.39  cudaMemcpyToSymbol (const char * symbol, const void * src, size_t count, size_t offset = 0, enum cudaMemcpyKind kind = cudaMemcpyHostToDevice)

Copies count bytes from the memory area pointed to by src to the memory area pointed to by offset bytes from the start of symbol symbol. The memory areas may not overlap. symbol can either be a variable that resides in global or constant memory space, or it can be a character string, naming a variable that resides in global or constant memory space. kind can be either cudaMemcpyHostToDevice or cudaMemcpyDeviceToDevice.

Parameters:

symbol - Symbol destination on device
src - Source memory address
count - Size in bytes to copy
offset - Offset from start of symbol in bytes
kind - Type of transfer

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSymbol, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

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

See also:
cudaMemcpy, cudaMemcpy2D, cudaMemcpyToSymbol, cudaMemcpy2DFromArray, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArraySync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync
4.8 Memory Management

4.8.2.40 cudaError_t cudaMemcpyToSymbolAsync (const char ∗symbol, const void ∗src, size_t count, size_t offset, enum cudaMemcpyKind kind, cudaStream_t stream = 0)

Copies count bytes from the memory area pointed to by src to the memory area pointed to by offset bytes from the start of symbol symbol. The memory areas may not overlap. symbol can either be a variable that resides in global or constant memory space, or it can be a character string, naming a variable that resides in global or constant memory space. kind can be either cudaMemcpyHostToDevice or cudaMemcpyDeviceToDevice.

cudaMemcpyToSymbolAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is cudaMemcpyHostToDevice and stream is non-zero, the copy may overlap with operations in other streams.

Parameters:

- symbol - Symbol destination on device
- src - Source memory address
- count - Size in bytes to copy
- offset - Offset from start of symbol in bytes
- kind - Type of transfer
- stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSymbol, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

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

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyFromSymbolAsync

4.8.2.41 cudaError_t cudaMemGetInfo (size_t ∗free, size_t ∗total)

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

Parameters:

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

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorLaunchFailure

Note:

Note that this function may also return error codes from previous, asynchronous launches.
4.8.2.42  cudaMemcpy (void *devPtr, int value, size_t count)

Fills the first \texttt{count} bytes of the memory area pointed to by \texttt{devPtr} with the constant byte value \texttt{value}.

Parameters:

\texttt{devPtr} - Pointer to device memory

\texttt{value} - Value to set for each byte of specified memory

\texttt{count} - Size in bytes to set

Returns:

\texttt{cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer}

Note:

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

See also:

cudaMemset2D, cudaMemset3D, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy3DAsync

4.8.2.43  cudaMemcpy2D (void *devPtr, size_t pitch, int value, size_t width, size_t height)

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

Parameters:

\texttt{devPtr} - Pointer to 2D device memory

\texttt{pitch} - Pitch in bytes of 2D device memory

\texttt{value} - Value to set for each byte of specified memory

\texttt{width} - Width of matrix set (columns in bytes)

\texttt{height} - Height of matrix set (rows)

Returns:

\texttt{cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer}

Note:

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

See also:

cudaMemset, cudaMemcpy3D, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy3DAsync
4.8 Memory Management

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

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

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

Parameters:

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

Returns:

- cudaSuccess
- cudaErrorInvalidValue
- cudaErrorInvalidDevicePointer

Note:

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

See also:

- cudaMemset, cudaMemset2D, cudaMemset3D, cudaMemsetAsync, cudaMemset3DAsync

4.8.2.45  cudaError_t cudaMemset3D (struct cudaPitchedPtr pitchedDevPtr, int value, struct cudaExtent extent)

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

The extents of the initialized region are specified as a width in bytes, a height in rows, and a depth in slices. Extents with width greater than or equal to the xsize of pitchedDevPtr may perform significantly faster than extents narrower than the xsize. Secondarily, extents with height equal to the ysize of pitchedDevPtr will perform faster than when the height is shorter than the ysize.

This function performs fastest when the pitchedDevPtr has been allocated by cudaMalloc3D().

Parameters:

- pitchedDevPtr - Pointer to pitched device memory
- value          - Value to set for each byte of specified memory
- extent         - Size parameters for where to set device memory (width field in bytes)
4.8.2.46 cudaError_t cudaMemset3DAsync (struct cudaPitchedPtr pitchedDevPtr, int value, struct cudaExtent extent, cudaStream_t stream = 0)

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

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

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

This function performs fastest when the pitchedDevPtr has been allocated by cudaMalloc3D().

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

Parameters:

- **pitchedDevPtr** - Pointer to pitched device memory
- **value** - Value to set for each byte of specified memory
- **extent** - Size parameters for where to set device memory (width field in bytes)
- **stream** - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer

Note:

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

See also:

cudaMemset, cudaMemset2D, cudaMemset3D, cudaMemsetAsync, cudaMemset2DAsync, cudaMalloc3D, make_cudaPitchedPtr, make_cudaExtent
4.8.2.47  `cudaError_t cudaMemsetAsync (void *devPtr, int value, size_t count, cudaStream_t stream = 0)`

Fills the first `count` bytes of the memory area pointed to by `devPtr` with the constant byte value `value`. `cudaMemsetAsync()` is asynchronous with respect to the host, so the call may return before the memset is complete. The operation can optionally be associated to a stream by passing a non-zero `stream` argument. If `stream` is non-zero, the operation may overlap with operations in other streams.

Parameters:

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

Returns:

- `cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidDevicePointer`

Note:

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

See also:

- `cudaMemset`, `cudaMemset2D`, `cudaMemset3D`, `cudaMemset2DAsync`, `cudaMemset3DAsync`

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

Returns a `cudaExtent` based on the specified input parameters `w`, `h`, and `d`.

Parameters:

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

Returns:

- `cudaExtent` specified by `w`, `h`, and `d`

See also:

- `make_cudaPitchedPtr`, `make_cudaPos`

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

Returns a `cudaPitchedPtr` based on the specified input parameters `d`, `p`, `xsz`, and `ysz`.

Parameters:

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

Generated for NVIDIA CUDA Library by Doxygen
xsiz - Logical width of allocation in elements
ysiz - Logical height of allocation in elements

Returns:

cudaPitchedPtr specified by d, p, xsiz, and ysz

See also:

make_cudaExtent, make_cudaPos

4.8.2.50 struct cudaPos make_cudaPos (size_t x, size_t y, size_t z) [read]

Returns a cudaPos based on the specified input parameters x, y, and z.

Parameters:

x - X position
y - Y position
z - Z position

Returns:

cudaPos specified by x, y, and z

See also:

make_cudaExtent, make_cudaPitchedPtr
4.9 Unified Addressing

Functions

- cudaError_t cudaPointerGetAttributes (struct cudaPointerAttributes *attributes, void *ptr)
  
  Returns attributes about a specified pointer.

4.9.1 Detailed Description

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

4.9.2 Overview

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

4.9.3 Supported Platforms

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

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

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

4.9.4 Looking Up Information from Pointer Values

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

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

4.9.5 Automatic Mapping of Host Allocated Host Memory

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

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

Note that this is not the case for memory allocated using the flag cudaHostAllocWriteCombined, as discussed below.
4.9.6 Direct Access of Peer Memory

Upon enabling direct access from a device that supports unified addressing to another peer device that supports unified addressing using `cudaDeviceEnablePeerAccess()` all memory allocated in the peer device using `cudaMalloc()` and `cudaMallocPitch()` will immediately be accessible by the current device. The device pointer value through which any peer’s memory may be accessed in the current device is the same pointer value through which that memory may be accessed from the peer device.

4.9.7 Exceptions, Disjoint Addressing

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

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

4.9.8 Function Documentation

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

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

The `cudaPointerAttributes` structure is defined as:

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

In this structure, the individual fields mean

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

- `device` is the device against which `ptr` was allocated. If `ptr` has memory type `cudaMemoryTypeDevice` then this identifies the device on which the memory referred to by `ptr` physically resides. If `ptr` has memory type `cudaMemoryTypeHost` then this identifies the device which was current when the allocation was made (and if that device is deinitialized then this allocation will vanish with that device’s state).

- `devicePointer` is the device pointer alias through which the memory referred to by `ptr` may be accessed on the current device. If the memory referred to by `ptr` cannot be accessed directly by the current device then this is NULL.

- `hostPointer` is the host pointer alias through which the memory referred to by `ptr` may be accessed on the host. If the memory referred to by `ptr` cannot be accessed directly by the host then this is NULL.
Parameters:

attributes - Attributes for the specified pointer
ptr - Pointer to get attributes for

Returns:

cudaSuccess, cudaErrorInvalidDevice

See also:

cudaGetDeviceCount, cudaGetDevice, cudaSetDevice, cudaChooseDevice
4.10 Peer Device Memory Access

Functions

- `cudaError_t cudaDeviceCanAccessPeer (int *canAccessPeer, int device, int peerDevice)`
  Queries if a device may directly access a peer device’s memory.

- `cudaError_t cudaDeviceDisablePeerAccess (int peerDevice)`
  Disables direct access to memory allocations on a peer device and unregisters any registered allocations from that device.

- `cudaError_t cudaDeviceEnablePeerAccess (int peerDevice, unsigned int flags)`
  Enables direct access to memory allocations on a peer device.

4.10.1 Detailed Description

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

4.10.2 Function Documentation

4.10.2.1 `cudaError_t cudaDeviceCanAccessPeer (int * canAccessPeer, int device, int peerDevice)`

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

Parameters:

- `canAccessPeer` - Returned access capability
- `device` - Device from which allocations on `peerDevice` are to be directly accessed.
- `peerDevice` - Device on which the allocations to be directly accessed by `device` reside.

Returns:

- `cudaSuccess`, `cudaErrorInvalidDevice`

Note:

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

See also:

- `cudaDeviceEnablePeerAccess`, `cudaDeviceDisablePeerAccess`

4.10.2.2 `cudaError_t cudaDeviceDisablePeerAccess (int peerDevice)`

Disables registering memory on `peerDevice` for direct access from the current device. If there are any allocations on `peerDevice` which were registered in the current device using `cudaPeerRegister()` then these allocations will be automatically unregistered.
4.10 Peer Device Memory Access

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

**Parameters:**

- `peerDevice` - Peer device to disable direct access to

**Returns:**

- `cudaSuccess`, `cudaErrorPeerAccessNotEnabled`, `cudaErrorInvalidDevice`

**Note:**

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

**See also:**

- `cudaDeviceCanAccessPeer`, `cudaDeviceEnablePeerAccess`

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

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

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

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

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

Returns `cudaErrorPeerAccessAlreadyEnabled` if direct access of `peerDevice` from the current device has already been enabled.

Returns `cudaErrorInvalidValue` if `flags` is not 0.

**Parameters:**

- `peerDevice` - Peer device to enable direct access to from the current device
- `flags` - Reserved for future use and must be set to 0

**Returns:**

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

**Note:**

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

**See also:**

- `cudaDeviceCanAccessPeer`, `cudaDeviceDisablePeerAccess`
4.11 OpenGL Interoperability

Modules

- OpenGL Interoperability [DEPRECATED]

Functions

- cudaError_t cudaGraphicsGLRegisterBuffer (struct cudaGraphicsResource **resource, GLuint buffer, unsigned int flags)
  
  Registers an OpenGL buffer object.

- cudaError_t cudaGraphicsGLRegisterImage (struct cudaGraphicsResource **resource, GLuint image, GLenum target, unsigned int flags)
  
  Register an OpenGL texture or renderbuffer object.

- cudaError_t cudaWGLGetDevice (int *device, HGPUNV hGpu)
  
  Gets the CUDA device associated with hGpu.

4.11.1 Detailed Description

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

4.11.2 Function Documentation

4.11.2.1 cudaError_t cudaGraphicsGLRegisterBuffer (struct cudaGraphicsResource **resource, GLuint buffer, unsigned int flags)

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

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

Parameters:

resource - Pointer to the returned object handle
buffer - name of buffer object to be registered
flags - Register flags

Returns:

 cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown
Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:

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

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

- cudaGraphicsRegisterFlagsNone: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.

- cudaGraphicsRegisterFlagsReadOnly: Specifies that CUDA will not write to this resource.

- cudaGraphicsRegisterFlagsWriteDiscard: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

- cudaGraphicsRegisterFlagsSurfaceLoadStore: Specifies that CUDA will bind this resource to a surface reference.

The following image classes are currently disallowed:

- Textures with borders
- Multisampled renderbuffers

Parameters:

- resource - Pointer to the returned object handle
- image - name of texture or renderbuffer object to be registered
- target - Identifies the type of object specified by image, and must be one of GL_TEXTURE_2D, GL_TEXTURE_RECTANGLE, GL_TEXTURE_CUBE_MAP, GL_TEXTURE_3D, GL_TEXTURE_2D_ARRAY, or GL_RENDERBUFFER.
- flags - Register flags

Returns:

- cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

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

See also:
4.11.2.3 cudaError_t cudaWGLGetDevice (int *device, HGPUNV hGpu)

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

Parameters:
- *device - Returns the device associated with hGpu, or -1 if hGpu is not a compute device.
- hGpu - Handle to a GPU, as queried via WGL_NV_gpu_affinity()

Returns:
cudaSuccess

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

See also:
WGL_NV_gpu_affinity, cudaGLSetGLDevice
4.12 Direct3D 9 Interoperability

Modules

- Direct3D 9 Interoperability [DEPRECATED]

Enumerations

- enum cudaD3D9DeviceList {
  cudaD3D9DeviceListAll = 1,
  cudaD3D9DeviceListCurrentFrame = 2,
  cudaD3D9DeviceListNextFrame = 3 }

Functions

- cudaError_t cudaD3D9GetDevice (int *device, const char *pszAdapterName)
  Gets the device number for an adapter.

- cudaError_t cudaD3D9GetDevices (unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, IDirect3DDevice9 *pD3D9Device, enum cudaD3D9DeviceList deviceList)
  Gets the CUDA devices corresponding to a Direct3D 9 device.

- cudaError_t cudaD3D9GetDirect3DDevice (IDirect3DDevice9 **ppD3D9Device)
  Gets the Direct3D device against which the current CUDA context was created.

- cudaError_t cudaD3D9SetDirect3DDevice (IDirect3DDevice9 *pD3D9Device, int device=-1)
  Sets the Direct3D 9 device to use for interoperability with a CUDA device.

- cudaError_t cudaGraphicsD3D9RegisterResource (struct cudaGraphicsResource **resource, IDirect3DResource9 *pD3DResource, unsigned int flags)
  Register a Direct3D 9 resource for access by CUDA.

4.12.1 Detailed Description

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

4.12.2 Enumeration Type Documentation

4.12.2.1 enum cudaD3D9DeviceList

CUDA devices corresponding to a D3D9 device

Enumerator:

- cudaD3D9DeviceListAll  The CUDA devices for all GPUs used by a D3D9 device
- cudaD3D9DeviceListCurrentFrame  The CUDA devices for the GPUs used by a D3D9 device in its currently rendering frame
cudaD3D9DeviceListNextFrame  The CUDA devices for the GPUs to be used by a D3D9 device in the next frame

4.12.3  Function Documentation

4.12.3.1  cudaError_t cudaD3D9GetDevice (int ∗device, const char ∗pszAdapterName)

Returns in ∗device the CUDA-compatible device corresponding to the adapter name pszAdapterName obtained from EnumDisplayDevices or IDirect3D9::GetAdapterIdentifier(). If no device on the adapter with name pszAdapterName is CUDA-compatible then the call will fail.

Parameters:

    device  - Returns the device corresponding to pszAdapterName
    pszAdapterName  - D3D9 adapter to get device for

Returns:

    cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown

Note:

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

See also:

    cudaD3D9SetDirect3DDevice, cudaGraphicsD3D9RegisterResource,

4.12.3.2  cudaError_t cudaD3D9GetDevices (unsigned int ∗pCudaDeviceCount, int ∗pCudaDevices, unsigned int cudaDeviceCount, IDirect3DDevice9 ∗pD3D9Device, enum cudaD3D9DeviceList deviceList)

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

If any of the GPUs being used to render pDevice are not CUDA capable then the call will return cudaErrorNoDevice.

Parameters:

    pCudaDeviceCount  - Returned number of CUDA devices corresponding to pD3D9Device
    pCudaDevices  - Returned CUDA devices corresponding to pD3D9Device
    cudaDeviceCount  - The size of the output device array pCudaDevices
    pD3D9Device  - Direct3D 9 device to query for CUDA devices
    deviceList  - The set of devices to return. This set may be cudaD3D9DeviceListAll for all devices, cudaD3D9DeviceListCurrentFrame for the devices used to render the current frame (in SLI), or cudaD3D9DeviceListNextFrame for the devices used to render the next frame (in SLI).

Returns:

    cudaSuccess, cudaErrorNoDevice, cudaErrorUnknown

Note:

    Note that this function may also return error codes from previous, asynchronous launches.
4.12 Direct3D 9 Interoperability

See also:


4.12.3.3 cudaError_t cudaD3D9GetDirect3DDevice (IDirect3DDevice9 **ppD3D9Device)

Returns in *ppD3D9Device the Direct3D device against which this CUDA context was created in cudaD3D9SetDirect3DDevice().

Parameters:

ppD3D9Device - Returns the Direct3D device for this thread

Returns:

cudaSuccess, cudaErrorUnknown

Note:

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

See also:

cudaD3D9SetDirect3DDevice

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

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

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

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

Parameters:

pD3D9Device - Direct3D device to use for this thread
device - The CUDA device to use. This device must be among the devices returned when querying cudaD3D9DeviceListAll from cudaD3D9GetDevices, may be set to -1 to automatically select an appropriate CUDA device.

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorSetOnActiveProcess

Note:

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

See also:

cudaD3D9GetDevice, cudaGraphicsD3D9RegisterResource, cudaDeviceReset

Generated for NVIDIA CUDA Library by Doxygen
4.12.3.5 cudaError_t cudaGraphicsD3D9RegisterResource (struct cudaGraphicsResource ** resource, IDirect3DResource9 * pD3DResource, unsigned int flags)

Registers the Direct3D 9 resource pD3DResource for access by CUDA.

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

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

The type of pD3DResource must be one of the following.

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

The flags argument may be used to specify additional parameters at register time. The valid values for this parameter are

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

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

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

If Direct3D interoperability is not initialized using cudaD3D9SetDirect3DDevice then cudaErrorInvalidDevice is returned. If pD3DResource is of incorrect type or is already registered, then cudaErrorInvalidResourceHandle is returned. If pD3DResource cannot be registered, then cudaErrorUnknown is returned.

Parameters:

resource - Pointer to returned resource handle
pd3DResource - Direct3D resource to register
flags - Parameters for resource registration

Returns:

cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown
Note:

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

See also:

4.13 Direct3D 10 Interoperability

Modules

- Direct3D 10 Interoperability [DEPRECATED]

Enumerations

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

Functions

- cudaError_t cudaD3D10GetDevice (int *device, DXGIAdapter *pAdapter)  
  Gets the device number for an adapter.

- cudaError_t cudaD3D10GetDevices (unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, ID3D10Device *pD3D10Device, enum cudaD3D10DeviceList deviceList)  
  Gets the CUDA devices corresponding to a Direct3D 10 device.

- cudaError_t cudaD3D10GetDirect3DDevice (ID3D10Device **pD3D10Device)  
  Gets the Direct3D device against which the current CUDA context was created.

- cudaError_t cudaD3D10SetDirect3DDevice (ID3D10Device *pD3D10Device, int device=-1)  
  Sets the Direct3D 10 device to use for interoperability with a CUDA device.

  Register a Direct3D 10 resource for access by CUDA.

4.13.1 Detailed Description

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

4.13.2 Enumeration Type Documentation

4.13.2.1 enum cudaD3D10DeviceList

CUDA devices corresponding to a D3D10 device

Enumerator:

- cudaD3D10DeviceListAll  The CUDA devices for all GPUs used by a D3D10 device
- cudaD3D10DeviceListCurrentFrame  The CUDA devices for the GPUs used by a D3D10 device in its currently rendering frame
4.13.3 Function Documentation

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

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

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

Returns:
cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown

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

See also:
cudaD3D10SetDirect3DDevice, cudaGraphicsD3D10RegisterResource,

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

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

If any of the GPUs being used to render pDevice are not CUDA capable then the call will return cudaErrorNoDevice.

Parameters:
- pCudaDeviceCount - Returned number of CUDA devices corresponding to pD3D10Device
- pCudaDevices - Returned CUDA devices corresponding to pD3D10Device
- cudaDeviceCount - The size of the output device array pCudaDevices
- pD3D10Device - Direct3D 10 device to query for CUDA devices
- deviceList - The set of devices to return. This set may be cudaD3D10DeviceListAll for all devices, cudaD3D10DeviceListCurrentFrame for the devices used to render the current frame (in SLI), or cudaD3D10DeviceListNextFrame for the devices used to render the next frame (in SLI).

Returns:
cudaSuccess, cudaErrorNoDevice, cudaErrorUnknown

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

    cudaGraphicsResourceGetMappedPointer

4.13.3.3 cudaError_t cudaD3D10GetDirect3DDevice (ID3D10Device ** ppD3D10Device)

Returns in *ppD3D10Device the Direct3D device against which this CUDA context was created in cudaD3D10SetDirect3DDevice().

Parameters:

    ppD3D10Device - Returns the Direct3D device for this thread

Returns:

    cudaSuccess, cudaErrorUnknown

Note:

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

See also:

    cudaD3D10SetDirect3DDevice

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

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

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

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

Parameters:

    pD3D10Device - Direct3D device to use for interoperability
    device - The CUDA device to use. This device must be among the devices returned when querying cudaD3D10DeviceListAll from cudaD3D10GetDevices, may be set to -1 to automatically select an appropriate CUDA device.

Returns:

    cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorSetOnActiveProcess

Note:

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

See also:

    cudaD3D10GetDevice, cudaGraphicsD3D10RegisterResource, cudaDeviceReset
4.13 Direct3D 10 Interoperability

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

Registers the Direct3D 10 resource `pD3DResource` for access by CUDA. If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through `cudaGraphicsUnregisterResource()`. Also on success, this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through `cudaGraphicsUnregisterResource()`.

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

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

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

The `flags` argument may be used to specify additional parameters at register time. The valid values for this parameter are

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

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

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

If Direct3D interoperability is not initialized using `cudaD3D10SetDirect3DDevice` then `cudaErrorInvalidDevice` is returned. If `pD3DResource` is of incorrect type or is already registered, then `cudaErrorInvalidResourceHandle` is returned. If `pD3DResource` cannot be registered, then `cudaErrorUnknown` is returned.

Parameters:

- `resource` - Pointer to returned resource handle
- `pD3DResource` - Direct3D resource to register
- `flags` - Parameters for resource registration

Returns:

- `cudaSuccess`, `cudaErrorInvalidDevice`, `cudaErrorInvalidValue`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`
Note:

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

See also:

4.14 Direct3D 11 Interoperability

Enumerations

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

Functions

- cudaError_t cudaD3D11GetDevice (int *device, IDXGIAdapter *pAdapter)
  Gets the device number for an adapter.

- cudaError_t cudaD3D11GetDevices (unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, ID3D11Device *pD3D11Device, enum cudaD3D11DeviceList deviceList)
  Gets the CUDA devices corresponding to a Direct3D 11 device.

- cudaError_t cudaD3D11GetDirect3DDevice (ID3D11Device **pD3D11Device)
  Gets the Direct3D device against which the current CUDA context was created.

- cudaError_t cudaD3D11SetDirect3DDevice (ID3D11Device *pD3D11Device, int device=-1)
  Sets the Direct3D 11 device to use for interoperability with a CUDA device.

  Register a Direct3D 11 resource for access by CUDA.

4.14.1 Detailed Description

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

4.14.2 Enumeration Type Documentation

4.14.2.1 enum cudaD3D11DeviceList

CUDA devices corresponding to a D3D11 device

Enumerator:

- cudaD3D11DeviceListAll  The CUDA devices for all GPUs used by a D3D11 device
- cudaD3D11DeviceListCurrentFrame  The CUDA devices for the GPUs used by a D3D11 device in its currently rendering frame
- cudaD3D11DeviceListNextFrame  The CUDA devices for the GPUs to be used by a D3D11 device in the next frame

Generated for NVIDIA CUDA Library by Doxygen
4.14.3 Function Documentation

4.14.3.1 cudaError_t cudaD3D11GetDevice (int * device, IDXGIAdapter * pAdapter)

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

Parameters:

   device - Returns the device corresponding to pAdapter
   pAdapter - D3D11 adapter to get device for

Returns:

   cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown

Note:

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

See also:

cudaGraphicsResourceGetMappedPointer

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

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

If any of the GPUs being used to render pDevice are not CUDA capable then the call will return cudaErrorNoDevice.

Parameters:

   pCudaDeviceCount - Returned number of CUDA devices corresponding to pD3D11Device
   pCudaDevices - Returned CUDA devices corresponding to pD3D11Device
   cudaDeviceCount - The size of the output device array pCudaDevices
   pD3D11Device - Direct3D 11 device to query for CUDA devices
   deviceList - The set of devices to return. This set may be cudaD3D11DeviceListAll for all devices, cu-
daD3D11DeviceListCurrentFrame for the devices used to render the current frame (in SLI), or cu-
daD3D11DeviceListNextFrame for the devices used to render the next frame (in SLI).

Returns:

   cudaSuccess, cudaErrorNoDevice, cudaErrorUnknown

Note:

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

See also:

cudaGraphicsResourceGetMappedPointer
4.14.3.3 cudaError_t cudaD3D11Get DIRECT3DDevice (ID3D11Device ** ppD3D11Device)

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

Parameters:

   ppD3D11Device - Returns the Direct3D device for this thread

Returns:

cudaSuccess, cudaErrorUnknown

Note:

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

See also:

cudaD3D11SetDirect3DDevice

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

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

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

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

Parameters:

   pD3D11Device - Direct3D device to use for interoperability

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

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorSetOnActiveProcess

Note:

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

See also:

4.14.3.5  cudaError_t cudaGraphicsD3D11RegisterResource (struct cudaGraphicsResource ** resource,  
            ID3D11Resource * pD3DResource, unsigned int flags)

Registers the Direct3D 11 resource pD3DResource for access by CUDA.

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

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

The type of pD3DResource must be one of the following.

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

The flags argument may be used to specify additional parameters at register time. The valid values for this parameter  
are

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

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

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

If Direct3D interoperability is not initialized using cudaD3D11SetDirect3DDevice then cudaErrorInvalidDevice is  
returned. If pD3DResource is of incorrect type or is already registered, then cudaErrorInvalidResourceHandle is  
returned. If pD3DResource cannot be registered, then cudaErrorUnknown is returned.

Parameters:

- resource - Pointer to returned resource handle
- pD3DResource - Direct3D resource to register
- flags - Parameters for resource registration

Returns:

- cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnk-  
  nown
4.14 Direct3D 11 Interoperability

Note:

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

See also:

4.15 VDPAU Interoperability

Functions

- **cudaError_t cudaGraphicsVDPAURegisterOutputSurface** (struct cudaGraphicsResource **resource, VdpOutputSurface vdpSurface, unsigned int flags)
  
  Register a VdpOutputSurface object.

- **cudaError_t cudaGraphicsVDPAURegisterVideoSurface** (struct cudaGraphicsResource **resource, VdpVideoSurface vdpSurface, unsigned int flags)

  Register a VdpVideoSurface object.

- **cudaError_t cudaVDPAUGetDevice** (int *device, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)

  Gets the CUDA device associated with a VdpDevice.

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

  Sets a CUDA device to use VDPAU interoperability.

4.15.1 Detailed Description

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

4.15.2 Function Documentation

4.15.2.1 **cudaError_t cudaGraphicsVDPAURegisterOutputSurface** (struct cudaGraphicsResource **resource, VdpOutputSurface vdpSurface, unsigned int flags)

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

- **cudaGraphicsMapFlagsNone**: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.

- **cudaGraphicsMapFlagsReadOnly**: Specifies that CUDA will not write to this resource.

- **cudaGraphicsMapFlagsWriteDiscard**: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

Parameters:

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

Returns:

- cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown
Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:

4.15.2.2 cudaError_t cudaGraphicsVDPAURegisterVideoSurface (struct cudaGraphicsResource ** resource, VdpVideoSurface vdpSurface, unsigned int flags)

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

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

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

Returns:
cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

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

See also:

4.15.2.3 cudaError_t cudaVDPAUGetDevice (int * device, VdpDevice vdpDevice, VdpGetProcAddress * vdpGetProcAddress)

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

Parameters:
device - Returns the device associated with vdpDevice, or -1 if the device associated with vdpDevice is not a compute device.
vdpDevice - A VdpDevice handle
vdpGetProcAddress - VDPAU’s VdpGetProcAddress function pointer
Returns:

    cudaSuccess

Note:

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

See also:

    cudaVDPAUSetVDPAUDevice

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

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

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

Parameters:

    device - Device to use for VDPAU interoperability
    vdpDevice - The VdpDevice to interoperate with
    vdpGetProcAddress - VDPAU’s VdpGetProcAddress function pointer

Returns:

    cudaSuccess, cudaErrorInvalidDevice, cudaErrorSetOnActiveProcess

Note:

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

See also:

    cudaGraphicsVDPAURegisterVideoSurface, cudaGraphicsVDPAURegisterOutputSurface, cudaDeviceReset
4.16 Graphics Interoperability

Functions

- `cudaError_t cudaGraphicsMapResources(int count, cudaGraphicsResource_t *resources, cudaStream_t stream=0)`
  Map graphics resources for access by CUDA.

- `cudaError_t cudaGraphicsResourceGetMappedPointer(void **devPtr, size_t *size, cudaGraphicsResource_t resource)`
  Get an device pointer through which to access a mapped graphics resource.

- `cudaError_t cudaGraphicsResourceSetMapFlags(cudaGraphicsResource_t resource, unsigned int flags)`
  Set usage flags for mapping a graphics resource.

- `cudaError_t cudaGraphicsSubResourceGetMappedArray(struct cudaArray **array, cudaGraphicsResource_t resource, unsigned int arrayIndex, unsigned int mipLevel)`
  Get an array through which to access a subresource of a mapped graphics resource.

- `cudaError_t cudaGraphicsUnmapResources(int count, cudaGraphicsResource_t *resources, cudaStream_t stream=0)`
  Unmap graphics resources.

- `cudaError_t cudaGraphicsUnregisterResource(cudaGraphicsResource_t resource)`
  Unregisters a graphics resource for access by CUDA.

4.16.1 Detailed Description

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

4.16.2 Function Documentation

4.16.2.1 `cudaError_t cudaGraphicsMapResources(int count, cudaGraphicsResource_t *resources, cudaStream_t stream = 0)`

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

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

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

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

Parameters:

- `count` - Number of resources to map
- `resources` - Resources to map for CUDA
stream - Stream for synchronization

**Returns:**

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

**Note:**

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

**See also:**


### 4.16.2.2 cudaError_t cudaGraphicsResourceGetMappedPointer (void **devPtr, size_t *size, cudaGraphicsResource_t resource)

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

If resource is not a buffer then it cannot be accessed via a pointer and cudaErrorUnknown is returned. If resource is not mapped then cudaErrorUnknown is returned.

**Parameters:**

- devPtr - Returned pointer through which \( \text{resource} \) may be accessed
- size - Returned size of the buffer accessible starting at *devPtr
- resource - Mapped resource to access

**Returns:**

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

**Note:**

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

**See also:**

cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray

### 4.16.2.3 cudaError_t cudaGraphicsResourceSetMapFlags (cudaGraphicsResource_t resource, unsigned int flags)

Set flags for mapping the graphics resource \( \text{resource} \).

Changes to flags will take effect the next time \( \text{resource} \) is mapped. The flags argument may be any of the following:

- cudaGraphicsMapFlagsNone: Specifies no hints about how \( \text{resource} \) will be used. It is therefore assumed that CUDA may read from or write to \( \text{resource} \).
- cudaGraphicsMapFlagsReadOnly: Specifies that CUDA will not write to \( \text{resource} \).
• cudaGraphicsMapFlagsWriteDiscard: Specifies CUDA will not read from resource and will write over the entire contents of resource, so none of the data previously stored in resource will be preserved.

If resource is presently mapped for access by CUDA then cudaErrorUnknown is returned. If flags is not one of the above values then cudaErrorInvalidValue is returned.

Parameters:

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

Returns:
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown,

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

See also:
cudaGraphicsMapResources

4.16.2.4 cudaError_t cudaGraphicsSubResourceGetMappedArray (struct cudaArray **array, cudaGraphicsResource_t resource, unsigned int arrayIndex, unsigned int mipLevel)

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

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

Parameters:

array - Returned array through which a subresource of resource may be accessed
resource - Mapped resource to access
arrayIndex - Array index for array textures or cubemap face index as defined by cudaGraphicsCubeFace for cubemap textures for the subresource to access
mipLevel - Mipmap level for the subresource to access

Returns:
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

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

See also:
cudaGraphicsResourceGetMappedPointer
4.16.2.5 cudaError_t cudaGraphicsUnmapResources (int count, cudaGraphicsResource_t *resources, cudaStream_t stream = 0)

Unmaps the count graphics resources in resources.

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

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

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

Parameters:

- count - Number of resources to unmap
- resources - Resources to unmap
- stream - Stream for synchronization

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

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

See also:

cudaGraphicsMapResources

4.16.2.6 cudaError_t cudaGraphicsUnregisterResource (cudaGraphicsResource_t resource)

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

If resource is invalid then cudaErrorInvalidResourceHandle is returned.

Parameters:

- resource - Resource to unregister

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

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

See also:

4.17 Texture Reference Management

Functions

- **cudaError_t cudaBindTexture**(size_t *offset, const struct textureReference *texref, const void *devPtr, const struct cudaChannelFormatDesc *desc, size_t size=UINT_MAX)
  
  Binds a memory area to a texture.

- **cudaError_t cudaBindTexture2D**(size_t *offset, const struct textureReference *texref, const void *devPtr, const struct cudaChannelFormatDesc *desc, size_t width, size_t height, size_t pitch)
  
  Binds a 2D memory area to a texture.

- **cudaError_t cudaBindTextureToArray**(const struct textureReference *texref, const struct cudaArray *array, const struct cudaChannelFormatDesc *desc)
  
  Binds an array to a texture.

- **struct cudaChannelFormatDesc cudaCreateChannelDesc**(int x, int y, int z, int w, enum cudaChannelFormatKind f)
  
  Returns a channel descriptor using the specified format.

- **cudaError_t cudaGetChannelDesc**(struct cudaChannelFormatDesc *desc, const struct cudaArray *array)
  
  Get the channel descriptor of an array.

- **cudaError_t cudaGetTextureAlignmentOffset**(size_t *offset, const struct textureReference *texref)
  
  Get the alignment offset of a texture.

- **cudaError_t cudaGetTextureReference**(const struct textureReference **texref, const char *symbol)
  
  Get the texture reference associated with a symbol.

- **cudaError_t cudaUnbindTexture**(const struct textureReference *texref)
  
  Unbinds a texture.

4.17.1 Detailed Description

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

4.17.2 Function Documentation

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

Binds size bytes of the memory area pointed to by devPtr to the texture reference texref. desc describes how the memory is interpreted when fetching values from the texture. Any memory previously bound to texref is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, cudaBindTexture() returns in *offset a byte offset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the
tex1Dfetch() function. If the device memory pointer was returned from cudaMemcpy(), the offset is guaranteed to be 0 and NULL may be passed as the offset parameter.

Parameters:

offset - Offset in bytes
texref - Texture to bind
devPtr - Memory area on device
desc - Channel format
size - Size of the memory area pointed to by devPtr

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

Note:

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

See also:

cudaCreateChannelDesc (C API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C++ API), 
cudaBindTexture2D (C API), cudaBindTextureToArray (C API), cudaUnbindTexture (C API), cudaGetTexture- 
AlignmentOffset (C API)

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

Binds the 2D memory area pointed to by devPtr to the texture reference texref. The size of the area is constrained by width in texel units, height in texel units, and pitch in byte units. desc describes how the memory is interpreted when fetching values from the texture. Any memory previously bound to texref is unbound.

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

Parameters:

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

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture
4.17 Texture Reference Management

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

See also:
cudaCreateChannelDesc (C API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C API),
cudaBindTexture2D (C++ API), cudaBindTexture2D (C++ API, inherited channel descriptor), cudaBindTextureTo
ToArray (C API), cudaBindTextureToArray (C API), cudaGetTextureAlignmentOffset (C API)

4.17.2.3 cudaError_t cudaBindTextureToArray (const struct textureReference *texref, const struct
cudaArray *array, const struct cudaChannelFormatDesc *desc)

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

Parameters:
texref - Texture to bind
array - Memory array on device
desc - Channel format

Returns:
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

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

See also:
cudaCreateChannelDesc (C API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C API),
cudaBindTexture2D (C API), cudaBindTextureToArray (C++ API), cudaUnbindTexture (C API), cudaGetTexture
AlignmentOffset (C API)

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

Returns a channel descriptor with format f and number of bits of each component x, y, z, and w. The cudaChan
nelFormatDesc is defined as:

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

where cudaChannelFormatKind is one of cudaChannelFormatKindSigned, cudaChannelFormatKindUnsigned, or cu
daChannelFormatKindFloat.

Parameters:
x - X component
y - Y component
$z$ - Z component

$w$ - W component

$f$ - Channel format

**Returns:**

Channel descriptor with format $f$

**See also:**

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

4.17.2.5  

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

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

**Parameters:**

*desc* - Channel format

*array* - Memory array on device

**Returns:**

cudaSuccess, cudaErrorInvalidValue

**Note:**

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

**See also:**

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

4.17.2.6  

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

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

**Parameters:**

*offset* - Offset of texture reference in bytes

*texref* - Texture to get offset of

**Returns:**

cudaSuccess, cudaErrorInvalidTexture, cudaErrorInvalidTextureBinding

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.
4.17 Texture Reference Management

See also:

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

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

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

Parameters:

texref - Texture associated with symbol

symbol - Symbol to find texture reference for

Returns:

cudaSuccess, cudaErrorInvalidTexture

Note:

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

See also:

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

4.17.2.8 cudaError_t cudaUnbindTexture (const struct textureReference *texref)

Unbinds the texture bound to texref.

Parameters:

texref - Texture to unbind

Returns:

cudaSuccess

Note:

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

See also:

cudaCreateChannelDesc (C API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C API),
cudaBindTexture2D (C API), cudaBindTextureToArray (C API), cudaUnbindTexture (C++ API), cudaGetTextureAlignmentOffset (C API)
4.18 Surface Reference Management

Functions

- `cudaError_t cudaBindSurfaceToArray (const struct surfaceReference *surfref, const struct cudaArray *array, const struct cudaChannelFormatDesc *desc)`
  Binds an array to a surface.

- `cudaError_t cudaGetSurfaceReference (const struct surfaceReference **surfref, const char *symbol)`
  Get the surface reference associated with a symbol.

4.18.1 Detailed Description

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

4.18.2 Function Documentation

4.18.2.1 `cudaError_t cudaBindSurfaceToArray (const struct surfaceReference *surfref, const struct cudaArray *array, const struct cudaChannelFormatDesc *desc)`

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

Parameters:

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

Returns:

- `cudaSuccess`, `cudaErrorInvalidValue`, `cudaErrorInvalidSurface`

Note:

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

See also:

- `cudaBindSurfaceToArray (C++ API)`, `cudaBindSurfaceToArray (C++ API, inherited channel descriptor)`, `cudaGetSurfaceReference`

4.18.2.2 `cudaError_t cudaGetSurfaceReference (const struct surfaceReference **surfref, const char *symbol)`

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

Parameters:

- `surfref` - Surface associated with symbol
symbol  - Symbol to find surface reference for

Returns:

cudaSuccess, cudaErrorInvalidSurface

Note:

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

See also:

cudaBindSurfaceToArray (C API)
4.19 Version Management

Functions

- **cudaError_t cudaDriverGetVersion (int \*driverVersion)**
  
  *Returns the CUDA driver version.*

- **cudaError_t cudaRuntimeGetVersion (int \*runtimeVersion)**
  
  *Returns the CUDA Runtime version.*

4.19.1 Function Documentation

4.19.1.1 cudaError_t cudaDriverGetVersion (int \*driverVersion)

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

Parameters:

  driverVersion - Returns the CUDA driver version.

Returns:

  cudaSuccess, cudaErrorInvalidValue

Note:

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

See also:

  cudaRuntimeGetVersion

4.19.1.2 cudaError_t cudaRuntimeGetVersion (int \*runtimeVersion)

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

Parameters:

  runtimeVersion - Returns the CUDA Runtime version.

Returns:

  cudaSuccess, cudaErrorInvalidValue

See also:

  cudaDriverGetVersion
4.20 C++ API Routines

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

Functions

• template<class T , int dim>
  cudaError_t cudaBindSurfaceToArray (const struct surface< T, dim > &surf, const struct cudaArray *array)
    [C++ API] Binds an array to a surface

• template<class T , int dim>
  cudaError_t cudaBindSurfaceToArray (const struct surface< T, dim > &surf, const struct cudaArray *array, const struct cudaChannelFormatDesc &desc)
    [C++ API] Binds an array to a surface

• template<class T , int dim, enum cudaTextureReadMode readMode>
  cudaError_t cudaBindTexture (size_t *offset, const struct texture< T, dim, readMode > &tex, const void *devPtr, size_t size=UINT_MAX)
    [C++ API] Binds a memory area to a texture

• template<class T , int dim, enum cudaTextureReadMode readMode>
  cudaError_t cudaBindTexture (size_t *offset, const struct texture< T, dim, readMode > &tex, const void *devPtr, const struct cudaChannelFormatDesc &desc, size_t size=UINT_MAX)
    [C++ API] Binds a memory area to a texture

• template<class T , int dim, enum cudaTextureReadMode readMode>
  cudaError_t cudaBindTexture2D (size_t *offset, const struct texture< T, dim, readMode > &tex, const void *devPtr, size_t width, size_t height, size_t pitch)
    [C++ API] Binds a 2D memory area to a texture

• template<class T , int dim, enum cudaTextureReadMode readMode>
  cudaError_t cudaBindTexture2D (size_t *offset, const struct texture< T, dim, readMode > &tex, const void *devPtr, const struct cudaChannelFormatDesc &desc, size_t width, size_t height, size_t pitch)
    [C++ API] Binds a 2D memory area to a texture

• template<class T , int dim, enum cudaTextureReadMode readMode>
  cudaError_t cudaBindTextureToArray (const struct texture< T, dim, readMode > &tex, const struct cudaArray *array)
    [C++ API] Binds an array to a texture

• template<class T , int dim, enum cudaTextureReadMode readMode>
  cudaError_t cudaBindTextureToArray (const struct texture< T, dim, readMode > &tex, const struct cudaArray *array, const struct cudaChannelFormatDesc &desc)
    [C++ API] Binds an array to a texture

• template<class T >
  cudaChannelFormatDesc cudaCreateChannelDesc (void)
    [C++ API] Returns a channel descriptor using the specified format

• cudaError_t cudaEventCreate (cudaEvent_t *event, unsigned int flags)
  [C++ API] Creates an event object with the specified flags
• template<class T>
cudaError_t cudaFuncGetAttributes (struct cudaFuncAttributes *attr, T *entry)

[C++ API] Find out attributes for a given function

• template<class T>
cudaError_t cudaFuncSetCacheConfig (T *func, enum cudaFuncCache cacheConfig)

Sets the preferred cache configuration for a device function.

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

[C++ API] Finds the address associated with a CUDA symbol

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

[C++ API] Finds the size of the object associated with a CUDA symbol

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

[C++ API] Get the alignment offset of a texture

• template<class T>
cudaError_t cudaLaunch (T *entry)

[C++ API] Launches a device function

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

[C++ API] Allocates page-locked memory on the host

• template<class T>
cudaError_t cudaSetupArgument (T arg, size_t offset)

[C++ API] Configure a device launch

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

[C++ API] Unbinds a texture

4.20.1 Detailed Description

This section describes the C++ high level API functions of the CUDA runtime application programming interface. To use these functions, your application needs to be compiled with the nvcc compiler.

4.20.2 Function Documentation

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

Binds the CUDA array array to the surface reference surf. The channel descriptor is inherited from the CUDA array. Any CUDA array previously bound to surf is unbound.
4.20 C++ API Routines

Parameters:

    surf - Surface to bind
    array - Memory array on device

Returns:

    cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSurface

Note:

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

See also:

    cudaMemcpy (C API), cudaMemcpy (C++ API)

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

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

Parameters:

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

Returns:

    cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSurface

Note:

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

See also:

    cudaMemcpy (C API), cudaMemcpy (C++ API, inherited channel descriptor)

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

Binds size bytes of the memory area pointed to by devPtr to texture reference tex. The channel descriptor is inherited from the texture reference type. The offset parameter is an optional byte offset as with the low-level cudaMemcpy(size_t*, const struct textureReference*, const void*, const struct cudaChannelFormatDesc*, size_t) function. Any memory previously bound to tex is unbound.

Parameters:

    offset - Offset in bytes
    tex - Texture to bind
devPtr - Memory area on device
size - Size of the memory area pointed to by devPtr

Returns:
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

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

See also:
cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C API),
cudaBindTexture (C++ API), cudaBindTexture2D (C++ API), cudaBindTexture2D (C++ API, inherited channel descriptor),
cudaBindTextureToArray (C++ API), cudaBindTextureToArray (C++ API, inherited channel descriptor),
cudaUnbindTexture (C++ API), cudaGetTextureAlignmentOffset (C++ API)

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

Binds size bytes of the memory area pointed to by devPtr to texture reference tex. desc describes how the memory is interpreted when fetching values from the texture. The offset parameter is an optional byte offset as with the low-level cudaBindTexture() function. Any memory previously bound to tex is unbound.

Parameters:
offset - Offset in bytes
tex - Texture to bind
devPtr - Memory area on device
desc - Channel format
size - Size of the memory area pointed to by devPtr

Returns:
cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

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

See also:
cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C API),
cudaBindTexture (C++ API, inherited channel descriptor), cudaBindTexture2D (C++ API), cudaBindTexture2D (C++ API, inherited channel descriptor),
cudaBindTextureToArray (C++ API), cudaBindTextureToArray (C++ API, inherited channel descriptor),
cudaUnbindTexture (C++ API), cudaGetTextureAlignmentOffset (C++ API)
4.20.2.5  template<class T, int dim, enum cudaTextureReadMode readMode> cudaError_t
cudaBindTexture2D (size_t * offset, const struct texture<T, dim, readMode> & tex, const void *
devPtr, size_t width, size_t height, size_t * pitch)

Binds the 2D memory area pointed to by devPtr to the texture reference tex. The size of the area is constrained by width in texel units, height in texel units, and pitch in byte units. The channel descriptor is inherited from the texture reference type. Any memory previously bound to tex is unbound.

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

Parameters:

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

Returns:

cudaSuccess, cudaMemcpyInvalidate, cudaMemcpyInvalidDevicePointer, cudaMemcpyInvalidTexture

Note:

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

See also:

cudaCreateChannelDesc (C++ API), cudaMemcpyChannelDesc, cudaMemcpyTextureReference, cudaMemcpyTexture (C++ API), cudaMemcpyTexture (C++ API, inherited channel descriptor), cudaMemcpyTexture2D (C API), cudaMemcpyTexture2D (C++ API), cudaMemcpyToArray (C++ API), cudaMemcpyToArray (C++ API, inherited channel descriptor), cudaMemcpyUnbindTexture (C++ API), cudaMemcpyTextureAlignmentOffset (C++ API)

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

Binds the 2D memory area pointed to by devPtr to the texture reference tex. The size of the area is constrained by width in texel units, height in texel units, and pitch in byte units. desc describes how the memory is interpreted when fetching values from the texture. Any memory previously bound to tex is unbound.

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

Parameters:

- offset - Offset in bytes
tex - Texture reference to bind

devPtr - 2D memory area on device

desc - Channel format

width - Width in texel units

height - Height in texel units

pitch - Pitch in bytes

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

Note:

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

See also:

cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C++ API), cudaBindTexture (C++ API, inherited channel descriptor), cudaBindTexture2D (C API), cudaBindTexture2D (C++ API, inherited channel descriptor), cudaBindTextureToArray (C++ API), cudaBindTextureToArray (C++ API, inherited channel descriptor), cudaUnbindTexture (C++ API), cudaGetTextureAlignmentOffset (C++ API)

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

Binds the CUDA array array to the texture reference tex. The channel descriptor is inherited from the CUDA array. Any CUDA array previously bound to tex is unbound.

Parameters:

tex - Texture to bind

array - Memory array on device

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

Note:

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

See also:

cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C++ API), cudaBindTexture (C++ API, inherited channel descriptor), cudaBindTexture2D (C API), cudaBindTexture2D (C++ API, inherited channel descriptor), cudaBindTextureToArray (C API), cudaBindTextureToArray (C++ API), cudaUnbindTexture (C++ API), cudaGetTextureAlignmentOffset (C++ API)
4.20.2.8 template<class T, int dim, enum cudaTextureReadMode readMode> cudaError_t
cudaBindTextureToArray (const struct texture<T, dim, readMode> & tex,
const struct cudaArray * array, const struct cudaChannelFormatDesc & desc)

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

Parameters:

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

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

Note:

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

See also:

cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C++ API),
cudaBindTexture (C++ API, inherited channel descriptor), cudaBindTexture2D (C++ API),
cudaBindTexture2D (C++ API, inherited channel descriptor),
cudaBindTextureToArray (C API),
cudaBindTextureToArray (C++ API, inherited channel descriptor),
cudaUnbindTexture (C++ API),
cudaGetTextureAlignmentOffset (C++ API)

4.20.2.9 template<class T> cudaChannelFormatDesc cudaCreateChannelDesc (void)

Returns a channel descriptor with format $f$ and number of bits of each component $x$, $y$, $z$, and $w$. The cudaChannelFormatDesc is defined as:

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

where cudaChannelFormatKind is one of cudaChannelFormatKindSigned, cudaChannelFormatKindUnsigned, or cudaChannelFormatKindFloat.

Returns:

Channel descriptor with format $f$

See also:

cudaCreateChannelDesc (Low level),
cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (High level),
cudaBindTexture (High level, inherited channel descriptor),
cudaBindTexture2D (High level),
cudaBindTexture2D (High level, inherited channel descriptor),
cudaBindTextureToArray (High level),
cudaBindTextureToArray (High level, inherited channel descriptor),
cudaUnbindTexture (High level),
cudaGetTextureAlignmentOffset (High level)
4.20.2.10  cudaError_t cudaEventCreate (cudaEvent_t * event, unsigned int flags)

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

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

Parameters:

- event - Newly created event
- flags - Flags for new event

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorLaunchFailure, cudaErrorMemoryAllocation

Note:

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

See also:

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

4.20.2.11  template<class T> cudaError_t cudaFuncGetAttributes (struct cudaFuncAttributes * attr, T * entry)

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

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

Parameters:

- attr - Return pointer to function’s attributes
- entry - Function to get attributes of

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidDeviceFunction

Note:

Note that this function may also return error codes from previous, asynchronous launches.
4.20 C++ API Routines

See also:

cudaConfigureCall, cudaMemcpy, cudaMemcpy2d, cudaMemcpy3d (C API), cudaMemcpy3dAsync (C API), cudaMemcpy3dBlocking (C API), cudaMemcpy3dStart (C API), cudaMemcpy3dStop (C API), cudaMemcpy3dComplete (C API), cudaMemcpy2d, cudaMemcpy2dAsync, cudaMemcpy2dBlocking, cudaMemcpy2dStart, cudaMemcpy2dStop

4.20.2.12 template<class T> cudaError_t cudaFuncSetCacheConfig (T *func, enum cudaFuncCache cacheConfig)

On devices where the L1 cache and shared memory use the same hardware resources, this sets through cacheConfig the preferred cache configuration for the function specified via func. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute func. func can either be a pointer to a function that executes on the device, or it can be a character string specifying the fully-decorated (C++) name for a function that executes on the device. The parameter specified by func must be declared as a __global__ function. If the specified function does not exist, then cudaErrorInvalidDeviceFunction is returned.

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

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

The supported cache configurations are:

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

Parameters:

func - Char string naming device function
cacheConfig - Requested cache configuration

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidDeviceFunction

Note:

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

See also:

cudaConfigureCall, cudaMemcpy, cudaMemcpy2d, cudaMemcpy3d (C API), cudaMemcpy3dAsync (C API), cudaMemcpy3dBlocking (C API), cudaMemcpy3dStart (C API), cudaMemcpy3dStop (C API), cudaMemcpy2d, cudaMemcpy2dAsync, cudaMemcpy2dBlocking, cudaMemcpy2dStart, cudaMemcpy2dStop

4.20.2.13 template<class T> cudaError_t cudaMemcpy (void **devPtr, const T &symbol)

Returns in *devPtr the address of symbol symbol on the device. symbol can either be a variable that resides in global or constant memory space, or it can be a character string, naming a variable that resides in global or constant memory space. If symbol cannot be found, or if symbol is not declared in the global or constant memory space, *devPtr is unchanged and the error cudaMemcpyInvalidSymbol is returned. If there are multiple global or constant variables with the same string name (from separate files) and the lookup is done via character string, cudaMemcpyDuplicateVariableName is returned.
Parameters:

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

Returns:

- `cudaSuccess`
- `cudaErrorInvalidSymbol`
- `cudaErrorDuplicateVariableName`

Note:

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

See also:

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

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

Returns in `*size` the size of symbol `symbol`. `symbol` can either be a variable that resides in global or constant memory space, or it can be a character string, naming a variable that resides in global or constant memory space. If `symbol` cannot be found, or if `symbol` is not declared in global or constant memory space, `*size` is unchanged and the error `cudaErrorInvalidSymbol` is returned. If there are multiple global variables with the same string name (from separate files) and the lookup is done via character string, `cudaErrorDuplicateVariableName` is returned.

Parameters:

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

Returns:

- `cudaSuccess`
- `cudaErrorInvalidSymbol`
- `cudaErrorDuplicateVariableName`

Note:

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

See also:

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

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

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

Parameters:

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

Returns:

- `cudaSuccess`
- `cudaErrorInvalidTexture`
- `cudaErrorInvalidTextureBinding`
4.20 C++ API Routines

Note:

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

See also:

cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C++ API), cudaBindTexture (C++ API, inherited channel descriptor), cudaBindTexture2D (C++ API), cudaBindTexture2D (C++ API, inherited channel descriptor), cudaBindTextureToArray (C++ API), cudaBindTextureToArray (C++ API, inherited channel descriptor), cudaUnbindTexture (C++ API), cudaGetTextureAlignmentOffset (C API)

4.20.2.16 template<class T> cudaError_t cudaLaunch (T * entry)

Launches the function entry on the device. The parameter entry can either be a function that executes on the device, or it can be a character string, naming a function that executes on the device. The parameter specified by entry must be declared as a __global__ function. cudaLaunch() must be preceded by a call to cudaConfigureCall() since it pops the data that was pushed by cudaConfigureCall() from the execution stack.

Parameters:

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

Returns:

cudaSuccess, cudaErrorInvalidDeviceFunction, cudaErrorInvalidConfiguration, cudaErrorLaunchFailure, cudaErrorLaunchTimeout, cudaErrorLaunchOutOfResouces, cudaErrorSharedObjectSymbolNotFound, cudaErrorSharedObjectInitFailed

Note:

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

See also:

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

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

Allocates size bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as cudaMemcpy().

Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as malloc(). Allocating excessive amounts of pinned memory may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to allocate staging areas for data exchange between host and device.

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

- cudaHostAllocDefault: This flag’s value is defined to be 0.
- cudaHostAllocPortable: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
• **cudaHostAllocMapped**: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling `cudaHostGetDevicePointer()`.

• **cudaHostAllocWriteCombined**: Allocates the memory as write-combined (WC). WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs. WC memory is a good option for buffers that will be written by the CPU and read by the device via mapped pinned memory or host->device transfers.

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

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

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

Memory allocated by this function must be freed with `cudaFreeHost()`.

**Parameters:**

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

**Returns:**

- `cudaSuccess`, `cudaErrorMemoryAllocation`

**Note:**

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

**See also:**

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

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

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

**Parameters:**

- `arg` - Argument to push for a kernel launch
- `offset` - Offset in argument stack to push new `arg`

**Returns:**

- `cudaSuccess`

**Note:**

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

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

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

Unbinds the texture bound to tex.

Parameters:

tex - Texture to unbind

Returns:

cudaSuccess

Note:

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

See also:

cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C++ API), cudaBindTexture (C++ API, inherited channel descriptor), cudaBindTexture2D (C++ API), cudaBindTexture2D (C++ API, inherited channel descriptor), cudaBindTextureToArray (C++ API), cudaBindTextureToArray (C++ API, inherited channel descriptor), cudaUnbindTexture (C API), cudaGetTextureAlignmentOffset (C++ API)
4.21 Interactions with the CUDA Driver API

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

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

4.21.1 Primary Contexts

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

4.21.2 Initialization and Tear-Down

CUDA Runtime API calls operate on the CUDA Driver API CUcontext which is current to the calling host thread. The function cudaSetDevice() makes the primary context for the specified device current to the calling thread by calling cuCtxSetCurrent().

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

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

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

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

4.21.3 Context Interoperability

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

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

The function cudaPointerGetAttributes() will return the error cudaErrorIncompatibleDriverContext if the pointer being queried was allocated by a non-primary context. The function cudaDeviceEnablePeerAccess() and the rest of the peer access API may not be called when a non-primary CUcontext is current. To use the pointer query and peer access APIs with a context created using the CUDA Driver API, it is necessary that the CUDA Driver API be used to access these features.
All CUDA Runtime API state (e.g., global variables’ addresses and values) travels with its underlying CUcontext. In particular, if a CUcontext is moved from one thread to another then all CUDA Runtime API state will move to that thread as well.

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

### 4.21.4 Interactions between CUstream and cudaStream_t

The types CUstream and cudaStream_t are identical and may be used interchangeably.

### 4.21.5 Interactions between CUevent and cudaEvent_t

The types CUevent and cudaEvent_t are identical and may be used interchangeably.

### 4.21.6 Interactions between CUarray and struct cudaArray *

The types CUarray and struct cudaArray * represent the same data type and may be used interchangeably by casting the two types between each other.

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

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

### 4.21.7 Interactions between CUgraphicsResource and cudaGraphicsResource_t

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

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

In order to use a cudaGraphicsResource_t in a CUDA Driver API function which takes a CUgraphicsResource, it is necessary to explicitly cast the cudaGraphicsResource_t to a CUgraphicsResource.
4.22 Direct3D 9 Interoperability [DEPRECATED]

Enumerations

- enum cudaD3D9MapFlags {
  cudaD3D9MapFlagsNone = 0,
  cudaD3D9MapFlagsReadOnly = 1,
  cudaD3D9MapFlagsWriteDiscard = 2
}

- enum cudaD3D9RegisterFlags {
  cudaD3D9RegisterFlagsNone = 0,
  cudaD3D9RegisterFlagsArray = 1
}

Functions

- cudaError_t cudaD3D9MapResources (int count, IDirect3DResource9 **ppResources)
  
  Map Direct3D resources for access by CUDA.

- cudaError_t cudaD3D9RegisterResource (IDirect3DResource9 *pResource, unsigned int flags)
  
  Registers a Direct3D resource for access by CUDA.

- cudaError_t cudaD3D9ResourceGetMappedArray (cudaArray **ppArray, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)
  
  Get an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

- cudaError_t cudaD3D9ResourceGetMappedPitch (size_t *pPitch, size_t *pPitchSlice, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)
  
  Get the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.

- cudaError_t cudaD3D9ResourceGetMappedPointer (void **pPointer, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)
  
  Get a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

- cudaError_t cudaD3D9ResourceGetMappedSize (size_t *pSize, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)
  
  Get the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.

- cudaError_t cudaD3D9ResourceGetSurfaceDimensions (size_t *pWidth, size_t *pHeight, size_t *pDepth, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)
  
  Get the dimensions of a registered Direct3D surface.

- cudaError_t cudaD3D9ResourceSetMapFlags (IDirect3DResource9 *pResource, unsigned int flags)
  
  Set usage flags for mapping a Direct3D resource.

- cudaError_t cudaD3D9UnmapResources (int count, IDirect3DResource9 **ppResources)
  
  Unmap Direct3D resources for access by CUDA.

- cudaError_t cudaD3D9UnregisterResource (IDirect3DResource9 *pResource)
  
  Unregisters a Direct3D resource for access by CUDA.
4.22 Direct3D 9 Interoperability [DEPRECATED]

### 4.22.1 Detailed Description

This section describes deprecated Direct3D 9 interoperability functions.

### 4.22.2 Enumeration Type Documentation

#### 4.22.2.1 enum cudaD3D9MapFlags

CUDA D3D9 Map Flags

**Enumerator:**

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

#### 4.22.2.2 enum cudaD3D9RegisterFlags

CUDA D3D9 Register Flags

**Enumerator:**

- `cudaD3D9RegisterFlagsNone` Default; Resource can be accessed through a void*
- `cudaD3D9RegisterFlagsArray` Resource can be accessed through a CUArray*

### 4.22.3 Function Documentation

#### 4.22.3.1 cudaError_t cudaD3D9MapResources (int count, IDirect3DResource9 ** ppResources)

**Deprecated**

This function is deprecated as of Cuda 3.0.

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

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

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

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

**Parameters:**

- `count` - Number of resources to map for CUDA
- `ppResources` - Resources to map for CUDA

**Returns:**

`cudaSuccess`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`
Note:

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

See also:

cudaGraphicsMapResources

4.22.3.2 cudaError_t cudaD3D9RegisterResource (IDirect3DResource9 * pResource, unsigned int flags)

Deprecated

This function is deprecated as of Cuda 3.0.

Registers the Direct3D resource pResource for access by CUDA.

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

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

The type of pResource must be one of the following.

- IDirect3DVertexBuffer9: No notes.
- IDirect3DIndexBuffer9: No notes.
- IDirect3DSurface9: Only stand-alone objects of type IDirect3DSurface9 may be explicitly shared. In particular, individual mipmap levels and faces of cube maps may not be registered directly. To access individual surfaces associated with a texture, one must register the base texture object.
- IDirect3DBaseTexture9: When a texture is registered, all surfaces associated with all mipmap levels of all faces of the texture will be accessible to CUDA.

The flags argument specifies the mechanism through which CUDA will access the Direct3D resource. The following value is allowed:

- cudaD3D9RegisterFlagsNone: Specifies that CUDA will access this resource through a void*. The pointer, size, and pitch for each subresource of this resource may be queried through cudaD3D9ResourceGetMappedPointer(), cudaD3D9ResourceGetMappedSize(), and cudaD3D9ResourceGetMappedPitch() respectively. This option is valid for all resource types.

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

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Any resources allocated in D3DPOOL_SYSTEMMEM or D3DPOOL_MANAGED may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.
4.22 Direct3D 9 Interoperability [DEPRECATED]

If Direct3D interoperability is not initialized on this context, then `cudaErrorInvalidDevice` is returned. If `pResource` is of incorrect type (e.g., is a non-stand-alone `IDirect3DSurface9`) or is already registered, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` cannot be registered then `cudaErrorUnknown` is returned.

Parameters:

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

Returns:

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

Note:

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

See also:

- `cudaGraphicsD3D9RegisterResource`

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

Deprecated

This function is deprecated as of Cuda 3.0.

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

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

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

Parameters:

- `ppArray` - Returned array corresponding to subresource
- `pResource` - Mapped resource to access
- `face` - Face of resource to access
- `level` - Level of resource to access

Returns:

- `cudaSuccess`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

Note:

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

See also:

- `cudaGraphicsSubResourceGetMappedArray`
4.22.3.4  cudaError_t cudaD3D9ResourceGetMappedPitch (size_t \* pPitch, size_t \* pPitchSlice, IDirect3DResource9 \* pResource, unsigned int face, unsigned int level)

Deprecated

This function is deprecated as of Cuda 3.0.

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

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

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

\[ y \times \text{pitch} + (\text{bytes per pixel}) \times x \]

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

\[ z \times \text{slicePitch} + y \times \text{pitch} + (\text{bytes per pixel}) \times x \]

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

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

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

Parameters:

- pPitch - Returned pitch of subresource
- pPitchSlice - Returned Z-slice pitch of subresource
- pResource - Mapped resource to access
- face - Face of resource to access
- level - Level of resource to access

Returns:

- cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

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

See also:

- cudaGraphicsResourceGetMappedPointer

4.22.3.5  cudaError_t cudaD3D9ResourceGetMappedPointer (void \*\* pPointer, IDirect3DResource9 \* pResource, unsigned int face, unsigned int level)

Deprecated

This function is deprecated as of Cuda 3.0.
Returns in *pPointer the base pointer of the subresource of the mapped Direct3D resource pResource, which corresponds to face and level. The value set in pPointer may change every time that pResource is mapped.

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

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

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

Parameters:

- **pPointer** - Returned pointer corresponding to subresource
- **pResource** - Mapped resource to access
- **face** - Face of resource to access
- **level** - Level of resource to access

Returns:

- cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

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

See also:

- cudaGraphicsResourceGetMappedPointer

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

Deprecated

This function is deprecated as of Cuda 3.0.

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

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

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

Parameters:

- **pSize** - Returned size of subresource
- **pResource** - Mapped resource to access
- **face** - Face of resource to access
- **level** - Level of resource to access
Returns:

    cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

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

See also:

    cudaGraphicsResourceGetMappedPointer

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

Deprecated

    This function is deprecated as of Cuda 3.0.

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

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

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

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

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

Parameters:

    pWidth - Returned width of surface
    pHeight - Returned height of surface
    pDepth - Returned depth of surface
    pResource - Registered resource to access
    face - Face of resource to access
    level - Level of resource to access

Returns:

    cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle,

Note:

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

See also:

    cudaGraphicsSubResourceGetMappedArray
4.22.3.8(cudaError_t cudaD3D9ResourceSetMapFlags (IDirect3DResource9 * pResource, unsigned int flags))

**Deprecated**

This function is deprecated as of Cuda 3.0.

Set flags for mapping the Direct3D resource `pResource`. Changes to flags will take effect the next time `pResource` is mapped. The `flags` argument may be any of the following:

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

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

**Parameters:**

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

**Returns:**

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

**Note:**

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

**See also:**

cudaInteropResourceSetMapFlags

4.22.3.9(cudaError_t cudaD3D9UnmapResources (int count, IDirect3DResource9 ** ppResources))

**Deprecated**

This function is deprecated as of Cuda 3.0.

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

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

- **count** - Number of resources to unmap for CUDA
- **ppResources** - Resources to unmap for CUDA

Returns:

- `cudaSuccess`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

Note:

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

See also:

- `cudaGraphicsUnmapResources`

### 4.22.3.10 cudaError_t cudaD3D9UnregisterResource (IDirect3DResource9 * pResource)

Deprecated

This function is deprecated as of Cuda 3.0.

Unregisters the Direct3D resource `pResource` so it is not accessible by CUDA unless registered again. If `pResource` is not registered, then `cudaErrorInvalidResourceHandle` is returned.

Parameters:

- **pResource** - Resource to unregister

Returns:

- `cudaSuccess`, `cudaErrorInvalidResourceHandle`, `cudaErrorUnknown`

Note:

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

See also:

- `cudaGraphicsUnregisterResource`
4.23 Direct3D 10 Interoperability [DEPRECATED]

Enumerations

- `enum cudaD3D10MapFlags {
  cudaD3D10MapFlagsNone = 0,
  cudaD3D10MapFlagsReadOnly = 1,
  cudaD3D10MapFlagsWriteDiscard = 2 }
- `enum cudaD3D10RegisterFlags {
  cudaD3D10RegisterFlagsNone = 0,
  cudaD3D10RegisterFlagsArray = 1 }

Functions

- `cudaError_t cudaD3D10MapResources (int count, ID3D10Resource **ppResources)
  Map Direct3D Resources for access by CUDA.

- `cudaError_t cudaD3D10RegisterResource (ID3D10Resource *pResource, unsigned int flags)
  Register a Direct3D 10 resource for access by CUDA.

- `cudaError_t cudaD3D10ResourceGetMappedArray (cudaArray **ppArray, ID3D10Resource *pResource, unsigned int subResource)
  Get an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

- `cudaError_t cudaD3D10ResourceGetMappedPitch (size_t *pPitch, size_t *pPitchSlice, ID3D10Resource *pResource, unsigned int subResource)
  Get the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.

  Get a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

- `cudaError_t cudaD3D10ResourceGetMappedSize (size_t *pSize, ID3D10Resource *pResource, unsigned int subResource)
  Get the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.

- `cudaError_t cudaD3D10ResourceGetSurfaceDimensions (size_t *pWidth, size_t *pHeight, size_t *pDepth, ID3D10Resource *pResource, unsigned int subResource)
  Get the dimensions of a registered Direct3D surface.

- `cudaError_t cudaD3D10ResourceSetMapFlags (ID3D10Resource *pResource, unsigned int flags)
  Set usage flags for mapping a Direct3D resource.

- `cudaError_t cudaD3D10UnmapResources (int count, ID3D10Resource **ppResources)
  Unmaps Direct3D resources.

- `cudaError_t cudaD3D10UnregisterResource (ID3D10Resource *pResource)
  Unregisters a Direct3D resource.
4.23.1 Detailed Description

This section describes deprecated Direct3D 10 interoperability functions.

4.23.2 Enumeration Type Documentation

4.23.2.1 enum cudaD3D10MapFlags

CUDA D3D10 Map Flags

Enumerator:

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

4.23.2.2 enum cudaD3D10RegisterFlags

CUDA D3D10 Register Flags

Enumerator:

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

4.23.3 Function Documentation

4.23.3.1 cudaError_t cudaD3D10MapResources (int count, ID3D10Resource **ppResources)

Deprecated

This function is deprecated as of Cuda 3.0.

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

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

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

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

Parameters:

- count - Number of resources to map for CUDA
- ppResources - Resources to map for CUDA

Returns:

- cudaSuccess
- cudaErrorInvalidResourceHandle
- cudaErrorUnknown
Note:

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

See also:

cudaGraphicsMapResources

### 4.23.3.2  
cudaError_t cudaD3D10RegisterResource (ID3D10Resource *pResource, unsigned int flags)

**Deprecated**

This function is deprecated as of Cuda 3.0.

Registers the Direct3D resource pResource for access by CUDA.

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

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

The type of pResource must be one of the following:

- **ID3D10Buffer**: Cannot be used with flags set to cudaD3D10RegisterFlagsArray.
- **ID3D10Texture1D**: No restrictions.
- **ID3D10Texture2D**: No restrictions.
- **ID3D10Texture3D**: No restrictions.

The flags argument specifies the mechanism through which CUDA will access the Direct3D resource. The following values are allowed.

- **cudaD3D10RegisterFlagsNone**: Specifies that CUDA will access this resource through a void*. The pointer, size, and pitch for each subresource of this resource may be queried through cudaD3D10ResourceGetMappedPointer(), cudaD3D10ResourceGetMappedSize(), and cudaD3D10ResourceGetMappedPitch() respectively. This option is valid for all resource types.

- **cudaD3D10RegisterFlagsArray**: Specifies that CUDA will access this resource through a CUArray queried on a sub-resource basis through cudaD3D10ResourceGetMappedArray(). This option is only valid for resources of type ID3D10Texture1D, ID3D10Texture2D, and ID3D10Texture3D.

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

- The primary render target may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.
If Direct3D interoperability is not initialized on this context then `cudaErrorInvalidDevice` is returned. If `pResource` is of incorrect type or is already registered then `cudaErrorInvalidResourceHandle` is returned. If `pResource` cannot be registered then `cudaErrorUnknown` is returned.

**Parameters:**

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

**Returns:**

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

**Note:**

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

**See also:**

`cudaGraphicsD3D10RegisterResource`

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

**Deprecated**

This function is deprecated as of Cuda 3.0.

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

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

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

**Parameters:**

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

**Returns:**

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

**Note:**

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

**See also:**

`cudaGraphicsSubResourceGetMappedArray`
4.23.3.4 \texttt{cudaError_t cudaD3D10ResourceGetMappedPitch (size_t \ast pPitch, size_t \ast pPitchSlice, ID3D10Resource \ast pResource, unsigned int subResource)}

\textbf{Deprecated}

This function is deprecated as of Cuda 3.0.

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

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

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

\[ y \ast \text{pitch} + (\text{bytes per pixel}) \ast x \]

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

\[ z \ast \text{slicePitch} + y \ast \text{pitch} + (\text{bytes per pixel}) \ast x \]

Both parameters \texttt{pPitch} and \texttt{pPitchSlice} are optional and may be set to NULL.

If \texttt{pResource} is not of type ID3D10Texture1D, ID3D10Texture2D, or ID3D10Texture3D, or if \texttt{pResource} has not been registered for use with CUDA, then \texttt{cudaErrorInvalidResourceHandle} is returned. If \texttt{pResource} was not registered with usage flags \texttt{cudaD3D10RegisterFlagsNone}, then \texttt{cudaErrorInvalidResourceHandle} is returned. If \texttt{pResource} is not mapped for access by CUDA then \texttt{cudaErrorUnknown} is returned.

For usage requirements of the \texttt{subResource} parameter see \texttt{cudaD3D10ResourceGetMappedPointer()}.  

\textbf{Parameters:}

- \texttt{pPitch} - Returned pitch of subresource
- \texttt{pPitchSlice} - Returned Z-slice pitch of subresource
- \texttt{pResource} - Mapped resource to access
- \texttt{subResource} - Subresource of \texttt{pResource} to access

\textbf{Returns:}

- \texttt{cudaSuccess}, \texttt{cudaErrorInvalidValue}, \texttt{cudaErrorInvalidResourceHandle}, \texttt{cudaErrorUnknown}

\textbf{Note:}

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

\textbf{See also:}

\texttt{cudaGraphicsSubResourceGetMappedArray}

4.23.3.5 \texttt{cudaError_t cudaD3D10ResourceGetMappedPointer (void **pPointer, ID3D10Resource \ast pResource, unsigned int subResource)}

\textbf{Deprecated}

This function is deprecated as of Cuda 3.0.

Returns in \*pPointer the base pointer of the subresource of the mapped Direct3D resource \texttt{pResource} which corresponds to \texttt{subResource}. The value set in \texttt{pPointer} may change every time that \texttt{pResource} is mapped.
If `pResource` is not registered, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` was not registered with usage flags `cudaD3D9RegisterFlagsNone`, then `cudaErrorInvalidResourceHandle` is returned. If `pResource` is not mapped then `cudaErrorUnknown` is returned.

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

**Parameters:**

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

**Returns:**

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

**Note:**

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

**See also:**

cudaGraphicsResourceGetMappedPointer

### 4.23.3.6 cudaError_t cudaD3D10ResourceGetMappedSize (size_t ∗ pSize, ID3D10Resource ∗ pResource, unsigned int subResource)

**Deprecated**

This function is deprecated as of Cuda 3.0.

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

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

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

**Parameters:**

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

**Returns:**

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

**Note:**

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

**See also:**

cudaGraphicsResourceGetMappedPointer
4.23 Direct3D 10 Interoperability [DEPRECATED]

4.23.3.7 cudaError_t cudaD3D10ResourceGetSurfaceDimensions (size_t *pWidth, size_t *pHeight, size_t *pDepth, ID3D10Resource *pResource, unsigned int subResource)

Deprecated

This function is deprecated as of Cuda 3.0.

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

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

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

If pResource is not of type ID3D10Texture1D, ID3D10Texture2D, or ID3D10Texture3D, or if pResource has not been registered for use with CUDA, then cudaErrorInvalidHandle is returned.

For usage requirements of subResource parameters see cudaD3D10ResourceGetMappedPointer().

Parameters:

- pWidth - Returned width of surface
- pHeight - Returned height of surface
- pDepth - Returned depth of surface
- pResource - Registered resource to access
- subResource - Subresource of pResource to access

Returns:

- cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle,

Note:

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

See also:

- cudaGraphicsSubResourceGetMappedArray

4.23.3.8 cudaError_t cudaD3D10ResourceSetMapFlags (ID3D10Resource *pResource, unsigned int flags)

Deprecated

This function is deprecated as of Cuda 3.0.

Set usage flags for mapping the Direct3D resource pResource.

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

- cudaD3D10MapFlagsNone: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- cudaD3D10MapFlagsReadOnly: Specifies that CUDA kernels which access this resource will not write to this resource.

Generated for NVIDIA CUDA Library by Doxygen
• cudaD3D10MapFlagsWriteDiscard: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

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

Parameters:

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

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown,

Note:

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

See also:

cudaGraphicsResourceSetMapFlags

4.23.3.9 cudaError_t cudaD3D10UnmapResources (int count, ID3D10Resource ** ppResources)

Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the count Direct3D resource in ppResources.

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

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

Parameters:

count - Number of resources to unmap for CUDA
ppResources - Resources to unmap for CUDA

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

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

See also:

cudaGraphicsUnmapResources
4.23 Direct3D 10 Interoperability [DEPRECATED] 147

4.23.3.10  cudaError_t cudaD3D10UnregisterResource (ID3D10Resource * pResource)

Deprecated

This function is deprecated as of Cuda 3.0.

Unregisters the Direct3D resource resource so it is not accessible by CUDA unless registered again. If pResource is not registered, then cudaErrorInvalidResourceHandle is returned.

Parameters:

pResource - Resource to unregister

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

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

See also:

cudaGraphicsUnregisterResource
## 4.24 OpenGL Interoperability [DEPRECATED]

### Enumerations

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

### Functions

- **cudaError_t cudaGLMapBufferObject (void **devPtr, GLuint bufObj)**
  
  Maps a buffer object for access by CUDA.

- **cudaError_t cudaGLMapBufferObjectAsync (void **devPtr, GLuint bufObj, cudaStream_t stream)**
  
  Maps a buffer object for access by CUDA.

- **cudaError_t cudaGLRegisterBufferObject (GLuint bufObj)**
  
  Registers a buffer object for access by CUDA.

- **cudaError_t cudaGLSetBufferObjectMapFlags (GLuint bufObj, unsigned int flags)**
  
  Set usage flags for mapping an OpenGL buffer.

- **cudaError_t cudaGLUnmapBufferObject (GLuint bufObj)**
  
  Unmaps a buffer object for access by CUDA.

- **cudaError_t cudaGLUnmapBufferObjectAsync (GLuint bufObj, cudaStream_t stream)**
  
  Unmaps a buffer object for access by CUDA.

- **cudaError_t cudaGLUnregisterBufferObject (GLuint bufObj)**
  
  Unregisters a buffer object for access by CUDA.

### 4.24.1 Detailed Description

This section describes deprecated OpenGL interoperability functionality.

### 4.24.2 Enumeration Type Documentation

#### 4.24.2.1 enum cudaGLMapFlags

CUDA GL Map Flags

**Enumerator:**

- **cudaGLMapFlagsNone** Default; Assume resource can be read/written
- **cudaGLMapFlagsReadOnly** CUDA kernels will not write to this resource
- **cudaGLMapFlagsWriteDiscard** CUDA kernels will only write to and will not read from this resource
4.24.3 Function Documentation

4.24.3.1 cudaError_t cudaGLMapBufferObject (void ** devPtr, GLuint bufObj)

Deprecated

This function is deprecated as of Cuda 3.0.

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

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

Parameters:
- **devPtr** - Returned device pointer to CUDA object
- **bufObj** - Buffer object ID to map

Returns:
- cudaSuccess, cudaErrorMapBufferObjectFailed

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

See also:
- cudaGraphicsMapResources

4.24.3.2 cudaError_t cudaGLMapBufferObjectAsync (void ** devPtr, GLuint bufObj, cudaStream_t stream)

Deprecated

This function is deprecated as of Cuda 3.0.

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

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

Parameters:
- **devPtr** - Returned device pointer to CUDA object
- **bufObj** - Buffer object ID to map
- **stream** - Stream to synchronize

Returns:
- cudaSuccess, cudaErrorMapBufferObjectFailed
Note:

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

See also:

cudaGraphicsMapResources

4.24.3.3 cudaError_t cudaGLRegisterBufferObject (GLuint bufObj)

Deprecated

This function is deprecated as of Cuda 3.0.

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

Parameters:

bufObj - Buffer object ID to register

Returns:

cudaSuccess, cudaErrorInitializationError

Note:

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

See also:

cudaGraphicsGLRegisterBuffer

4.24.3.4 cudaError_t cudaGLSetBufferObjectMapFlags (GLuint bufObj, unsigned int flags)

Deprecated

This function is deprecated as of Cuda 3.0.

Set flags for mapping the OpenGL buffer bufObj

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

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

If bufObj has not been registered for use with CUDA, then cudaErrorInvalidResourceHandle is returned. If bufObj is presently mapped for access by CUDA, then cudaErrorUnknown is returned.
Parameters:

bufObj - Registered buffer object to set flags for
flags - Parameters for buffer mapping

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

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

See also:

cudaGraphicsResourceSetMapFlags

4.24.3.5 cudaError_t cudaGLUnmapBufferObject (GLuint bufObj)

Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the buffer object of ID bufObj for access by CUDA. When a buffer is unmapped, the base address returned by cudaGLMapBufferObject() is invalid and subsequent references to the address result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

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

Parameters:

bufObj - Buffer object to unmap

Returns:

cudaSuccess, cudaErrorInvalidDevicePointer, cudaErrorUnmapBufferObjectFailed

Note:

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

See also:

cudaGraphicsUnmapResources

4.24.3.6 cudaError_t cudaGLUnmapBufferObjectAsync (GLuint bufObj, cudaStream_t stream)

Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the buffer object of ID bufObj for access by CUDA. When a buffer is unmapped, the base address returned by cudaGLMapBufferObject() is invalid and subsequent references to the address result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

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

bufObj - Buffer object to unmap

stream - Stream to synchronize

Returns:

cudaSuccess, cudaErrorInvalidDevicePointer, cudaErrorUnmapBufferObjectFailed

Note:

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

See also:

cudaGraphicsUnmapResources

4.24.3.7 cudaError_t cudaGLUnregisterBufferObject (GLuint bufObj)

Deprecated

This function is deprecated as of Cuda 3.0.

Unregisters the buffer object of ID bufObj for access by CUDA and releases any CUDA resources associated with the buffer. Once a buffer is unregistered, it may no longer be mapped by CUDA. The GL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

Parameters:

bufObj - Buffer object to unregister

Returns:

cudaSuccess

Note:

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

See also:

cudaGraphicsUnregisterResource
4.25 Data types used by CUDA Runtime

Data Structures

- struct cudaChannelFormatDesc
- struct cudaDeviceProp
- struct cudaExtent
- struct cudaFuncAttributes
- struct cudaMemcpy3DParms
- struct cudaMemcpy3DPeerParms
- struct cudaPitchedPtr
- struct cudaPointerAttributes
- struct cudaPos
- struct surfaceReference
- struct textureReference

Enumerations

- enum cudaSurfaceBoundaryMode {
  cudaBoundaryModeZero = 0,
  cudaBoundaryModeClamp = 1,
  cudaBoundaryModeTrap = 2
}
- enum cudaSurfaceFormatMode {
  cudaFormatModeForced = 0,
  cudaFormatModeAuto = 1
}
- enum cudaTextureAddressMode {
  cudaAddressModeWrap = 0,
  cudaAddressModeClamp = 1,
  cudaAddressModeMirror = 2,
  cudaAddressModeBorder = 3
}
- enum cudaTextureFilterMode {
  cudaFilterModePoint = 0,
  cudaFilterModeLinear = 1
}
- enum cudaTextureReadMode {
  cudaReadModeElementType = 0,
  cudaReadModeNormalizedFloat = 1
}
• enum cudaChannelFormatKind {
    cudaChannelFormatKindSigned = 0,
    cudaChannelFormatKindUnsigned = 1,
    cudaChannelFormatKindFloat = 2,
    cudaChannelFormatKindNone = 3
}
• enum cudaComputeMode {
    cudaComputeModeDefault = 0,
    cudaComputeModeExclusive = 1,
    cudaComputeModeProhibited = 2,
    cudaComputeModeExclusiveProcess = 3
}
• enum cudaError {
    cudaSuccess = 0,
    cudaErrorMissingConfiguration = 1,
    cudaErrorMemoryAllocation = 2,
    cudaErrorInitializationError = 3,
    cudaErrorLaunchFailure = 4,
    cudaErrorPriorLaunchFailure = 5,
    cudaErrorLaunchTimeout = 6,
    cudaErrorLaunchOutOfResources = 7,
    cudaErrorInvalidDeviceFunction = 8,
    cudaErrorInvalidConfiguration = 9,
    cudaErrorInvalidDevice = 10,
    cudaErrorInvalidValue = 11,
    cudaErrorInvalidPitchValue = 12,
    cudaErrorInvalidSymbol = 13,
    cudaErrorMapBufferObjectFailed = 14,
    cudaErrorUnmapBufferObjectFailed = 15,
    cudaErrorInvalidHostPointer = 16,
    cudaErrorInvalidDevicePointer = 17,
    cudaErrorInvalidTexture = 18,
    cudaErrorInvalidTextureBinding = 19,
    cudaErrorInvalidChannelDescriptor = 20,
    cudaErrorInvalidMemcpyDirection = 21,
    cudaErrorAddressOfConstant = 22,
    cudaErrorTextureFetchFailed = 23,
    cudaErrorTextureNotBound = 24,
    cudaErrorSynchronizationError = 25,
    cudaErrorInvalidFilterSetting = 26,
    cudaErrorInvalidNormSetting = 27,
    cudaErrorMixedDeviceExecution = 28,
    cudaErrorCudartUnloading = 29,
cudaErrorUnknown = 30,
cudaErrorNotYetImplemented = 31,
cudaErrorMemoryValueTooLarge = 32,
cudaErrorInvalidResourceHandle = 33,
cudaErrorNotReady = 34,
cudaErrorInsufficientDriver = 35,
cudaErrorSetOnActiveProcess = 36,
cudaErrorInvalidSurface = 37,
cudaErrorNoDevice = 38,
cudaErrorECCUncorrectable = 39,
cudaErrorSharedObjectSymbolNotFound = 40,
cudaErrorSharedObjectInitFailed = 41,
cudaErrorUnsupportedLimit = 42,
cudaErrorDuplicateVariableName = 43,
cudaErrorDuplicateTextureName = 44,
cudaErrorDuplicateSurfaceName = 45,
cudaErrorDevicesUnavailable = 46,
cudaErrorInvalidKernelImage = 47,
cudaErrorNoKernelImageForDevice = 48,
cudaErrorIncompatibleDriverContext = 49,
cudaErrorPeerAccessAlreadyEnabled = 50,
cudaErrorPeerAccessNotEnabled = 51,
cudaErrorDeviceAlreadyInUse = 54,
cudaErrorProfilerDisabled = 55,
cudaErrorProfilerNotInitialized = 56,
cudaErrorProfilerAlreadyStarted = 57,
cudaErrorProfilerAlreadyStopped = 58,
cudaErrorStartupFailure = 0x7f,
cudaErrorApiFailureBase = 10000 }
• enum cudaFuncCache {
  cudaFuncCachePreferNone = 0,
  cudaFuncCachePreferShared = 1,
  cudaFuncCachePreferL1 = 2 }
• enum cudaGraphicsCubeFace {
  cudaGraphicsCubeFacePositiveX = 0x00,
  cudaGraphicsCubeFaceNegativeX = 0x01,
  cudaGraphicsCubeFacePositiveY = 0x02,
  cudaGraphicsCubeFaceNegativeY = 0x03,
  cudaGraphicsCubeFacePositiveZ = 0x04,
  cudaGraphicsCubeFaceNegativeZ = 0x05 }
enum cudaGraphicsMapFlags {
    cudaGraphicsMapFlagsNone = 0,
    cudaGraphicsMapFlagsReadOnly = 1,
    cudaGraphicsMapFlagsWriteDiscard = 2
}

enum cudaGraphicsRegisterFlags {
    cudaGraphicsRegisterFlagsNone = 0,
    cudaGraphicsRegisterFlagsReadOnly = 1,
    cudaGraphicsRegisterFlagsWriteDiscard = 2,
    cudaGraphicsRegisterFlagsSurfaceLoadStore = 4
}

enum cudaLimit {
    cudaLimitStackSize = 0x00,
    cudaLimitPrintfFifoSize = 0x01,
    cudaLimitMallocHeapSize = 0x02
}

enum cudaMemcpyKind {
    cudaMemcpyHostToHost = 0,
    cudaMemcpyHostToDevice = 1,
    cudaMemcpyDeviceToHost = 2,
    cudaMemcpyDeviceToDevice = 3,
    cudaMemcpyDefault = 4
}

enum cudaMemcpyKind {
    cudaMemcpyHostToHost = 0,
    cudaMemcpyHostToDevice = 1,
    cudaMemcpyDeviceToHost = 2,
    cudaMemcpyDeviceToDevice = 3,
    cudaMemcpyDefault = 4
}

enum cudaMemcpyKind {
    cudaMemcpyHostToHost = 0,
    cudaMemcpyHostToDevice = 1,
    cudaMemcpyDeviceToHost = 2,
    cudaMemcpyDeviceToDevice = 3,
    cudaMemcpyDefault = 4
}

enum cudaMemcpyKind {
    cudaMemcpyHostToHost = 0,
    cudaMemcpyHostToDevice = 1,
    cudaMemcpyDeviceToHost = 2,
    cudaMemcpyDeviceToDevice = 3,
    cudaMemcpyDefault = 4
}

typedef enum cudaError cudaError_t
typedef struct CUevent_st * cudaEvent_t
typedef struct cudaGraphicsResource * cudaGraphicsResource_t
typedef enum cudaOutputMode cudaOutputMode_t
typedef struct CUstream_st * cudaStream_t
typedef struct CUuuid_st cudaUUID_t
#define cudaArrayDefault 0x00
#define cudaArrayLayered 0x01
#define cudaArraySurfaceLoadStore 0x02
#define cudaDeviceBlockingSync 0x04
#define cudaDeviceLmemResizeToMax 0x10
#define cudaDeviceMapHost 0x08
#define cudaDeviceMask 0x1f
#define cudaDevicePropDontCare
#define cudaDeviceScheduleAuto 0x00
#define cudaDeviceScheduleBlockingSync 0x04
#define cudaDeviceScheduleSpin 0x01
#define cudaDeviceScheduleYield 0x02
#define cudaEventBlockingSync 0x01
#define cudaEventDefault 0x00
#define cudaEventDisableTiming 0x02
#define cudaHostAllocDefault 0x00
#define cudaHostAllocMapped 0x02
4.25 Data types used by CUDA Runtime

- #define cudaHostAllocPortable 0x01
- #define cudaHostAllocWriteCombined 0x04
- #define cudaHostRegisterDefault 0x00
- #define cudaHostRegisterMapped 0x02
- #define cudaHostRegisterPortable 0x01
- #define cudaPeerAccessDefault 0x00

### 4.25.1 Define Documentation

#### 4.25.1.1 #define cudaArrayDefault 0x00

Default CUDA array allocation flag

#### 4.25.1.2 #define cudaArrayLayered 0x01

Must be set in cudaMalloc3DArray to create a layered texture

#### 4.25.1.3 #define cudaArraySurfaceLoadStore 0x02

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

#### 4.25.1.4 #define cudaDeviceBlockingSync 0x04

Device flag - Use blocking synchronization

**Deprecated**

#### 4.25.1.5 #define cudaDeviceLmemResizeToMax 0x10

Device flag - Keep local memory allocation after launch

#### 4.25.1.6 #define cudaDeviceMapHost 0x08

Device flag - Support mapped pinned allocations

#### 4.25.1.7 #define cudaDeviceMask 0x1f

Device flags mask

#### 4.25.1.8 #define cudaDevicePropDontCare

Empty device properties

#### 4.25.1.9 #define cudaDeviceScheduleAuto 0x00

Device flag - Automatic scheduling
4.25.1.10  
#define cudaDeviceScheduleBlockingSync 0x04
Device flag - Use blocking synchronization

4.25.1.11  
#define cudaDeviceScheduleSpin 0x01
Device flag - Spin default scheduling

4.25.1.12  
#define cudaDeviceScheduleYield 0x02
Device flag - Yield default scheduling

4.25.1.13  
#define cudaEventBlockingSync 0x01
Event uses blocking synchronization

4.25.1.14  
#define cudaEventDefault 0x00
Default event flag

4.25.1.15  
#define cudaEventDisableTiming 0x02
Event will not record timing data

4.25.1.16  
#define cudaHostAllocDefault 0x00
Default page-locked allocation flag

4.25.1.17  
#define cudaHostAllocMapped 0x02
Map allocation into device space

4.25.1.18  
#define cudaHostAllocPortable 0x01
Pinned memory accessible by all CUDA contexts

4.25.1.19  
#define cudaHostAllocWriteCombined 0x04
Write-combined memory

4.25.1.20  
#define cudaHostRegisterDefault 0x00
Default host memory registration flag

4.25.1.21  
#define cudaHostRegisterMapped 0x02
Map registered memory into device space
4.25 Data types used by CUDA Runtime

4.25.1.22 #define cudaHostRegisterPortable 0x01
Pinned memory accessible by all CUDA contexts

4.25.1.23 #define cudaPeerAccessDefault 0x00
Default peer addressing enable flag

4.25.2 Typedef Documentation

4.25.2.1 typedef enum cudaError cudaError_t
CUDA Error types

4.25.2.2 typedef struct CUevent_st* cudaEvent_t
CUDA event types

4.25.2.3 typedef struct cudaGraphicsResource* cudaGraphicsResource_t
CUDA graphics resource types

4.25.2.4 typedef enum cudaOutputMode cudaOutputMode_t
CUDA output file modes

4.25.2.5 typedef struct CUstream_st* cudaStream_t
CUDA stream

4.25.2.6 typedef struct CUuuid_st cudaUUID_t
CUDA UUID types

4.25.3 Enumeration Type Documentation

4.25.3.1 enum cudaChannelFormatKind
Channel format kind

Enumerator:

- `cudaChannelFormatKindSigned` Signed channel format
- `cudaChannelFormatKindUnsigned` Unsigned channel format
- `cudaChannelFormatKindFloat` Float channel format
- `cudaChannelFormatKindNone` No channel format
4.25.3.2 enum cudaComputeMode

CUDA device compute modes

Enumerator:

cudaComputeModeDefault  Default compute mode (Multiple threads can use cudaSetDevice() with this device)

cudaComputeModeExclusive  Compute-exclusive-thread mode (Only one thread in one process will be able to use cudaSetDevice() with this device)

cudaComputeModeProhibited  Compute-prohibited mode (No threads can use cudaSetDevice() with this device)

cudaComputeModeExclusiveProcess  Compute-exclusive-process mode (Many threads in one process will be able to use cudaSetDevice() with this device)

4.25.3.3 enum cudaError

CUDA error types

Enumerator:

cudaSuccess  The API call returned with no errors. In the case of query calls, this can also mean that the operation being queried is complete (see cudaEventQuery() and cudaStreamQuery()).

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

cudaErrorMemoryAllocation  The API call failed because it was unable to allocate enough memory to perform the requested operation.

cudaErrorInitializationError  The API call failed because the CUDA driver and runtime could not be initialized.

cudaErrorLaunchFailure  An exception occurred on the device while executing a kernel. Common causes include dereferencing an invalid device pointer and accessing out of bounds shared memory. The device cannot be used until cudaThreadExit() is called. All existing device memory allocations are invalid and must be reconstructed if the program is to continue using CUDA.

cudaErrorPriorLaunchFailure  This indicated that a previous kernel launch failed. This was previously used for device emulation of kernel launches.

Deprecated

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

cudaErrorLaunchTimeout  This indicates that the device kernel took too long to execute. This can only occur if timeouts are enabled - see the device property kernelExecTimeoutEnabled for more information. The device cannot be used until cudaThreadExit() is called. All existing device memory allocations are invalid and must be reconstructed if the program is to continue using CUDA.

cudaErrorLaunchOutOfResources  This indicates that a launch did not occur because it did not have appropriate resources. Although this error is similar to cudaErrorInvalidConfiguration, this error usually indicates that the user has attempted to pass too many arguments to the device kernel, or the kernel launch specifies too many threads for the kernel’s register count.

cudaErrorInvalidDeviceFunction  The requested device function does not exist or is not compiled for the proper device architecture.
cudaErrorInvalidConfiguration  This indicates that a kernel launch is requesting resources that can never be satisfied by the current device. Requesting more shared memory per block than the device supports will trigger this error, as will requesting too many threads or blocks. See cudaDeviceProp for more device limitations.

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

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

cudaErrorInvalidPitchValue  This indicates that one or more of the pitch-related parameters passed to the API call is not within the acceptable range for pitch.

cudaErrorInvalidSymbol  This indicates that the symbol name/identifier passed to the API call is not a valid name or identifier.

cudaErrorMapBufferObjectFailed  This indicates that the buffer object could not be mapped.

cudaErrorUnmapBufferObjectFailed  This indicates that the buffer object could not be unmapped.

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

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

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

cudaErrorInvalidTextureBinding  This indicates that the texture binding is not valid. This occurs if you call cudaGetTextureAlignmentOffset() with an unbound texture.

cudaErrorInvalidChannelDescriptor  This indicates that the channel descriptor passed to the API call is not valid. This occurs if the format is not one of the formats specified by cudaChannelFormatKind, or if one of the dimensions is invalid.

cudaErrorInvalidMemcpyDirection  This indicates that the direction of the memcpy passed to the API call is not one of the types specified by cudaMemcpyKind.

cudaErrorAddressOfConstant  This indicated that the user has taken the address of a constant variable, which was forbidden up until the CUDA 3.1 release.

Deprecated
This error return is deprecated as of CUDA 3.1. Variables in constant memory may now have their address taken by the runtime via cudaGetSymbolAddress().

cudaErrorTextureFetchFailed  This indicated that a texture fetch was not able to be performed. This was previously used for device emulation of texture operations.

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

cudaErrorTextureNotBound  This indicated that a texture was not bound for access. This was previously used for device emulation of texture operations.

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

cudaErrorSynchronizationError  This indicated that a synchronization operation had failed. This was previously used for some device emulation functions.

Deprecated
This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.
cudaErrorInvalidFilterSetting This indicates that a non-float texture was being accessed with linear filtering. This is not supported by CUDA.

cudaErrorInvalidNormSetting This indicates that an attempt was made to read a non-float texture as a normalized float. This is not supported by CUDA.

cudaErrorMixedDeviceExecution Mixing of device and device emulation code was not allowed.

Deprecated

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

cudaErrorCudartUnloading This indicated an issue with calling API functions during the unload process of the CUDA runtime in prior releases.

Deprecated

This error return is deprecated as of CUDA 3.2.

cudaErrorUnknown This indicates that an unknown internal error has occurred.

cudaErrorNotYetImplemented This indicates that the API call is not yet implemented. Production releases of CUDA will never return this error.

cudaErrorMemoryValueTooLarge This indicated that an emulated device pointer exceeded the 32-bit address range.

Deprecated

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

cudaErrorInvalidResourceHandle This indicates that a resource handle passed to the API call was not valid. Resource handles are opaque types like cudaStream_t and cudaEvent_t.

cudaErrorNotReady This indicates that asynchronous operations issued previously have not completed yet. This result is not actually an error, but must be indicated differently than cudaSuccess (which indicates completion). Calls that may return this value include cudaMemcpy() and cudaMemcpyAsync() and cudaMemcpy2D() and cudaMemcpy2DAsync() and cudaMemcpyAsync() and cudaMemcpy2DAsync() and cudaMemcpy2DAsync().

cudaErrorInsufficientDriver This indicates that the installed NVIDIA CUDA driver is older than the CUDA runtime library. This is not a supported configuration. Users should install an updated NVIDIA display driver to allow the application to run.

cudaErrorSetOnActiveProcess This indicates that the user has called cudaSetDevice(), cudaSetValidDevices(), cudaSetDeviceFlags(), cudaD3D9SetDirect3DDevice(), cudaD3D10SetDirect3DDevice, cudaD3D11SetDirect3DDevice(), * or cudaVDPAUSetVDPAUDevice() after initializing the CUDA runtime by calling non-device management operations (allocating memory and launching kernels are examples of non-device management operations). This error can also be returned if using runtime/driver interoperability and there is an existing CUcontext active on the host thread.

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

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

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

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

cudaErrorSharedObjectInitFailed This indicates that initialization of a shared object failed.

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

cudaErrorDuplicateVariableName This indicates that multiple global or constant variables (across separate CUDA source files in the application) share the same string name.

cudaErrorDuplicateTextureName This indicates that multiple textures (across separate CUDA source files in the application) share the same string name.
cudaErrorDuplicateSurfaceName  This indicates that multiple surfaces (across separate CUDA source files in the application) share the same string name.

cudaErrorDevicesUnavailable  This indicates that all CUDA devices are busy or unavailable at the current time. Devices are often busy/unavailable due to use of cudaComputeModeExclusive, cudaComputeModeProhibited or when long running CUDA kernels have filled up the GPU and are blocking new work from starting. They can also be unavailable due to memory constraints on a device that already has active CUDA work being performed.

cudaErrorInvalidKernelImage  This indicates that the device kernel image is invalid.

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

cudaErrorIncompatibleDriverContext  This indicates that the current context is not compatible with this the CUDA Runtime. This can only occur if you are using CUDA Runtime/Driver interoperability and have created an existing Driver context using the driver API. The Driver context may be incompatible either because the Driver context was created using an older version of the API, because the Runtime API call expects a primary driver context and the Driver context is not primary, or because the Driver context has been destroyed. Please see Interactions with the CUDA Driver API for more information.

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

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

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

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

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

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

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

cudaErrorStartupFailure  This indicates an internal startup failure in the CUDA runtime.

cudaErrorApiFailureBase  Any unhandled CUDA driver error is added to this value and returned via the runtime. Production releases of CUDA should not return such errors.

4.25.3.4  enum cudaFuncCache

CUDA function cache configurations

Enumerator:

  cudaFuncCachePreferNone  Default function cache configuration, no preference
  cudaFuncCachePreferShared  Prefer larger shared memory and smaller L1 cache
  cudaFuncCachePreferL1  Prefer larger L1 cache and smaller shared memory
4.25.3.5  enum cudaGraphicsCubeFace

CUDA graphics interop array indices for cube maps

**Enumerator:**

- `cudaGraphicsCubeFacePositiveX`  Positive X face of cubemap
- `cudaGraphicsCubeFaceNegativeX`  Negative X face of cubemap
- `cudaGraphicsCubeFacePositiveY`  Positive Y face of cubemap
- `cudaGraphicsCubeFaceNegativeY`  Negative Y face of cubemap
- `cudaGraphicsCubeFacePositiveZ`  Positive Z face of cubemap
- `cudaGraphicsCubeFaceNegativeZ`  Negative Z face of cubemap

4.25.3.6  enum cudaGraphicsMapFlags

CUDA graphics interop map flags

**Enumerator:**

- `cudaGraphicsMapFlagsNone`  Default: Assume resource can be read/written
- `cudaGraphicsMapFlagsReadOnly`  CUDA will not write to this resource
- `cudaGraphicsMapFlagsWriteDiscard`  CUDA will only write to and will not read from this resource

4.25.3.7  enum cudaGraphicsRegisterFlags

CUDA graphics interop register flags

**Enumerator:**

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

4.25.3.8  enum cudaLimit

CUDA Limits

**Enumerator:**

- `cudaLimitStackSize`  GPU thread stack size
- `cudaLimitPrintfFifoSize`  GPU printf FIFO size
- `cudaLimitMallocHeapSize`  GPU malloc heap size
4.25.3.9 enum cudaMemcpyKind

CUDA memory copy types

Enumerator:

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

4.25.3.10 enum cudaMemoryType

CUDA memory types

Enumerator:

- cudaMemcpyTypeHost  Host memory
- cudaMemcpyTypeDevice  Device memory

4.25.3.11 enum cudaOutputMode

CUDA Profiler Output modes

4.25.3.12 enum cudaSurfaceBoundaryMode

CUDA Surface boundary modes

Enumerator:

- cudaMemcpyBoundaryModeZero  Zero boundary mode
- cudaMemcpyBoundaryModeClamp  Clamp boundary mode
- cudaMemcpyBoundaryModeTrap  Trap boundary mode

4.25.3.13 enum cudaSurfaceFormatMode

CUDA Surface format modes

Enumerator:

- cudaMemcpyFormatModeForced  Forced format mode
- cudaMemcpyFormatModeAuto  Auto format mode
4.25.3.14  enum cudaTextureAddressMode

CUDA texture address modes

Enumerator:

- `cudaAddressModeWrap` Wrapping address mode
- `cudaAddressModeClamp` Clamp to edge address mode
- `cudaAddressModeMirror` Mirror address mode
- `cudaAddressModeBorder` Border address mode

4.25.3.15  enum cudaTextureFilterMode

CUDA texture filter modes

Enumerator:

- `cudaFilterModePoint` Point filter mode
- `cudaFilterModeLinear` Linear filter mode

4.25.3.16  enum cudaTextureReadMode

CUDA texture read modes

Enumerator:

- `cudaReadModeElementType` Read texture as specified element type
- `cudaReadModeNormalizedFloat` Read texture as normalized float
4.26 CUDA Driver API

Modules

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

4.26.1 Detailed Description

This section describes the low-level CUDA driver application programming interface.
4.27 Data types used by CUDA driver

Data Structures

- `struct CUDA_ARRAY3D_DESCRIPTOR_st`
- `struct CUDA_ARRAY_DESCRIPTOR_st`
- `struct CUDA_MEMCPY2D_st`
- `struct CUDA_MEMCPY3D_PEER_st`
- `struct CUDA_MEMCPY3D_st`
- `struct CUdevprop_st`

Defines

- `#define CU_LAUNCH_PARAM_BUFFER_POINTER ((void*)0x01)`
- `#define CU_LAUNCH_PARAM_BUFFER_SIZE ((void*)0x02)`
- `#define CU_LAUNCH_PARAM_END ((void*)0x00)`
- `#define CU_MEMHOSTALLOC_DEVICEMAP 0x02`
- `#define CU_MEMHOSTALLOC_PORTABLE 0x01`
- `#define CU_MEMHOSTALLOC_WRITECOMBINED 0x04`
- `#define CU_MEMHOSTREGISTER_DEVICEMAP 0x02`
- `#define CU_MEMHOSTREGISTER_PORTABLE 0x01`
- `#define CU_PARAM_TR_DEFAULT -1`
- `#define CU_TRSA_OVERRIDE_FORMAT 0x01`
- `#define CU_TRSF_NORMALIZED_COORDINATES 0x02`
- `#define CU_TRSF_READ_AS_INTEGER 0x01`
- `#define CU_TRSF_SRGB 0x10`
- `#define CUDA_ARRAY3D_2DARRAY 0x01`
- `#define CUDA_ARRAY3D_LAYERED 0x01`
- `#define CUDA_ARRAY3D_SURFACE_LDST 0x02`
- `#define CUDA_VERSION 4000`

Typedefs

- `typedef enum CUaddress_mode_enum CUaddress_mode`
- `typedef struct CUarray_st *CUarray`
- `typedef enum CUint32_t_enum CUint32_t`
- `typedef struct CUDA_ARRAY3D_DESCRIPTOR_st *CUDA_ARRAY3D_DESCRIPTOR`
- `typedef struct CUDA_ARRAY_DESCRIPTOR_st *CUDA_ARRAY_DESCRIPTOR`
- `typedef struct CUDA_MEMCPY2D_st *CUDA_MEMCPY2D`
- `typedef struct CUDA_MEMCPY3D_st *CUDA_MEMCPY3D`
- `typedef structure CUDA_MEMCPY3D_PEER_st *CUDA_MEMCPY3D_PEER`
- `typedef int CUdevice`
- `typedef enum CUdevice_attribute_enum CUdevice_attribute`
- `typedef unsigned int CUdeviceptr`
- `typedef struct CUdevprop_st CUdevprop`
4.27 Data types used by CUDA driver

• typedef struct CUevent_st * CUevent
• typedef enum CUevent_flags_enum CUevent_flags
• typedef enum CUfilter_mode_enum CUfilter_mode
• typedef enum CUfunc_cache_enum CUfunc_cache
• typedef struct CUfunc_st * CUfunction
• typedef enum CUfunction_attribute_enum CUfunction_attribute
• typedef enum CUgraphicsMapResourceFlags_enum CUgraphicsMapResourceFlags
• typedef enum CUgraphicsRegisterFlags_enum CUgraphicsRegisterFlags
• typedef struct CUgraphicsResource_st * CUgraphicsResource
• typedef enum CUidjit_fallback_enum CUidjit_fallback
• typedef enum CUidjit_option_enum CUidjit_option
• typedef enum CUidjit_target_enum CUidjit_target
• typedef enum CUidlimit_enum CUidlimit
• typedef enum CUidmemorytype_enum CUidmemorytype
• typedef struct CUidmodule_st * CUidmodule
• typedef enum CUidpointer_attribute_enum CUidpointer_attribute
• typedef enum cudaError_enum CUidresult
• typedef struct CUidstream_st * CUidstream
• typedef struct CUidsurfref_st * CUidsurfref
• typedef struct CUidtexref_st * CUidtexref

Enumerations

• enum CUidaddress_mode_enum {
  CU_TR_ADDRESS_MODE_WRAP = 0,
  CU_TR_ADDRESS_MODE_CLAMP = 1,
  CU_TR_ADDRESS_MODE_MIRROR = 2,
  CU_TR_ADDRESS_MODE_BORDER = 3 } 
• enum CUidarray_cubemap_face_enum {
  CU_CUBEMAP_FACE_POSITIVE_X = 0x00,
  CU_CUBEMAP_FACE_NEGATIVE_X = 0x01,
  CU_CUBEMAP_FACE_POSITIVE_Y = 0x02,
  CU_CUBEMAP_FACE_NEGATIVE_Y = 0x03,
  CU_CUBEMAP_FACE_POSITIVE_Z = 0x04,
  CU_CUBEMAP_FACE_NEGATIVE_Z = 0x05 } 
• enum CUidarray_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_HALFF = 0x10,
  CU_AD_FORMAT_FLOAT = 0x20 } 

Generated for NVIDIA CUDA Library by Doxygen
• enum CUcomputemode_enum {
    CU_COMPUTEMODE_DEFAULT = 0,
    CU_COMPUTEMODE_EXCLUSIVE = 1,
    CU_COMPUTEMODE_PROHIBITED = 2,
    CU_COMPUTEMODE_EXCLUSIVE_PROCESS = 3 }
• enum CUctx_flags_enum {
    CU_CTX_SCHED_AUTO = 0x00,
    CU_CTX_SCHED_SPIN = 0x01,
    CU_CTX_SCHED_YIELD = 0x02,
    CU_CTX_SCHED_BLOCKING_SYNC = 0x04,
    CU_CTX_BLOCKING_SYNC = 0x04,
    CU_CTX_MAP_HOST = 0x08,
    CU_CTX_LMEM_RESIZE_TO_MAX = 0x10 }
• enum cudaError_enum {
    CUDA_SUCCESS = 0,
    CUDA_ERROR_INVALID_VALUE = 1,
    CUDA_ERROR_OUT_OF_MEMORY = 2,
    CUDA_ERROR_NOT_INITIALIZED = 3,
    CUDA_ERROR_DEINITIALIZED = 4,
    CUDA_ERROR_PROFILER_DISABLED = 5,
    CUDA_ERROR_PROFILER_NOT_INITIALIZED = 6,
    CUDA_ERROR_PROFILER_ALREADY_STARTED = 7,
    CUDA_ERROR_PROFILER_ALREADY_STOPPED = 8,
    CUDA_ERROR_NO_DEVICE = 100,
    CUDA_ERROR_INVALID_DEVICE = 101,
    CUDA_ERROR_INVALID_IMAGE = 200,
    CUDA_ERROR_INVALID_CONTEXT = 201,
    CUDA_ERROR_CONTEXT_ALREADY_CURRENT = 202,
    CUDA_ERROR_MAP_FAILED = 205,
    CUDA_ERROR_UNMAP_FAILED = 206,
    CUDA_ERROR_ARRAY_IS_MAPPED = 207,
    CUDA_ERROR_ALREADY_MAPPED = 208,
    CUDA_ERROR_NO_BINARY_FOR_GPU = 209,
    CUDA_ERROR_ALREADY_ACQUIRED = 210,
    CUDA_ERROR_NOT_MAPPED = 211,
    CUDA_ERROR_NOT_MAPPED_AS_ARRAY = 212,
    CUDA_ERROR_NOT_MAPPED_AS_POINTER = 213,
    CUDA_ERROR_ECC_UNCORRECTABLE = 214,
    CUDA_ERROR_UNSUPPORTED_LIMIT = 215,
    CUDA_ERROR_CONTEXT_ALREADY_IN_USE = 216,
    CUDA_ERROR_INVALID_SOURCE = 300,
4.27 Data types used by CUDA driver

CUDA_ERROR_FILE_NOT_FOUND = 301,
CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND = 302,
CUDA_ERROR_SHARED_OBJECT_INIT_FAILED = 303,
CUDA_ERROR_OPERATING_SYSTEM = 304,
CUDA_ERROR_INVALID_HANDLE = 400,
CUDA_ERROR_NOT_FOUND = 500,
CUDA_ERROR_NOT_READY = 600,
CUDA_ERROR_LAUNCH_FAILED = 700,
CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES = 701,
CUDA_ERROR_LAUNCH_TIMEOUT = 702,
CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING = 703,
CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED = 704,
CUDA_ERROR_PEER_ACCESS_NOT_ENABLED = 705,
CUDA_ERROR_PRIMARY_CONTEXT_ACTIVE = 708,
CUDA_ERROR_CONTEXT_IS_DESTROYED = 709,
CUDA_ERROR_UNKNOWN = 999

• enum CUdevice_attribute_enum {
  CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK = 1,
  CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X = 2,
  CU DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y = 3,
  CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z = 4,
  CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X = 5,
  CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y = 6,
  CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z = 7,
  CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK = 8,
  CU_DEVICE_ATTRIBUTE_SHARED_MEMORY_PER_BLOCK = 8,
  CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY = 9,
  CU_DEVICE_ATTRIBUTE_WARP_SIZE = 10,
  CU_DEVICE_ATTRIBUTE_MAX_PITCH = 11,
  CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK = 12,
  CU_DEVICE_ATTRIBUTE_REGISTERS_PER_BLOCK = 12,
  CU_DEVICE_ATTRIBUTE_CLOCK_RATE = 13,
  CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT = 14,
  CU_DEVICE_ATTRIBUTE_GPU_OVERLAP = 15,
  CUDEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT = 16,
  CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT = 17,
  CU_DEVICE_ATTRIBUTE_INTEGRATED = 18,
  CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY = 19,
  CU_DEVICE_ATTRIBUTE_COMPUTE_MODE = 20,
  CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH = 21,
  CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_WIDTH = 22,

Generated for NVIDIA CUDA Library by Doxygen
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_HEIGHT = 23,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH = 24,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT = 25,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH = 26,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH = 27,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT = 28,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS = 29,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_WIDTH = 27,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_HEIGHT = 28,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_NUMSLICES = 29,
CUDEVICE_ATTRIBUTE_SURFACE_ALIGNMENT = 30,
CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS = 31,
CU_DEVICE_ATTRIBUTE_ECC_ENABLED = 32,
CU_DEVICE_ATTRIBUTE_PCI_BUS_ID = 33,
CU_DEVICE_ATTRIBUTE_PCI_DEVICE_ID = 34,
CU_DEVICE_ATTRIBUTE_TCC_DRIVER = 35,
CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE = 36,
CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH = 37,
CUDEVICE_ATTRIBUTE_L2_CACHE_SIZE = 38,
CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR = 39,
CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT = 40,
CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING = 41,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH = 42,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS = 43,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_NUMSLICES = 50 }

• enum CUEvent_flags_enum {
    CU_EVENT_DEFAULT = 0,
    CU_EVENT_BLOCKING_SYNC = 1,
    CU_EVENT_DISABLE_TIMING = 2 }

• enum CUfilter_mode_enum {
    CU_TR_FILTER_MODE_POINT = 0,
    CU_TR_FILTER_MODE_LINEAR = 1 }

• enum CUFunc_cache_enum {
    CU_FUNC_CACHE_PREFER_NONE = 0x00,
    CU_FUNC_CACHE_PREFER_SHARED = 0x01,
    CU_FUNC_CACHE_PREFER_L1 = 0x02 }

• enum CUFункци_attribute_enum {
    CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK = 0,
    CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES = 1,
    CU_FUNC_ATTRIBUTE_CONST_SIZE_BYTES = 2,
    CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES = 3,
    CU_FUNC_ATTRIBUTE_NUM_REGS = 4,
CU_FUNC_ATTRIBUTE_PTX_VERSION = 5,
CU_FUNC_ATTRIBUTE_BINARY_VERSION = 6 }

• enum CUgraphicsMapResourceFlags_enum
• enum CUgraphicsRegisterFlags_enum
• enum CUjit_fallback_enum {
  CU_PREFER_PTX = 0,
  CU_PREFER_BINARY }
• enum CUjit_option_enum {
  CU_JIT_MAX_REGISTERS = 0,
  CU_JIT_THREADS_PER_BLOCK,
  CU_JIT_WALL_TIME,
  CU_JIT_INFO_LOG_BUFFER,
  CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES,
  CU_JIT_ERROR_LOG_BUFFER,
  CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES,
  CU_JIT_OPTIMIZATION_LEVEL,
  CU_JIT_TARGET_FROM_CUCONTEXT,
  CU_JIT_TARGET,
  CU_JIT_FALLBACK_STRATEGY }
• enum CUjit_target_enum {
  CU_TARGET_COMPUTE_10 = 0,
  CU_TARGET_COMPUTE_11,
  CU_TARGET_COMPUTE_12,
  CU_TARGET_COMPUTE_13,
  CU_TARGET_COMPUTE_20,
  CU_TARGET_COMPUTE_21 }
• enum CUlimit_enum {
  CU_LIMIT_STACK_SIZE = 0x00,
  CU_LIMIT_PRINTF_FIFO_SIZE = 0x01,
  CU_LIMIT_MALLOC_HEAP_SIZE = 0x02 }
• enum CUmemorytype_enum {
  CU_MEMORYTYPE_HOST = 0x01,
  CU_MEMORYTYPE_DEVICE = 0x02,
  CU_MEMORYTYPE_ARRAY = 0x03,
  CU_MEMORYTYPE_UNIFIED = 0x04 }
• enum CUpointer_attribute_enum {
  CU_POINTER_ATTRIBUTE_CONTEXT = 1,
  CU_POINTER_ATTRIBUTE_MEMORY_TYPE = 2,
  CU_POINTER_ATTRIBUTE_DEVICE_POINTER = 3,
  CU_POINTER_ATTRIBUTE_HOST_POINTER = 4 }
4.27.1 Define Documentation

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

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

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

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

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

End of array terminator for the extra parameter to cuLaunchKernel

4.27.1.4 #define CU_MEMHOSTALLOC_DEVICEMAP 0x02

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

4.27.1.5 #define CU_MEMHOSTALLOC_PORTABLE 0x01

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

4.27.1.6 #define CU_MEMHOSTALLOC_WRITECOMBINED 0x04

If set, host memory is allocated as write-combined - fast to write, faster to DMA, slow to read except via SSE4 streaming load instruction (MOVNTDQA). Flag for cuMemHostAlloc()

4.27.1.7 #define CU_MEMHOSTREGISTER_DEVICEMAP 0x02

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

4.27.1.8 #define CU_MEMHOSTREGISTER_PORTABLE 0x01

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

4.27.1.9 #define CU_PARAM_TR_DEFAULT -1

For texture references loaded into the module, use default texunit from texture reference.
4.27 Data types used by CUDA driver

4.27.1.10 define CU_TRSA_OVERRIDE_FORMAT 0x01
Override the texref format with a format inferred from the array. Flag for cuTexRefSetArray()

4.27.1.11 define CU_TRSF_NORMALIZED_COORDINATES 0x02
Use normalized texture coordinates in the range [0,1) instead of [0,dim). Flag for cuTexRefSetFlags()

4.27.1.12 define CU_TRSF_READ_AS_INTEGER 0x01
Read the texture as integers rather than promoting the values to floats in the range [0,1]. Flag for cuTexRefSetFlags()

4.27.1.13 define CU_TRSF_SRGB 0x10
Perform sRGB->linear conversion during texture read. Flag for cuTexRefSetFlags()

4.27.1.14 define CUDA_ARRAY3D_2DARRAY 0x01
Deprecated, use CUDA_ARRAY3D_LAYERED

4.27.1.15 define CUDA_ARRAY3D_LAYERED 0x01
If set, the CUDA array is a collection of layers, where each layer is either a 1D or a 2D array and the Depth member of CUDA_ARRAY3D_DESCRIPTOR specifies the number of layers, not the depth of a 3D array.

4.27.1.16 define CUDA_ARRAY3D_SURFACE_LDST 0x02
This flag must be set in order to bind a surface reference to the CUDA array

4.27.1.17 define CUDA_VERSION 4000
CUDA API version number

4.27.2 Typedef Documentation

4.27.2.1 typedef enum CUaddress_mode_enum CUaddress_mode
Texture reference addressing modes

4.27.2.2 typedef struct CUarray_st∗ CUarray
CUDA array

4.27.2.3 typedef enum CUarray_cubemap_face_enum CUarray_cubemap_face
Array indices for cube faces
4.27.2.4 typedef enum CUarray_format_enum CUarray_format

Array formats

4.27.2.5 typedef enum CUcomputemode_enum CUcomputemode

Compute Modes

4.27.2.6 typedef struct CUctx_st* CUcontext

CUDA context

4.27.2.7 typedef enum CUctx_flags_enum CUctx_flags

Context creation flags

4.27.2.8 typedef struct CUDA_ARRAY3D_DESCRIPTOR_st CUDA_ARRAY3D_DESCRIPTOR

3D array descriptor

4.27.2.9 typedef struct CUDA_ARRAY_DESCRIPTOR_st CUDA_ARRAY_DESCRIPTOR

Array descriptor

4.27.2.10 typedef struct CUDA_MEMCPY2D_st CUDA_MEMCPY2D

2D memory copy parameters

4.27.2.11 typedef struct CUDA_MEMCPY3D_st CUDA_MEMCPY3D

3D memory copy parameters

4.27.2.12 typedef struct CUDA_MEMCPY3D_PEER_st CUDA_MEMCPY3D_PEER

3D memory cross-context copy parameters

4.27.2.13 typedef int CUdevice

CUDA device

4.27.2.14 typedef enum CUdevice_attribute_enum CUdevice_attribute

Device properties

4.27.2.15 typedef unsigned int CUdeviceptr

CUDA device pointer
4.27 Data types used by CUDA driver

4.27.2.16 typedef struct CUdevprop_st CUdevprop

Legacy device properties

4.27.2.17 typedef struct CUevent_st∗ CUevent

CUDA event

4.27.2.18 typedef enum CUevent_flags_enum CUevent_flags

Event creation flags

4.27.2.19 typedef enum CUfilter_mode_enum CUfilter_mode

Texture reference filtering modes

4.27.2.20 typedef enum CUfunc_cache_enum CUfunc_cache

Function cache configurations

4.27.2.21 typedef struct CUfunc_st∗ CUfunction

CUDA function

4.27.2.22 typedef enum CUfunction_attribute_enum CUfunction_attribute

Function properties

4.27.2.23 typedef enum CUgraphicsMapResourceFlags_enum CUgraphicsMapResourceFlags

Flags for mapping and unmapping interop resources

4.27.2.24 typedef enum CUgraphicsRegisterFlags_enum CUgraphicsRegisterFlags

Flags to register a graphics resource

4.27.2.25 typedef struct CUgraphicsResource_st∗ CUgraphicsResource

CUDA graphics interop resource

4.27.2.26 typedef enum CUjit_fallback_enum CUjit_fallback

Cubin matching fallback strategies

4.27.2.27 typedef enum CUjit_option_enum CUjit_option

Online compiler options

Generated for NVIDIA CUDA Library by Doxygen
4.27.2.28  typedef enum CUjit_target_enum CUjit_target

Online compilation targets

4.27.2.29  typedef enum CUlimit_enum CUlimit

Limits

4.27.2.30  typedef enum CUmemorytype_enum CUmemorytype

Memory types

4.27.2.31  typedef struct CUmod_st* CUmodule

CUDA module

4.27.2.32  typedef enum CUpointer_attribute_enum CUpointer_attribute

Pointer information

4.27.2.33  typedef enum cudaError_enum CUresult

Error codes

4.27.2.34  typedef struct CUsream_st* CUsream

CUDA stream

4.27.2.35  typedef struct CUsurfref_st* CUsurfref

CUDA surface reference

4.27.2.36  typedef struct CUtrefref_st* CUtrefref

CUDA texture reference

4.27.3  Enumeration Type Documentation

4.27.3.1  enum CUmaddress_mode_enum

Texture reference addressing modes

Enumerator:

\begin{itemize}
  \item \texttt{CU_TR_ADDRESS_MODE_WRAP} \hspace{1cm} Wrapping address mode
  \item \texttt{CU_TR_ADDRESS_MODE_CLAMP} \hspace{1cm} Clamp to edge address mode
  \item \texttt{CU_TR_ADDRESS_MODE_MIRROR} \hspace{1cm} Mirror address mode
  \item \texttt{CU_TR_ADDRESS_MODE_BORDER} \hspace{1cm} Border address mode
\end{itemize}
4.27 Data types used by CUDA driver

4.27.3.2 enum CUarray_cubemap_face_enum

Array indices for cube faces

Enumerator:

- **CU_CUBEMAP_FACE_POSITIVE_X** Positive X face of cubemap
- **CU_CUBEMAP_FACE_NEGATIVE_X** Negative X face of cubemap
- **CU_CUBEMAP_FACE_POSITIVE_Y** Positive Y face of cubemap
- **CU_CUBEMAP_FACE_NEGATIVE_Y** Negative Y face of cubemap
- **CU_CUBEMAP_FACE_POSITIVE_Z** Positive Z face of cubemap
- **CU_CUBEMAP_FACE_NEGATIVE_Z** Negative Z face of cubemap

4.27.3.3 enum CUarray_format_enum

Array formats

Enumerator:

- **CU_AD_FORMAT_UNSIGNED_INT8** Unsigned 8-bit integers
- **CU_AD_FORMAT_UNSIGNED_INT16** Unsigned 16-bit integers
- **CU_AD_FORMAT_UNSIGNED_INT32** Unsigned 32-bit integers
- **CU_AD_FORMAT_SIGNED_INT8** Signed 8-bit integers
- **CU_AD_FORMAT_SIGNED_INT16** Signed 16-bit integers
- **CU_AD_FORMAT_SIGNED_INT32** Signed 32-bit integers
- **CU_AD_FORMAT_HALF** 16-bit floating point
- **CU_AD_FORMAT_FLOAT** 32-bit floating point

4.27.3.4 enum CUcomputemode_enum

Compute Modes

Enumerator:

- **CU_COMPUTEMODE_DEFAULT** Default compute mode (Multiple contexts allowed per device)
- **CU_COMPUTEMODE_EXCLUSIVE** Compute-exclusive-thread mode (Only one context used by a single thread can be present on this device at a time)
- **CU_COMPUTEMODE_PROHIBITED** Compute-prohibited mode (No contexts can be created on this device at this time)
- **CU_COMPUTEMODE_EXCLUSIVE_PROCESS** Compute-exclusive-process mode (Only one context used by a single process can be present on this device at a time)

4.27.3.5 enum CUctx_flags_enum

Context creation flags
Enumerators:

- `CU_CTX_SCHED_AUTO`  Automatic scheduling
- `CU_CTX_SCHED_SPIN`  Set spin as default scheduling
- `CU_CTX_SCHED_YIELD`  Set yield as default scheduling
- `CU_CTX_SCHED_BLOCKING_SYNC`  Set blocking synchronization as default scheduling
- `CU_CTX_BLOCKING_SYNC`  Set blocking synchronization as default scheduling

**Deprecated**

- `CU_CTX_MAP_HOST`  Support mapped pinned allocations
- `CU_CTX_LMEM_RESIZE_TO_MAX`  Keep local memory allocation after launch

### 4.27.3.6 Enum cudaError_enum

Error codes

**Enumerators:**

- `CUDA_SUCCESS`  The API call returned with no errors. In the case of query calls, this can also mean that the operation being queried is complete (see `cuEventQuery()` and `cuStreamQuery()`).
- `CUDA_ERROR_INVALID_VALUE`  This indicates that one or more of the parameters passed to the API call is not within an acceptable range of values.
- `CUDA_ERROR_OUT_OF_MEMORY`  The API call failed because it was unable to allocate enough memory to perform the requested operation.
- `CUDA_ERROR_NOT_INITIALIZED`  This indicates that the CUDA driver has not been initialized with `cuInit()` or that initialization has failed.
- `CUDA_ERROR_DEINITIALIZED`  This indicates that the CUDA driver is in the process of shutting down.
- `CUDA_ERROR_PROFILER_DISABLED`  This indicates profiling APIs are called while application is running in visual profiler mode.
- `CUDA_ERROR_PROFILER_NOT_INITIALIZED`  This indicates profiling has not been initialized for this context. Call `cuProfilerInitialize()` to resolve this.
- `CUDA_ERROR_PROFILER_ALREADY_STARTED`  This indicates profiler has already been started and probably `cuProfilerStart()` is incorrectly called.
- `CUDA_ERROR_PROFILER_ALREADY_STOPPED`  This indicates profiler has already been stopped and probably `cuProfilerStop()` is incorrectly called.
- `CUDA_ERROR_NO_DEVICE`  This indicates that no CUDA-capable devices were detected by the installed CUDA driver.
- `CUDA_ERROR_INVALID_DEVICE`  This indicates that the device ordinal supplied by the user does not correspond to a valid CUDA device.
- `CUDA_ERROR_INVALID_IMAGE`  This indicates that the device kernel image is invalid. This can also indicate an invalid CUDA module.
- `CUDA_ERROR_INVALID_CONTEXT`  This most frequently indicates that there is no context bound to the current thread. This can also be returned if the context passed to an API call is not a valid handle (such as a context that has had `cuCtxDestroy()` invoked on it). This can also be returned if a user mixes different API versions (i.e. 3010 context with 3020 API calls). See `cuCtxGetApiVersion()` for more details.
- `CUDA_ERROR_CONTEXT_ALREADY_CURRENT`  This indicated that the context being supplied as a parameter to the API call was already the active context.
4.27 Data types used by CUDA driver

**Deprecated**

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

**CUDA_ERROR_MAP_FAILED** This indicates that a map or register operation has failed.

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

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

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

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

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

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

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

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

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

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

**CUDA_ERROR_CONTEXT_ALREADY_IN_USE** This indicates that the `CUcontext` passed to the API call can only be bound to a single CPU thread at a time but is already bound to a CPU thread.

**CUDA_ERROR_INVALID_SOURCE** This indicates that the device kernel source is invalid.

**CUDA_ERROR_FILE_NOT_FOUND** This indicates that the file specified was not found.

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

**CUDA_ERROR_SHARED_OBJECT_INIT_FAILED** This indicates that initialization of a shared object failed.

**CUDA_ERROR_OPERATING_SYSTEM** This indicates that an OS call failed.

**CUDA_ERROR_INVALID_HANDLE** This indicates that a resource handle passed to the API call was not valid. Resource handles are opaque types like `CUstream` and `CUevent`.

**CUDA_ERROR_NOT_FOUND** This indicates that a named symbol was not found. Examples of symbols are global/constant variable names, texture names, and surface names.

**CUDA_ERROR_NOT_READY** This indicates that asynchronous operations issued previously have not completed yet. This result is not actually an error, but must be indicated differently than `CUDA_SUCCESS` (which indicates completion). Calls that may return this value include `cuEventQuery()` and `cuStreamQuery()`.

**CUDA_ERROR_LAUNCH_FAILED** An exception occurred on the device while executing a kernel. Common causes include dereferencing an invalid device pointer and accessing out of bounds shared memory. The context cannot be used, so it must be destroyed (and a new one should be created). All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.

**CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES** This indicates that a launch did not occur because it did not have appropriate resources. This error usually indicates that the user has attempted to pass too many arguments to the device kernel, or the kernel launch specifies too many threads for the kernel's register count. Passing arguments of the wrong size (i.e. a 64-bit pointer when a 32-bit int is expected) is equivalent to passing too many arguments and can also result in this error.
CUDA_ERROR_LAUNCH_TIMEOUT  This indicates that the device kernel took too long to execute. This can only occur if timeouts are enabled - see the device attribute CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT for more information. The context cannot be used (and must be destroyed similar to CUDA_ERROR_LAUNCH_FAILED). All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.

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

CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED  This error indicates that a call to cuCtxEnablePeerAccess() is trying to re-enable peer access to a context which has already had peer access to it enabled.

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

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

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

CUDA_ERROR_UNKNOWN  This indicates that an unknown internal error has occurred.

4.27.3.7  enum CUdevice_attribute_enum

Device properties

Enumerator:

CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK  Maximum number of threads per block
CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X  Maximum block dimension X
CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y  Maximum block dimension Y
CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z  Maximum block dimension Z
CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X  Maximum grid dimension X
CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y  Maximum grid dimension Y
CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z  Maximum grid dimension Z
CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK  Maximum shared memory available per block in bytes
cuDeviceAttribute_shared_memory_per_block  Deprecated, use CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK
CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY  Memory available on device for __constant__ variables in a CUDA C kernel in bytes
CU_DEVICE_ATTRIBUTE_WARP_SIZE  Warp size in threads
CU_DEVICE_ATTRIBUTE_MAX_PITCH  Maximum pitch in bytes allowed by memory copies
CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK  Maximum number of 32-bit registers available per block
CU_DEVICE_ATTRIBUTE_REGISTERS_PER_BLOCK  Deprecated, use CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK
CU_DEVICE_ATTRIBUTE_CLOCK_RATE  Peak clock frequency in kilohertz
CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT  Alignment requirement for textures
CU_DEVICE_ATTRIBUTE_GPU_OVERLAP  Device can possibly copy memory and execute a kernel concurrently. Deprecated. Use instead CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT.
4.27 Data types used by CUDA driver

- **CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT**: Number of multiprocessors on device
- **CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT**: Specifies whether there is a runtime limit on kernels
- **CU_DEVICE_ATTRIBUTE_INTEGRATED**: Device is integrated with host memory
- **CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY**: Device can map host memory into CUDA address space
- **CU_DEVICE_ATTRIBUTE_COMPUTE_MODE**: Compute mode (See CUcomputeMode for details)
- **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH**: Maximum 1D texture width
- **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_WIDTH**: Maximum 2D texture width
- **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_HEIGHT**: Maximum 2D texture height
- **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH**: Maximum 3D texture width
- **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT**: Maximum 3D texture height
- **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH**: Maximum 3D texture depth
- **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH**: Maximum 2D layered texture width
- **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT**: Maximum 2D layered texture height
- **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS**: Maximum layers in a 2D layered texture
- **CUDEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_WIDTH**: Deprecated, use **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH**
- **CUDEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_HEIGHT**: Deprecated, use **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT**
- **CUDEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_NUMSPLICES**: Deprecated, use **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS**
- **CU_DEVICE_ATTRIBUTE_SURFACE_ALIGNMENT**: Alignment requirement for surfaces
- **CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS**: Device can possibly execute multiple kernels concurrently
- **CU_DEVICE_ATTRIBUTE_ECC_ENABLED**: Device has ECC support enabled
- **CU_DEVICE_ATTRIBUTE_PCI_BUS_ID**: PCI bus ID of the device
- **CU_DEVICE_ATTRIBUTE_PCI_DEVICE_ID**: PCI device ID of the device
- **CU_DEVICE_ATTRIBUTE_TCC_DRIVER**: Device is using TCC driver model
- **CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE**: Peak memory clock frequency in kilohertz
- **CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH**: Global memory bus width in bits
- **CU_DEVICE_ATTRIBUTE_L2_CACHE_SIZE**: Size of L2 cache in bytes
- **CUDEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR**: Maximum resident threads per multiprocessor
- **CU_DEVICE_ATTRIBUTE_ASYNC ENGINE_COUNT**: Number of asynchronous engines
- **CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING**: Device shares a unified address space with the host
- **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH**: Maximum 1D layered texture width
- **CUDEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS**: Maximum layers in a 1D layered texture
- **CUDEVICE_ATTRIBUTE_PCI_DOMAIN_ID**: PCI domain ID of the device
4.27.3.8 enum CUevent_flags_enum

Event creation flags

**Enumerator:**

- **CU_EVENT_DEFAULT**  Default event flag
- **CU_EVENT_BLOCKING_SYNC**  Event uses blocking synchronization
- **CU_EVENT_DISABLE_TIMING**  Event will not record timing data

4.27.3.9 enum CUfilter_mode_enum

Texture reference filtering modes

**Enumerator:**

- **CU_TR_FILTER_MODE_POINT**  Point filter mode
- **CU_TR_FILTER_MODE_LINEAR**  Linear filter mode

4.27.3.10 enum CUfunc_cache_enum

Function cache configurations

**Enumerator:**

- **CU_FUNC_CACHE_PREFER_NONE**  no preference for shared memory or L1 (default)
- **CU_FUNC_CACHE_PREFER_SHARED**  prefer larger shared memory and smaller L1 cache
- **CU_FUNC_CACHE_PREFER_L1**  prefer larger L1 cache and smaller shared memory

4.27.3.11 enum CUfunction_attribute_enum

Function properties

**Enumerator:**

- **CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK**  The maximum number of threads per block, beyond which a launch of the function would fail. This number depends on both the function and the device on which the function is currently loaded.
- **CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES**  The size in bytes of statically-allocated shared memory required by this function. This does not include dynamically-allocated shared memory requested by the user at runtime.
- **CU_FUNC_ATTRIBUTE_CONST_SIZE_BYTES**  The size in bytes of user-allocated constant memory required by this function.
- **CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES**  The size in bytes of local memory used by each thread of this function.
- **CU_FUNC_ATTRIBUTE_NUM_REGS**  The number of registers used by each thread of this function.
- **CU_FUNC_ATTRIBUTE_PTX_VERSION**  The PTX virtual architecture version for which the function was compiled. This value is the major PTX version \times 10 + the minor PTX version, so a PTX version 1.3 function would return the value 13. Note that this may return the undefined value of 0 for cubins compiled prior to CUDA 3.0.
**CU_FUNC_ATTRIBUTE_BINARY_VERSION** The binary architecture version for which the function was compiled. This value is the major binary version + 10 + the minor binary version, so a binary version 1.3 function would return the value 13. Note that this will return a value of 10 for legacy cubins that do not have a properly-encoded binary architecture version.

### 4.27.3.12 enum CUgraphicsMapResourceFlags_enum

Flags for mapping and unmapping interop resources

### 4.27.3.13 enum CUgraphicsRegisterFlags_enum

Flags to register a graphics resource

### 4.27.3.14 enum CUjit_fallback_enum

Cubin matching fallback strategies

**Enumerator:**

- **CU_PREFER_PTX** Prefer to compile ptx
- **CU_PREFER_BINARY** Prefer to fall back to compatible binary code

### 4.27.3.15 enum CUjit_option_enum

Online compiler options

**Enumerator:**

- **CU_JIT_MAX_REGISTERS** Max number of registers that a thread may use.
  Option type: unsigned int
- **CU_JIT_THREADS_PER_BLOCK** IN: Specifies minimum number of threads per block to target compilation for
  OUT: Returns the number of threads the compiler actually targeted. This restricts the resource utilization of the compiler (e.g. max registers) such that a block with the given number of threads should be able to launch based on register limitations. Note, this option does not currently take into account any other resource limitations, such as shared memory utilization.
  Option type: unsigned int
- **CU_JIT_WALL_TIME** Returns a float value in the option of the wall clock time, in milliseconds, spent creating the cubin.
  Option type: float
- **CU_JIT_INFO_LOG_BUFFER** Pointer to a buffer in which to print any log messages from PTXAS that are informational in nature (the buffer size is specified via option **CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES**)
  Option type: char*
- **CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES** IN: Log buffer size in bytes. Log messages will be capped at this size (including null terminator)
  OUT: Amount of log buffer filled with messages
  Option type: unsigned int
**CU_JIT_ERROR_LOG_BUFFER**  Pointer to a buffer in which to print any log messages from PTXAS that reflect errors (the buffer size is specified via option **CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES**)
   Option type: char*

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

**CU_JIT_OPTIMIZATION_LEVEL**  Level of optimizations to apply to generated code (0 - 4), with 4 being the default and highest level of optimizations.
   Option type: unsigned int

**CU_JIT_TARGET_FROM_CUCONTEXT**  No option value required. Determines the target based on the current attached context (default)
   Option type: No option value needed

**CU_JIT_TARGET**  Target is chosen based on supplied **CUjit_target_enum**.
   Option type: unsigned int for enumerated type **CUjit_target_enum**

**CU_JIT_FALLBACK_STRATEGY**  Specifies choice of fallback strategy if matching cubin is not found. Choice is based on supplied **CUjit_fallback_enum**.
   Option type: unsigned int for enumerated type **CUjit_fallback_enum**

### 4.27.3.16  enum CUjit_target_enum

Online compilation targets

**Enumerator:**

- **CU_TARGET_COMPUTE_10**  Compute device class 1.0
- **CU_TARGET_COMPUTE_11**  Compute device class 1.1
- **CU_TARGET_COMPUTE_12**  Compute device class 1.2
- **CU_TARGET_COMPUTE_13**  Compute device class 1.3
- **CU_TARGET_COMPUTE_20**  Compute device class 2.0
- **CU_TARGET_COMPUTE_21**  Compute device class 2.1

### 4.27.3.17  enum CUlimit_enum

Limits

**Enumerator:**

- **CU_LIMIT_STACK_SIZE**  GPU thread stack size
- **CU_LIMIT_PRINTF_FIFO_SIZE**  GPU printf FIFO size
- **CU_LIMIT_MALLOC_HEAP_SIZE**  GPU malloc heap size
4.27.3.18  enum CUmemorytype_enum

Memory types

Enumerator:

- \texttt{CU_MEMORYTYPE_HOST}  Host memory
- \texttt{CU_MEMORYTYPE_DEVICE}  Device memory
- \texttt{CU_MEMORYTYPE_ARRAY}  Array memory
- \texttt{CU_MEMORYTYPE_UNIFIED}  Unified device or host memory

4.27.3.19  enum CUpointer_attribute_enum

Pointer information

Enumerator:

- \texttt{CU_POINTER_ATTRIBUTE_CONTEXT}  The \texttt{CUcontext} on which a pointer was allocated or registered
- \texttt{CU_POINTER_ATTRIBUTE_MEMORY_TYPE}  The \texttt{CUmemorytype} describing the physical location of a pointer
- \texttt{CU_POINTER_ATTRIBUTE_DEVICE_POINTER}  The address at which a pointer’s memory may be accessed on the device
- \texttt{CU_POINTER_ATTRIBUTE_HOST_POINTER}  The address at which a pointer’s memory may be accessed on the host
4.28 Initialization

Functions

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

4.28.1 Detailed Description

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

4.28.2 Function Documentation

4.28.2.1 CUresult cuInit (unsigned int Flags)

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

Parameters:

Flags - Initialization flag for CUDA.

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Note:

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

Functions

- CUresult cuDriverGetVersion (int *driverVersion)
  
  Returns the CUDA driver version.

4.29.1 Detailed Description

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

4.29.2 Function Documentation

4.29.2.1 CUresult cuDriverGetVersion (int *driverVersion)

Returns in *driverVersion the version number of the installed CUDA driver. This function automatically returns CUDA_ERROR_INVALID_VALUE if the driverVersion argument is NULL.

Parameters:

  driverVersion - Returns the CUDA driver version

Returns:

  CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE

Note:

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

Functions

- CUresult cuDeviceComputeCapability (int *major, int *minor, CUdevice dev)
  
  Returns the compute capability of the device.

- CUresult cuDeviceGet (CUdevice *device, int ordinal)
  
  Returns a handle to a compute device.

- CUresult cuDeviceGetAttribute (int *pi, CUdevice_attribute attrib, CUdevice dev)
  
  Returns information about the device.

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

- CUresult cuDeviceGetName (char *name, int len, CUdevice dev)
  
  Returns an identifier string for the device.

- CUresult cuDeviceGetProperties (CDevprop *prop, CUdevice dev)
  
  Returns properties for a selected device.

- CUresult cuDeviceTotalMem (size_t *bytes, CUdevice dev)
  
  Returns the total amount of memory on the device.

4.30.1 Detailed Description

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

4.30.2 Function Documentation

4.30.2.1 CUresult cuDeviceComputeCapability (int *major, int *minor, CUdevice dev)

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

Parameters:

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

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Note:

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

- cuDeviceGetAttribute
- cuDeviceGetCount
- cuDeviceGetName
- cuDeviceGet
- cuDeviceGetProperties
- cuDeviceTotalMem

### 4.30.2.2 CUresult cuDeviceGet (CUdevice * device, int ordinal)

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

**Parameters:**

- `device` - Returned device handle
- `ordinal` - Device number to get handle for

**Returns:**

- `CUDA_SUCCESS`
- `CUDA_ERROR_DEINITIALIZED`
- `CUDA_ERROR_NOT_INITIALIZED`
- `CUDA_ERROR_INVALID_CONTEXT`
- `CUDA_ERROR_INVALID_VALUE`
- `CUDA_ERROR_INVALID_DEVICE`

**Note:**

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

See also:

- cuDeviceComputeCapability
- cuDeviceGetAttribute
- cuDeviceGetCount
- cuDeviceGetName
- cuDeviceGetProperties
- cuDeviceTotalMem

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

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

- `CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK`: Maximum number of threads per block;
- `CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X`: Maximum x-dimension of a block;
- `CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y`: Maximum y-dimension of a block;
- `CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z`: Maximum z-dimension of a block;
- `CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X`: Maximum x-dimension of a grid;
- `CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y`: Maximum y-dimension of a grid;
- `CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z`: Maximum z-dimension of a grid;
- `CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK`: Maximum amount of shared memory available to a thread block in bytes; this amount is shared by all thread blocks simultaneously resident on a multiprocessor;
- `CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY`: Memory available on device for `__constant_` variables in a CUDA C kernel in bytes;
- `CU_DEVICE_ATTRIBUTE_WARP_SIZE`: Warp size in threads;
- `CU_DEVICE_ATTRIBUTE_MAX_PITCH`: Maximum pitch in bytes allowed by the memory copy functions that involve memory regions allocated through `cuMemAllocPitch();`
• **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH**: Maximum 1D texture width;

• **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_WIDTH**: Maximum 2D texture width;

• **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_HEIGHT**: Maximum 2D texture height;

• **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH**: Maximum 3D texture width;

• **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT**: Maximum 3D texture height;

• **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH**: Maximum 3D texture depth;

• **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH**: Maximum 1D layered texture width;

• **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS**: Maximum layers in a 1D layered texture;

• **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH**: Maximum 2D layered texture width;

• **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT**: Maximum 2D layered texture height;

• **CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS**: Maximum layers in a 2D layered texture;

• **CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK**: Maximum number of 32-bit registers available to a thread block; this number is shared by all thread blocks simultaneously resident on a multiprocessor;

• **CU_DEVICE_ATTRIBUTE_CLOCK_RATE**: Peak clock frequency in kilohertz;

• **CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT**: Alignment requirement; texture base addresses aligned to textureAlign bytes do not need an offset applied to texture fetches;

• **CU_DEVICE_ATTRIBUTE_GPU_OVERLAP**: 1 if the device can concurrently copy memory between host and device while executing a kernel, or 0 if not;

• **CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT**: Number of multiprocessors on the device;

• **CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT**: 1 if there is a run time limit for kernels executed on the device, or 0 if not;

• **CU_DEVICE_ATTRIBUTE_INTEGRATED**: 1 if the device is integrated with the memory subsystem, or 0 if not;

• **CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY**: 1 if the device can map host memory into the CUDA address space, or 0 if not;

• **CU_DEVICE_ATTRIBUTE_COMPUTE_MODE**: Compute mode that device is currently in. Available modes are as follows:
  
  – **CU_COMPUTEMODE_DEFAULT**: Default mode - Device is not restricted and can have multiple CUDA contexts present at a single time.

  – **CU_COMPUTEMODE_EXCLUSIVE**: Compute-exclusive mode - Device can have only one CUDA context present on it at a time.

  – **CU_COMPUTEMODE_PROHIBITED**: Compute-prohibited mode - Device is prohibited from creating new CUDA contexts.

  – **CU_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_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;

• **CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING**: 1 if the device shares a unified address space with the host, or 0 if not;

**Parameters:**

\[
pi - Returned device attribute value \\
attrib - Device attribute to query \\
dev - Device handle
\]

**Returns:**

CUDASUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

**Note:**

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

**See also:**

cuDeviceComputeCapability, cuDeviceGetCount, cuDeviceGetName, cuDeviceGet, cuDeviceGetProperties, cuDeviceTotalMem

### 4.30.2.4 CUresult cuDeviceGetCount (int *count)

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

**Parameters:**

\[
\text{count} - Returned number of compute-capable devices
\]
Returns:

\[
\text{CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE}
\]

Note:

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

See also:

- cuDeviceComputeCapability
- cuDeviceGetAttribute
- cuDeviceGetName
- cuDeviceGet
- cuDeviceGetProperties
- cuDeviceTotalMem

4.30.2.5 \texttt{CUresult cuDeviceGetName (char * name, int len, CUdevice dev)}

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

Parameters:

- \texttt{name} - Returned identifier string for the device
- \texttt{len} - Maximum length of string to store in \texttt{name}
- \texttt{dev} - Device to get identifier string for

Returns:

\[
\text{CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_INVALID\_DEVICE}
\]

Note:

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

See also:

- cuDeviceComputeCapability
- cuDeviceGetAttribute
- cuDeviceGetCount
- cuDeviceGet
- cuDeviceGetProperties
- cuDeviceTotalMem

4.30.2.6 \texttt{CUresult cuDeviceGetProperties (CUdevprop * prop, CUdevice dev)}

Returns in \texttt{prop} the properties of device \texttt{dev}. The \texttt{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.

Parameters:

- `prop` - Returned properties of device
- `dev` - Device to get properties for

Returns:

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Note:

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

See also:

cuDeviceComputeCapability, cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGetName, cuDeviceGet, cuDeviceTotalMem

4.30.2.7 CUresult cuDeviceTotalMem (size_t ∗ bytes, CUdevice dev)

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

Parameters:

- `bytes` - Returned memory available on device in bytes
- `dev` - Device handle

Returns:

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE
Note:

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

See also:

cuDeviceComputeCapability, cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGetName, cuDeviceGet, cuDeviceGetProperties,
4.31 Context Management

Modules

- Context Management [DEPRECATED]

Functions

- CUresult cuCtxCreate (CUcontext *pctx, unsigned int flags, CUdevice dev)
  
  Create a CUDA context.

- CUresult cuCtxDestroy (CUcontext ctx)
  
  Destroy a CUDA context.

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

- CUresult cuCtxGetCacheConfig (CUfunc_cache *pconfig)
  
  Returns the preferred cache configuration for the current context.

- CUresult cuCtxGetCurrent (CUcontext *pctx)
  
  Returns the CUDA context bound to the calling CPU thread.

- CUresult cuCtxGetDevice (CUdevice *device)
  
  Returns the device ID for the current context.

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

- CUresult cuCtxPopCurrent (CUcontext *pctx)
  
  Pops the current CUDA context from the current CPU thread.

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

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

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

- CUresult cuCtxSetLimit (CUlimit limit, size_t value)
  
  Set resource limits.

- CUresult cuCtxSynchronize (void)
  
  Block for a context's tasks to complete.
4.31.1 Detailed Description

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

4.31.2 Function Documentation

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

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

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

- **CU_CTX_SCHED_AUTO**: The default value if the flags parameter is zero, uses a heuristic based on the number of active CUDA contexts in the process \( C \) and the number of logical processors in the system \( P \). If \( C > P \), then CUDA will yield to other OS threads when waiting for the GPU, otherwise CUDA will not yield while waiting for results and actively spin on the processor.

- **CU_CTX_SCHED_SPIN**: Instruct CUDA to actively spin when waiting for results from the GPU. This can decrease latency when waiting for the GPU, but may lower the performance of CPU threads if they are performing work in parallel with the CUDA thread.

- **CU_CTX_SCHED_YIELD**: Instruct CUDA to yield its thread when waiting for results from the GPU. This can increase latency when waiting for the GPU, but can increase the performance of CPU threads performing work in parallel with the GPU.

- **CU_CTX_SCHED_BLOCKING_SYNC**: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the GPU to finish work.

- **CU_CTX_BLOCKING_SYNC**: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the GPU to finish work.

**Deprecated**

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

- **CU_CTX_MAP_HOST**: Instruct CUDA to support mapped pinned allocations. This flag must be set in order to allocate pinned host memory that is accessible to the GPU.

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

**Note to Linux users:**

Context creation will fail with **CUDA_ERROR_UNKNOWN** if the compute mode of the device is **CU_COMPUTEMODE_PROHIBITED**. Similarly, context creation will also fail with **CUDA_ERROR_UNKNOWN** if the compute mode for the device is set to **CU_COMPUTEMODE_EXCLUSIVE** and there is already an active context on the device. The function cuDeviceGetAttribute() can be used with **CU_DEVICE_ATTRIBUTE_COMPUTEMODE** 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.
4.31 Context Management

Parameters:

\[ pctx \] - Returned context handle of the new context
\[ flags \] - Context creation flags
\[ dev \] - Device to create context on

Returns:

\[ \text{CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_DEVICE, CUDA\_ERROR\_INVALID\_VALUE, CUDA\_ERROR\_OUT\_OF\_MEMORY, CUDA\_ERROR\_UNKNOWN} \]

Note:

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

See also:

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

4.31.2.2 CUresult cuCtxDestroy (CUcontext ctx)

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

If \[ ctx \] is current to the calling thread then \[ ctx \] will also be popped from the current thread’s context stack (as though \[ cuCtxPopCurrent() \] were called). If \[ ctx \] is current to other threads, then \[ ctx \] will remain current to those threads, and attempting to access \[ ctx \] from those threads will result in the error \[ CUDA\_ERROR\_CONTEXT\_IS\_DESTROYED \].

Parameters:

\[ ctx \] - Context to destroy

Returns:

\[ \text{CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_DEVICE, CUDA\_ERROR\_INVALID\_VALUE} \]

Note:

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

See also:

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

4.31.2.3 CUresult cuCtxGetApiVersion (CUcontext ctx, unsigned int ∗ version)

Returns the API version used to create \[ ctx \] in \[ version \]. If \[ ctx \] is NULL, returns the API version used to create the currently bound context.

This will return the API version used to create a context (for example, 3010 or 3020), which library developers can use to direct callers to a specific API version. Note that this API version may not be the same as returned by cuDriverGetVersion.
Parameters:

\( \text{ctx} \) - Context to check

\( \text{version} \) - Pointer to version

Returns:

\( \text{CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_UNKNOWN} \)

Note:

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

4.31.2.4 \textbf{CUresult cuCtxGetCacheConfig (CUfunc_cache \( \ast \) pconfig)}

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

This will return a \( pconfig \) of \textbf{CU_FUNC_CACHE_PREFER_NONE} on devices where the size of the L1 cache and shared memory are fixed.

The supported cache configurations are:

\begin{itemize}
\item \textbf{CU_FUNC_CACHE_PREFER_NONE}: no preference for shared memory or L1 (default)
\item \textbf{CU_FUNC_CACHE_PREFER_SHARED}: prefer larger shared memory and smaller L1 cache
\item \textbf{CU_FUNC_CACHE_PREFER_L1}: prefer larger L1 cache and smaller shared memory
\end{itemize}

Parameters:

\( pconfig \) - Returned cache configuration

Returns:

\( \text{CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE} \)

Note:

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize, cuFuncSetCacheConfig
4.31 Context Management

4.31.2.5 `CUresult cuCtxGetCurrent (CUcontext ∗ pctx)`

Returns in ∗pctx the CUDA context bound to the calling CPU thread. If no context is bound to the calling CPU thread then ∗pctx is set to NULL and CUDA_SUCCESS is returned.

Parameters:

pctx - Returned context handle

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,

Note:

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

See also:

cuCtxSetCurrent, cuCtxCreate, cuCtxDestroy

4.31.2.6 `CUresult cuCtxGetDevice (CUdevice ∗ device)`

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

Parameters:

device - Returned device ID for the current context

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE,

Note:

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

4.31.2.7 `CUresult cuCtxGetLimit (size_t ∗ pvalue, CUlimit limit)`

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

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

Parameters:

limit - Limit to query
pvalue - Returned size in bytes of limit

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_UNSUPPORTED_LIMIT

Note:

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

See also:

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

4.31.2.8 CUresult cuCtxPopCurrent (CUcontext *pctx)

Pops the current CUDA context from the CPU thread and passes back the old context handle in *pctx. That context may then be made current to a different CPU thread by calling cuCtxPushCurrent(). If a context was current to the CPU thread before cuCtxCreate() or cuCtxPushCurrent() was called, this function makes that context current to the CPU thread again.

Parameters:

pctx - Returned new context handle

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Note:

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit,
cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

4.31.2.9 CUresult cuCtxPushCurrent (CUcontext ctx)

Pushes the given context ctx onto the CPU thread’s stack of current contexts. The specified context becomes the CPU thread’s current context, so all CUDA functions that operate on the current context are affected. The previous current context may be made current again by calling cuCtxDestroy() or cuCtxPopCurrent().

Parameters:

ctx - Context to push

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE
Note:
Note that this function may also return error codes from previous, asynchronous launches.

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

4.31.2.10 CUresult cuCtxSetCacheConfig (CUfunc_cache config)

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

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

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

The supported cache configurations are:

- **CU_FUNC_CACHE_PREFER_NONE**: no preference for shared memory or L1 (default)
- **CU_FUNC_CACHE_PREFER_SHARED**: prefer larger shared memory and smaller L1 cache
- **CU_FUNC_CACHE_PREFER_L1**: prefer larger L1 cache and smaller shared memory

Parameters:

`config` - Requested cache configuration

Returns:

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

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

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

4.31.2.11 CUresult cuCtxSetCurrent (CUcontext ctx)

Binds the specified CUDA context to the calling CPU thread. If `ctx` is NULL then the CUDA context previously bound to the calling CPU thread is unbound and CUDA_SUCCESS is returned.

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

\( \text{ctx} \) - Context to bind to the calling CPU thread

Returns:

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Note:

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

See also:

- cuCtxGetCurrent, cuCtxCreate, cuCtxDestroy

4.31.2.12 CUresult cuCtxSetLimit (CUlimit limit, size_t value)

Setting \( \text{limit} \) to \( \text{value} \) 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 \( \text{cuCtxGetLimit()} \) to find out exactly what the limit has been set to.

Setting each \( \text{CUlimit} \) has its own specific restrictions, so each is discussed here.

- \( \text{CU_LIMIT_STACK_SIZE} \) controls the stack size of each GPU thread. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error CUDA_ERROR_UNSUPPORTED_LIMIT being returned.

- \( \text{CU_LIMIT_PRINTF_FIFO_SIZE} \) controls the size of the FIFO used by the printf() device system call. Setting \( \text{CU_LIMIT_PRINTF_FIFO_SIZE} \) must be performed before launching any kernel that uses the printf() device system call, otherwise CUDA_ERROR_INVALID_VALUE will be returned. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error CUDA_ERROR_UNSUPPORTED_LIMIT being returned.

- \( \text{CU_LIMIT_MALLOC_HEAP_SIZE} \) controls the size of the heap used by the malloc() and free() device system calls. Setting \( \text{CU_LIMIT_MALLOC_HEAP_SIZE} \) must be performed before launching any kernel that uses the malloc() or free() device system calls, otherwise CUDA_ERROR_INVALID_VALUE will be returned. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error CUDA_ERROR_UNSUPPORTED_LIMIT being returned.

Parameters:

- \( \text{limit} \) - Limit to set
- \( \text{value} \) - Size in bytes of limit

Returns:

- CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_UNSUPPORTED_LIMIT

Note:

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

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSynchronize

4.31.2.13 CUresult cuCtxSynchronize (void)

Blocks until the device has completed all preceding requested tasks. cuCtxSynchronize() returns an error if one of the preceding tasks failed. If the context was created with the CU_CTX_SCHED_BLOCKING_SYNC flag, the CPU thread will block until the GPU context has finished its work.

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Note:

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent cuCtxSetCacheConfig, cuCtxSetLimit
4.32 Context Management [DEPRECATED]

Functions

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

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

4.32.1 Detailed Description

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

4.32.2 Function Documentation

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

*Deprecated*

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

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

Currently, the **flags** parameter must be 0.

**Parameters:**

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

**Returns:**

- **CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE**

**Note:**

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

**See also:**

- cuCtxCreate, cuCtxDestroy, cuCtxDetach, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize
4.32 Context Management [DEPRECATED]

4.32.2 CUresult cuCtxDetach (CUcontext ctx)

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.

Parameters:

ctx - Context to destroy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Note:

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, 
cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize
4.33 Module Management

Functions

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

- **CUresult cuModuleGetGlobal (CUdeviceptr *dptr, size_t *bytes, CUmodule hmod, const char *name)**
  Returns a global pointer from a module.

- **CUresult cuModuleGetSurfRef (CUsurfref *pSurfRef, CUmodule hmod, const char *name)**
  Returns a handle to a surface reference.

- **CUresult cuModuleGetTexRef (CUtexref *pTexRef, CUmodule hmod, const char *name)**
  Returns a handle to a texture reference.

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

- **CUresult cuModuleLoadData (CUmodule *module, const void *image)**
  Load a module’s data.

- **CUresult cuModuleLoadDataEx (CUmodule *module, const void *image, unsigned int numOptions, CUjit_option *options, void **optionValues)**
  Load a module’s data with options.

- **CUresult cuModuleLoadFatBinary (CUmodule *module, const void *fatCubin)**
  Load a module’s data.

- **CUresult cuModuleUnload (CUmodule hmod)**
  Unloads a module.

4.33.1 Detailed Description

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

4.33.2 Function Documentation

4.33.2.1 **CUresult cuModuleGetFunction (CUfunction *hfunc, CUmodule hmod, const char *name)**

Returns in *hfunc the handle of the function of name *name located in module *hmod. If no function of that name exists, cuModuleGetFunction() returns CUDA_ERROR_NOT_FOUND.

Parameters:

  - **hfunc** - Returned function handle
  - **hmod** - Module to retrieve function from
  - **name** - Name of function to retrieve

Generated for NVIDIA CUDA Library by Doxygen
4.33 Module Management

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND

Note:

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

See also:

cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadData, cuModuleLoadDataEx, cuModuleLoadFatBinary, cuModuleUnload

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

Returns in *dptr and *bytes the base pointer and size of the global of name name located in module hmod. If no variable of that name exists, cuModuleGetGlobal() returns CUDA_ERROR_NOT_FOUND. Both parameters dptr and bytes are optional. If one of them is NULL, it is ignored.

Parameters:

dptr - Returned global device pointer
bytes - Returned global size in bytes
hmod - Module to retrieve global from
name - Name of global to retrieve

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND

Note:

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

See also:

cuModuleGetFunction, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadData, cuModuleLoadDataEx, cuModuleLoadFatBinary, cuModuleUnload

4.33.2.3 CUresult cuModuleGetSurfRef (CUsurfref *pSurfRef, CUmodule hmod, const char *name)

Returns in *pSurfRef the handle of the surface reference of name name in the module hmod. If no surface reference of that name exists, cuModuleGetSurfRef() returns CUDA_ERROR_NOT_FOUND.

Parameters:

pSurfRef - Returned surface reference
hmod - Module to retrieve surface reference from
name - Name of surface reference to retrieve
Returns:

\[
\begin{align*}
\text{CUDA_SUCCESS}, & \quad \text{CUDA_ERROR_DEINITIALIZED}, \quad \text{CUDA_ERROR_NOT_INITIALIZED}, \quad \text{CUDA_ERROR_INVALID_CONTEXT}, \quad \text{CUDA_ERROR_INVALID_VALUE}, \quad \text{CUDA_ERROR_NOT_FOUND}
\end{align*}
\]

Note:

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

See also:

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

4.33.2.4 CUresult cuModuleGetTexRef (CUtexref \textbf{* pTexRef}, CUmodule \textbf{hmod}, const char \textbf{* name})

Returns in \textbf{* pTexRef} the handle of the texture reference of name \textbf{name} in the module \textbf{hmod}. If no texture reference of that name exists, \textbf{cuModuleGetTexRef()} returns \textbf{CUDA_ERROR_NOT_FOUND}. This texture reference handle should not be destroyed, since it will be destroyed when the module is unloaded.

Parameters:

\textbf{pTexRef} - Returned texture reference

\textbf{hmod} - Module to retrieve texture reference from

\textbf{name} - Name of texture reference to retrieve

Returns:

\[
\begin{align*}
\text{CUDA_SUCCESS}, & \quad \text{CUDA_ERROR_DEINITIALIZED}, \quad \text{CUDA_ERROR_NOT_INITIALIZED}, \quad \text{CUDA_ERROR_INVALID_CONTEXT}, \quad \text{CUDA_ERROR_INVALID_VALUE}, \quad \text{CUDA_ERROR_NOT_FOUND}
\end{align*}
\]

Note:

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

See also:

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetSurfRef, cuModuleLoad, cuModuleLoadData, cuModuleLoadDataEx, cuModuleLoadFatBinary, cuModuleUnload

4.33.2.5 CUresult cuModuleLoad (CUmodule \textbf{* module}, const char \textbf{* fname})

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

Parameters:

\textbf{module} - Returned module

\textbf{fname} - Filename of module to load
Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_FILE_NOT_FOUND, CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Note:

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

See also:

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoadData, cuModuleLoadDataEx, cuModuleLoadFatBinary, cuModuleUnload

4.33.2.6 CUresult cuModuleLoadData (CUmodule * module, const void * image)

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

Parameters:

module - Returned module
image - Module data to load

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Note:

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

See also:

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadDataEx, cuModuleLoadFatBinary, cuModuleUnload

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

Takes a pointer image and loads the corresponding module module into the current context. The pointer may be obtained by mapping a cubin or PTX or fatbin file, passing a cubin or PTX or fatbin file as a NULL-terminated text string, or incorporating a cubin or fatbin object into the executable resources and using operating system calls such as Windows FindResource() to obtain the pointer. Options are passed as an array via options and any corresponding parameters are passed in optionValues. The number of total options is supplied via numOptions. Any outputs will be returned via optionValues. Supported options are (types for the option values are specified in parentheses after the option name):
• **CU_JIT_MAX_REGISTERS**: (unsigned int) input specifies the maximum number of registers per thread;

• **CU_JIT_THREADS_PER_BLOCK**: (unsigned int) input specifies number of threads per block to target compilation for; output returns the number of threads the compiler actually targeted;

• **CU_JIT_WALL_TIME**: (float) output returns the float value of wall clock time, in milliseconds, spent compiling the PTX code;

• **CU_JIT_INFO_LOG_BUFFER**: (char*) input is a pointer to a buffer in which to print any informational log messages from PTX assembly (the buffer size is specified via option **CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES**);

• **CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES**: (unsigned int) input is the size in bytes of the buffer; output is the number of bytes filled with messages;

• **CU_JIT_ERROR_LOG_BUFFER**: (char*) input is a pointer to a buffer in which to print any error log messages from PTX assembly (the buffer size is specified via option **CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES**);

• **CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES**: (unsigned int) input is the size in bytes of the buffer; output is the number of bytes filled with messages;

• **CU_JIT_OPTIMIZATION_LEVEL**: (unsigned int) input is the level of optimization to apply to generated code (0 - 4), with 4 being the default and highest level;

• **CU_JIT_TARGET_FROM_CUCONTEXT**: (No option value) causes compilation target to be determined based on current attached context (default);

• **CU_JIT_TARGET**: (unsigned int for enumerated type **CUjit_target_enum**) input is the compilation target based on supplied **CUjit_target_enum**; possible values are:

  - **CU_TARGET_COMPUTE_10**
  - **CU_TARGET_COMPUTE_11**
  - **CU_TARGET_COMPUTE_12**
  - **CU_TARGET_COMPUTE_13**
  - **CU_TARGET_COMPUTE_20**

• **CU_JIT_FALLBACK_STRATEGY**: (unsigned int for enumerated type **CUjit_fallback_enum**) chooses fallback strategy if matching cubin is not found; possible values are:

  - **CU_PREFER_PTX**
  - **CU_PREFER_BINARY**

**Parameters:**

- **module** - Returned module
- **image** - Module data to load
- **numOptions** - Number of options
- **options** - Options for JIT
- **optionValues** - Option values for JIT

**Returns:**

- **CUDA_SUCCESS**
- **CUDA_ERROR_DEINITIALIZED**
- **CUDA_ERROR_NOT_INITIALIZED**
- **CUDA_ERROR_INVALID_CONTEXT**
- **CUDA_ERROR_INVALID_VALUE**
- **CUDA_ERROR_OUT_OF_MEMORY**
- **CUDA_ERROR_NO_BINARY_FOR_GPU**
- **CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND**
- **CUDA_ERROR_SHARED_OBJECT_INIT_FAILED**
Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:

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

### 4.33.2.8 CUresult cuModuleLoadFatBinary (CUmodule * module, const void * fatCubin)

Takes a pointer `fatCubin` and loads the corresponding module `module` into the current context. The pointer represents a `fat binary` object, which is a collection of different `cubin` and/or `PTX` files, all representing the same device code, but compiled and optimized for different architectures.

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

**Parameters:**

- `module` - Returned module
- `fatCubin` - Fat binary to load

**Returns:**

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_NO_BINARY_FOR_GPU, CUDA_ERROR_-SHARED_OBJECT_SYMBOL_NOT_FOUND, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

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

See also:

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

### 4.33.2.9 CUresult cuModuleUnload (CUmodule hmod)

Unloads a module `hmod` from the current context.

**Parameters:**

- `hmod` - Module to unload

**Returns:**

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

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

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadData, cuModuleLoadDataEx, cuModuleLoadFatBinary
4.34 Memory Management

Functions

• CUresult cuArray3DCreate (CUarray *pHandle, const CUDA_ARRAY3D_DESCRIPTOR *pAllocateArray)
  Creates a 3D CUDA array.

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

• CUresult cuArrayCreate (CUarray *pHandle, const CUDA_ARRAY_DESCRIPTOR *pAllocateArray)
  Creates a 1D or 2D CUDA array.

• CUresult cuArrayDestroy (CUarray hArray)
  Destroys a CUDA array.

• CUresult cuArrayGetDescriptor (CUDA_ARRAY_DESCRIPTOR *pArrayDescriptor, CUarray hArray)
  Get a 1D or 2D CUDA array descriptor.

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

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

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

• CUresult cuMemcpy (CUdeviceptr dst, CUdeviceptr src, size_t ByteCount)
  Copies memory.

• CUresult cuMemcpy2D (const CUDA_MEMCPY2D *pCopy)
  Copies memory for 2D arrays.

• CUresult cuMemcpy2DAsync (const CUDA_MEMCPY2D *pCopy, CUstream hStream)
  Copies memory for 2D arrays.

• CUresult cuMemcpy2DUnaligned (const CUDA_MEMCPY2D *pCopy)
  Copies memory for 2D arrays.

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

• CUresult cuMemcpy3DAsync (const CUDA_MEMCPY3D *pCopy, CUstream hStream)
  Copies memory for 3D arrays.

• CUresult cuMemcpy3DPeer (const CUDA_MEMCPY3D_PEER *pCopy)
  Copies memory between contexts.
• CUresult cuMemcpy3DPeerAsync (const CUDA_MEMCPY3D_PEER *pCopy, CUstream hStream)
  Copies memory between contexts asynchronously.

• CUresult cuMemcpyAsync (CUdeviceptr dst, CUdeviceptr src, size_t ByteCount, CUstream hStream)
  Copies memory asynchronously.

• CUresult cuMemcpyAtoA (CUarray dstArray, size_t dstOffset, CUarray srcArray, size_t srcOffset, size_t ByteCount)
  Copies memory from Array to Array.

• CUresult cuMemcpyAtoD (CUdeviceptr dstDevice, CUarray srcArray, size_t srcOffset, size_t ByteCount)
  Copies memory from Array to Device.

• CUresult cuMemcpyAtoH (void *dstHost, CUarray srcArray, size_t srcOffset, size_t ByteCount)
  Copies memory from Array to Host.

• CUresult cuMemcpyAtoHAsync (void *dstHost, CUarray srcArray, size_t srcOffset, size_t ByteCount, CUstream hStream)
  Copies memory from Array to Host.

• CUresult cuMemcpyDtoA (CUarray dstArray, size_t dstOffset, CUdeviceptr srcDevice, size_t ByteCount)
  Copies memory from Device to Array.

• CUresult cuMemcpyDtoD (CUdeviceptr dstDevice, CUdeviceptr srcDevice, size_t ByteCount)
  Copies memory from Device to Device.

• CUresult cuMemcpyDtoDAsync (CUdeviceptr dstDevice, CUdeviceptr srcDevice, size_t ByteCount, CUstream hStream)
  Copies memory from Device to Device.

• CUresult cuMemcpyDtoH (void *dstHost, CUdeviceptr srcDevice, size_t ByteCount)
  Copies memory from Device to Host.

• CUresult cuMemcpyDtoHAsync (void *dstHost, CUdeviceptr srcDevice, size_t ByteCount, CUstream hStream)
  Copies memory from Device to Host.

• CUresult cuMemcpyHtoA (CUarray dstArray, size_t dstOffset, const void *srcHost, size_t ByteCount)
  Copies memory from Host to Array.

• CUresult cuMemcpyHtoAAasync (CUarray dstArray, size_t dstOffset, const void *srcHost, size_t ByteCount, CUstream hStream)
  Copies memory from Host to Array.

• CUresult cuMemcpyHtoD (CUdeviceptr dstDevice, const void *srcHost, size_t ByteCount)
  Copies memory from Host to Device.

• CUresult cuMemcpyHtoDAasync (CUdeviceptr dstDevice, const void *srcHost, size_t ByteCount, CUstream hStream)
Copies memory from Host to Device.

- CUresult cuMemcpyPeer (CUdeviceptr dstDevice, CUcontext dstContext, CUdeviceptr srcDevice, CUcontext srcContext, size_t ByteCount)
  Copies device memory between two contexts.

- CUresult cuMemcpyPeerAsync (CUdeviceptr dstDevice, CUcontext dstContext, CUdeviceptr srcDevice, CUcontext srcContext, size_t ByteCount, CUsream hStream)
  Copies device memory between two contexts asynchronously.

- CUresult cuMemFree (CUdeviceptr dptr)
  Frees device memory.

- CUresult cuMemFreeHost (void *p)
  Frees page-locked host memory.

- CUresult cuMemGetAddressRange (CUdeviceptr *pbase, size_t *psize, CUdeviceptr dptr)
  Get information on memory allocations.

- CUresult cuMemGetInfo (size_t *free, size_t *total)
  Gets free and total memory.

- CUresult cuMemHostAlloc (void **pp, size_t bytesize, unsigned int Flags)
  Allocates page-locked host memory.

- CUresult cuMemHostGetDevicePointer (CUdeviceptr *pdptr, void *p, unsigned int Flags)
  Passes back device pointer of mapped pinned memory.

- CUresult cuMemHostGetFlags (unsigned int *pFlags, void *p)
  Passes back flags that were used for a pinned allocation.

- CUresult cuMemHostRegister (void *p, size_t bytesize, unsigned int Flags)
  Registers an existing host memory range for use by CUDA.

- CUresult cuMemHostUnregister (void *p)
  Unregisters a memory range that was registered with cuMemHostRegister().

- CUresult cuMemsetD16 (CUdeviceptr dstDevice, unsigned short us, size_t N)
  Initializes device memory.

- CUresult cuMemsetD16Async (CUdeviceptr dstDevice, unsigned short us, size_t N, CUsream hStream)
  Sets device memory.

- CUresult cuMemsetD2D16 (CUdeviceptr dstDevice, size_t dstPitch, unsigned short us, size_t Width, size_t Height)
  Initializes device memory.

- CUresult cuMemsetD2D16Async (CUdeviceptr dstDevice, size_t dstPitch, unsigned short us, size_t Width, size_t Height, CUsream hStream)
  Sets device memory.
• **CUresult cuMemsetD2D32** (CUdeviceptr dstDevice, size_t dstPitch, unsigned int ui, size_t Width, size_t Height)
  
  *Initializes device memory.*

• **CUresult cuMemsetD2D32Async** (CUdeviceptr dstDevice, size_t dstPitch, unsigned int ui, size_t Width, size_t Height, CUstream hStream)
  
  *Sets device memory.*

• **CUresult cuMemsetD2D8** (CUdeviceptr dstDevice, size_t dstPitch, unsigned char uc, size_t Width, size_t Height)
  
  *Initializes device memory.*

• **CUresult cuMemsetD2D8Async** (CUdeviceptr dstDevice, size_t dstPitch, unsigned char uc, size_t Width, size_t Height, CUstream hStream)
  
  *Sets device memory.*

• **CUresult cuMemsetD32** (CUdeviceptr dstDevice, unsigned int ui, size_t N)
  
  *Initializes device memory.*

• **CUresult cuMemsetD32Async** (CUdeviceptr dstDevice, unsigned int ui, size_t N, CUstream hStream)
  
  *Sets device memory.*

• **CUresult cuMemsetD8** (CUdeviceptr dstDevice, unsigned char uc, size_t N)
  
  *Initializes device memory.*

• **CUresult cuMemsetD8Async** (CUdeviceptr dstDevice, unsigned char uc, size_t N, CUstream hStream)
  
  *Sets device memory.*

### 4.34.1 Detailed Description

This section describes the memory management functions of the low-level CUDA driver application programming interface.

### 4.34.2 Function Documentation

#### 4.34.2.1 **CUresult cuArray3DCreate** (CUarray ∗ pHandle, const CUDA_ARRAY3D_DESCRIPTOR ∗ pAllocateArray)

Creates a CUDA array according to the CUDA_ARRAY3D_DESCRIPTOR structure pAllocateArray and returns a handle to the new CUDA array in ∗ pHandle. The CUDA_ARRAY3D_DESCRIPTOR is defined as:

```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 CUDA array is one-dimensional if height and depth are 0, two-dimensional if depth is 0, and three-dimensional otherwise; If the CUDA_ARRAY3D_LAYERED flag is set, then the CUDA array is a collection of layers, where Depth indicates the number of layers. Each layer is a 1D array if Height is 0, and a 2D array otherwise.

• Format specifies the format of the elements; CUarray_format is defined as:

```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.

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;
```

Parameters:

  pHandle - Returned array
  pAllocateArray - 3D array descriptor
Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:


4.34.2.2 CUresult cuArray3DGetDescriptor (CUDA_ARRAY3D_DESCRIPTOR *pArrayDescriptor, CUarray hArray)

Returns in *pArrayDescriptor a descriptor containing information on the format and dimensions of the CUDA array hArray. It is useful for subroutines that have been passed a CUDA array, but need to know the CUDA array parameters for validation or other purposes.

This function may be called on 1D and 2D arrays, in which case the Height and/or Depth members of the descriptor struct will be set to 0.

Parameters:

pArrayDescriptor - Returned 3D array descriptor
hArray - 3D array to get descriptor of

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

4.34 Memory Management

4.34.2.3 CUresult cuArrayCreate (CUarray *pHandle, const CUDA_ARRAY_DESCRIPTOR *pAllocateArray)

Creates a CUDA array according to the CUDA_ARRAY_DESCRIPTOR structure pAllocateArray and returns a handle to the new CUDA array in *pHandle. The CUDA_ARRAY_DESCRIPTOR is defined as:

```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;
```

Parameters:
- `pHandle`: Returned array
- `pAllocateArray`: Array descriptor

Returns:

- CUDA_SUCCESS
- CUDA_ERROR_DEINITIALIZED
- CUDA_ERROR_NOT_INITIALIZED
- CUDA_ERROR_INVALID_CONTEXT
- CUDA_ERROR_INVALID_VALUE
- CUDA_ERROR_OUT_OF_MEMORY
- CUDA_ERROR_UNKNOWN

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
- cuArray3DCreate
- cuArray3DGetDescriptor
- cuArrayDestroy
- cuArrayGetDescriptor
- cuMemAlloc
- cuMemAllocHost
- cuMemAllocPitch
- cuMemcpy2D
- cuMemcpy2DAsync
- cuMemcpy2DUnaligned
- cuMemcpy3D
- cuMemcpy3DAsync
- cuMemcpyAtoA
- cuMemcpyAtoD
- cuMemcpyAtoH
- cuMemcpyAtoHAsync
- cuMempyDtoA
- cuMempyDtoD
- cuMempyDtoDAsync
- cuMempyDtoH
- cuMempyDtoHAsync
- cuMempyHtoA
- cuMempyHtoAAsync
- cuMempyHtoD
- cuMempyHtoDAsync
- cuMemFree
- cuMemFreeHost
- cuMemGetAddressRange
- cuMemGetInfo
- cuMemHostAlloc
- cuMemHostGetDevicePointer
- cuMemsetD2D8
- cuMemsetD2D16
- cuMemsetD2D32
- cuMemsetD8
- cuMemsetD16
- cuMemsetD32

4.34.2.4 CUresult cuArrayDestroy (CUarray hArray)

Destroys the CUDA array `hArray`.

Parameters:
- `hArray`: Array to destroy

Returns:

- CUDA_SUCCESS
- CUDA_ERROR_DEINITIALIZED
- CUDA_ERROR_NOT_INITIALIZED
- CUDA_ERROR_INVALID_CONTEXT
- CUDA_ERROR_INVALID_HANDLE
- CUDA_ERROR_ARRAY_IS-MAPPED

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
- cuArray3DCreate
- cuArray3DGetDescriptor
- cuArrayCreate
- cuArrayGetDescriptor
- cuMemAlloc
- cuMemAllocHost
- cuMemAllocPitch
- cuMemcpy2D
- cuMemcpy2DAsync
- cuMemcpy2DUnaligned
- cuMemcpy3D
- cuMemcpy3DAsync
- cuMemcpyAtoA
- cuMemcpyAtoD
- cuMemcpyAtoH
- cuMemcpyAtoHAsync
- cuMempyDtoA
- cuMempyDtoD
- cuMempyDtoDAsync
- cuMempyDtoH
- cuMempyDtoHAsync
- cuMempyHtoA
- cuMempyHtoAAsync
- cuMempyHtoD
- cuMempyHtoDAsync
- cuMemFree
- cuMemFreeHost
- cuMemGetAddressRange
- cuMemGetInfo
- cuMemHostAlloc
- cuMemHostGetDevicePointer
- cuMemsetD2D8
- cuMemsetD2D16
- cuMemsetD2D32
- cuMemsetD8
- cuMemsetD16
- cuMemsetD32
4.34 Memory Management

4.34.2.5 CUresult cuArrayGetDescriptor (CUDA_ARRAY_DESCRIPTOR *pArrayDescriptor, CUarray hArray)

Returns in *pArrayDescriptor a descriptor containing information on the format and dimensions of the CUDA array hArray. It is useful for subroutines that have been passed a CUDA array, but need to know the CUDA array parameters for validation or other purposes.

Parameters:

- *pArrayDescriptor - Returned array descriptor
- hArray - Array to get descriptor of

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cudaMemcpy2D, cudaMemcpy2DAsync, cudaMemcpy2DUnaligned, cudaMemcpy3D, cudaMemcpy3DAsync, cudaMemcpyDtoA, cudaMemcpyDtoD, cudaMemcpyDtoHASync, cudaMemcpyDtoH, cudaMemcpyDtoHAsync, cudaMemcpyHtoA, cudaMemcpyHtoAAsync, cudaMemcpyHtoD, cudaMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

4.34.2.6 CUresult cuMemAlloc (CUdeviceptr *dptr, size_t bytesize)

Allocates bytesize bytes of linear memory on the device and returns in *dptr a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. If bytesize is 0, cuMemAlloc() returns CUDA_ERROR_INVALID_VALUE.

Parameters:

- dptr - Returned device pointer
- bytesize - Requested allocation size in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAllocHost, cuMemAllocPitch, cudaMemcpy2D, cudaMemcpy2DAsync, cudaMemcpy2DUnaligned, cudaMemcpy3D,
4.34.2.7 \textbf{CUresult cuMemAllocHost (void **pp, size_t \textit{bytesize})}

Allocates \textit{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 \texttt{cuMemcpy()}. Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as malloc(). Allocating excessive amounts of memory with \texttt{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 \texttt{cuMemHostAlloc()} will automatically be immediately accessible to all contexts on all devices which support unified addressing (as may be queried using \texttt{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 \texttt{*pp}. See \textit{Unified Addressing} for additional details.

\textbf{Parameters:}

- \textit{pp} - Returned host pointer to page-locked memory
- \textit{bytesize} - Requested allocation size in bytes

\textbf{Returns:}

- \texttt{CUDA_SUCCESS}
- \texttt{CUDA_ERROR_DEINITIALIZED}
- \texttt{CUDA_ERROR_NOT_INITIALIZED}
- \texttt{CUDA_ERROR_INVALID_CONTEXT}
- \texttt{CUDA_ERROR_INVALID_VALUE}
- \texttt{CUDA_ERROR_OUT_OF_MEMORY}

\textbf{Note:}

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{cuMemAllocPitch}, \texttt{cuMemcpy2D}, \texttt{cuMemcpy2DAsync}, \texttt{cuMemcpy2DUnaligned}, \texttt{cuMemcpy3D}, \texttt{cuMemcpy3DAsync}, \texttt{cuMemcpyAtoA}, \texttt{cuMemcpyAtoD}, \texttt{cuMemcpyAtoH}, \texttt{cuMemcpyDtoAAsync}, \texttt{cuMemcpyDtoDAsync}, \texttt{cuMemcpyDtoHAsync}, \texttt{cuMemcpyHtoAAsync}, \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}

4.34.2.8 \textbf{CUresult cuMemAllocPitch (CUdeviceptr * \textit{dptr}, size_t \textit{pPitch}, size_t \textit{WidthInBytes}, size_t \textit{Height},
unsigned int \textit{ElementSizeBytes})}

Allocates at least \textit{WidthInBytes} \textit{Height} bytes of linear memory on the device and returns in \texttt{*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. \textit{ElementSizeBytes} specifies the size of the largest reads and writes that will be performed on the memory range. \textit{ElementSizeBytes} may be 4, 8 or 16 (since coalesced memory transactions are not possible on other data sizes). If \textit{ElementSizeBytes} is smaller than the actual read/write size of a kernel, the kernel will run correctly, but possibly...
4.34 Memory Management

at reduced speed. The pitch returned in *pPitch by cuMem AllocPitch() is the width in bytes of the allocation. The intended usage of pitch is as a separate parameter of the allocation, used to compute addresses within the 2D array. Given the row and column of an array element of type T, the address is computed as:

```
T* pElement = (T*)((char*)BaseAddress + Row * Pitch) + Column;
```

The pitch returned by cuMem AllocPitch() is guaranteed to work with cuMemcpy2D() under all circumstances. For allocations of 2D arrays, it is recommended that programmers consider performing pitch allocations using cuMem AllocPitch(). 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 cuMem AllocPitch() is guaranteed to match or exceed the alignment requirement for texture binding with cuTexRefSetAddress2D().

Parameters:

- **dptr** - Returned device pointer
- **pPitch** - Returned pitch of allocation in bytes
- **WidthInBytes** - Requested allocation width in bytes
- **Height** - Requested allocation height in rows
- **ElementSizeBytes** - Size of largest reads/writes for range

Returns:

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:


4.34.2.9 CUresult cuMemcpy (CUdeviceptr dst, CUdeviceptr src, size_t ByteCount)

Copies data between two pointers. dst and src are base pointers of the destination and source, respectively. ByteCount specifies the number of bytes to copy. Note that this function infers the type of the transfer (host to host, host to device, device to device, or device to host) from the pointer values. This function is only allowed in contexts which support unified addressing. Note that this function is synchronous.

Parameters:

- **dst** - Destination unified virtual address space pointer
- **src** - Source unified virtual address space pointer
- **ByteCount** - Size of memory copy in bytes
Returns:

- CUDA_SUCCESS,
- CUDA_ERROR_DEINITIALIZED,
- CUDA_ERROR_NOT_INITIALIZED,
- CUDA_ERROR_INVALID_CONTEXT,
- CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:


4.34.2.10 CUresult cuMemcpy2D (const CUDA_MEMCPY2D *pCopy)

Perform a 2D memory copy according to the parameters specified in pCopy. The CUDA_MEMCPY2D structure is defined as:

```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_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.

\texttt{cuMemcpy2D()} returns an error if any pitch is greater than the maximum allowed (CU\_DEVICE\_ATTRIBUTE\_MAX\_PITCH). \texttt{cuMemAllocPitch()} passes back pitches that always work with \texttt{cuMemcpy2D()}. On intra-device memory copies (device to device, CUDA array to device, CUDA array to CUDA array), \texttt{cuMemcpy2D()} may fail for pitches not computed by \texttt{cuMemAllocPitch()}. \texttt{cuMemcpy2DUnaligned()} does not have this restriction, but may run significantly slower in the cases where \texttt{cuMemcpy2D()} would have returned an error code.

**Parameters:**

\texttt{pCopy} - Parameters for the memory copy

**Returns:**

CUDA\_SUCCESS, CUDA\_ERROR\_DEINITIALIZED, CUDA\_ERROR\_NOT\_INITIALIZED, CUDA\_ERROR\_INVALID\_CONTEXT, CUDA\_ERROR\_INVALID\_VALUE

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**


**4.34.2.11 CUresult cuMemcpy2DAsync (const CUDA\_MEMCPY2D \*pCopy, CUstream hStream)**

Perform a 2D memory copy according to the parameters specified in pCopy. The CUDA\_MEMCPY2D structure is defined as:

```c
typedef struct CUDA_MEMCPY2D_st {
    unsigned int srcXInBytes, srcY;
    CUmArrayType srcMemoryType;
    const void *srcHost;
    CUdeviceptr srcDevice;
    CUiArray srcArray;
    unsigned int srcPitch;
    unsigned int dstXInBytes, dstY;
    CUmArrayType dstMemoryType;
    void *dstHost;
    CUdeviceptr dstDevice;
    CUiArray 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_MEMORYTYPEDEVICE = 0x02,
    CU_MEMORYTYPEARRAY = 0x03,
    CU_MEMORYTYPEUNIFIED = 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
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.

`cuMemcpy2D()` returns an error if any pitch is greater than the maximum allowed (CU_DEVICE_ATTRIBUTE_MAX_PITCH). `cuMemAllocPitch()` passes back pitches that always work with `cuMemcpy2D()`. On intra-device memory copies (device to device, CUDA array to device, CUDA array to CUDA array), `cuMemcpy2D()` may fail for pitches not computed by `cuMemAllocPitch()`. `cuMemcpy2DUnaligned()` does not have this restriction, but may run significantly slower in the cases where `cuMemcpy2D()` would have returned an error code.

`cuMemcpy2DAsync()` is asynchronous and can optionally be associated to a stream by passing a non-zero `hStream` argument. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input.

**Parameters:**

- `pCopy` - Parameters for the memory copy
- `hStream` - Stream identifier

**Returns:**

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.
4.34 Memory Management

See also:

- cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor,
cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DUnaligned, cuMemcpy3D,
cuMemcpy3DAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoHASync, cuMemcpyDtoH, cuMemcpyDtoHASync,
cuMemcpyHtoA, cuMemcpyHtoASync, cuMemcpyHtoD, cuMemcpyHtoHASync, cuMemcpyHtoHASync,
cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8,
cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async,
cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

4.34.2.12 CUresult cuMemcpy2DUnaligned (const CUDA_MEMCPY2D *pCopy)

Perform a 2D memory copy according to the parameters specified in pCopy. The CUDA_MEMCPY2D structure is defined as:

```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_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`. 
4.34 Memory Management

`cuMemcpy2D()` returns an error if any pitch is greater than the maximum allowed (`CU_DEVICE_ATTRIBUTE_MAX_PITCH`). `cuMemAllocPitch()` passes back pitches that always work with `cuMemcpy2D()`. On intra-device memory copies (device to device, CUDA array to device, CUDA array to CUDA array), `cuMemcpy2D()` may fail for pitches not computed by `cuMemAllocPitch()`. `cuMemcpy2DUnaligned()` does not have this restriction, but may run significantly slower in the cases where `cuMemcpy2D()` would have returned an error code.

**Parameters:**

`pCopy` - Parameters for the memory copy

**Returns:**

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**


4.34.2.13 `CUresult cuMemcpy3D (const CUDA_MEMCPY3D *pCopy)`

Perform a 3D memory copy according to the parameters specified in `pCopy`. The `CUDA_MEMCPY3D` structure is defined as:

```c
typedef struct CUDA_MEMCPY3D_st {
    unsigned int srcXInBytes, srcY, srcZ;
    unsigned int srcLOD;
    CUmemorytype srcMemoryType;
    const void *srcHost;
    CUdeviceptr srcDevice;
    CArray 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;
    CArray 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

```c
void* Start = (void*)((char*)srcHost + (srcZ * srcHeight + srcY) * srcPitch + srcXInBytes);
```

For device pointers, the starting address is
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

```c
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.

**Parameters:**

- `pCopy` - Parameters for the memory copy

**Returns:**

- `CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

4.34.2.14  CUresult cuMemcpy3DAsync (const CUDA_MEMCPY3D *pCopy, CUstream hStream)

Perform a 3D memory copy according to the parameters specified in pCopy. The CUDA_MEMCPY3D structure is defined as:

```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

```c
void* Start = (void*)((char*)srcHost+(srcZ*srcHeight+srcY)*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```c
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

```c
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).

cuMemcpy3DAsync() is asynchronous and can optionally be associated to a stream by passing a non-zero hStream argument. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input.

The srcLOD and dstLOD members of the CUDA_MEMCPY3D structure must be set to 0.

Parameters:

- `pCopy` - Parameters for the memory copy
- `hStream` - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:


4.34.2.15 CUresult cuMemcpy3DPeer (const CUDA_MEMCPY3D_PEER *pCopy)

Perform a 3D memory copy according to the parameters specified in pCopy. See the definition of the CUDA_MEMCPY3D_PEER structure for documentation of its parameters.

Note that this function is synchronous with respect to the host only if the source or destination memory is of type CU_MEMORYTYPE_HOST. Note also that this copy is serialized with respect all pending and future asynchronous work in to the current context, the copy’s source context, and the copy’s destination context (use cuMemcpy3DPeerAsync to avoid this synchronization).

Parameters:

- `pCopy` - Parameters for the memory copy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.
4.34 Memory Management

See also:

`cuMemcpyDtoD`, `cuMemcpyPeer`, `cuMemcpyDtoDAsync`, `cuMemcpyPeerAsync`, `cuMemcpy3DPeerAsync`

### 4.34.2.16 CUresult cuMemcpy3DPeerAsync (const CUDA_MEMCPY3D_PEER * `pCopy`, CUstream `hStream`)

Perform a 3D memory copy according to the parameters specified in `pCopy`. See the definition of the `CUDA_MEMCPY3D_PEER` structure for documentation of its parameters.

**Parameters:**

- `pCopy` - Parameters for the memory copy
- `hStream` - Stream identifier

**Returns:**

- `CUDA_SUCCESS`
- `CUDA_ERROR_DEINITIALIZED`
- `CUDA_ERROR_NOT_INITIALIZED`
- `CUDA_ERROR_INVALID_CONTEXT`
- `CUDA_ERROR_INVALID_VALUE`

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

See also:

`cuMemcpyDtoD`, `cuMemcpyPeer`, `cuMemcpyDtoDAsync`, `cuMemcpyPeerAsync`, `cuMemcpy3DPeerAsync`

### 4.34.2.17 CUresult cuMemcpyAsync (CUdeviceptr `dst`, CUdeviceptr `src`, size_t `ByteCount`, CUstream `hStream`)

Copies data between two pointers. `dst` and `src` are base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy. Note that this function infers the type of the transfer (host to host, host to device, device to device, or device to host) from the pointer values. This function is only allowed in contexts which support unified addressing. Note that this function is asynchronous and can optionally be associated to a stream by passing a non-zero `hStream` argument.

**Parameters:**

- `dst` - Destination unified virtual address space pointer
- `src` - Source unified virtual address space pointer
- `ByteCount` - Size of memory copy in bytes
- `hStream` - Stream identifier

**Returns:**

- `CUDA_SUCCESS`
- `CUDA_ERROR_DEINITIALIZED`
- `CUDA_ERROR_NOT_INITIALIZED`
- `CUDA_ERROR_INVALID_CONTEXT`
- `CUDA_ERROR_INVALID_VALUE`

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.
4.34.2.18 CUresult cuMemcpyAtoA (CUarray dstArray, size_t dstOffset, CUarray srcArray, size_t srcOffset, size_t ByteCount)

Copies from one 1D CUDA array to another. dstArray and srcArray specify the handles of the destination and source CUDA arrays for the copy, respectively. dstOffset and srcOffset specify the destination and source offsets in bytes into the CUDA arrays. ByteCount is the number of bytes to be copied. The size of the elements in the CUDA arrays need not be the same format, but the elements must be the same size; and count must be evenly divisible by that size.

Parameters:

- **dstArray** - Destination array
- **dstOffset** - Offset in bytes of destination array
- **srcArray** - Source array
- **srcOffset** - Offset in bytes of source array
- **ByteCount** - Size of memory copy in bytes

Returns:

- CUDA_SUCCESS
- CUDA_ERROR_DEINITIALIZED
- CUDA_ERROR_NOT_INITIALIZED
- CUDA_ERROR_INVALID_CONTEXT
- CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:


4.34.2.19 CUresult cuMemcpyAtoD (CUdeviceptr dstDevice, CUarray srcArray, size_t srcOffset, size_t ByteCount)

Copies from one 1D CUDA array to device memory. dstDevice specifies the base pointer of the destination and must be naturally aligned with the CUDA array elements. srcArray and srcOffset specify the CUDA array handle and the offset in bytes into the array where the copy is to begin. ByteCount specifies the number of bytes to copy and must be evenly divisible by the array element size.
Parameters:

\[\begin{align*}
\text{dstDevice} &\quad \text{Destination device pointer} \\
\text{srcArray} &\quad \text{Source array} \\
\text{srcOffset} &\quad \text{Offset in bytes of source array} \\
\text{ByteCount} &\quad \text{Size of memory copy in bytes}
\end{align*}\]

Returns:

\[\text{CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE}\]

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

\text{cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAt0A, cuMemcpyAt0H, cuMemcpyAt0HASync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHASync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32}\]

4.34.2.20 \text{CUresult cuMemcpyAt0H (void * \text{dstHost}, CUarray \text{srcArray}, size_t \text{srcOffset}, size_t \text{ByteCount})}

Copies from one 1D CUDA array to host memory. \text{dstHost} specifies the base pointer of the destination. \text{srcArray} and \text{srcOffset} specify the CUDA array handle and starting offset in bytes of the source data. \text{ByteCount} specifies the number of bytes to copy.

Parameters:

\[\begin{align*}
\text{dstHost} &\quad \text{Destination device pointer} \\
\text{srcArray} &\quad \text{Source array} \\
\text{srcOffset} &\quad \text{Offset in bytes of source array} \\
\text{ByteCount} &\quad \text{Size of memory copy in bytes}
\end{align*}\]

Returns:

\[\text{CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE}\]

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

\text{cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAt0A, cuMemcpyAt0H, cuMemcpyAt0HASync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHASync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32}
CUresult cuMemcpyAtoHAsync (void * dstHost, CUarray srcArray, size_t srcOffset, size_t ByteCount, CUstream hStream)

Copies from one 1D CUDA array to host memory. `dstHost` specifies the base pointer of the destination. `srcArray` and `srcOffset` specify the CUDA array handle and starting offset in bytes of the source data. `ByteCount` specifies the number of bytes to copy.

`cuMemcpyAtoHAsync()` is asynchronous and can optionally be associated to a stream by passing a non-zero `stream` argument. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input.

Parameters:
- `dstHost` - Destination pointer
- `srcArray` - Source array
- `srcOffset` - Offset in bytes of source array
- `ByteCount` - Size of memory copy in bytes
- `hStream` - Stream identifier

Returns:
- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:

CUresult cuMemcpyDtoA (CUarray dstArray, size_t dstOffset, CUdeviceptr srcDevice, size_t ByteCount)

Copies from device memory to a 1D CUDA array. `dstArray` and `dstOffset` specify the CUDA array handle and starting index of the destination data. `srcDevice` specifies the base pointer of the source. `ByteCount` specifies the number of bytes to copy.

Parameters:
- `dstArray` - Destination array
- `dstOffset` - Offset in bytes of destination array
- `srcDevice` - Source device pointer
- `ByteCount` - Size of memory copy in bytes
4.34 Memory Management

Returns:

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:


4.34.2.23 CUresult cuMemcpyDtoD (CUdeviceptr dstDevice, CUdeviceptr srcDevice, size_t ByteCount)

Copies from device memory to device memory. dstDevice and srcDevice are the base pointers of the destination and source, respectively. ByteCount specifies the number of bytes to copy. Note that this function is asynchronous.

Parameters:

- **dstDevice** - Destination device pointer
- **srcDevice** - Source device pointer
- **ByteCount** - Size of memory copy in bytes

Returns:

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:


4.34.2.24 CUresult cuMemcpyDtoDAsync (CUdeviceptr dstDevice, CUdeviceptr srcDevice, size_t ByteCount, CUstream hStream)

Copies from device memory to device memory. dstDevice and srcDevice are the base pointers of the destination and source, respectively. ByteCount specifies the number of bytes to copy. Note that this function is asynchronous and can optionally be associated to a stream by passing a non-zero hStream argument.
Parameters:

- **dstDevice** - Destination device pointer
- **srcDevice** - Source device pointer
- **ByteCount** - Size of memory copy in bytes
- **hStream** - Stream identifier

Returns:

- CUDA_SUCCESS
- CUDA_ERROR_DEINITIALIZED
- CUDA_ERROR_NOT_INITIALIZED
- CUDA_ERROR_INVALID_CONTEXT
- CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuArray3DCreate
- cuArray3DGetDescriptor
- cuArrayCreate
- cuArrayDestroy
- cuArrayGetDescriptor
- cuMemAlloc
- cuMemAllocHost
- cuMemAllocPitch
- cuMemcpy2D
- cuMemcpy2DAsync
- cuMemcpy2DUnaligned
- cuMemcpy3D
- cuMemcpy3DAsync
- cuMemcpyAtoA
- cuMemcpyAtoD
- cuMemcpyAtoH
- cuMemcpyAtoHAVec
- cuMemcpyDtoA
- cuMemcpyDtoD
- cuMemcpyDtoH
- cuMemcpyDtoHAVec
- cuMemcpyDtoHAsync
- cuMemcpyHtoA
- cuMemcpyHtoDAAsync
- cuMemcpyHtoH
- cuMemcpyHtoHAVec
- cuMemcpyHtoHAVecAsync
- cuMemcpyHtoHAsync
- cuMemFree
- cuMemGetAddressRange
- cuMemGetInfo
- cuMemHostAlloc
- cuMemHostGetDevicePointer
- cuMemsetD2D8
- cuMemsetD2D8Async
- cuMemsetD2D16
- cuMemsetD2D16Async
- cuMemsetD2D32
- cuMemsetD2D32Async
- cuMemsetD8
- cuMemsetD8Async
- cuMemsetD16
- cuMemsetD16Async
- cuMemsetD32
- cuMemsetD32Async

4.34.2.25 **CUresult cuMemcpyDtoH (void \* dstHost, CUdeviceptr srcDevice, size_t ByteCount)**

Copies from device to host memory. **dstHost** and **srcDevice** specify the base pointers of the destination and source, respectively. **ByteCount** specifies the number of bytes to copy. Note that this function is synchronous.

Parameters:

- **dstHost** - Destination host pointer
- **srcDevice** - Source device pointer
- **ByteCount** - Size of memory copy in bytes

Returns:

- CUDA_SUCCESS
- CUDA_ERROR_DEINITIALIZED
- CUDA_ERROR_NOT_INITIALIZED
- CUDA_ERROR_INVALID_CONTEXT
- CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuArray3DCreate
- cuArray3DGetDescriptor
- cuArrayCreate
- cuArrayDestroy
- cuArrayGetDescriptor
- cuMemAlloc
- cuMemAllocHost
- cuMemAllocPitch
- cuMemcpy2D
- cuMemcpy2DAsync
- cuMemcpy2DUnaligned
- cuMemcpy3D
- cuMemcpy3DAsync
- cuMemcpyAtoA
- cuMemcpyAtoD
- cuMemcpyAtoH
- cuMemcpyAtoHAVec
- cuMemcpyDtoA
- cuMemcpyDtoD
- cuMemcpyDtoH
- cuMemcpyDtoHAVec
- cuMemcpyDtoHAVecAsync
- cuMemcpyHtoA
- cuMemcpyHtoDAAsync
- cuMemcpyHtoH
- cuMemcpyHtoHAVec
- cuMemcpyHtoHAVecAsync
- cuMemcpyHtoHAsync
- cuMemFree
- cuMemGetAddressRange
- cuMemGetInfo
- cuMemHostAlloc
- cuMemHostGetDevicePointer
- cuMemsetD2D8
- cuMemsetD2D16
- cuMemsetD2D32
- cuMemsetD8
- cuMemsetD16
- cuMemsetD32
- cuMemsetD32Async
4.34.2.26  CUresult cuMemcpyDtoHAsync (void * dstHost, CUdeviceptr srcDevice, size_t ByteCount, CUstream hStream)

Copies from device to host memory. dstHost and srcDevice specify the base pointers of the destination and source, respectively. ByteCount specifies the number of bytes to copy.

cuMemcpyDtoHAsync() is asynchronous and can optionally be associated to a stream by passing a non-zero hStream argument. It only works on page-locked memory and returns an error if a pointer to pageable memory is passed as input.

Parameters:

- **dstHost** - Destination host pointer
- **srcDevice** - Source device pointer
- **ByteCount** - Size of memory copy in bytes
- **hStream** - Stream identifier

Returns:

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

- Note that this function may also return error codes from previous, asynchronous launches.

See also:


4.34.2.27  CUresult cuMemcpyHtoA (CUarray dstArray, size_t dstOffset, const void * srcHost, size_t ByteCount)

Copies from host memory to a 1D CUDA array. dstArray and dstOffset specify the CUDA array handle and starting offset in bytes of the destination data. pSrc specifies the base address of the source. ByteCount specifies the number of bytes to copy.

Parameters:

- **dstArray** - Destination array
- **dstOffset** - Offset in bytes of destination array
- **srcHost** - Source host pointer
- **ByteCount** - Size of memory copy in bytes

Returns:

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE
Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuArray3DCreate
- cuArray3DGetDescriptor
- cuArrayCreate
- cuArrayDestroy
- cuArrayGetDescriptor
- cuMemAlloc
- cuMemAllocHost
- cuMemAllocPitch
- cuMemcpy2D
- cuMemcpy2DAsync
- cuMemcpy3D
- cuMemcpy3DAsync
- cuMemcpyAtoA
- cuMemcpyAtoD
- cuMemcpyAtoH
- cuMemcpyAtoHAsync
- cuMemcpyDtoA
- cuMemcpyDtoD
- cuMemcpyDtoDAsync
- cuMemcpyDtoH
- cuMemcpyDtoHAsync
- cuMemcpyHtoAAsync
- cuMemcpyHtoD
- cuMemcpyHtoDAsync
- cuMemFree
- cuMemFreeHost
- cuMemGetAddressRange
- cuMemGetInfo
- cuMemHostAlloc
- cuMemHostGetDevicePointer
- cuMemsetD2D8
- cuMemsetD2D8Async
- cuMemsetD2D16
- cuMemsetD2D32
- cuMemsetD8
- cuMemsetD8Async
- cuMemsetD16
- cuMemsetD16Async
- cuMemsetD32
- cuMemsetD32Async

4.34.28  

CUresult cuMemcpyHtoAAsync (CUarray dstArray, size_t dstOffset, const void * srcHost, size_t ByteCount, CUstream hStream)

Copies from host memory to a 1D CUDA array. dstArray and dstOffset specify the CUDA array handle and starting offset in bytes of the destination data. srcHost specifies the base address of the source. ByteCount specifies the number of bytes to copy.

cuMemcpyHtoAAsync() is asynchronous and can optionally be associated to a stream by passing a non-zero hStream argument. It only works on page-locked memory and returns an error if a pointer to pageable memory is passed as input.

Parameters:

- **dstArray** - Destination array
- **dstOffset** - Offset in bytes of destination array
- **srcHost** - Source host pointer
- **ByteCount** - Size of memory copy in bytes
- **hStream** - Stream identifier

Returns:

- CUDA_SUCCESS
- CUDA_ERROR_DEINITIALIZED
- CUDA_ERROR_NOT_INITIALIZED
- CUDA_ERROR_INVALID_CONTEXT
- CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuArray3DCreate
- cuArray3DGetDescriptor
- cuArrayCreate
- cuArrayDestroy
- cuArrayGetDescriptor
- cuMemAlloc
- cuMemAllocHost
- cuMemAllocPitch
- cuMemcpy2D
- cuMemcpy2DAsync
- cuMemcpy3D
- cuMemcpy3DAsync
- cuMemcpyAtoA
- cuMemcpyAtoD
- cuMemcpyAtoH
- cuMemcpyAtoHAsync
- cuMemcpyDtoA
- cuMemcpyDtoD
- cuMemcpyDtoDAsync
- cuMemcpyDtoH
- cuMemcpyDtoHAsync
- cuMemcpyHtoAAsync
- cuMemcpyHtoD
- cuMemcpyHtoDAsync
- cuMemFree
- cuMemFreeHost
- cuMemGetAddressRange
- cuMemGetInfo
- cuMemHostAlloc
- cuMemHostGetDevicePointer
- cuMemsetD2D8
- cuMemsetD2D8Async
- cuMemsetD2D16
- cuMemsetD2D16Async
- cuMemsetD2D32
- cuMemsetD2D32Async
- cuMemsetD8
- cuMemsetD8Async
- cuMemsetD16
- cuMemsetD16Async
- cuMemsetD32
- cuMemsetD32Async
- cuMemsetD8Async
- cuMemsetD16Async
- cuMemsetD32Async
- cuMemsetD32Async
4.34 Memory Management

4.34.2.29  

CUresult cuMemcpyHtoD (CUdeviceptr dstDevice, const void * srcHost, size_t ByteCount)

Copies from host memory to device memory. dstDevice and srcHost are the base addresses of the destination and source, respectively. ByteCount specifies the number of bytes to copy. Note that this function is synchronous.

Parameters:

- **dstDevice** - Destination device pointer
- **srcHost** - Source host pointer
- **ByteCount** - Size of memory copy in bytes

Returns:

- CUDA_SUCCESS
- CUDA_ERROR_DEINITIALIZED
- CUDA_ERROR_NOT_INITIALIZED
- CUDA_ERROR_INVALID_CONTEXT
- CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyDtoAAsync, cuMemcpyDtoAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

4.34.2.30  

CUresult cuMemcpyHtoDAsync (CUdeviceptr dstDevice, const void * srcHost, size_t ByteCount, CUsream hStream)

Copies from host memory to device memory. dstDevice and srcHost are the base addresses of the destination and source, respectively. ByteCount specifies the number of bytes to copy.

cuMemcpyHtoDAsync() is asynchronous and can optionally be associated to a stream by passing a non-zero hStream argument. It only works on page-locked memory and returns an error if a pointer to pageable memory is passed as input.

Parameters:

- **dstDevice** - Destination device pointer
- **srcHost** - Source host pointer
- **ByteCount** - Size of memory copy in bytes
- **hStream** - Stream identifier

Returns:

- CUDA_SUCCESS
- CUDA_ERROR_DEINITIALIZED
- CUDA_ERROR_NOT_INITIALIZED
- CUDA_ERROR_INVALID_CONTEXT
- CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.
See also:


4.34.2.31 CUresult cuMemcpyPeer (CUdeviceptr dstDevice, CUcontext dstContext, CUdeviceptr srcDevice, CUcontext srcContext, size_t ByteCount)

Copies from device memory in one context to device memory in another context. dstDevice is the base device pointer of the destination memory and dstContext is the destination context. srcDevice is the base device pointer of the source memory and srcContext is the source pointer. ByteCount specifies the number of bytes to copy.

Note that this function is asynchronous with respect to the host, but serialized with respect all pending and future asynchronous work in to the current context, srcContext, and dstContext (use cuMemcpyPeerAsync to avoid this synchronization).

Parameters:

- **dstDevice** - Destination device pointer
- **dstContext** - Destination context
- **srcDevice** - Source device pointer
- **srcContext** - Source context
- **ByteCount** - Size of memory copy in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuMemcpyDtoD, cuMemcpy3DPeer, cuMemcpyDtoDAsync, cuMemcpyPeerAsync, cuMemcpy3DPeerAsync

4.34.2.32 CUresult cuMemcpyPeerAsync (CUdeviceptr dstDevice, CUcontext dstContext, CUdeviceptr srcDevice, CUcontext srcContext, size_t ByteCount, CUstream hStream)

Copies from device memory in one context to device memory in another context. dstDevice is the base device pointer of the destination memory and dstContext is the destination context. srcDevice is the base device pointer of the source memory and srcContext is the source pointer. ByteCount specifies the number of bytes to copy. Note that this function is asynchronous with respect to the host and all work in other streams in other devices.
4.34 Memory Management

Parameters:

- **dstDevice** - Destination device pointer
- **dstContext** - Destination context
- **srcDevice** - Source device pointer
- **srcContext** - Source context
- **ByteCount** - Size of memory copy in bytes
- **hStream** - Stream identifier

Returns:

- `CUDA_SUCCESS`
- `CUDA_ERROR_DEINITIALIZED`
- `CUDA_ERROR_NOT_INITIALIZED`
- `CUDA_ERROR_INVALID_CONTEXT`
- `CUDA_ERROR_INVALID_VALUE`

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuMemcpyDtoD
- cuMemcpyPeer
- cuMemcpy3DPeer
- cuMemcpyDtoDAsync
- cuMemcpy3DPeerAsync

4.34.2.33 CUresult cuMemFree (CUdeviceptr dptr)

Frees the memory space pointed to by **dptr**, which must have been returned by a previous call to **cuMemAlloc()** or **cuMemAllocPitch()**.

Parameters:

- **dptr** - Pointer to memory to free

Returns:

- `CUDA_SUCCESS`
- `CUDA_ERROR_DEINITIALIZED`
- `CUDA_ERROR_NOT_INITIALIZED`
- `CUDA_ERROR_INVALID_CONTEXT`
- `CUDA_ERROR_INVALID_VALUE`

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuArray3DCreate
- cuArray3DGetDescriptor
- cuArrayCreate
- cuArrayDestroy
- cuArrayGetDescriptor
- cuMemAlloc
- cuMemAllocHost
- cuMemAllocPitch
- cuMemcpy2D
- cuMemcpy2DAsync
- cuMemcpyAtoA
- cuMemcpyAtoD
- cuMemcpyAtoH
- cuMemcpyAtoHAsync
- cuMemcpyDtoA
- cuMemcpyDtoD
- cuMemcpyDtoDAsync
- cuMemcpyDtoH
- cuMemcpyDtoHAsync
- cuMemcpyHtoA
- cuMemcpyHtoAAsync
- cuMemcpyHtoD
- cuMemcpyHtoDAsync
- cuMemFreeHost
- cuMemGetAddressRange
- cuMemGetInfo
- cuMemHostAlloc
- cuMemHostGetDevicePointer
- cuMemsetD2D8
- cuMemsetD2D16
- cuMemsetD2D32
- cuMemsetD8
- cuMemsetD16
- cuMemsetD32
4.34.2.34 **CUresult cuMemFreeHost (void *p)**

Frees the memory space pointed to by p, which must have been returned by a previous call to cuMemAllocHost().

**Parameters:**

p - Pointer to memory to free

**Returns:**

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**


4.34.2.35 **CUresult cuMemGetAddressRange (CUdeviceptr *pbase, size_t *psize, CUdeviceptr dptr)**

Returns the base address in *pbase and size in *psize of the allocation by cuMemAlloc() or cuMemAllocPitch() that contains the input pointer dptr. Both parameters pbase and psize are optional. If one of them is NULL, it is ignored.

**Parameters:**

pbase - Returned base address

psize - Returned size of device memory allocation

dptr - Device pointer to query

**Returns:**

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

4.34.2.36 **CUresult cuMemGetInfo (size_t *free, size_t *total)**

Returns in *free* and *total* respectively, the free and total amount of memory available for allocation by the CUDA context, in bytes.

**Parameters:**

- *free* - Returned free memory in bytes
- *total* - Returned total memory in bytes

**Returns:**

- CUDA_SUCCESS
- CUDA_ERROR_DEINITIALIZED
- CUDA_ERROR_NOT_INITIALIZED
- CUDA_ERROR_INVALID_CONTEXT
- CUDA_ERROR_INVALID_VALUE

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**


4.34.2.37 **CUresult cuMemHostAlloc (void **pp, size_t bytesize, unsigned int Flags)**

Allocates bytesize bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as cuMemcpyHtoD(). Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as malloc(). Allocating excessive amounts of pinned memory may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to allocate staging areas for data exchange between host and device.

The Flags parameter enables different options to be specified that affect the allocation, as follows.

- **CU_MEMHOSTALLOC_PORTABLE**: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.

- **CU_MEMHOSTALLOC_DEVICEMAP**: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling cuMemHostGetDevicePointer(). This feature is available only on GPUs with compute capability greater than or equal to 1.1.

- **CU_MEMHOSTALLOC_WRITECOMBINED**: Allocates the memory as write-combined (WC). WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs. WC memory is a good option for buffers that will be written by the CPU and read by the GPU via mapped pinned memory or host->device transfers.

All of these flags are orthogonal to one another: a developer may allocate memory that is portable, mapped and/or write-combined with no restrictions.
The CUDA context must have been created with the `CU_CTX_MAP_HOST` flag in order for the `CU_MEMHOSTALLOC_MAPPED` flag to have any effect.

The `CU_MEMHOSTALLOC_MAPPED` flag may be specified on CUDA contexts for devices that do not support mapped pinned memory. The failure is deferred to `cuMemHostGetDevicePointer()` because the memory may be mapped into other CUDA contexts via the `CU_MEMHOSTALLOC_PORTABLE` flag.

The memory allocated by this function must be freed with `cuMemFreeHost()`.

Note all host memory allocated using `cuMemHostAlloc()` will automatically be immediately accessible to all contexts on all devices which support unified addressing (as may be queried using `CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING`). Unless the flag `CU_MEMHOSTALLOC_WRITECOMBINED` is specified, the device pointer that may be used to access this host memory from those contexts is always equal to the returned host pointer *pp*. If the flag `CU_MEMHOSTALLOC_WRITECOMBINED` is specified, then the function `cuMemHostGetDevicePointer()` must be used to query the device pointer, even if the context supports unified addressing. See Unified Addressing for additional details.

**Parameters:**

- `pp` - Returned host pointer to page-locked memory
- `bytesize` - Requested allocation size in bytes
- `Flags` - Flags for allocation request

**Returns:**

- `CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_OUT_OF_MEMORY`

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**


**4.34.2.38** `CUresult cuMemHostGetDevicePointer (CUdeviceptr *pdptr, void *p, unsigned int Flags)`

Passes back the device pointer `pdptr` corresponding to the mapped, pinned host buffer `p` allocated by `cuMemHostAlloc`.

`cuMemHostGetDevicePointer()` will fail if the `CU_MEMALLOCHOST_DEVICEMAP` flag was not specified at the time the memory was allocated, or if the function is called on a GPU that does not support mapped pinned memory.

`Flags` provides for future releases. For now, it must be set to 0.

**Parameters:**

- `pdptr` - Returned device pointer
- `p` - Host pointer
4.34 Memory Management

Flags - Options (must be 0)

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:


4.34.2.39 CUresult cuMemHostGetFlags (unsigned int *pFlags, void *p)

Passes back the flags pFlags that were specified when allocating the pinned host buffer p allocated by cuMemHostAlloc.

cuMemHostGetFlags() will fail if the pointer does not reside in an allocation performed by cuMemAllocHost() or cuMemHostAlloc().

Parameters:

pFlags - Returned flags word

p - Host pointer

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuMemAllocHost, cuMemHostAlloc

4.34.2.40 CUresult cuMemHostRegister (void *p, size_t bytesize, unsigned int Flags)

Page-locks the memory range specified by p and bytesize and maps it for the device(s) as specified by Flags. This memory range also is added to the same tracking mechanism as cuMemHostAlloc to automatically accelerate calls to functions such as cuMemcpyHtoD(). Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory that has not been registered. Page-locking excessive amounts of memory may degrade system performance, since it reduces the amount of memory available to the system.
for paging. As a result, this function is best used sparingly to register staging areas for data exchange between host and device.

This function is not yet supported on Mac OS X.

The `Flags` parameter enables different options to be specified that affect the allocation, as follows.

- **CU_MEMHOSTREGISTER_PORTABLE**: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.

- **CU_MEMHOSTREGISTER_DEVICEMAP**: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling `cuMemHostGetDevicePointer()`. This feature is available only on GPUs with compute capability greater than or equal to 1.1.

All of these flags are orthogonal to one another: a developer may page-lock memory that is portable or mapped with no restrictions.

The CUDA context must have been created with the `CU_CTX_MAP_HOST` flag in order for the `CU_MEMHOSTREGISTER_DEVICEMAP` flag to have any effect.

The `CU_MEMHOSTREGISTER_DEVICEMAP` flag may be specified on CUDA contexts for devices that do not support mapped pinned memory. The failure is deferred to `cuMemHostGetDevicePointer()` because the memory may be mapped into other CUDA contexts via the `CU_MEMHOSTREGISTER_PORTABLE` flag.

The pointer `p` and size `bytesize` must be aligned to the host page size (4 KB).

The memory page-locked by this function must be unregistered with `cuMemHostUnregister()`.

**Parameters:**

- `p` - Host pointer to memory to page-lock
- `bytesize` - Size in bytes of the address range to page-lock
- `Flags` - Flags for allocation request

**Returns:**

- `CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_OUT_OF_MEMORY`

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

- `cuMemHostUnregister`, `cuMemHostGetFlags`, `cuMemHostGetDevicePointer`

### 4.34.2.41 CUresult cuMemHostUnregister (void *p)

Unmaps the memory range whose base address is specified by `p`, and makes it pageable again.

The base address must be the same one specified to `cuMemHostRegister()`.

**Parameters:**

- `p` - Host pointer to memory to unregister
Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuMemHostRegister

4.34.2.42 CUresult cuMemsetD16 (CUdeviceptr dstDevice, unsigned short us, size_t N)

Sets the memory range of N 16-bit values to the specified value us.

Parameters:

dstDevice - Destination device pointer
us - Value to set
N - Number of elements

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpy3DToA, cuMemcpy3DToD, cuMemcpy3DToH, cuMemcpy3DToHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyDtoHAsync, cuMemcpyHtoAAsync, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

4.34.2.43 CUresult cuMemsetD16Async (CUdeviceptr dstDevice, unsigned short us, size_t N, CUstream hStream)

Sets the memory range of N 16-bit values to the specified value us. cuMemsetD16Async() is asynchronous and can optionally be associated to a stream by passing a non-zero stream argument.

Parameters:

dstDevice - Destination device pointer
us - Value to set
N - Number of elements
hStream - Stream identifier

Returns:
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcp2D, cuMemcp2DAsync, cuMemcp2DUnaligned, cuMemcp3D, cuMemcp3DAsync, cuMemcpAtoA, cuMemcpAtoD, cuMemcpAtoH, cuMemcpAtoHasync, cuMemcpDtoA, cuMemcpDtoD, cuMemcpDtoDAsync, cuMemcpDtoH, cuMemcpDtoHasync, cuMemcpHtoADAsync, cuMemcpHtoD, cuMemcpHtoDAsync, cuMemFree, cuMemHostAlloc, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16, cuMemsetD32, cuMemsetD32Async

4.34.2.44 CUresult cuMemsetD2D16 (CUdeviceptr dstDevice, size_t dstPitch, unsigned short us, size_t Width, size_t Height)

Sets the 2D memory range of Width 16-bit values to the specified value us. Height specifies the number of rows to set, and dstPitch specifies the number of bytes between each row. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

Parameters:
dstDevice - Destination device pointer
dstPitch - Pitch of destination device pointer
us - Value to set
Width - Width of row
Height - Number of rows

Returns:
CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:
Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcp2D, cuMemcp2DAsync, cuMemcp2DUnaligned, cuMemcp3D, cuMemcp3DAsync, cuMemcpAtoA, cuMemcpAtoD, cuMemcpAtoH, cuMemcpAtoHasync, cuMemcpDtoA, cuMemcpDtoD, cuMemcpDtoDAsync, cuMemcpDtoH, cuMemcpDtoHasync,
4.34.2.45  CUresult cuMemsetD2D16Async (CUdeviceptr dstDevice, size_t dstPitch, unsigned short us, size_t Width, size_t Height, CUstream hStream)

Sets the 2D memory range of Width 16-bit values to the specified value us. Height specifies the number of rows to set, and dstPitch specifies the number of bytes between each row. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

cuMemsetD2D16Async() is asynchronous and can optionally be associated to a stream by passing a non-zero stream argument.

Parameters:

- **dstDevice** - Destination device pointer
- **dstPitch** - Pitch of destination device pointer
- **us** - Value to set
- **Width** - Width of row
- **Height** - Number of rows
- **hStream** - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:


4.34.2.46  CUresult cuMemsetD2D32 (CUdeviceptr dstDevice, size_t dstPitch, unsigned int ui, size_t Width, size_t Height)

Sets the 2D memory range of Width 32-bit values to the specified value ui. Height specifies the number of rows to set, and dstPitch specifies the number of bytes between each row. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

Parameters:

- **dstDevice** - Destination device pointer
**dstPitch** - Pitch of destination device pointer

**ui** - Value to set

**Width** - Width of row

**Height** - Number of rows

**Returns:**

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**


**4.34.2.47** CUresult cuMemsetD2D32Async (CUdeviceptr dstDevice, size_t dstPitch, unsigned int ui, size_t Width, size_t Height, CUstream hStream)

Sets the 2D memory range of `Width` 32-bit values to the specified value `ui`. `Height` specifies the number of rows to set, and `dstPitch` specifies the number of bytes between each row. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

cuMemsetD2D32Async() is asynchronous and can optionally be associated to a stream by passing a non-zero `stream` argument.

**Parameters:**

- **dstDevice** - Destination device pointer
- **dstPitch** - Pitch of destination device pointer
- **ui** - Value to set
- **Width** - Width of row
- **Height** - Number of rows
- **hStream** - Stream identifier

**Returns:**

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.
4.34 Memory Management

See also:

4.34.2.48 CUresult cuMemsetD2D8 (CUdeviceptr dstDevice, size_t dstPitch, unsigned char uc, size_t Width, size_t Height)

Sets the 2D memory range of Width 8-bit values to the specified value uc. Height specifies the number of rows to set, and dstPitch specifies the number of bytes between each row. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

Parameters:

  - **dstDevice** - Destination device pointer
  - **dstPitch** - Pitch of destination device pointer
  - **uc** - Value to set
  - **Width** - Width of row
  - **Height** - Number of rows

Returns:

  CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

4.34.2.49 CUresult cuMemsetD2D8Async (CUdeviceptr dstDevice, size_t dstPitch, unsigned char uc, size_t Width, size_t Height, CUstream hStream)

Sets the 2D memory range of Width 8-bit values to the specified value uc. Height specifies the number of rows to set, and dstPitch specifies the number of bytes between each row. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

cuMemsetD2D8Async() is asynchronous and can optionally be associated to a stream by passing a non-zero stream argument.

Generated for NVIDIA CUDA Library by Doxygen
Parameters:

- **dstDevice** - Destination device pointer
- **dstPitch** - Pitch of destination device pointer
- **uc** - Value to set
- **Width** - Width of row
- **Height** - Number of rows
- **hStream** - Stream identifier

Returns:

- CUDA_SUCCESS
- CUDA_ERROR_DEINITIALIZED
- CUDA_ERROR_NOT_INITIALIZED
- CUDA_ERROR_INVALID_CONTEXT
- CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:


4.3.4.2.50 CUresult cuMemsetD32 (CUdeviceptr dstDevice, unsigned int ui, size_t N)

Sets the memory range of \( N \) 32-bit values to the specified value \( ui \).

Parameters:

- **dstDevice** - Destination device pointer
- **ui** - Value to set
- **N** - Number of elements

Returns:

- CUDA_SUCCESS
- CUDA_ERROR_DEINITIALIZED
- CUDA_ERROR_NOT_INITIALIZED
- CUDA_ERROR_INVALID_CONTEXT
- CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMalloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHasync, cuMemcpyDtoAAsync, cuMemcpyDtoDAsync, cuMemcpyDtoHasync, cuMemcpyDtoAAsync, cuMemcpyDtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async
4.34 Memory Management

4.34.2.51 CUresult cuMemsetD32Async (CUdeviceptr dstDevice, unsigned int ui, size_t N, CUstream hStream)

Sets the memory range of \(N\) 32-bit values to the specified value `ui`. 

`cuMemsetD32Async()` is asynchronous and can optionally be associated to a stream by passing a non-zero `stream` argument.

**Parameters:**
- `dstDevice` - Destination device pointer
- `ui` - Value to set
- `N` - Number of elements
- `hStream` - Stream identifier

**Returns:**
- `CUDA_SUCCESS`
- `CUDA_ERROR_DEINITIALIZED`
- `CUDA_ERROR_NOT_INITIALIZED`
- `CUDA_ERROR_INVALID_CONTEXT`
- `CUDA_ERROR_INVALID_VALUE`

**Note:**
Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

4.34.2.52 CUresult cuMemsetD8 (CUdeviceptr dstDevice, unsigned char uc, size_t N)

Sets the memory range of \(N\) 8-bit values to the specified value `uc`.

**Parameters:**
- `dstDevice` - Destination device pointer
- `uc` - Value to set
- `N` - Number of elements

**Returns:**
- `CUDA_SUCCESS`
- `CUDA_ERROR_DEINITIALIZED`
- `CUDA_ERROR_NOT_INITIALIZED`
- `CUDA_ERROR_INVALID_CONTEXT`
- `CUDA_ERROR_INVALID_VALUE`

**Note:**
Note that this function may also return error codes from previous, asynchronous launches.
4.34.2.53  

CUresult cuMemsetD8Async (CUdeviceptr dstDevice, unsigned char uc, size_t N, CUstream hStream)  

Sets the memory range of N 8-bit values to the specified value uc. 

cuMemsetD8Async() is asynchronous and can optionally be associated to a stream by passing a non-zero stream argument.

Parameters:  

- dstDevice - Destination device pointer  
- uc - Value to set  
- N - Number of elements  
- hStream - Stream identifier

Returns:  

CU_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:  

Note that this function may also return error codes from previous, asynchronous launches.

See also:

4.35 Unified Addressing

Functions

- CUresult cuPointerGetAttribute (void *data, CUpointer_attribute attribute, CUdeviceptr ptr)

  Returns information about a pointer.

4.35.1 Detailed Description

This section describes the unified addressing functions of the low-level CUDA driver application programming interface.

4.35.2 Overview

CUDA devices can share a unified address space with the host. For these devices there is no distinction between a device pointer and a host pointer – the same pointer value may be used to access memory from the host program and from a kernel running on the device (with exceptions enumerated below).

4.35.3 Supported Platforms

Whether or not a device supports unified addressing may be queried by calling cuDeviceGetAttribute() with the device attribute CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING.

Unified addressing is automatically enabled in 64-bit processes on devices with compute capability greater than or equal to 2.0.

Unified addressing is not yet supported on Windows Vista or Windows 7 for devices that do not use the TCC driver model.

4.35.4 Looking Up Information from Pointer Values

It is possible to look up information about the memory which backs a pointer value. For instance, one may want to know if a pointer points to host or device memory. As another example, in the case of device memory, one may want to know on which CUDA device the memory resides. These properties may be queried using the function cuPointerGetAttribute().

Because pointers are unique, it is not necessary to specify information about the pointers specified to the various copy functions in the CUDA API. The function cuMemcpy() may be used to perform a copy between two pointers, ignoring whether they point to host or device memory (making cuMemcpyHtoD(), cuMemcpyDtoD(), and cuMemcpyDtoH() unnecessary for devices supporting unified addressing). For multidimensional copies, the memory type CU_MEMORYTYPE_UNIFIED may be used to specify that the CUDA driver should infer the location of the pointer from its value.

4.35.5 Automatic Mapping of Host Allocated Host Memory

All host memory allocated in all contexts using cuMemAllocHost() and cuMemHostAlloc() is always directly accessible from all contexts on all devices that support unified addressing. This is the case regardless of whether or not the flags CU_MEMHOSTALLOC_PORTABLE and CU_MEMHOSTALLOC_DEVICEMAP are specified.
The pointer value through which allocated host memory may be accessed in kernels on all devices that support unified addressing is the same as the pointer value through which that memory is accessed on the host, so it is not necessary to call `cuMemHostGetDevicePointer()` to get the device pointer for these allocations.

Note that this is not the case for memory allocated using the flag `CU_MEMHOSTALLOC_WRITECOMBINED`, as discussed below.

### 4.35.6 Automatic Registration of Peer Memory

Upon enabling direct access from a context that supports unified addressing to another peer context that supports unified addressing using `cuCtxEnablePeerAccess()` all memory allocated in the peer context using `cuMemAlloc()` and `cuMemAllocPitch()` will immediately be accessible by the current context. 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.

### 4.35.7 Exceptions, Disjoint Addressing

Not all memory may be accessed on devices through the same pointer value through which they are accessed on the host. These exceptions are host memory registered using `cuMemHostRegister()` and host memory allocated using the flag `CU_MEMHOSTALLOC_WRITECOMBINED`. For these exceptions, there exists a distinct host and device address for the memory. The device address is guaranteed to not overlap any valid host pointer range and is guaranteed to have the same value across all contexts that support unified addressing.

This device address may be queried using `cuMemHostGetDevicePointer()` when a context using unified addressing is current. Either the host or the unified device pointer value may be used to refer to this memory through `cuMemcpy()` and similar functions using the `CU_MEMORYTYPE_UNIFIED` memory type.

### 4.35.8 Function Documentation

#### 4.35.8.1 CUresult cuPointerGetAttribute (void *data, CUpointer_attribute attribute, CUdeviceptr ptr)

The supported attributes are:

- **CU_POINTER_ATTRIBUTE_CONTEXT**:

  Returns in *data the `CUcontext` in which *ptr* was allocated or registered. The type of `data` must be `CUcontext` *.

  If *ptr* was not allocated by, mapped by, or registered with a `CUcontext` which uses unified virtual addressing then `CUDA_ERROR_INVALID_VALUE` is returned.

- **CU_POINTER_ATTRIBUTE_MEMORY_TYPE**:

  Returns in *data the physical memory type of the memory that *ptr* addresses as a `CUmemorytype` enumerated value. The type of `data` must be unsigned int.

  If *ptr* addresses device memory then *data is set to `CU_MEMORYTYPE_DEVICE`. The particular `CUdevice` on which the memory resides is the `CUdevice` of the `CUcontext` returned by the `CU_POINTER_ATTRIBUTE_CONTEXT` attribute of *ptr*.

  If *ptr* addresses host memory then *data is set to `CU_MEMORYTYPE_HOST`.

  If *ptr* was not allocated by, mapped by, or registered with a `CUcontext` which uses unified virtual addressing then `CUDA_ERROR_INVALID_VALUE` is returned.
If the current \texttt{CUcontext} does not support unified virtual addressing then \texttt{CUDA_ERROR_INVALID_CONTEXT} is returned.

- \textbf{CU_POINTER_ATTRIBUTE_DEVICE_POINTER}:

Returns in \texttt{*data} the device pointer value through which \texttt{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 \texttt{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 \texttt{*data} will equal the input value \texttt{ptr}.

- \textbf{CU_POINTER_ATTRIBUTE_HOST_POINTER}:

Returns in \texttt{*data} the host pointer value through which \texttt{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 \texttt{ptr} then \texttt{CUDA_ERROR_INVALID_VALUE} is returned.

Except in the exceptional disjoint addressing cases discussed below, the value returned in \texttt{*data} will equal the input value \texttt{ptr}.

Note that for most allocations in the unified virtual address space the host and device pointer for accessing the allocation will be the same. The exceptions to this are

- user memory registered using \texttt{cuMemHostRegister}

- host memory allocated using \texttt{cuMemHostAlloc} with the \texttt{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 \texttt{CU_POINTER_ATTRIBUTE_HOST_POINTER} and \texttt{CU_POINTER_ATTRIBUTEDEVICE_POINTER} may be used to retrieve the host and device addresses from either address.

\textbf{Parameters:}

\begin{itemize}
  \item \textit{data} - Returned pointer attribute value
  \item \textit{attribute} - Pointer attribute to query
  \item \textit{ptr} - Pointer
\end{itemize}

\textbf{Returns:}

\begin{itemize}
  \item \texttt{CUDA_SUCCESS}, \texttt{CUDA_ERROR_DEINITIALIZED}, \texttt{CUDA_ERROR_NOT_INITIALIZED}, \texttt{CUDA_ERROR_INVALID_CONTEXT}, \texttt{CUDA_ERROR_INVALID_VALUE}, \texttt{CUDA_ERROR_INVALID_DEVICE}
\end{itemize}
Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuMemAlloc, cuMemFree, cuMemAllocHost, cuMemFreeHost, cuMemHostAlloc, cuMemHostRegister, cuMemHostUnregister
4.36 Stream Management

Functions

- CUresult cuStreamCreate (CUstream *phStream, unsigned int Flags)
  Create a stream.

- CUresult cuStreamDestroy (CUstream hStream)
  Destroys a stream.

- CUresult cuStreamQuery (CUstream hStream)
  Determine status of a compute stream.

- CUresult cuStreamSynchronize (CUstream hStream)
  Wait until a stream's tasks are completed.

- CUresult cuStreamWaitEvent (CUstream hStream, CUevent hEvent, unsigned int Flags)
  Make a compute stream wait on an event.

4.36.1 Detailed Description

This section describes the stream management functions of the low-level CUDA driver application programming interface.

4.36.2 Function Documentation

4.36.2.1 CUresult cuStreamCreate (CUstream *phStream, unsigned int Flags)

Creates a stream and returns a handle in phStream. Flags is required to be 0.

Parameters:

  phStream - Returned newly created stream

  Flags - Parameters for stream creation (must be 0)

Returns:

  CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Note:

  Note that this function may also return error codes from previous, asynchronous launches.

See also:

  cuStreamDestroy, cuStreamWaitEvent, cuStreamQuery, cuStreamSynchronize
4.36.2.2 CUresult cuStreamDestroy (CUstream hStream)

Destroys the stream specified by hStream.

In the case that the device is still doing work in the stream hStream when cuStreamDestroy() is called, the function will return immediately and the resources associated with hStream will be released automatically once the device has completed all work in hStream.

Parameters:

hStream - Stream to destroy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuStreamCreate, cuStreamWaitEvent, cuStreamQuery, cuStreamSynchronize

4.36.2.3 CUresult cuStreamQuery (CUstream hStream)

Returns CUDA_SUCCESS if all operations in the stream specified by hStream have completed, or CUDA_ERROR_NOT_READY if not.

Parameters:

hStream - Stream to query status of

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_READY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuStreamCreate, cuStreamWaitEvent, cuStreamDestroy, cuStreamSynchronize

4.36.2.4 CUresult cuStreamSynchronize (CUstream hStream)

Waits until the device has completed all operations in the stream specified by hStream. If the context was created with the CU_CTX_SCHED_BLOCKING_SYNC flag, the CPU thread will block until the stream is finished with all of its tasks.

Parameters:

hStream - Stream to wait for
4.36 Stream Management

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuStreamCreate, cuStreamDestroy, cuStreamWaitEvent, cuStreamQuery

4.36.2.5 CUresult cuStreamWaitEvent (CUstream hStream, CUevent hEvent, unsigned int Flags)

Makes all future work submitted to hStream wait until hEvent reports completion before beginning execution. This synchronization will be performed efficiently on the device. The event hEvent may be from a different context than hStream, in which case this function will perform cross-device synchronization.

The stream hStream will wait only for the completion of the most recent host call to cuEventRecord() on hEvent. Once this call has returned, any functions (including cuEventRecord() and cuEventDestroy()) may be called on hEvent again, and the subsequent calls will not have any effect on hStream.

If hStream is 0 (the NULL stream) any future work submitted in any stream will wait for hEvent to complete before beginning execution. This effectively creates a barrier for all future work submitted to the context.

If cuEventRecord() has not been called on hEvent, this call acts as if the record has already completed, and so is a functional no-op.

Parameters:

hStream - Stream to wait
hEvent - Event to wait on (may not be NULL)
Flags - Parameters for the operation (must be 0)

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuStreamCreate, cuEventRecord, cuStreamQuery, cuStreamSynchronize, cuStreamDestroy
4.37  Event Management

Functions

- **CUresult cuEventCreate (CUevent *phEvent, unsigned int Flags)**
  Creates an event.

- **CUresult cuEventDestroy (CUevent hEvent)**
  Destroys an event.

- **CUresult cuEventElapsedTime (float *pMilliseconds, CUevent hStart, CUevent hEnd)**
  Computes the elapsed time between two events.

- **CUresult cuEventQuery (CUevent hEvent)**
  Queries an event’s status.

- **CUresult cuEventRecord (CUevent hEvent, CUstream hStream)**
  Records an event.

- **CUresult cuEventSynchronize (CUevent hEvent)**
  Waits for an event to complete.

4.37.1  Detailed Description

This section describes the event management functions of the low-level CUDA driver application programming interface.

4.37.2  Function Documentation

4.37.2.1  **CUresult cuEventCreate (CUevent * phEvent, unsigned int Flags)**

Creates an event *phEvent with the flags specified via Flags. Valid flags include:

- **CU_EVENT_DEFAULT**: Default event creation flag.
- **CU_EVENT_BLOCKING_SYNC**: Specifies that the created event should use blocking synchronization. A CPU thread that uses cuEventSynchronize() to wait on an event created with this flag will block until the event has actually been recorded.
- **CU_EVENT_DISABLE_TIMING**: Specifies that the created event does not need to record timing data. Events created with this flag specified and the CU_EVENT_BLOCKING_SYNC flag not specified will provide the best performance when used with cuStreamWaitEvent() and cuEventQuery().

Parameters:

- **phEvent** - Returns newly created event
- **Flags** - Event creation flags

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY
4.37 Event Management

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuEventRecord, cuEventQuery, cuEventSynchronize, cuEventDestroy, cuEventElapsedTime

4.37.2.2 CUresult cuEventDestroy (CUevent hEvent)

Destroys the event specified by hEvent.

In the case that hEvent has been recorded but has not yet been completed when cuEventDestroy() is called, the function will return immediately and the resources associated with hEvent will be released automatically once the device has completed hEvent.

Parameters:

hEvent - Event to destroy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuEventCreate, cuEventRecord, cuEventQuery, cuEventSynchronize, cuEventElapsedTime

4.37.2.3 CUresult cuEventElapsedTime (float *pMilliseconds, CUevent hStart, CUevent hEnd)

Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds).

If either event was last recorded in a non-NULL stream, the resulting time may be greater than expected (even if both used the same stream handle). This happens because the cuEventRecord() operation takes place asynchronously and there is no guarantee that the measured latency is actually just between the two events. Any number of other different stream operations could execute in between the two measured events, thus altering the timing in a significant way.

If cuEventRecord() has not been called on either event then CUDA_ERROR_INVALID_HANDLE is returned. If cuEventRecord() has been called on both events but one or both of them has not yet been completed (that is, cuEventQuery() would return CUDA_ERROR_NOT_READY on at least one of the events), CUDA_ERROR_NOT_READY is returned. If either event was created with the CU_EVENT_DISABLE_TIMING flag, then this function will return CUDA_ERROR_INVALID_HANDLE.

Parameters:

pMilliseconds - Time between hStart and hEnd in ms

hStart - Starting event

hEnd - Ending event
Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_READY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuEventCreate, cuEventRecord, cuEventQuery, cuEventSynchronize, cuEventDestroy

4.37.2.4 CUresult cuEventQuery (CUevent hEvent)

Query the status of all device work preceding the most recent call to cuEventRecord() (in the appropriate compute streams, as specified by the arguments to cuEventRecord()).

If this work has successfully been completed by the device, or if cuEventRecord() has not been called on hEvent, then CUDA_SUCCESS is returned. If this work has not yet been completed by the device then CUDA_ERROR_NOT_READY is returned.

Parameters:

hEvent - Event to query

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_READY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuEventCreate, cuEventRecord, cuEventSynchronize, cuEventDestroy, cuEventElapsedTime

4.37.2.5 CUresult cuEventRecord (CUevent hEvent, CUstream hStream)

Records an event. If hStream is non-zero, the event is recorded after all preceding operations in hStream have been completed; otherwise, it is recorded after all preceding operations in the CUDA context have been completed. Since operation is asynchronous, cuEventQuery and/or cuEventSynchronize() must be used to determine when the event has actually been recorded.

If cuEventRecord() has previously been called on hEvent, then this call will overwrite any existing state in hEvent. Any subsequent calls which examine the status of hEvent will only examine the completion of this most recent call to cuEventRecord().

It is necessary that hEvent and hStream be created on the same context.

Parameters:

hEvent - Event to record

hStream - Stream to record event for
4.37 Event Management

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuEventCreate, cuEventQuery, cuEventSynchronize, cuStreamWaitEvent, cuEventDestroy, cuEventElapsedTime

4.37.2.6 CUresult cuEventSynchronize (CUevent hEvent)

Wait until the completion of all device work preceding the most recent call to cuEventRecord() (in the appropriate compute streams, as specified by the arguments to cuEventRecord()).

If cuEventRecord() has not been called on hEvent, CUDA_SUCCESS is returned immediately.

Waiting for an event that was created with the CU_EVENT_BLOCKING_SYNC flag will cause the calling CPU thread to block until the event has been completed by the device. If the CU_EVENT_BLOCKING_SYNC flag has not been set, then the CPU thread will busy-wait until the event has been completed by the device.

Parameters:

hEvent - Event to wait for

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuEventCreate, cuEventRecord, cuEventQuery, cuEventDestroy, cuEventElapsedTime
4.38 Execution Control

Modules

- Execution Control [DEPRECATED]

Functions

- CUresult cuFuncGetAttribute (int *pi, CUfunction_attribute attrib, CUfunction hfunc)
  
  Returns information about a function.
  
- CUresult cuFuncSetCacheConfig (CUfunction hfunc, CUfunc_cache config)
  
  Sets the preferred cache configuration for a device function.
  
- CUresult cuLaunchKernel (CUfunction f, unsigned int gridDimX, unsigned int gridDimY, unsigned int gridDimZ, unsigned int blockDimX, unsigned int blockDimY, unsigned int blockDimZ, unsigned int sharedMemBytes, CUstream hStream, void **kernelParams, void **extra)
  
  Launches a CUDA function.

4.38.1 Detailed Description

This section describes the execution control functions of the low-level CUDA driver application programming interface.

4.38.2 Function Documentation

4.38.2.1 CUresult cuFuncGetAttribute (int *pi, CUfunction_attribute attrib, CUfunction hfunc)

Returns in *pi the integer value of the attribute attrib on the kernel given by hfunc. The supported attributes are:

- **CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK**: The maximum number of threads per block, beyond which a launch of the function would fail. This number depends on both the function and the device on which the function is currently loaded.

- **CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES**: The size in bytes of statically-allocated shared memory per block required by this function. This does not include dynamically-allocated shared memory requested by the user at runtime.

- **CU_FUNC_ATTRIBUTE_CONST_SIZE_BYTES**: The size in bytes of user-allocated constant memory required by this function.

- **CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES**: The size in bytes of local memory used by each thread of this function.

- **CU_FUNC_ATTRIBUTE_NUM_REGS**: The number of registers used by each thread of this function.

- **CU_FUNC_ATTRIBUTE_PTX_VERSION**: The PTX virtual architecture version for which the function was compiled. This value is the major PTX version * 10 + the minor PTX version, so a PTX version 1.3 function would return the value 13. Note that this may return the undefined value of 0 for cubins compiled prior to CUDA 3.0.
• **CU_FUNC_ATTRIBUTE_BINARY_VERSION**: The binary architecture version for which the function was compiled. This value is the major binary version \* 10 + the minor binary version, so a binary version 1.3 function would return the value 13. Note that this will return a value of 10 for legacy cubins that do not have a properly-encoded binary architecture version.

**Parameters:**

- `pi` - Returned attribute value
- `attrib` - Attribute requested
- `hfunc` - Function to query attribute of

**Returns:**

- `CUDA_SUCCESS`
- `CUDA_ERROR_DEINITIALIZED`
- `CUDA_ERROR_NOT_INITIALIZED`
- `CUDA_ERROR_INVALID_CONTEXT`
- `CUDA_ERROR_INVALID_HANDLE`
- `CUDA_ERROR_INVALID_VALUE`

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

cuCtxGetCacheConfig, cuCtxSetCacheConfig, cuFuncSetCacheConfig, cuLaunchKernel

---

### 4.38.2.2 CUresult cuFuncSetCacheConfig (CUfunction `hfunc`, CUfunc_cache `config`)

On devices where the L1 cache and shared memory use the same hardware resources, this sets through `config` the preferred cache configuration for the device function `hfunc`. This is only a preference. The driver will use the requested configuration if possible, but it is free to choose a different configuration if required to execute `hfunc`. Any context-wide preference set via `cuCtxSetCacheConfig()` will be overridden by this per-function setting unless the per-function setting is `CU_FUNC_CACHE_PREFER_NONE`. In that case, the current context-wide setting will be used.

This setting does nothing on devices where the size of the L1 cache and shared memory are fixed.

Launching a kernel with a different preference than the most recent preference setting may insert a device-side synchronization point.

The supported cache configurations are:

- **CU_FUNC_CACHE_PREFER_NONE**: no preference for shared memory or L1 (default)
- **CU_FUNC_CACHE_PREFER_SHARED**: prefer larger shared memory and smaller L1 cache
- **CU_FUNC_CACHE_PREFER_L1**: prefer larger L1 cache and smaller shared memory

**Parameters:**

- `hfunc` - Kernel to configure cache for
- `config` - Requested cache configuration

**Returns:**

- `CUDA_SUCCESS`
- `CUDA_ERROR_DEINITIALIZED`
- `CUDA_ERROR_NOT_INITIALIZED`
- `CUDA_ERROR_INVALID_CONTEXT`

---

Generated for NVIDIA CUDA Library by Doxygen
Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxGetCacheConfig, cuCtxSetCacheConfig, cuFuncGetAttribute, cuLaunchKernel

4.38.2.3 CUresult cuLaunchKernel (CUfunction f, unsigned int gridDimX, unsigned int gridDimY, unsigned int gridDimZ, unsigned int blockDimX, unsigned int blockDimY, unsigned int blockDimZ, unsigned int sharedMemBytes, CUstream hStream, void **kernelParams, void **extra)

Invokes the kernel f on a gridDimX x gridDimY x gridDimZ grid of blocks. Each block contains blockDimX x blockDimY x blockDimZ threads.

sharedMemBytes sets the amount of dynamic shared memory that will be available to each thread block.

cuLaunchKernel() can optionally be associated to a stream by passing a non-zero hStream argument.

Kernel parameters to f can be specified in one of two ways:

1) Kernel parameters can be specified via kernelParams. If f has N parameters, then kernelParams needs to be an array of N pointers. Each of kernelParams[0] through kernelParams[N-1] must point to a region of memory from which the actual kernel parameter will be copied. The number of kernel parameters and their offsets and sizes do not need to be specified as that information is retrieved directly from the kernel’s image.

2) Kernel parameters can also be packaged by the application into a single buffer that is passed in via the extra parameter. This places the burden on the application of knowing each kernel parameter’s size and alignment/padding within the buffer. Here is an example of using the extra parameter in this manner:

```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 extra parameter exists to allow cuLaunchKernel to take additional less commonly used arguments. extra specifies a list of names of extra settings and their corresponding values. Each extra setting name is immediately followed by the corresponding value. The list must be terminated with either NULL or CU_LAUNCH_PARAM_-END.

- CU_LAUNCH_PARAM_END, which indicates the end of the extra array;
- CU_LAUNCH_PARAM_BUFFER_POINTER, which specifies that the next value in extra will be a pointer to a buffer containing all the kernel parameters for launching kernel f;
- CU_LAUNCH_PARAM_BUFFER_SIZE, which specifies that the next value in extra will be a pointer to a size_t containing the size of the buffer specified with CU_LAUNCH_PARAM_BUFFER_POINTER;

The error CUDA_ERROR_INVALID_VALUE will be returned if kernel parameters are specified with both kernelParams and extra (i.e. both kernelParams and extra are non-NULL).

Calling cuLaunchKernel() sets persistent function state that is the same as function state set through the following deprecated APIs:
cuFuncSetBlockShape() cuFuncSetSharedSize() cuParamSetSize() cuParamSeti() cuParamSetf() cuParamSetv()

When the kernel \( f \) is launched via cuLaunchKernel(), the previous block shape, shared size and parameter info associated with \( f \) is overwritten.

Note that to use cuLaunchKernel(), the kernel \( f \) must either have been compiled with toolchain version 3.2 or later so that it will contain kernel parameter information, or have no kernel parameters. If either of these conditions is not met, then cuLaunchKernel() will return CUDA_ERROR_INVALID_IMAGE.

Parameters:

- \( f \) - Kernel to launch
- \( \text{gridDim}_X \) - Width of grid in blocks
- \( \text{gridDim}_Y \) - Height of grid in blocks
- \( \text{gridDim}_Z \) - Depth of grid in blocks
- \( \text{blockDim}_X \) - X dimension of each thread block
- \( \text{blockDim}_Y \) - Y dimension of each thread block
- \( \text{blockDim}_Z \) - Z dimension of each thread block
- \( \text{sharedMemBytes} \) - Dynamic shared-memory size per thread block in bytes
- \( \text{hStream} \) - Stream identifier
- \( \text{kernelParams} \) - Array of pointers to kernel parameters
- \( \text{extra} \) - Extra options

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_IMAGE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_LAUNCH_FAILED, CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES, CUDA_ERROR_LAUNCH_TIMEOUT, CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxGetCacheConfig, cuCtxSetCacheConfig, cuFuncSetCacheConfig, cuFuncGetAttribute,
4.39  Execution Control [DEPRECATED]

Functions

- **CUresult cuFuncSetBlockShape (CUfunction hfunc, int x, int y, int z)**
  
  Sets the block-dimensions for the function.

- **CUresult cuFuncSetSharedSize (CUfunction hfunc, unsigned int bytes)**
  
  Sets the dynamic shared-memory size for the function.

- **CUresult cuLaunch (CUfunction f)**
  
  Launches a CUDA function.

- **CUresult cuLaunchGrid (CUfunction f, int grid_width, int grid_height)**
  
  Launches a CUDA function.

- **CUresult cuLaunchGridAsync (CUfunction f, int grid_width, int grid_height, CUstream hStream)**
  
  Launches a CUDA function.

- **CUresult cuParamSetf (CUfunction hfunc, int offset, float value)**
  
  Adds a floating-point parameter to the function’s argument list.

- **CUresult cuParamSeti (CUfunction hfunc, int offset, unsigned int value)**
  
  Adds an integer parameter to the function’s argument list.

- **CUresult cuParamSetSize (CUfunction hfunc, unsigned int numbytes)**
  
  Sets the parameter size for the function.

- **CUresult cuParamSetTexRef (CUfunction hfunc, int texunit, CUtexref hTexRef)**
  
  Adds a texture-reference to the function’s argument list.

- **CUresult cuParamSetv (CUfunction hfunc, int offset, void *ptr, unsigned int numbytes)**
  
  Adds arbitrary data to the function’s argument list.

---

4.39.1  Detailed Description

This section describes the deprecated execution control functions of the low-level CUDA driver application programming interface.

4.39.2  Function Documentation

4.39.2.1  **CUresult cuFuncSetBlockShape (CUfunction hfunc, int x, int y, int z)**

**Deprecated**

Specifies the $x$, $y$, and $z$ dimensions of the thread blocks that are created when the kernel given by $hfunc$ is launched.
Parameters:

\( hfunc \) - Kernel to specify dimensions of
\( x \) - X dimension
\( y \) - Y dimension
\( z \) - Z dimension

Returns:

\texttt{CUDA\_SUCCESS}, \texttt{CUDA\_ERROR\_DEINITIALIZED}, \texttt{CUDA\_ERROR\_NOT\_INITIALIZED}, \texttt{CUDA\_ERROR\_INVALID\_CONTEXT}, \texttt{CUDA\_ERROR\_INVALID\_HANDLE}, \texttt{CUDA\_ERROR\_INVALID\_VALUE}

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

\texttt{cuFuncSetSharedSize}, \texttt{cuFuncSetCacheConfig}, \texttt{cuFuncGetAttribute}, \texttt{cuParamSetSize}, \texttt{cuParamSeti}, \texttt{cuParamSetf}, \texttt{cuParamSetv}, \texttt{cuLaunch}, \texttt{cuLaunchGrid}, \texttt{cuLaunchGridAsync}, \texttt{cuLaunchKernel}

4.39.2.2 \texttt{CUresult cuFuncSetSharedSize (CUfunction hfunc, unsigned int bytes)}

Deprecated

Sets through \( \texttt{bytes} \) the amount of dynamic shared memory that will be available to each thread block when the kernel given by \( \texttt{hfunc} \) is launched.

Parameters:

\( \texttt{hfunc} \) - Kernel to specify dynamic shared-memory size for
\( \texttt{bytes} \) - Dynamic shared-memory size per thread in bytes

Returns:

\texttt{CUDA\_SUCCESS}, \texttt{CUDA\_ERROR\_DEINITIALIZED}, \texttt{CUDA\_ERROR\_NOT\_INITIALIZED}, \texttt{CUDA\_ERROR\_INVALID\_CONTEXT}, \texttt{CUDA\_ERROR\_INVALID\_HANDLE}, \texttt{CUDA\_ERROR\_INVALID\_VALUE}

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

\texttt{cuFuncSetBlockShape}, \texttt{cuFuncSetCacheConfig}, \texttt{cuFuncGetAttribute}, \texttt{cuParamSetSize}, \texttt{cuParamSeti}, \texttt{cuParamSetf}, \texttt{cuParamSetv}, \texttt{cuLaunch}, \texttt{cuLaunchGrid}, \texttt{cuLaunchGridAsync}, \texttt{cuLaunchKernel}

4.39.2.3 \texttt{CUresult cuLaunch (CUfunction f)}

Deprecated
Invokes the kernel $f$ on a $1 \times 1 \times 1$ grid of blocks. The block contains the number of threads specified by a previous call to `cuFuncSetBlockShape()`.

Parameters:

$f$ - Kernel to launch

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_LAUNCH_FAILED, CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES, CUDA_ERROR_LAUNCH_TIMEOUT, CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSeti, cuParamSetv, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel

### 4.39.2.4 CUresult cuLaunchGrid (CUfunction $f$, int grid_width, int grid_height)

Deprecated

Invokes the kernel $f$ on a grid_width x grid_height grid of blocks. Each block contains the number of threads specified by a previous call to `cuFuncSetBlockShape()`.

Parameters:

$f$ - Kernel to launch

grid_width - Width of grid in blocks

grid_height - Height of grid in blocks

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_LAUNCH_FAILED, CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES, CUDA_ERROR_LAUNCH_TIMEOUT, CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSeti, cuParamSetv, cuLaunch, cuLaunchGridAsync, cuLaunchKernel
4.39 Execution Control [DEPRECATED]

4.39.2.5 CResult cuLaunchGridAsync (CUfunction f, int grid_width, int grid_height, CUstream hStream)

Deprecated

Invokes the kernel f on a grid_width x grid_height grid of blocks. Each block contains the number of threads specified by a previous call to cuFuncSetBlockShape().

cuLaunchGridAsync() can optionally be associated to a stream by passing a non-zero hStream argument.

Parameters:

- f - Kernel to launch
- grid_width - Width of grid in blocks
- grid_height - Height of grid in blocks
- hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_LAUNCH_FAILED, CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES, CUDA_ERROR_LAUNCH_TIMEOUT, CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:


4.39.2.6 CResult cuParamSetf (CUfunction hfunc, int offset, float value)

Deprecated

Sets a floating-point parameter that will be specified the next time the kernel corresponding to hfunc will be invoked. offset is a byte offset.

Parameters:

- hfunc - Kernel to add parameter to
- offset - Offset to add parameter to argument list
- value - Value of parameter

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE
**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

`cuFuncSetBlockShape`, `cuFuncSetSharedSize`, `cuFuncGetAttribute`, `cuParamSetSize`, `cuParamSeti`, `cuParamSetv`, `cuLaunch`, `cuLaunchGrid`, `cuLaunchGridAsync`, `cuLaunchKernel`

### 4.39.2.7 CUresult cuParamSeti (CUfunction hfunc, int offset, unsigned int value)

**Deprecated**

Sets an integer parameter that will be specified the next time the kernel corresponding to hfunc will be invoked. offset is a byte offset.

**Parameters:**

- `hfunc` - Kernel to add parameter to
- `offset` - Offset to add parameter to argument list
- `value` - Value of parameter

**Returns:**

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

`cuFuncSetBlockShape`, `cuFuncSetSharedSize`, `cuFuncGetAttribute`, `cuParamSetSize`, `cuParamSetf`, `cuParamSetv`, `cuLaunch`, `cuLaunchGrid`, `cuLaunchGridAsync`, `cuLaunchKernel`

### 4.39.2.8 CUresult cuParamSetSize (CUfunction hfunc, unsigned int numbytes)

**Deprecated**

Sets through numbytes the total size in bytes needed by the function parameters of the kernel corresponding to hfunc.

**Parameters:**

- `hfunc` - Kernel to set parameter size for
- `numbytes` - Size of parameter list in bytes

**Returns:**

`CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`
Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetf, cuParamSeti, cuParamSetv, cuLaunch, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel

4.39.2.9 CUresult cuParamSetTexRef (CUfunction hfunc, int texunit, CUtexref hTexRef)

Deprecated

Makes the CUDA array or linear memory bound to the texture reference hTexRef available to a device program as a texture. In this version of CUDA, the texture-reference must be obtained via cuModuleGetTexRef() and the texunit parameter must be set to CU_PARAM_TR_DEFAULT.

Parameters:

  hfunc  - Kernel to add texture-reference to
  texunit - Texture unit (must be CU_PARAM_TR_DEFAULT)
  hTexRef - Texture-reference to add to argument list

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

4.39.2.10 CUresult cuParamSetv (CUfunction hfunc, int offset, void * ptr, unsigned int numbytes)

Deprecated

Copies an arbitrary amount of data (specified in numbytes) from ptr into the parameter space of the kernel corresponding to hfunc. offset is a byte offset.

Parameters:

  hfunc  - Kernel to add data to
  offset - Offset to add data to argument list
  ptr - Pointer to arbitrary data
  numbytes - Size of data to copy in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE
Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuFuncSetBlockSize, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSeti,
cuLaunch, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel
4.40 Texture Reference Management

Modules

- Texture Reference Management [DEPRECATED]

Functions

- CUresult cuTexRefGetAddress (CUdeviceptr *pdpTR, CUtexref hTexRef)
  
  Gets the address associated with a texture reference.

- CUresult cuTexRefGetAddressMode (CUaddress_mode *pam, CUtexref hTexRef, int dim)
  
  Gets the addressing mode used by a texture reference.

- CUresult cuTexRefGetArray (CUarray *phArray, CUtexref hTexRef)
  
  Gets the array bound to a texture reference.

- CUresult cuTexRefGetFilterMode (CUfilter_mode *pfm, CUtexref hTexRef)
  
  Gets the filter-mode used by a texture reference.

- CUresult cuTexRefGetFlags (unsigned int *pFlags, CUtexref hTexRef)
  
  Gets the flags used by a texture reference.

- CUresult cuTexRefGetFormat (CUarray_format *pFormat, int *pNumChannels, CUtexref hTexRef)
  
  Gets the format used by a texture reference.

- CUresult cuTexRefSetAddress (size_t *ByteOffset, CUtexref hTexRef, CUdeviceptr dpTR, size_t bytes)
  
  Binds an address as a texture reference.

- CUresult cuTexRefSetAddress2D (CUtexref hTexRef, const CUDA_ARRAY_DESCRIPTOR *desc, CUdeviceptr dpTR, size_t Pitch)
  
  Binds an address as a 2D texture reference.

- CUresult cuTexRefSetAddressMode (CUtexref hTexRef, int dim, CUaddress_mode am)
  
  Sets the addressing mode for a texture reference.

- CUresult cuTexRefSetArray (CUtexref hTexRef, CUarray hArray, unsigned int Flags)
  
  Binds an array as a texture reference.

- CUresult cuTexRefSetFilterMode (CUtexref hTexRef, CUfilter_mode fm)
  
  Sets the filtering mode for a texture reference.

- CUresult cuTexRefSetFlags (CUtexref hTexRef, unsigned int Flags)
  
  Sets the flags for a texture reference.

- CUresult cuTexRefSetFormat (CUtexref hTexRef, CUarray_format fmt, int NumPackedComponents)
  
  Sets the format for a texture reference.
4.40.1 Detailed Description

This section describes the texture reference management functions of the low-level CUDA driver application programming interface.

4.40.2 Function Documentation

4.40.2.1 CUresult cuTexRefGetAddress (CUdeviceptr *pdptr, CUtexref hTexRef)

Returns in *pdptr the base address bound to the texture reference hTexRef, or returns CUDA_ERROR_-INVALID_VALUE if the texture reference is not bound to any device memory range.

Parameters:

- pdptr - Returned device address
- hTexRef - Texture reference

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

4.40.2.2 CUresult cuTexRefGetAddressMode (CUaddress_mode *pam, CUtexref hTexRef, int dim)

Returns in *pam the addressing mode corresponding to the dimension dim of the texture reference hTexRef. Currently, the only valid value for dim are 0 and 1.

Parameters:

- pam - Returned addressing mode
- hTexRef - Texture reference
- dim - Dimension

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat
4.40.2.3 \textbf{CUresult cuTexRefGetArray (CUarray \textbullet \textit{phArray}, CUtexref \textit{hTexRef})}

Returns in *\textit{phArray} the CUDA array bound to the texture reference \textit{hTexRef}, or returns \texttt{CUDA_ERROR_-INVALID_VALUE} if the texture reference is not bound to any CUDA array.

Parameters:

\textit{phArray} - Returned array

\textit{hTexRef} - Texture reference

Returns:

\texttt{CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE}

See also:

\texttt{cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFil-
terMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRe-
fGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat}

4.40.2.4 \textbf{CUresult cuTexRefGetFilterMode (CUfilter_mode \textbullet \textit{pfm}, CUtexref \textit{hTexRef})}

Returns in *\textit{pfm} the filtering mode of the texture reference \textit{hTexRef}.

Parameters:

\textit{pfm} - Returned filtering mode

\textit{hTexRef} - Texture reference

Returns:

\texttt{CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE}

See also:

\texttt{cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFil-
terMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRe-
fGetArray, cuTexRefGetFlags, cuTexRefGetFormat}

4.40.2.5 \textbf{CUresult cuTexRefGetFlags (unsigned int \textbullet \textit{pFlags}, CUtexref \textit{hTexRef})}

Returns in *\textit{pFlags} the flags of the texture reference \textit{hTexRef}.

Parameters:

\textit{pFlags} - Returned flags

\textit{hTexRef} - Texture reference

Returns:

\texttt{CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE}
See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFormat

4.40.2.6 CUresult cuTexRefGetFormat (CUarray_format *pFormat, int *pNumChannels, CUtexref hTexRef)

Returns in *pFormat and *pNumChannels the format and number of components of the CUDA array bound to the texture reference hTexRef. If pFormat or pNumChannels is NULL, it will be ignored.

Parameters:

pFormat - Returned format
pNumChannels - Returned number of components
hTexRef - Texture reference

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags

4.40.2.7 CUresult cuTexRefSetAddress (size_t *ByteOffset, CUtexref hTexRef, CUdeviceptr dptr, size_t bytes)

Binds a linear address range to the texture reference hTexRef. Any previous address or CUDA array state associated with the texture reference is superseded by this function. Any memory previously bound to hTexRef is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, cuTexRefSetAddress() passes back a byte offset in *ByteOffset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the tex1Dfetch() function.

If the device memory pointer was returned from cuMemAlloc(), the offset is guaranteed to be 0 and NULL may be passed as the ByteOffset parameter.

Parameters:

ByteOffset - Returned byte offset
hTexRef - Texture reference to bind
dptr - Device pointer to bind
bytes - Size of memory to bind in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE
4.40 Texture Reference Management

See also:

cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

4.40.2.8 CUresult cuTexRefSetAddress2D (CUtexref hTexRef, const CUDA_ARRAY_DESCRIPTOR *desc, C Ud eviceptr dptr, size_t Pitch)

Binds a linear address range to the texture reference hTexRef. Any previous address or CUDA array state associated with the texture reference is superseded by this function. Any memory previously bound to hTexRef is unbound.

Using a tex2D() function inside a kernel requires a call to either cuTexRefSetArray() to bind the corresponding texture reference to an array, or cuTexRefSetAddress2D() to bind the texture reference to linear memory.

Function calls to cuTexRefSetFormat() cannot follow calls to cuTexRefSetAddress2D() for the same texture reference.

It is required that dptr be aligned to the appropriate hardware-specific texture alignment. You can query this value using the device attribute CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT. If an unaligned dptr is supplied, CUDA_ERROR_INVALID_VALUE is returned.

Parameters:

- **hTexRef** - Texture reference to bind
- **desc** - Descriptor of CUDA array
- **dptr** - Device pointer to bind
- **Pitch** - Line pitch in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

4.40.2.9 CUresult cuTexRefSetAddressMode (CUtexref hTexRef, int dim, CUaddress_mode am)

Specifies the addressing mode am for the given dimension dim of the texture reference hTexRef. If dim is zero, the addressing mode is applied to the first parameter of the functions used to fetch from the texture; if dim is 1, the second, and so on. CUaddress_mode is defined as:

```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.
Parameters:

- \textit{hTexRef} - Texture reference
- \textit{dim} - Dimension
- \textit{am} - Addressing mode to set

Returns:

- \textit{CUDA_SUCCESS}, \textit{CUDA_ERROR_DEINITIALIZED}, \textit{CUDA_ERROR_NOT_INITIALIZED}, \textit{CUDA_ERROR_INVALID_CONTEXT}, \textit{CUDA_ERROR_INVALID_VALUE}

See also:

- cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

4.40.2.10 \texttt{CUresult cuTexRefSetArray (CUtexref hTexRef, CUarray hArray, unsigned int Flags)}

Binds the CUDA array \textit{hArray} to the texture reference \textit{hTexRef}. Any previous address or CUDA array state associated with the texture reference is superseded by this function. \textit{Flags} must be set to \texttt{CU_TRSA_OVERRIDE_FORMAT}. Any CUDA array previously bound to \textit{hTexRef} is unbound.

Parameters:

- \textit{hTexRef} - Texture reference to bind
- \textit{hArray} - Array to bind
- \textit{Flags} - Options (must be \texttt{CU_TRSA_OVERRIDE_FORMAT})

Returns:

- \textit{CUDA_SUCCESS}, \textit{CUDA_ERROR_DEINITIALIZED}, \textit{CUDA_ERROR_NOT_INITIALIZED}, \textit{CUDA_ERROR_INVALID_CONTEXT}, \textit{CUDA_ERROR_INVALID_VALUE}

See also:

- cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

4.40.2.11 \texttt{CUresult cuTexRefSetFilterMode (CUtexref hTexRef, CUfilter_mode fm)}

Specifies the filtering mode \textit{fm} to be used when reading memory through the texture reference \textit{hTexRef}. \texttt{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 \textit{hTexRef} is bound to linear memory.

Parameters:

- \textit{hTexRef} - Texture reference
4.40 Texture Reference Management

\( fm \) - Filtering mode to set

Returns:

- CUDA_SUCCESS,
- CUDA_ERROR_DEINITIALIZED,
- CUDA_ERROR_NOT_INITIALIZED,
- CUDA_ERROR_INVALID_CONTEXT,
- CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

4.40.2.12 CUresult cuTexRefSetFlags (CUtexref hTexRef, unsigned int Flags)

Specifies optional flags via Flags to specify the behavior of data returned through the texture reference hTexRef. The valid flags are:

- \texttt{CU_TRSF\_READ\_AS\_INTEGER}, which suppresses the default behavior of having the texture promote integer data to floating point data in the range \([0, 1]\);
- \texttt{CU_TRSF\_NORMALIZED\_COORDINATES}, which suppresses the default behavior of having the texture coordinates range from \([0, \text{Dim}]\) where Dim is the width or height of the CUDA array. Instead, the texture coordinates \([0, 1.0]\) reference the entire breadth of the array dimension;

Parameters:

- \textit{hTexRef} - Texture reference
- \textit{Flags} - Optional flags to set

Returns:

- CUDA_SUCCESS,
- CUDA_ERROR_DEINITIALIZED,
- CUDA_ERROR_NOT_INITIALIZED,
- CUDA_ERROR_INVALID_CONTEXT,
- CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

4.40.2.13 CUresult cuTexRefSetFormat (CUtexref hTexRef, CUarray_format fmt, int NumPackedComponents)

Specifies the format of the data to be read by the texture reference hTexRef. fmt and NumPackedComponents are exactly analogous to the Format and NumChannels members of the CUDA\_ARRAY\__DESCRIPTOR structure. They specify the format of each component and the number of components per array element.

Parameters:

- \textit{hTexRef} - Texture reference
- \textit{fmt} - Format to set
- \textit{NumPackedComponents} - Number of components per array element

Generated for NVIDIA CUDA Library by Doxygen
Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat
4.41 Texture Reference Management [DEPRECATED]

Functions

- **CUresult cuTexRefCreate (CUtexref *pTexRef)**
  
  Creates a texture reference.

- **CUresult cuTexRefDestroy (CUtexref hTexRef)**
  
  Destroys a texture reference.

4.41.1 Detailed Description

This section describes the deprecated texture reference management functions of the low-level CUDA driver application programming interface.

4.41.2 Function Documentation

4.41.2.1 **CUresult cuTexRefCreate (CUtexref * pTexRef)**

*Deprecated*

Creates a texture reference and returns its handle in *pTexRef*. Once created, the application must call cuTexRefSetArray() or cuTexRefSetAddress() to associate the reference with allocated memory. Other texture reference functions are used to specify the format and interpretation (addressing, filtering, etc.) to be used when the memory is read through this texture reference.

Parameters:

- pTexRef - Returned texture reference

Returns:

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

- cuTexRefDestroy

4.41.2.2 **CUresult cuTexRefDestroy (CUtexref hTexRef)**

*Deprecated*

Destroys the texture reference specified by hTexRef.

Parameters:

- hTexRef - Texture reference to destroy
Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefCreate
4.42 Surface Reference Management

Functions

- CUresult cuSurfRefGetArray (CUarray *phArray, CUsurfref hSurfRef)
  
  Passes back the CUDA array bound to a surface reference.

- CUresult cuSurfRefSetArray (CUsurfref hSurfRef, CUarray hArray, unsigned int Flags)
  
  Sets the CUDA array for a surface reference.

4.42.1 Detailed Description

This section describes the surface reference management functions of the low-level CUDA driver application programming interface.

4.42.2 Function Documentation

4.42.2.1 CUresult cuSurfRefGetArray (CUarray *phArray, CUsurfref hSurfRef)

Returns in *phArray the CUDA array bound to the surface reference hSurfRef, or returns CUDA_ERROR_INVALID_VALUE if the surface reference is not bound to any CUDA array.

Parameters:

  *phArray - Surface reference handle
  hSurfRef - Surface reference handle

Returns:

  CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

  cuModuleGetSurfRef, cuSurfRefSetArray

4.42.2.2 CUresult cuSurfRefSetArray (CUsurfref hSurfRef, CUarray hArray, unsigned int Flags)

Sets the CUDA array hArray to be read and written by the surface reference hSurfRef. Any previous CUDA array state associated with the surface reference is superseded by this function. Flags must be set to 0. The CUDA_3D_SURFACE_LDST flag must have been set for the CUDA array. Any CUDA array previously bound to hSurfRef is unbound.

Parameters:

  hSurfRef - Surface reference handle
  hArray - CUDA array handle
  Flags - set to 0
Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuModuleGetSurfRef, cuSurfRefGetArray
4.43 Peer Context Memory Access

Functions

- **CUresult cuCtxDisablePeerAccess (CUcontext peerContext)**
  
  Disables direct access to memory allocations in a peer context and unregisters any registered allocations.

- **CUresult cuCtxEnablePeerAccess (CUcontext peerContext, unsigned int Flags)**
  
  Enables direct access to memory allocations in a peer context.

- **CUresult cuDeviceCanAccessPeer (int *canAccessPeer, CUdevice dev, CUdevice peerDev)**
  
  Queries if a device may directly access a peer device's memory.

4.43.1 Detailed Description

This section describes the direct peer context memory access functions of the low-level CUDA driver application programming interface.

4.43.2 Function Documentation

4.43.2.1 **CUresult cuCtxDisablePeerAccess (CUcontext peerContext)**

Returns **CUDA_ERROR_PEER_ACCESS_NOT_ENABLED** if direct peer access has not yet been enabled from **peerContext** to the current context.

Returns **CUDA_ERROR_INVALID_CONTEXT** if there is no current context, or if **peerContext** is not a valid context.

Parameters:

- **peerContext** - Peer context to disable direct access to

Returns:

- **CUDA_SUCCESS**, **CUDA_ERROR_DEINITIALIZED**, **CUDA_ERROR_NOT_INITIALIZED**, **CUDA_-ERROR_PEER_ACCESS_NOT_ENABLED**, **CUDA_ERROR_INVALID_CONTEXT**,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuDeviceCanAccessPeer, cuCtxEnablePeerAccess

4.43.2.2 **CUresult cuCtxEnablePeerAccess (CUcontext peerContext, unsigned int Flags)**

If both the current context and **peerContext** are on devices which support unified addressing (as may be queried using **CU DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING**), then on success all allocations from **peerContext** will immediately be accessible by the current context. See Unified Addressing for additional details.
Note that access granted by this call is unidirectional and that in order to access memory from the current context in `peerContext`, a separate symmetric call to `cuCtxEnablePeerAccess()` is required.

Returns `CUDA_ERROR_INVALID_DEVICE` if `cuDeviceCanAccessPeer()` indicates that the `CUdevice` of the current context cannot directly access memory from the `CUdevice` of `peerContext`.

Returns `CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED` if direct access of `peerContext` from the current context has already been enabled.

Returns `CUDA_ERROR_INVALID_CONTEXT` if there is no current context, `peerContext` is not a valid context, or if the current context is `peerContext`.

Returns `CUDA_ERROR_INVALID_VALUE` if `Flags` is not 0.

Parameters:

- `peerContext` - Peer context to enable direct access to from the current context
- `Flags` - Reserved for future use and must be set to 0

Returns:

- `CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_DEVICE`, `CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- `cuDeviceCanAccessPeer`, `cuCtxDisablePeerAccess`

4.43.2.3 `CUresult cuDeviceCanAccessPeer (int * canAccessPeer, CUdevice dev, CUdevice peerDev)`

Returns in `*canAccessPeer` a value of 1 if contexts on `dev` are capable of directly accessing memory from contexts on `peerDev` and 0 otherwise. If direct access of `peerDev` from `dev` is possible, then access may be enabled on two specific contexts by calling `cuCtxEnablePeerAccess()`.

Parameters:

- `canAccessPeer` - Returned access capability
- `dev` - Device from which allocations on `peerDev` are to be directly accessed.
- `peerDev` - Device on which the allocations to be directly accessed by `dev` reside.

Returns:

- `CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_DEVICE`

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- `cuCtxEnablePeerAccess`, `cuCtxDisablePeerAccess`
4.44 Graphics Interoperability

Functions

- CUresult cuGraphicsMapResources (unsigned int count, CUgraphicsResource *resources, CUstream hStream)
  Map graphics resources for access by CUDA.

- CUresult cuGraphicsResourceGetMappedPointer (CUdeviceptr *pDevPtr, size_t *pSize, CUgraphicsResource resource)
  Get a device pointer through which to access a mapped graphics resource.

- CUresult cuGraphicsResourceSetMapFlags (CUgraphicsResource resource, unsigned int flags)
  Set usage flags for mapping a graphics resource.

- CUresult cuGraphicsSubResourceGetMappedArray (CUarray *pArray, CUgraphicsResource resource, unsigned int arrayIndex, unsigned int mipLevel)
  Get an array through which to access a subresource of a mapped graphics resource.

- CUresult cuGraphicsUnmapResources (unsigned int count, CUgraphicsResource *resources, CUstream hStream)
  Unmap graphics resources.

- CUresult cuGraphicsUnregisterResource (CUgraphicsResource resource)
  Unregisters a graphics resource for access by CUDA.

4.44.1 Detailed Description

This section describes the graphics interoperability functions of the low-level CUDA driver application programming interface.

4.44.2 Function Documentation

4.44.2.1 CUresult cuGraphicsMapResources (unsigned int count, CUgraphicsResource *resources, CUstream hStream)

Maps the count graphics resources in resources for access by CUDA.

The resources in resources may be accessed by CUDA until they are unmapped. The graphics API from which resources were registered should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any graphics calls issued before cuGraphicsMapResources() will complete before any subsequent CUDA work issued in stream begins.

If resources includes any duplicate entries then CUDA_ERROR_INVALID_HANDLE is returned. If any of resources are presently mapped for access by CUDA then CUDA_ERROR_ALREADY_MAPPED is returned.

Parameters:

  count - Number of resources to map
resources - Resources to map for CUDA usage
hStream - Stream with which to synchronize

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

4.44.2.2 CUresult cuGraphicsResourceGetMappedPointer (CUdeviceptr *pDevPtr, size_t *pSize, CUgraphicsResource resource)

Returns in *pDevPtr a pointer through which the mapped graphics resource resource may be accessed. Returns in pSize the size of the memory in bytes which may be accessed from that pointer. The value set in pPointer may change every time that resource is mapped.

If resource is not a buffer then it cannot be accessed via a pointer and CUDA_ERROR_NOT_MAPPED_AS_POINTER is returned. If resource is not mapped then CUDA_ERROR_NOT_MAPPED is returned.

Parameters:

pDevPtr - Returned pointer through which resource may be accessed
pSize - Returned size of the buffer accessible starting at *pPointer
resource - Mapped resource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED CUDA_ERROR_NOT_MAPPED_AS_POINTER

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:
cuGraphicsMapResources, cuGraphicsSubResourceGetMappedArray

4.44.2.3 CUresult cuGraphicsResourceSetMapFlags (CUgraphicsResource resource, unsigned int flags)

Set flags for mapping the graphics resource resource.

Changes to flags will take effect the next time resource is mapped. The flags argument may be any of the following:
4.44 Graphics Interoperability

- **CU_GRAPHICS_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_GRAPHICS_MAPRESOURCE_FLAGS_READONLY**: Specifies that CUDA kernels which access this resource will not write to this resource.

- **CU_GRAPHICS_MAPRESOURCE_FLAGS_WRITEDISCARD**: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If `resource` is presently mapped for access by CUDA then `CUDA_ERROR_ALREADY_MAPPED` is returned. If `flags` is not one of the above values then `CUDA_ERROR_INVALID_VALUE` is returned.

**Parameters:**

- `resource` - Registered resource to set flags for
- `flags` - Parameters for resource mapping

**Returns:**

- `CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_INVALID_HANDLE`, `CUDA_ERROR_ALREADY_MAPPED`

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

- `cuGraphicsMapResources`

### 4.44.2.4 CUresult cuGraphicsSubResourceGetMappedArray (CUarray *pArray, CUgraphicsResource resource, unsigned int arrayIndex, unsigned int mipLevel)

Returns in `pArray` an array through which the subresource of the mapped graphics resource `resource` which corresponds to array index `arrayIndex` and mipmap level `mipLevel` may be accessed. The value set in `pArray` may change every time that `resource` is mapped.

If `resource` is not a texture then it cannot be accessed via an array and `CUDA_ERROR_NOT_MAPPED_AS_ARRAY` is returned. If `arrayIndex` is not a valid array index for `resource` then `CUDA_ERROR_INVALID_VALUE` is returned. If `mipLevel` is not a valid mipmap level for `resource` then `CUDA_ERROR_INVALID_VALUE` is returned. If `resource` is not mapped then `CUDA_ERROR_NOT_MAPPED` is returned.

**Parameters:**

- `pArray` - Returned array through which a subresource of `resource` may be accessed
- `resource` - Mapped resource to access
- `arrayIndex` - Array index for array textures or cubemap face index as defined by `CUarray_cubemap_face` for cubemap textures for the subresource to access
- `mipLevel` - Mipmap level for the subresource to access

Generated for NVIDIA CUDA Library by Doxygen
Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED CUDA_ERROR_NOT_MAPPED_AS_ARRAY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsResourceGetMappedPointer

4.44.2.5 CUresult cuGraphicsUnmapResources (unsigned int count, CUgraphicsResource *resources, CUstream hStream)

Unmaps the count graphics resources in resources.

Once unmapped, the resources in resources may not be accessed by CUDA until they are mapped again.

This function provides the synchronization guarantee that any CUDA work issued in stream before cuGraphicsUnmapResources() will complete before any subsequently issued graphics work begins.

If resources includes any duplicate entries then CUDA_ERROR_INVALID_HANDLE is returned. If any of resources are not presently mapped for access by CUDA then CUDA_ERROR_NOT_MAPPED is returned.

Parameters:

  count - Number of resources to unmap
  resources - Resources to unmap
  hStream - Stream with which to synchronize

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsMapResources

4.44.2.6 CUresult cuGraphicsUnregisterResource (CUgraphicsResource resource)

Unregisters the graphics resource resource so it is not accessible by CUDA unless registered again.

If resource is invalid then CUDA_ERROR_INVALID_HANDLE is returned.

Parameters:

  resource - Resource to unregister
Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

4.45 OpenGL Interoperability

Modules

- OpenGL Interoperability [DEPRECATED]

Functions

- **CUresult cuGLCtxCreate (CUcontext *pCtx, unsigned int Flags, CUdevice device)**
  
  Create a CUDA context for interoperability with OpenGL.

- **CUresult cuGraphicsGLRegisterBuffer (CUgraphicsResource *pCudaResource, GLuint buffer, unsigned int Flags)**
  
  Registers an OpenGL buffer object.

- **CUresult cuGraphicsGLRegisterImage (CUgraphicsResource *pCudaResource, GLuint image, GLenum target, unsigned int Flags)**
  
  Register an OpenGL texture or renderbuffer object.

- **CUresult cuWGLGetDevice (CUdevice *pDevice, HGPUNV hGpu)**
  
  Gets the CUDA device associated with hGpu.

4.45.1 Detailed Description

This section describes the OpenGL interoperability functions of the low-level CUDA driver application programming interface.

4.45.2 Function Documentation

4.45.2.1 **CUresult cuGLCtxCreate (CUcontext *pCtx, unsigned int Flags, CUdevice device)**

Creates a new CUDA context, initializes OpenGL interoperability, and associates the CUDA context with the calling thread. It must be called before performing any other OpenGL interoperability operations. It may fail if the needed OpenGL driver facilities are not available. For usage of the Flags parameter, see cuCtxCreate().

Parameters:

- pCtx - Returned CUDA context
- Flags - Options for CUDA context creation
- device - Device on which to create the context

Returns:

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Note:

Note that this function may also return error codes from previous, asynchronous launches.
4.45 OpenGL Interoperability

See also:

cuCtxCreate, cuGLInit, cuGLMapBufferObject, cuGLRegisterBufferObject, cuGLUnmapBufferObject, cuGLUnregisterBufferObject, cuGLMapBufferObjectAsync, cuGLUnmapBufferObjectAsync, cuGLSetBufferObjectMapFlags, cuWGLGetDevice

4.45.2.2 CUresult cuGraphicsGLRegisterBuffer (CUgraphicsResource *pCudaResource, GLuint buffer, unsigned int Flags)

Registers the buffer object specified by buffer for access by CUDA. A handle to the registered object is returned as pCudaResource. The register flags Flags specify the intended usage, as follows:

- CU_GRAPHICS_REGISTER_FLAGS_NONE: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- CU_GRAPHICS_REGISTER_FLAGS_READ_ONLY: Specifies that CUDA will not write to this resource.
- CU_GRAPHICS_REGISTER_FLAGS_WRITE_DISCARD: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

Parameters:

pCudaResource - Pointer to the returned object handle
buffer - name of buffer object to be registered
Flags - Register flags

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_INVALID_CONTEXT.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:


4.45.2.3 CUresult cuGraphicsGLRegisterImage (CUgraphicsResource *pCudaResource, GLuint image, GLenum target, unsigned int Flags)

Registers the texture or renderbuffer object specified by image for access by CUDA. target must match the type of the object. A handle to the registered object is returned as pCudaResource. The register flags Flags specify the intended usage, as follows:

- CU_GRAPHICS_REGISTER_FLAGS_NONE: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- CU_GRAPHICS_REGISTER_FLAGS_READ_ONLY: Specifies that CUDA will not write to this resource.
• CU_GRAPHICS_REGISTER_FLAGS_WRITE_DISCARD: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

• CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST: Specifies that CUDA will bind this resource to a surface reference.

The following image classes are currently disallowed:

• Textures with borders
• Multisampled renderbuffers

Parameters:

- pCudaResource - Pointer to the returned object handle
- image - name of texture or renderbuffer object to be registered
- target - Identifies the type of object specified by image, and must be one of GL_TEXTURE_2D, GL_TEXTURE_RECTANGLE, GL_TEXTURE_CUBE_MAP, GL_TEXTURE_3D, GL_TEXTURE_2D_ARRAY, or GL_RENDERBUFFER.
- Flags - Register flags

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_mapped, CUDA_ERROR_INVALID_CONTEXT.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:


4.45.2.4 CUresult cuWGLGetDevice (CUdevice *pDevice, HGPUNV hGpu)

Returns in *pDevice the CUDA device associated with a hGpu, if applicable.

Parameters:

- pDevice - Device associated with hGpu
- hGpu - Handle to a GPU, as queried via WGL_NV_gpu_affinity()

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGLCtxCreate, cuGLInit, cuGLMapBufferObject, cuGLRegisterBufferObject, cuGLUnmapBufferObject, cuGLUnregisterBufferObject, cuGLUnmapBufferObjectAsync, cuGLSetBufferObjectMapFlags
4.46 OpenGL Interoperability [DEPRECATED]

Typedefs

- typedef enum CUGLmap_flags_enum CUGLmap_flags

Enumerations

- enum CUGLmap_flags_enum

Functions

- CUresult cuGLInit (void)
  
  Initializes OpenGL interoperability.

- CUresult cuGLMapBufferObject (CUdeviceptr *dptr, size_t *size, GLuint buffer)
  
  Maps an OpenGL buffer object.

- CUresult cuGLMapBufferObjectAsync (CUdeviceptr *dptr, size_t *size, GLuint buffer, CUstream hStream)
  
  Maps an OpenGL buffer object.

- CUresult cuGLRegisterBufferObject (GLuint buffer)
  
  Registers an OpenGL buffer object.

- CUresult cuGLSetBufferObjectMapFlags (GLuint buffer, unsigned int Flags)
  
  Set the map flags for an OpenGL buffer object.

- CUresult cuGLUnmapBufferObject (GLuint buffer)
  
  Unmaps an OpenGL buffer object.

- CUresult cuGLUnmapBufferObjectAsync (GLuint buffer, CUstream hStream)
  
  Unmaps an OpenGL buffer object.

- CUresult cuGLUnregisterBufferObject (GLuint buffer)
  
  Unregister an OpenGL buffer object.

4.46.1 Detailed Description

This section describes deprecated OpenGL interoperability functionality.

4.46.2 Typedef Documentation

4.46.2.1 typedef enum CUGLmap_flags_enum CUGLmap_flags

Flags to map or unmap a resource
4.46.3 Enumeration Type Documentation

4.46.3.1 enum CUGLmap_flags_enum

Flags to map or unmap a resource

4.46.4 Function Documentation

4.46.4.1 CUresult cuGLInit (void)

Deprecated

This function is deprecated as of Cuda 3.0.

Initializes OpenGL interoperability. This function is deprecated and calling it is no longer required. It may fail if the needed OpenGL driver facilities are not available.

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGLCtxCreate, cuGLMapBufferObject, cuGLRegisterBufferObject, cuGLUnmapBufferObject, cuGLUnregisterBufferObject, cuGLMapBufferObjectAsync, cuGLUnmapBufferObjectAsync, cuGLSetBufferObjectMapFlags, cuWGLGetDevice

4.46.4.2 CUresult cuGLMapBufferObject (CUdeviceptr *dptr, size_t *size, GLuint buffer)

Deprecated

This function is deprecated as of Cuda 3.0.

Maps the buffer object specified by buffer into the address space of the current CUDA context and returns in *dptr and *size the base pointer and size of the resulting mapping.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

All streams in the current CUDA context are synchronized with the current GL context.

Parameters:

dptr - Returned mapped base pointer
size - Returned size of mapping
buffer - The name of the buffer object to map

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_MAP_FAILED
4.46 OpenGL Interoperability [DEPRECATED]

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsMapResources

4.46.4.3 CUresult cuGLMapBufferObjectAsync (CUdeviceptr * dptr, size_t * size, GLuint buffer, CUstream hStream)

Deprecated

This function is deprecated as of Cuda 3.0.

Maps the buffer object specified by buffer into the address space of the current CUDA context and returns in *dptr and *size the base pointer and size of the resulting mapping.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

Stream hStream in the current CUDA context is synchronized with the current GL context.

Parameters:

dptr - Returned mapped base pointer
size - Returned size of mapping
buffer - The name of the buffer object to map
hStream - Stream to synchronize

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_MAP_FAILED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsMapResources

4.46.4.4 CUresult cuGLRegisterBufferObject (GLuint buffer)

Deprecated

This function is deprecated as of Cuda 3.0.

Registers the buffer object specified by buffer for access by CUDA. This function must be called before CUDA can map the buffer object. There must be a valid OpenGL context bound to the current thread when this function is called, and the buffer name is resolved by that context.

Parameters:

buffer - The name of the buffer object to register.
Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_ALREADY_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsGLRegisterBuffer

4.46.4.5 CUresult cuGLSetBufferObjectMapFlags (GLuint buffer, unsigned int Flags)

Deprecated

This function is deprecated as of Cuda 3.0.

Sets the map flags for the buffer object specified by buffer.

Changes to Flags will take effect the next time buffer is mapped. The Flags argument may be any of the following:

- CU_GL_MAP_RESOURCE_FLAGS_NONE: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- CU_GL_MAP_RESOURCE_FLAGS_READ_ONLY: Specifies that CUDA kernels which access this resource will not write to this resource.
- CU_GL_MAP_RESOURCE_FLAGS_WRITE_DISCARD: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If buffer has not been registered for use with CUDA, then CUDA_ERROR_INVALID_HANDLE is returned. If buffer is presently mapped for access by CUDA, then CUDA_ERROR_ALREADY_MAPPED is returned.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

Parameters:

buffer - Buffer object to unmap

Flags - Map flags

Returns:

CUDA_SUCCESS, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_INVALID_CONTEXT,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsResourceSetMapFlags
4.46 Interoperability [DEPRECATED]

4.46.4.6 CUresult cuGLUnmapBufferObject (GLuint buffer)

Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the buffer object specified by buffer for access by CUDA.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

All streams in the current CUDA context are synchronized with the current GL context.

Parameters:

buffer - Buffer object to unmap

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsUnmapResources

4.46.4.7 CUresult cuGLUnmapBufferObjectAsync (GLuint buffer, CUSTream hStream)

Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the buffer object specified by buffer for access by CUDA.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

Stream hStream in the current CUDA context is synchronized with the current GL context.

Parameters:

buffer - Name of the buffer object to unmap

hStream - Stream to synchronize

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsUnmapResources
4.46.4.8 Curesult cuGLUnregisterBufferObject (GLuint buffer)

Deprecated

This function is deprecated as of Cuda 3.0.

Unregisters the buffer object specified by buffer. This releases any resources associated with the registered buffer. After this call, the buffer may no longer be mapped for access by CUDA.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

Parameters:

buffer - Name of the buffer object to unregister

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsUnregisterResource
4.47 Direct3D 9 Interoperability

Modules

- Direct3D 9 Interoperability [DEPRECATED]

Typedefs

- typedef enum C Ud3d9DeviceList_enum CUd3d9DeviceList

Enumerations

- enum CUd3d9DeviceList_enum {
  CU_D3D9_DEVICE_LIST_ALL = 0x01,
  CU_D3D9_DEVICE_LIST_CURRENT_FRAME = 0x02,
  CU_D3D9_DEVICE_LIST_NEXT_FRAME = 0x03
}

Functions

- CUresult cuD3D9CtxCreate (CUcontext *pCtx, CUdevice *pCudaDevice, unsigned int Flags, IDirect3DDevice9 *pD3DDevice)
  
  Create a CUDA context for interoperability with Direct3D 9.

- CUresult cuD3D9CtxCreateOnDevice (CUcontext *pCtx, unsigned int flags, IDirect3DDevice9 *pD3DDevice, CUdevice cudaDevice)
  
  Create a CUDA context for interoperability with Direct3D 9.

- CUresult cuD3D9GetDevice (CUdevice *pCudaDevice, const char *pszAdapterName)
  
  Gets the CUDA device corresponding to a display adapter.

- CUresult cuD3D9GetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, IDirect3DDevice9 *pD3D9Device, CUd3d9DeviceList deviceList)
  
  Gets the CUDA devices corresponding to a Direct3D 9 device.

- CUresult cuD3D9GetDirect3DDevice (IDirect3DDevice9 **ppD3DDevice)
  
  Get the Direct3D 9 device against which the current CUDA context was created.

- CUresult cuGraphicsD3D9RegisterResource (CUgraphicsResource *pCudaResource, IDirect3DResource9 *pD3DResource, unsigned int Flags)
  
  Register a Direct3D 9 resource for access by CUDA.

4.47.1 Detailed Description

This section describes the Direct3D 9 interoperability functions of the low-level CUDA driver application programming interface.
4.47.2 Typedef Documentation

4.47.2.1 typedef enum CUd3d9DeviceList_enum CUd3d9DeviceList

CUDA devices corresponding to a D3D9 device

4.47.3 Enumeration Type Documentation

4.47.3.1 enum CUd3d9DeviceList_enum

CUDA devices corresponding to a D3D9 device

Enumerator:

- **CU_D3D9_DEVICE_LIST_ALL**  The CUDA devices for all GPUs used by a D3D9 device
- **CU_D3D9_DEVICE_LIST_CURRENT_FRAME**  The CUDA devices for the GPUs used by a D3D9 device in its currently rendering frame
- **CU_D3D9_DEVICE_LIST_NEXT_FRAME**  The CUDA devices for the GPUs to be used by a D3D9 device in the next frame

4.47.4 Function Documentation

4.47.4.1 CUresult cuD3D9CtxCreate (CUcontext * pCtx, CUdevice * pCudaDevice, unsigned int Flags, IDirect3DDevice9 * pD3DDevice)

Creates a new CUDA context, enables interoperability for that context with the Direct3D device pD3DDevice, and associates the created CUDA context with the calling thread. The created CUcontext will be returned in *pCtx. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context. If pCudaDevice is non-NULL then the CUdevice on which this CUDA context was created will be returned in *pCudaDevice.

On success, this call will increase the internal reference count on pD3DDevice. This reference count will be decremented upon destruction of this context through cuCtxDestroy(). This context will cease to function if pD3DDevice is destroyed or encounters an error.

Parameters:

- **pCtx**  - Returned newly created CUDA context
- **pCudaDevice**  - Returned pointer to the device on which the context was created
- **Flags**  - Context creation flags (see cuCtxCreate() for details)
- **pD3DDevice**  - Direct3D device to create interoperability context with

Returns:

- **CUDA_SUCCESS**
- **CUDA_ERROR_DEINITIALIZED**
- **CUDA_ERROR_NOT_INITIALIZED**
- **CUDA_ERROR_INVALID_VALUE**
- **CUDA_ERROR_OUT_OF_MEMORY**
- **CUDA_ERROR_UNKNOWN**

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuD3D9GetDevice, cuGraphicsD3D9RegisterResource
4.47.4.2 **CUresult cuD3D9CtxCreateOnDevice (CUcontext *pCtx, unsigned int flags, IDirect3DDevice9 *pD3DDevice, CUdevice cudaDevice)**

Creates a new CUDA context, enables interoperability for that context with the Direct3D device pD3DDevice, and associates the created CUDA context with the calling thread. The created CUcontext will be returned in *pCtx. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context.

On success, this call will increase the internal reference count on pD3DDevice. This reference count will be decremented upon destruction of this context through cuCtxDestroy(). This context will cease to function if pD3DDevice is destroyed or encounters an error.

**Parameters:**

- **pCtx** - Returned newly created CUDA context
- **flags** - Context creation flags (see cuCtxCreate() for details)
- **pD3DDevice** - Direct3D device to create interoperability context with
- **cudaDevice** - The CUDA device on which to create the context. This device must be among the devices returned when querying CU_D3D9_DEVICES_ALL from cuD3D9GetDevices.

**Returns:**

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

cuD3D9GetDevices, cuGraphicsD3D9RegisterResource

4.47.4.3 **CUresult cuD3D9GetDevice (CUdevice *pCudaDevice, const char *pszAdapterName)**

Returns in *pCudaDevice the CUDA-compatible device corresponding to the adapter name pszAdapterName obtained from EnumDisplayDevices() or IDirect3D9::GetAdapterIdentifier().

If no device on the adapter with name pszAdapterName is CUDA-compatible, then the call will fail.

**Parameters:**

- **pCudaDevice** - Returned CUDA device corresponding to pszAdapterName
- **pszAdapterName** - Adapter name to query for device

**Returns:**

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_UNKNOWN

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

cuD3D9CtxCreate
4.47.4.4 CUresult cuD3D9GetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, IDirect3DDevice9 *pD3D9Device, CUd3d9DeviceList deviceList)

Returns in *pCudaDeviceCount the number of CUDA-compatible device corresponding to the Direct3D 9 device pD3D9Device. Also returns in *pCudaDevices at most cudaDeviceCount of the the CUDA-compatible devices corresponding to the Direct3D 9 device pD3D9Device.

If any of the GPUs being used to render pDevice are not CUDA capable then the call will return CUDA_ERROR_NO_DEVICE.

Parameters:

- pCudaDeviceCount - Returned number of CUDA devices corresponding to pD3D9Device
- pCudaDevices - Returned CUDA devices corresponding to pD3D9Device
- cudaDeviceCount - The size of the output device array pCudaDevices
- pD3D9Device - Direct3D 9 device to query for CUDA devices
- deviceList - The set of devices to return. This set may be CU_D3D9_DEVICE_LIST_ALL for all devices, CU_D3D9_DEVICE_LIST_CURRENT_FRAME for the devices used to render the current frame (in SLI), or CU_D3D9_DEVICE_LIST_NEXT_FRAME for the devices used to render the next frame (in SLI).

Returns:

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_NO_DEVICE, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D9CtxCreate

4.47.4.5 CUresult cuD3D9GetDirect3DDevice (IDirect3DDevice9 **ppD3DDevice)

Returns in *ppD3DDevice the Direct3D device against which this CUDA context was created in cuD3D9CtxCreate().

Parameters:

- ppD3DDevice - Returned Direct3D device corresponding to CUDA context

Returns:

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D9GetDevice
4.47.4.6 `CUresult cuGraphicsD3D9RegisterResource (CUgraphicsResource * pCudaResource,
               IDirect3DResource9 * pD3DResource, unsigned int Flags)`

Registers the Direct3D 9 resource `pD3DResource` for access by CUDA and returns a CUDA handle to `pD3DResource` in `pCudaResource`. The handle returned in `pCudaResource` may be used to map and unmap this resource until it is unregistered. On success this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through `cuGraphicsUnregisterResource()`.

This call is potentially high-overhead and should not be called every frame in interactive applications.

The type of `pD3DResource` must be one of the following:

- IDirect3DVertexBuffer9: may be accessed through a device pointer
- IDirect3DIndexBuffer9: may be accessed through a device pointer
- IDirect3DSurface9: may be accessed through an array. Only stand-alone objects of type IDirect3DSurface9 may be explicitly shared. In particular, individual mipmap levels and faces of cube maps may not be registered directly. To access individual surfaces associated with a texture, one must register the base texture object.
- IDirect3DBaseTexture9: individual surfaces on this texture may be accessed through an array.

The `Flags` argument may be used to specify additional parameters at register time. The valid values for this parameter are

- `CU_GRAPHICS_REGISTER_FLAGS_NONE`: Specifies no hints about how this resource will be used.
- `CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST`: Specifies that CUDA will bind this resource to a surface reference.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary render target may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized for this context using `cuD3D9CtxCreate` then `CUDA_ERROR_INVALID_CONTEXT` is returned. If `pD3DResource` is of incorrect type or is already registered then `CUDA_ERROR_INVALID_HANDLE` is returned. If `pD3DResource` cannot be registered then `CUDA_ERROR_UNKNOWN` is returned. If `Flags` is not one of the above specified value then `CUDA_ERROR_INVALID_VALUE` is returned.

**Parameters:**

- `pCudaResource` - Returned graphics resource handle
- `pD3DResource` - Direct3D resource to register
- `Flags` - Parameters for resource registration
Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

4.48 Direct3D 9 Interoperability [DEPRECATED]

Typedefs

- typedef enum CUd3d9map_flags_enum CUd3d9map_flags
- typedef enum CUd3d9register_flags_enum CUd3d9register_flags

Enumerations

- enum CUd3d9map_flags_enum
- enum CUd3d9register_flags_enum

Functions

- CUresult cuD3D9MapResources (unsigned int count, IDirect3DResource9 **ppResource)
  Map Direct3D resources for access by CUDA.

- CUresult cuD3D9RegisterResource (IDirect3DResource9 *pResource, unsigned int Flags)
  Register a Direct3D resource for access by CUDA.

- CUresult cuD3D9ResourceGetMappedArray (CUarray *pArray, IDirect3DResource9 *pResource, unsigned int Face, unsigned int Level)
  Get an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

- CUresult cuD3D9ResourceGetMappedPitch (size_t *pPitch, size_t *pPitchSlice, IDirect3DResource9 *pResource, unsigned int Face, unsigned int Level)
  Get the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.

- CUresult cuD3D9ResourceGetMappedPointer (CUdeviceptr *pDevPtr, IDirect3DResource9 *pResource, unsigned int Face, unsigned int Level)
  Get the pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

- CUresult cuD3D9ResourceGetMappedSize (size_t *pSize, IDirect3DResource9 *pResource, unsigned int Face, unsigned int Level)
  Get the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.

- CUresult cuD3D9ResourceSurfaceDimensions (size_t *pWidth, size_t *pHeight, size_t *pDepth, IDirect3DResource9 *pResource, unsigned int Face, unsigned int Level)
  Get the dimensions of a registered surface.

- CUresult cuD3D9ResourceSetMapFlags (IDirect3DResource9 *pResource, unsigned int Flags)
  Set usage flags for mapping a Direct3D resource.

- CUresult cuD3D9UnmapResources (unsigned int count, IDirect3DResource9 **ppResource)
  Unmaps Direct3D resources.

- CUresult cuD3D9UnregisterResource (IDirect3DResource9 *pResource)
  Unregister a Direct3D resource.
4.48.1 Detailed Description

This section describes deprecated Direct3D 9 interoperability functionality.

4.48.2 Typedef Documentation

4.48.2.1 typedef enum CUd3d9map_flags_enum CUd3d9map_flags

Flags to map or unmap a resource

4.48.2.2 typedef enum CUd3d9register_flags_enum CUd3d9register_flags

Flags to register a resource

4.48.3 Enumeration Type Documentation

4.48.3.1 enum CUd3d9map_flags_enum

Flags to map or unmap a resource

4.48.3.2 enum CUd3d9register_flags_enum

Flags to register a resource

4.48.4 Function Documentation

4.48.4.1 CUresult cuD3D9MapResources (unsigned int count, IDirect3DResource9 **ppResource)

Deprecated

This function is deprecated as of Cuda 3.0.

Maps the count Direct3D resources in ppResource for access by CUDA.

The resources in ppResource may be accessed in CUDA kernels until they are unmapped. Direct3D should not access any resources while they are mapped by CUDA. If an application does so the results are undefined.

This function provides the synchronization guarantee that any Direct3D calls issued before cuD3D9MapResources() will complete before any CUDA kernels issued after cuD3D9MapResources() begin.

If any of ppResource have not been registered for use with CUDA or if ppResource contains any duplicate entries, then CUDA_ERROR_INVALID_HANDLE is returned. If any of ppResource are presently mapped for access by CUDA, then CUDA_ERROR_ALREADY_MAPPED is returned.

Parameters:

count - Number of resources in ppResource

ppResource - Resources to map for CUDA usage
Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsMapResources

4.48.4.2 CUresult cuD3D9RegisterResource (IDirect3DResource9 * pResource, unsigned int Flags)

Deprecated

This function is deprecated as of Cuda 3.0.

Registers the Direct3D resource pResource for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through cuD3D9UnregisterResource(). Also on success, this call will increase the internal reference count on pResource. This reference count will be decremented when this resource is unregistered through cuD3D9UnregisterResource().

This call is potentially high-overhead and should not be called every frame in interactive applications.

The type of pResource must be one of the following.

- IDirect3DVertexBuffer9: Cannot be used with Flags set to CU_D3D9_REGISTER_FLAGS_ARRAY.
- IDirect3DIndexBuffer9: Cannot be used with Flags set to CU_D3D9_REGISTER_FLAGS_ARRAY.
- IDirect3DSurface9: Only stand-alone objects of type IDirect3DSurface9 may be explicitly shared. In particular, individual mipmap levels and faces of cube maps may not be registered directly. To access individual surfaces associated with a texture, one must register the base texture object. For restrictions on the Flags parameter, see type IDirect3DBaseTexture9.
- IDirect3DBaseTexture9: When a texture is registered, all surfaces associated with the all mipmap levels of all faces of the texture will be accessible to CUDA.

The Flags argument specifies the mechanism through which CUDA will access the Direct3D resource. The following values are allowed.

- CU_D3D9_REGISTER_FLAGS_NONE: Specifies that CUDA will access this resource through a CUdeviceptr. The pointer, size, and (for textures), pitch for each subresource of this allocation may be queried through cuD3D9ResourceGetMappedPointer(), cuD3D9ResourceGetMappedSize(), and cuD3D9ResourceGetMappedPitch() respectively. This option is valid for all resource types.
- CU_D3D9_REGISTER_FLAGS_ARRAY: Specifies that CUDA will access this resource through a CUarray queried on a sub-resource basis through cuD3D9ResourceGetMappedArray(). This option is only valid for resources of type IDirect3DSurface9 and subtypes of IDirect3DBaseTexture9.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.
The primary rendertarget may not be registered with CUDA.

Resources allocated as shared may not be registered with CUDA.

Any resources allocated in D3DPOOL_SYSTEMMEM or D3DPOOL_MANAGED may not be registered with CUDA.

Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.

Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized on this context, then CUDA_ERROR_INVALID_CONTEXT is returned. If pResource is of incorrect type (e.g. is a non-stand-alone IDirect3DSurface9) or is already registered, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource cannot be registered then CUDA_ERROR_UNKNOWN is returned.

Parameters:

- **pResource** - Resource to register for CUDA access
- **Flags** - Flags for resource registration

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsD3D9RegisterResource

4.48.4.3 CUresult cuD3D9ResourceGetMappedArray (CUarray *pArray, IDirect3DResource9 *pResource, unsigned int Face, unsigned int Level)

Deprecated

This function is deprecated as of Cuda 3.0.

Returns in *pArray an array through which the subresource of the mapped Direct3D resource pResource which corresponds to Face and Level may be accessed. The value set in pArray may change every time that pResource is mapped.

If pResource is not registered then CUDA_ERROR_INVALID_HANDLE is returned. If pResource was not registered with usage flags CU_D3D9_REGISTER_FLAGS_ARRAY then CUDA_ERROR_INVALID_HANDLE is returned. If pResource is not mapped then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of Face and Level parameters, see cuD3D9ResourceGetMappedPointer().

Parameters:

- **pArray** - Returned array corresponding to subresource
- **pResource** - Mapped resource to access
4.48 Direct3D 9 Interoperability [DEPRECATED] 323

Face - Face of resource to access
Level - Level of resource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsSubResourceGetMappedArray

4.48.4.4 CUresult cuD3D9ResourceGetMappedPitch (size_t ∗pPitch, size_t ∗pPitchSlice, IDirect3DResource9 ∗pResource, unsigned int Face, unsigned int Level)

Deprecated

This function is deprecated as of Cuda 3.0.

Returns in ∗pPitch and ∗pPitchSlice the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource pResource, which corresponds to Face and Level. The values set in pPitch and pPitchSlice may change every time that pResource is mapped.

The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position $x$, $y$ from the base pointer of the surface is:

$$y \times \text{pitch} + (\text{bytes per pixel}) \times x$$

For a 3D surface, the byte offset of the sample at position $x$, $y$, $z$ from the base pointer of the surface is:

$$z \times \text{slicePitch} + y \times \text{pitch} + (\text{bytes per pixel}) \times x$$

Both parameters pPitch and pPitchSlice are optional and may be set to NULL.

If pResource is not of type IDirect3DBaseTexture9 or one of its sub-types or if pResource has not been registered for use with CUDA, then cudaErrorInvalidResourceHandle is returned. If pResource was not registered with usage flags CU_D3D9_REGISTER_FLAGS_NONE, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource is not mapped for access by CUDA then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of Face and Level parameters, see cuD3D9ResourceGetMappedPointer().

Parameters:

pPitch - Returned pitch of subresource
pPitchSlice - Returned Z-slice pitch of subresource
pResource - Mapped resource to access
Face - Face of resource to access
Level - Level of resource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Generated for NVIDIA CUDA Library by Doxygen
Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsSubResourceGetMappedArray

4.48.4.5 CUresult cuD3D9ResourceGetMappedPointer (CUdeviceptr *pDevPtr, IDirect3DResource9 *pResource, unsigned int Face, unsigned int Level)

Deprecated

This function is deprecated as of CUDA 3.0.

Returns in *pDevPtr the base pointer of the subresource of the mapped Direct3D resource pResource, which corresponds to Face and Level. The value set in pDevPtr may change every time that pResource is mapped.

If pResource is not registered, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource was not registered with usage flags CU_D3D9_REGISTER_FLAGS_NONE, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource is not mapped, then CUDA_ERROR_NOT_MAPPED is returned.

If pResource is of type IDirect3DCubeTexture9, then Face must one of the values enumerated by type D3DCUBEMAP_FACES. For all other types Face must be 0. If Face is invalid, then CUDA_ERROR_INVALID_VALUE is returned.

If pResource is of type IDirect3DBaseTexture9, then Level must correspond to a valid mipmap level. At present only mipmap level 0 is supported. For all other types Level must be 0. If Level is invalid, then CUDA_ERROR_INVALID_VALUE is returned.

Parameters:

- pDevPtr - Returned pointer corresponding to subresource
- pResource - Mapped resource to access
- Face - Face of resource to access
- Level - Level of resource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsResourceGetMappedPointer

4.48.4.6 CUresult cuD3D9ResourceGetMappedSize (size_t *pSize, IDirect3DResource9 *pResource, unsigned int Face, unsigned int Level)

Deprecated

This function is deprecated as of CUDA 3.0.
4.48 Direct3D 9 Interoperability [DEPRECATED] 325

Returns in *pSize the size of the subresource of the mapped Direct3D resource pResource, which corresponds to Face and Level. The value set in pSize may change every time that pResource is mapped.

If pResource has not been registered for use with CUDA, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource was not registered with usage flags CU_D3D9_REGISTER_FLAGS_NONE, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource is not mapped for access by CUDA, then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of Face and Level parameters, see cuD3D9ResourceGetMappedPointer.

Parameters:

- **pSize** - Returned size of subresource
- **pResource** - Mapped resource to access
- **Face** - Face of resource to access
- **Level** - Level of resource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsResourceGetMappedPointer

4.48.4.7 CUresult cuD3D9ResourceGetSurfaceDimensions (size_t * pWidth, size_t * pHeight, size_t * pDepth, IDirect3DResource9 * pResource, unsigned int Face, unsigned int Level)

Deprecated

This function is deprecated as of Cuda 3.0.

Returns in *pWidth, *pHeight, and *pDepth the dimensions of the subresource of the mapped Direct3D resource pResource, which corresponds to Face and Level.

Because anti-aliased surfaces may have multiple samples per pixel, it is possible that the dimensions of a resource will be an integer factor larger than the dimensions reported by the Direct3D runtime.

The parameters pWidth, pHeight, and pDepth are optional. For 2D surfaces, the value returned in *pDepth will be 0.

If pResource is not of type IDirect3DBaseTexture9 or IDirect3DSurface9 or if pResource has not been registered for use with CUDA, then CUDA_ERROR_INVALID_HANDLE is returned.

For usage requirements of Face and Level parameters, see cuD3D9ResourceGetMappedPointer().

Parameters:

- **pWidth** - Returned width of surface
- **pHeight** - Returned height of surface
- **pDepth** - Returned depth of surface
**pResource** - Registered resource to access  
**Face** - Face of resource to access  
**Level** - Level of resource to access

Returns:

- CUDA_SUCCESS  
- CUDA_ERROR_DEINITIALIZED  
- CUDA_ERROR_NOT_INITIALIZED  
- CUDA_ERROR_INVALID_CONTEXT  
- CUDA_ERROR_INVALID_VALUE  
- CUDA_ERROR_INVALID_HANDLE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuGraphicsSubResourceGetMappedArray

### 4.48.4.8 CUresult cuD3D9ResourceSetMapFlags (IDirect3DResource9 * pResource, unsigned int Flags)

**Deprecated**

This function is deprecated as of Cuda 3.0.

Set **Flags** for mapping the Direct3D resource **pResource**. Changes to **Flags** will take effect the next time **pResource** is mapped. The **Flags** argument may be any of the following:

- **CU_D3D9_MAPRESOURCE_FLAGS_NONE**: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- **CU_D3D9_MAPRESOURCE_FLAGS_READONLY**: Specifies that CUDA kernels which access this resource will not write to this resource.
- **CU_D3D9_MAPRESOURCE_FLAGS_WRITEDISCARD**: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If **pResource** has not been registered for use with CUDA, then **CUDA_ERROR_INVALID_HANDLE** is returned. If **pResource** is presently mapped for access by CUDA, then **CUDA_ERROR_ALREADY_MAPPED** is returned.

Parameters:

- **pResource** - Registered resource to set flags for  
- **Flags** - Parameters for resource mapping

Returns:

- CUDA_SUCCESS  
- CUDA_ERROR_DEINITIALIZED  
- CUDA_ERROR_NOT_INITIALIZED  
- CUDA_ERROR_INVALID_CONTEXT  
- CUDA_ERROR_INVALID_VALUE  
- CUDA_ERROR_INVALID_HANDLE  
- CUDA_ERROR_ALREADY_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuGraphicsResourceSetMapFlags
4.48.4.9  CResult cuD3D9UnmapResources (unsigned int count, IDirect3DResource9 ** ppResource)

Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the count Direct3D resources in ppResource.

This function provides the synchronization guarantee that any CUDA kernels issued before cuD3D9UnmapResources() will complete before any Direct3D calls issued after cuD3D9UnmapResources() begin.

If any of ppResource have not been registered for use with CUDA or if ppResource contains any duplicate entries, then CUDA_ERROR_INVALID_HANDLE is returned. If any of ppResource are not presently mapped for access by CUDA, then CUDA_ERROR_NOT_MAPPED is returned.

Parameters:

- **count** - Number of resources to unmap for CUDA
- **ppResource** - Resources to unmap for CUDA

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsUnmapResources

4.48.4.10  CResult cuD3D9UnregisterResource (IDirect3DResource9 * pResource)

Deprecated

This function is deprecated as of Cuda 3.0.

Unregisters the Direct3D resource pResource so it is not accessible by CUDA unless registered again.

If pResource is not registered, then CUDA_ERROR_INVALID_HANDLE is returned.

Parameters:

- **pResource** - Resource to unregister

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsUnregisterResource
4.49 Direct3D 10 Interoperability

Modules

- Direct3D 10 Interoperability [DEPRECATED]

Typedefs

- typedef enum CUd3d10DeviceList_enum C Ud3d10DeviceList

Enumerations

- enum CUd3d10DeviceList_enum {
  CU_D3D10_DEVICE_LIST_ALL = 0x01,
  CU_D3D10_DEVICE_LIST_CURRENT_FRAME = 0x02,
  CU_D3D10_DEVICE_LIST_NEXT_FRAME = 0x03 }

Functions

- CUresult cuD3D10CtxCreate (CUcontext *pCtx, CUdevice *pCudaDevice, unsigned int Flags, ID3D10Device *pD3DDevice)
  
  Create a CUDA context for interoperability with Direct3D 10.

- CUresult cuD3D10CtxCreateOnDevice (CUcontext *pCtx, unsigned int flags, ID3D10Device *pD3DDevice, CUdevice cudaDevice)
  
  Create a CUDA context for interoperability with Direct3D 10.

- CUresult cuD3D10GetDevice (CUdevice *pCudaDevice, IDXGIAdapter *pAdapter)
  
  Gets the CUDA device corresponding to a display adapter.

- CUresult cuD3D10GetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, ID3D10Device *pD3D10Device, CUd3d10DeviceList deviceList)
  
  Gets the CUDA devices corresponding to a Direct3D 10 device.

- CUresult cuD3D10GetDirect3DDevice (ID3D10Device **ppD3DDevice)
  
  Get the Direct3D 10 device against which the current CUDA context was created.

  
  Register a Direct3D 10 resource for access by CUDA.

4.49.1 Detailed Description

This section describes the Direct3D 10 interoperability functions of the low-level CUDA driver application programming interface.
4.49 Direct3D 10 Interoperability

4.49.2 Typedef Documentation

4.49.2.1 typedef enum CUd3d10DeviceList_enum CUd3d10DeviceList

CUDA devices corresponding to a D3D10 device

4.49.3 Enumeration Type Documentation

4.49.3.1 enum CUd3d10DeviceList_enum

CUDA devices corresponding to a D3D10 device

 Enumerator:

`CU_D3D10_DEVICE_LIST_ALL` The CUDA devices for all GPUs used by a D3D10 device

`CU_D3D10_DEVICE_LIST_CURRENT_FRAME` The CUDA devices for the GPUs used by a D3D10 device in its currently rendering frame

`CU_D3D10_DEVICE_LIST_NEXT_FRAME` The CUDA devices for the GPUs to be used by a D3D10 device in the next frame

4.49.4 Function Documentation

4.49.4.1 CUresult cuD3D10CtxCreate (CUcontext *pCtx, CUdevice *pCudaDevice, unsigned int Flags, ID3D10Device *pD3DDevice)

Creates a new CUDA context, enables interoperability for that context with the Direct3D device `pD3DDevice`, and associates the created CUDA context with the calling thread. The created `CUcontext` will be returned in `*pCtx`. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context. If `pCudaDevice` is non-NULL then the `CUdevice` on which this CUDA context was created will be returned in `*pCudaDevice`.

On success, this call will increase the internal reference count on `pD3DDevice`. This reference count will be decremented upon destruction of this context through `cuCtxDestroy()`. This context will cease to function if `pD3DDevice` is destroyed or encounters an error.

Parameters:

- `pCtx` - Returned newly created CUDA context
- `pCudaDevice` - Returned pointer to the device on which the context was created
- `Flags` - Context creation flags (see `cuCtxCreate()` for details)
- `pD3DDevice` - Direct3D device to create interoperability context with

Returns:

- `CUDA_SUCCESS`
- `CUDA_ERROR_DEINITIALIZED`
- `CUDA_ERROR_NOT_INITIALIZED`
- `CUDA_ERROR_INVALID_VALUE`
- `CUDA_ERROR_OUT_OF_MEMORY`
- `CUDA_ERROR_UNKNOWN`

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- `cuD3D10GetDevice`, `cuGraphicsD3D10RegisterResource`
4.49.4.2 CUresult cuD3D10CtxCreateOnDevice (CUcontext *pCtx, unsigned int flags, ID3D10Device *pD3DDevice, CUdevice cudaDevice)

Creates a new CUDA context, enables interoperability for that context with the Direct3D device pD3DDevice, and associates the created CUDA context with the calling thread. The created CUcontext will be returned in *pCtx. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context.

On success, this call will increase the internal reference count on pD3DDevice. This reference count will be decremented upon destruction of this context through cuCtxDestroy(). This context will cease to function if pD3DDevice is destroyed or encounters an error.

Parameters:

- **pCtx** - Returned newly created CUDA context
- **flags** - Context creation flags (see cuCtxCreate() for details)
- **pD3DDevice** - Direct3D device to create interoperability context with
- **cudaDevice** - The CUDA device on which to create the context. This device must be among the devices returned when querying CU_D3D10_DEVICES_ALL from cuD3D10GetDevices.

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D10GetDevices, cuGraphicsD3D10RegisterResource

4.49.4.3 CUresult cuD3D10GetDevice (CUdevice *pCudaDevice, IDXGIAdapter *pAdapter)

Returns in *pCudaDevice the CUDA-compatible device corresponding to the adapter pAdapter obtained from IDXGIFactory::EnumAdapters.

If no device on pAdapter is CUDA-compatible then the call will fail.

Parameters:

- **pCudaDevice** - Returned CUDA device corresponding to pAdapter
- **pAdapter** - Adapter to query for CUDA device

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D10CtxCreate
4.49 Direct3D 10 Interoperability

4.49.4.4 CUresult cuD3D10GetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, ID3D10Device *pD3D10Device, CUd3d10DeviceList deviceList)

Returns in *pCudaDeviceCount the number of CUDA-compatible device corresponding to the Direct3D 10 device pD3D10Device. Also returns in *pCudaDevices at most cudaDeviceCount of the the CUDA-compatible devices corresponding to the Direct3D 10 device pD3D10Device.

If any of the GPUs being used to render pDevice are not CUDA capable then the call will return CUDA_ERROR_NO_DEVICE.

Parameters:

pCudaDeviceCount - Returned number of CUDA devices corresponding to pD3D10Device
pCudaDevices - Returned CUDA devices corresponding to pD3D10Device
cudaDeviceCount - The size of the output device array pCudaDevices
pD3D10Device - Direct3D 10 device to query for CUDA devices
deviceList - The set of devices to return. This set may be CU_D3D10DEVICE_LIST_ALL for all devices, CU_D3D10DEVICE_LIST_CURRENT_FRAME for the devices used to render the current frame (in SLI), or CU_D3D10DEVICE_LIST_NEXT_FRAME for the devices used to render the next frame (in SLI).

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_NO_DEVICE, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D10CtxCreate

4.49.4.5 CUresult cuD3D10GetDirect3DDevice (ID3D10Device **ppD3DDevice)

Returns in *ppD3DDevice the Direct3D device against which this CUDA context was created in cuD3D10CtxCreate().

Parameters:

ppD3DDevice - Returned Direct3D device corresponding to CUDA context

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D10GetDevice
4.49.4.6 **CUresult cuGraphicsD3D10RegisterResource (CUgraphicsResource * pCudaResource, ID3D10Resource * pD3DResource, unsigned int Flags)**

Registers the Direct3D 10 resource `pD3DResource` for access by CUDA and returns a CUDA handle to `pD3DResource` in `pCudaResource`. The handle returned in `pCudaResource` may be used to map and unmap this resource until it is unregistered. On success this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through `cuGraphicsUnregisterResource()`.

This call is potentially high-overhead and should not be called every frame in interactive applications.

The type of `pD3DResource` must be one of the following.

- ID3D10Buffer: may be accessed through a device pointer.
- ID3D10Texture1D: individual subresources of the texture may be accessed via arrays
- ID3D10Texture2D: individual subresources of the texture may be accessed via arrays
- ID3D10Texture3D: individual subresources of the texture may be accessed via arrays

The `Flags` argument may be used to specify additional parameters at register time. The valid values for this parameter are

- `CU_GRAPHICS_REGISTER_FLAGS_NONE`: Specifies no hints about how this resource will be used.
- `CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST`: Specifies that CUDA will bind this resource to a surface reference.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized for this context using `cuD3D10CtxCreate` then `CUDA_ERROR_InvalidContext` is returned. If `pD3DResource` is of incorrect type or is already registered then `CUDA_ERROR_INVALID_HANDLE` is returned. If `pD3DResource` cannot be registered then `CUDA_ERROR_UNKNOWN` is returned. If `Flags` is not one of the above specified value then `CUDA_ERROR_INVALID_VALUE` is returned.

**Parameters:**

- `pCudaResource` - Returned graphics resource handle
- `pD3DResource` - Direct3D resource to register
- `Flags` - Parameters for resource registration

**Returns:**

- `CUDA_SUCCESS`, `CUDA_ERROR_DEINITIALIZED`, `CUDA_ERROR_NOT_INITIALIZED`, `CUDA_ERROR_INVALID_CONTEXT`, `CUDA_ERROR_INVALID_VALUE`, `CUDA_ERROR_INVALID_HANDLE`, `CUDA_ERROR_OUT_OF_MEMORY`, `CUDA_ERROR_UNKNOWN`
Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

4.50  Direct3D 10 Interoperability [DEPRECATED]

Typedefs

• typedef enum CUD3D10map_flags_enum CUD3D10map_flags
• typedef enum CUD3D10register_flags_enum CUD3D10register_flags

Enumerations

• enum CUD3D10map_flags_enum
• enum CUD3D10register_flags_enum

Functions

• CUresult cuD3D10MapResources (unsigned int count, ID3D10Resource **ppResources)
  Map Direct3D resources for access by CUDA.

• CUresult cuD3D10RegisterResource (ID3D10Resource *pResource, unsigned int Flags)
  Register a Direct3D resource for access by CUDA.

• CUresult cuD3D10ResourceGetMappedArray (CUarray *pArray, ID3D10Resource *pResource, unsigned int SubResource)
  Get an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

• CUresult cuD3D10ResourceGetMappedPitch (size_t *pPitch, size_t *pPitchSlice, ID3D10Resource *pResource, unsigned int SubResource)
  Get the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.

• CUresult cuD3D10ResourceGetMappedPointer (CUdeviceptr *pDevPtr, ID3D10Resource *pResource, unsigned int SubResource)
  Get a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

• CUresult cuD3D10ResourceGetMappedSize (size_t *pSize, ID3D10Resource *pResource, unsigned int SubResource)
  Get the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.

• CUresult cuD3D10ResourceGetSurfaceDimensions (size_t *pWidth, size_t *pHeight, size_t *pDepth, ID3D10Resource *pResource, unsigned int SubResource)
  Get the dimensions of a registered surface.

• CUresult cuD3D10ResourceSetMapFlags (ID3D10Resource *pResource, unsigned int Flags)
  Set usage flags for mapping a Direct3D resource.

• CUresult cuD3D10UnmapResources (unsigned int count, ID3D10Resource **ppResources)
  Unmap Direct3D resources.

• CUresult cuD3D10UnregisterResource (ID3D10Resource *pResource)
  Unregister a Direct3D resource.
4.50 Direct3D 10 Interoperability [DEPRECATED]

4.50.1 Detailed Description

This section describes deprecated Direct3D 10 interoperability functionality.

4.50.2 Typedef Documentation

4.50.2.1 typedef enum CUD3D10map_flags_enum CUD3D10map_flags

Flags to map or unmap a resource

4.50.2.2 typedef enum CUD3D10register_flags_enum CUD3D10register_flags

Flags to register a resource

4.50.3 Enumeration Type Documentation

4.50.3.1 enum CUD3D10map_flags_enum

Flags to map or unmap a resource

4.50.3.2 enum CUD3D10register_flags_enum

Flags to register a resource

4.50.4 Function Documentation

4.50.4.1 CUresult cuD3D10MapResources (unsigned int count, ID3D10Resource ** ppResources)

Deprecated

This function is deprecated as of Cuda 3.0.

Maps the `count` Direct3D resources in `ppResources` for access by CUDA.

The resources in `ppResources` may be accessed in CUDA kernels until they are unmapped. Direct3D should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any Direct3D calls issued before `cuD3D10MapResources()` will complete before any CUDA kernels issued after `cuD3D10MapResources()` begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries, then `CUDA_ERROR_INVALID_HANDLE` is returned. If any of `ppResources` are presently mapped for access by CUDA, then `CUDA_ERROR_ALREADY_MAPPED` is returned.

Parameters:

- `count` - Number of resources to map for CUDA
- `ppResources` - Resources to map for CUDA

Generated for NVIDIA CUDA Library by Doxygen
Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsMapResources

4.50.4.2 CUresult cuD3D10RegisterResource (ID3D10Resource *pResource, unsigned int Flags)

Deprecated

This function is deprecated as of Cuda 3.0.

Registers the Direct3D resource pResource for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through cuD3D10UnregisterResource(). Also on success, this call will increase the internal reference count on pResource. This reference count will be decremented when this resource is unregistered through cuD3D10UnregisterResource().

This call is potentially high-overhead and should not be called every frame in interactive applications.

The type of pResource must be one of the following.

- ID3D10Buffer: Cannot be used with Flags set to CU_D3D10_REGISTER_FLAGS_ARRAY.
- ID3D10Texture1D: No restrictions.
- ID3D10Texture2D: No restrictions.
- ID3D10Texture3D: No restrictions.

The Flags argument specifies the mechanism through which CUDA will access the Direct3D resource. The following values are allowed.

- CU_D3D10_REGISTER_FLAGS_NONE: Specifies that CUDA will access this resource through a CUdeviceptr. The pointer, size, and (for textures), pitch for each subresource of this allocation may be queried through cuD3D10ResourceGetMappedPointer(), cuD3D10ResourceGetMappedSize(), and cuD3D10ResourceGetMappedPitch() respectively. This option is valid for all resource types.
- CU_D3D10_REGISTER_FLAGS_ARRAY: Specifies that CUDA will access this resource through a CUarray queried on a sub-resource basis through cuD3D10ResourceGetMappedArray(). This option is only valid for resources of type ID3D10Texture1D, ID3D10Texture2D, and ID3D10Texture3D.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.

Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized on this context then **CUDA_ERROR_INVALID_CONTEXT** is returned. If **pResource** is of incorrect type or is already registered, then **CUDA_ERROR_INVALID_HANDLE** is returned. If **pResource** cannot be registered, then **CUDA_ERROR_UNKNOWN** is returned.

**Parameters:**

- **pResource** - Resource to register
- **Flags** - Parameters for resource registration

**Returns:**

**CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN**

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

cuGraphicsD3D10RegisterResource

4.50.4.3 **CUresult cuD3D10ResourceGetMappedArray (CUarray * pArray, ID3D10Resource * pResource, unsigned int SubResource)**

**Deprecated**

This function is deprecated as of Cuda 3.0.

Returns in **pArray** an array through which the subresource of the mapped Direct3D resource **pResource**, which corresponds to **SubResource** may be accessed. The value set in **pArray** may change every time that **pResource** is mapped.

If **pResource** is not registered, then **CUDA_ERROR_INVALID_HANDLE** is returned. If **pResource** was not registered with usage flags **CU_D3D10_REGISTER_FLAGS_ARRAY**, then **CUDA_ERROR_INVALID_HANDLE** is returned. If **pResource** is not mapped, then **CUDA_ERROR_NOT_MAPPED** is returned.

For usage requirements of the **SubResource** parameter, see cuD3D10ResourceGetMappedPointer().

**Parameters:**

- **pArray** - Returned array corresponding to subresource
- **pResource** - Mapped resource to access
- **SubResource** - Subresource of **pResource** to access

**Returns:**

**CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED**
Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsSubResourceGetMappedArray

4.50.4.4 CUresult cuD3D10ResourceGetMappedPitch (size_t∗pPitch, size_t∗pPitchSlice, ID3D10Resource∗pResource, unsigned intSubResource)

Deprecated

This function is deprecated as of Cuda 3.0.

Returns in ∗pPitch and ∗pPitchSlice the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource pResource, which corresponds to SubResource. The values set in pPitch and pPitchSlice may change every time that pResource is mapped.

The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position x, y from the base pointer of the surface is:

y ∗ pitch + (bytes per pixel) ∗ x

For a 3D surface, the byte offset of the sample at position x, y, z from the base pointer of the surface is:

z ∗ slicePitch + y ∗ pitch + (bytes per pixel) ∗ x

Both parameters pPitch and pPitchSlice are optional and may be set to NULL.

If pResource is not of type IDirect3DBaseTexture10 or one of its sub-types or if pResource has not been registered for use with CUDA, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource was not registered with usage flags CU_D3D10_REGISTER_FLAGS_NONE, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource is not mapped for access by CUDA, then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of the SubResource parameter, see cuD3D10ResourceGetMappedPointer().

Parameters:

pPitch - Returned pitch of subresource
pPitchSlice - Returned Z-slice pitch of subresource
pResource - Mapped resource to access
SubResource - Subresource of pResource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsSubResourceGetMappedArray
4.50 Direct3D 10 Interoperability [DEPRECATED] 339

4.50.4.5 CUresult cuD3D10ResourceGetMappedPointer (CUdeviceptr *pDevPtr, ID3D10Resource *pResource, unsigned int SubResource)

Deprecated

This function is deprecated as of Cuda 3.0.

Returns in *pDevPtr the base pointer of the subresource of the mapped Direct3D resource pResource, which corresponds to SubResource. The value set in pDevPtr may change every time that pResource is mapped.

If pResource is not registered, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource was not registered with usage flags CU_D3D10_REGISTER_FLAGS_NONE, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource is not mapped, then CUDA_ERROR_NOT_MAPPED is returned.

If pResource is of type ID3D10Buffer, then SubResource must be 0. If pResource is of any other type, then the value of SubResource must come from the subresource calculation in D3D10CalcSubResource().

Parameters:

pDevPtr - Returned pointer corresponding to subresource
pResource - Mapped resource to access
SubResource - Subresource of pResource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsResourceGetMappedPointer

4.50.4.6 CUresult cuD3D10ResourceGetMappedSize (size_t *pSize, ID3D10Resource *pResource, unsigned int SubResource)

Deprecated

This function is deprecated as of Cuda 3.0.

Returns in *pSize the size of the subresource of the mapped Direct3D resource pResource, which corresponds to SubResource. The value set in pSize may change every time that pResource is mapped.

If pResource has not been registered for use with CUDA, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource was not registered with usage flags CU_D3D10_REGISTER_FLAGS_NONE, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource is not mapped for access by CUDA, then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of the SubResource parameter, see cuD3D10ResourceGetMappedPointer().

Parameters:

pSize - Returned size of subresource


\texttt{pResource} - Mapped resource to access

\texttt{SubResource} - Subresource of \texttt{pResource} to access

Returns:

\texttt{CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED}

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

\texttt{cuGraphicsResourceGetMappedPointer}

4.50.4.7 \texttt{CUresult cuD3D10ResourceGetSurfaceDimensions(size_t \ast pWidth, size_t \ast pHeight, size_t \ast pDepth, ID3D10Resource \ast pResource, unsigned int SubResource)}

Deprecated

This function is deprecated as of Cuda 3.0.

Returns in \ast pWidth, \ast pHeight, and \ast pDepth the dimensions of the subresource of the mapped Direct3D resource \texttt{pResource}, which corresponds to \texttt{SubResource}.

Because anti-aliased surfaces may have multiple samples per pixel, it is possible that the dimensions of a resource will be an integer factor larger than the dimensions reported by the Direct3D runtime.

The parameters \texttt{pWidth, pHeight, and pDepth} are optional. For 2D surfaces, the value returned in \ast pDepth will be 0.

If \texttt{pResource} is not of type IDirect3DBaseTexture10 or IDirect3DSurface10 or if \texttt{pResource} has not been registered for use with CUDA, then \texttt{CUDA_ERROR_INVALID_HANDLE} is returned.

For usage requirements of the \texttt{SubResource} parameter, see \texttt{cuD3D10ResourceGetMappedPointer()}. Parameters:

\texttt{pWidth} - Returned width of surface
\texttt{pHeight} - Returned height of surface
\texttt{pDepth} - Returned depth of surface
\texttt{pResource} - Registered resource to access
\texttt{SubResource} - Subresource of \texttt{pResource} to access

Returns:

\texttt{CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE}

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

\texttt{cuGraphicsSubResourceGetMappedArray}
4.50 Direct3D 10 Interoperability [DEPRECATED] 341

4.50.4.8 CUresult cuD3D10ResourceSetMapFlags (ID3D10Resource * pResource, unsigned int Flags)

Deprecated

This function is deprecated as of Cuda 3.0.

Set flags for mapping the Direct3D resource pResource.

Changes to flags will take effect the next time pResource is mapped. The Flags argument may be any of the following.

- CU_D3D10_MAPRESOURCE_FLAGS_NONE: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- CU_D3D10_MAPRESOURCE_FLAGS_READONLY: Specifies that CUDA kernels which access this resource will not write to this resource.
- CU_D3D10_MAPRESOURCE_FLAGS_WRITEDISCARD: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If pResource has not been registered for use with CUDA, then CUDA_ERROR_INVALID_HANDLE is returned.
If pResource is presently mapped for access by CUDA then CUDA_ERROR_ALREADY_MAPPED is returned.

Parameters:

- pResource - Registered resource to set flags for
- Flags - Parameters for resource mapping

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsResourceSetMapFlags

4.50.4.9 CUresult cuD3D10UnmapResources (unsigned int count, ID3D10Resource ** ppResources)

Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the count Direct3D resources in ppResources.

This function provides the synchronization guarantee that any CUDA kernels issued before cuD3D10UnmapResources() will complete before any Direct3D calls issued after cuD3D10UnmapResources() begin.

If any of ppResources have not been registered for use with CUDA or if ppResources contains any duplicate entries, then CUDA_ERROR_INVALID_HANDLE is returned. If any of ppResources are not presently mapped for access by CUDA, then CUDA_ERROR_NOT_MAPPED is returned.
Parameters:

- **count** - Number of resources to unmap for CUDA
- **ppResources** - Resources to unmap for CUDA

Returns:

- CUDA_SUCCESS
- CUDA_ERROR_DEINITIALIZED
- CUDA_ERROR_NOT_INITIALIZED
- CUDA_ERROR_INVALID_CONTEXT
- CUDA_ERROR_INVALID_VALUE
- CUDA_ERROR_INVALID_HANDLE
- CUDA_ERROR_NOT_MAPPED
- CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuGraphicsUnmapResources

---

### 4.50.4.10 CUresult cuD3D10UnregisterResource (ID3D10Resource * pResource)

**Deprecated**

This function is deprecated as of Cuda 3.0.

Unregisters the Direct3D resource `pResource` so it is not accessible by CUDA unless registered again.

If `pResource` is not registered, then CUDA_ERROR_INVALID_HANDLE is returned.

Parameters:

- **pResource** - Resources to unregister

Returns:

- CUDA_SUCCESS
- CUDA_ERROR_DEINITIALIZED
- CUDA_ERROR_NOT_INITIALIZED
- CUDA_ERROR_INVALID_CONTEXT
- CUDA_ERROR_INVALID_HANDLE
- CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- cuGraphicsUnregisterResource
4.51  Direct3D 11 Interoperability

Typedefs

- typedef enum CUd3d11DeviceList_enum CUd3d11DeviceList

Enumerations

- enum CUd3d11DeviceList_enum {
  CU_D3D11_DEVICE_LIST_ALL = 0x01,
  CU_D3D11_DEVICE_LIST_CURRENT_FRAME = 0x02,
  CU_D3D11_DEVICE_LIST_NEXT_FRAME = 0x03
}

Functions

- CUresult cuD3D11CtxCreate (CUcontext *pCtx, CUdevice *pCudaDevice, unsigned int Flags, ID3D11Device *pD3DDevice)
  
  Create a CUDA context for interoperability with Direct3D 11.

- CUresult cuD3D11CtxCreateOnDevice (CUcontext *pCtx, unsigned int flags, ID3D11Device *pD3DDevice, CUdevice cudaDevice)
  
  Create a CUDA context for interoperability with Direct3D 11.

- CUresult cuD3D11GetDevice (CUdevice *pCudaDevice, IDXGIAdapter *pAdapter)
  
  Gets the CUDA device corresponding to a display adapter.

- CUresult cuD3D11GetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, ID3D11Device *pD3D11Device, CUd3d11DeviceList deviceList)
  
  Gets the CUDA devices corresponding to a Direct3D 11 device.

- CUresult cuD3D11GetDirect3DDevice (ID3D11Device **ppD3DDevice)
  
  Get the Direct3D 11 device against which the current CUDA context was created.

  
  Register a Direct3D 11 resource for access by CUDA.

4.51.1  Detailed Description

This section describes the Direct3D 11 interoperability functions of the low-level CUDA driver application programming interface.

4.51.2  Typedef Documentation

4.51.2.1  typedef enum CUd3d11DeviceList_enum CUd3d11DeviceList

CUDA devices corresponding to a D3D11 device
4.51.3 Enumeration Type Documentation

4.51.3.1 enum CUd3d11DeviceList_enum

CUDA devices corresponding to a D3D11 device

**Enumerator:**

- **CU_D3D11_DEVICE_LIST_ALL**  
  The CUDA devices for all GPUs used by a D3D11 device

- **CU_D3D11_DEVICE_LIST_CURRENT_FRAME**  
  The CUDA devices for the GPUs used by a D3D11 device in its currently rendering frame

- **CU_D3D11_DEVICE_LIST_NEXT_FRAME**  
  The CUDA devices for the GPUs to be used by a D3D11 device in the next frame

4.51.4 Function Documentation

4.51.4.1 CUresult cuD3D11CtxCreate (CUcontext *pCtx, CUdevice *pCudaDevice, unsigned int Flags, ID3D11Device *pD3DDevice)

Creates a new CUDA context, enables interoperability for that context with the Direct3D device pD3DDevice, and associates the created CUDA context with the calling thread. The created CUcontext will be returned in *pCtx. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context. If pCudaDevice is non-NULL then the CUdevice on which this CUDA context was created will be returned in *pCudaDevice.

On success, this call will increase the internal reference count on pD3DDevice. This reference count will be decremented upon destruction of this context through cuCtxDestroy(). This context will cease to function if pD3DDevice is destroyed or encounters an error.

**Parameters:**

- **pCtx** - Returned newly created CUDA context
- **pCudaDevice** - Returned pointer to the device on which the context was created
- **Flags** - Context creation flags (see cuCtxCreate() for details)
- **pD3DDevice** - Direct3D device to create interoperability context with

**Returns:**

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

**Note:**

Note that this function may also return error codes from previous, asynchronous launches.

**See also:**

cuD3D11GetDevice, cuGraphicsD3D11RegisterResource

4.51.4.2 CUresult cuD3D11CtxCreateOnDevice (CUcontext *pCtx, unsigned int flags, ID3D11Device *pD3DDevice, CUdevice cudaDevice)

Creates a new CUDA context, enables interoperability for that context with the Direct3D device pD3DDevice, and associates the created CUDA context with the calling thread. The created CUcontext will be returned in *pCtx. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context.
On success, this call will increase the internal reference count on `pD3DDevice`. This reference count will be decremented upon destruction of this context through `cuCtxDestroy()`. This context will cease to function if `pD3DDevice` is destroyed or encounters an error.

Parameters:

- `pCtx` - Returned newly created CUDA context
- `flags` - Context creation flags (see `cuCtxCreate()` for details)
- `pD3DDevice` - Direct3D device to create interoperability context with
- `cudaDevice` - The CUDA device on which to create the context. This device must be among the devices returned when querying CU_D3D11_DEVICES_ALL from `cuD3D11GetDevices`.

Returns:

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- `cuD3D11GetDevices`, `cuGraphicsD3D11RegisterResource`

### 4.51.4.3 CUresult cuD3D11GetDevice (CUdevice * `pCudaDevice`, IDXGIAdapter * `pAdapter`)

Returns in `pCudaDevice` the CUDA-compatible device corresponding to the adapter `pAdapter` obtained from `IDXGIFactory::EnumAdapters`.

If no device on `pAdapter` is CUDA-compatible the call will return CUDA_ERROR_NO_DEVICE.

Parameters:

- `pCudaDevice` - Returned CUDA device corresponding to `pAdapter`
- `pAdapter` - Adapter to query for CUDA device

Returns:

- CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_NO_DEVICE, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- `cuD3D11CtxCreate`
4.51.4.4  

CUresult cuD3D11GetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, ID3D11Device *pD3D11Device, CUd3d11DeviceList deviceList)

Returns in *pCudaDeviceCount the number of CUDA-compatible device corresponding to the Direct3D 11 device pD3D11Device. Also returns in *pCudaDevices at most cudaDeviceCount of the the CUDA-compatible devices corresponding to the Direct3D 11 device pD3D11Device.

If any of the GPUs being used to render pDevice are not CUDA capable then the call will return CUDA_ERROR_-NO_DEVICE.

Parameters:

- *pCudaDeviceCount - Returned number of CUDA devices corresponding to pD3D11Device
- *pCudaDevices - Returned CUDA devices corresponding to pD3D11Device
- cudaDeviceCount - The size of the output device array pCudaDevices
- *pD3D11Device - Direct3D 11 device to query for CUDA devices
- deviceList - The set of devices to return. This set may be CU_D3D11_DEVICE_LIST_ALL for all devices, CU_D3D11_DEVICE_LIST_CURRENT_FRAME for the devices used to render the current frame (in SLI), or CU_D3D11_DEVICE_LIST_NEXT_FRAME for the devices used to render the next frame (in SLI).

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_NO_DEVICE, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D11CtxCreate

4.51.4.5  

CUresult cuD3D11GetDirect3DDevice (ID3D11Device **ppD3DDevice)

Returns in *ppD3DDevice the Direct3D device against which this CUDA context was created in cuD3D11CtxCreate().

Parameters:

- **ppD3DDevice - Returned Direct3D device corresponding to CUDA context

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D11GetDevice
4.51.4.6 CUresult cuGraphicsD3D11RegisterResource (CUgraphicsResource * pCudaResource, ID3D11Resource * pD3DResource, unsigned int Flags)

Registers the Direct3D 11 resource pD3DResource for access by CUDA and returns a CUDA handle to pD3Dresource in pCudaResource. The handle returned in pCudaResource may be used to map and unmap this resource until it is unregistered. On success this call will increase the internal reference count on pD3DResource. This reference count will be decremented when this resource is unregistered through cuGraphicsUnregisterResource().

This call is potentially high-overhead and should not be called every frame in interactive applications.

The type of pD3DResource must be one of the following.

- ID3D11Buffer: may be accessed through a device pointer.
- ID3D11Texture1D: individual subresources of the texture may be accessed via arrays
- ID3D11Texture2D: individual subresources of the texture may be accessed via arrays
- ID3D11Texture3D: individual subresources of the texture may be accessed via arrays

The Flags argument may be used to specify additional parameters at register time. The valid values for this parameter are

- CU_GRAPHICS_REGISTER_FLAGS_NONE: Specifies no hints about how this resource will be used.
- CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST: Specifies that CUDA will bind this resource to a surface reference.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary render target may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized for this context using cuD3D11CtxCreate then CUDA_ERROR_INVALID_CONTEXT is returned. If pD3DResource is of incorrect type or is already registered then CUDA_ERROR_INVALID_HANDLE is returned. If pD3DResource cannot be registered then CUDA_ERROR_UNKNOWN is returned. If Flags is not one of the above specified value then CUDA_ERROR_INVALID_VALUE is returned.

Parameters:

- pCudaResource - Returned graphics resource handle
- pD3DResource - Direct3D resource to register
- Flags - Parameters for resource registration

Returns:

- CUDA_SUCCESS,
- CUDA_ERROR_DEINITIALIZED,
- CUDA_ERROR_NOT_INITIALIZED,
- CUDA_ERROR_INVALID_CONTEXT,
- CUDA_ERROR_INVALID_VALUE,
- CUDA_ERROR_INVALID_HANDLE,
- CUDA_ERROR_OUT_OF_MEMORY,
- CUDA_ERROR_UNKNOWN

Generated for NVIDIA CUDA Library by Doxygen
Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:
4.52 VDPAU Interoperability

Functions

- CUresult cuGraphicsVDPAURegisterOutputSurface (CUgraphicsResource *pCudaResource, VdpOutputSurface vdpSurface, unsigned int flags)

  Registers a VDPAU VdpOutputSurface object.

- CUresult cuGraphicsVDPAURegisterVideoSurface (CUgraphicsResource *pCudaResource, VdpVideoSurface vdpSurface, unsigned int flags)

  Registers a VDPAU VdpVideoSurface object.

- CUresult cuVDPAUCtxCreate (CUcontext *pCtx, unsigned int flags, CUdevice device, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)

  Create a CUDA context for interoperability with VDPAU.

- CUresult cuVDPAUGetDevice (CUdevice *pDevice, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)

  Gets the CUDA device associated with a VDPAU device.

4.52.1 Detailed Description

This section describes the VDPAU interoperability functions of the low-level CUDA driver application programming interface.

4.52.2 Function Documentation

4.52.2.1 CUresult cuGraphicsVDPAURegisterOutputSurface (CUgraphicsResource * pCudaResource, VdpOutputSurface vdpSurface, unsigned int flags)

Registers the VdpOutputSurface specified by vdpSurface for access by CUDA. A handle to the registered object is returned as pCudaResource. The surface’s intended usage is specified using flags, as follows:

- CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.

- CU_GRAPHICS_MAP_RESOURCE_FLAGS_READ_ONLY: Specifies that CUDA will not write to this resource.

- CU_GRAPHICS_MAP_RESOURCE_FLAGS_WRITE_DISCARD: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

The VdpOutputSurface is presented as an array of subresources that may be accessed using pointers returned by cuGraphicsSubResourceGetMappedArray. The exact number of valid arrayIndex values depends on the VDPAU surface format. The mapping is shown in the table below. mipLevel must be 0.

<table>
<thead>
<tr>
<th>VdpRGBAFormat</th>
<th>arrayIndex</th>
<th>Size</th>
<th>Format</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDP_RGBA_FORMAT_B8G8R8A8</td>
<td>0</td>
<td>w x h</td>
<td>ARGB8</td>
<td>Entire surface</td>
</tr>
<tr>
<td>VDP_RGBA_FORMAT_R10G10B10A2</td>
<td>0</td>
<td>w x h</td>
<td>A2BGR10</td>
<td>Entire surface</td>
</tr>
</tbody>
</table>

Generated for NVIDIA CUDA Library by Doxygen
Parameters:

- `pCudaResource` - Pointer to the returned object handle
- `vdpSurface` - The VdpOutputSurface to be registered
- `flags` - Map flags

Returns:

- `CUDA_SUCCESS`
- `CUDA_ERROR_INVALID_HANDLE`
- `CUDA_ERROR_ALREADY_MAPPED`
- `CUDA_ERROR_INVALID_CONTEXT`

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

- `cuCtxCreate`
- `cuVDPAUCtxCreate`
- `cuGraphicsVDPAURegisterVideoSurface`
- `cuGraphicsUnregisterResource`
- `cuGraphicsResourceSetMapFlags`
- `cuGraphicsMapResources`
- `cuGraphicsUnmapResources`
- `cuGraphicsSubResourceGetMappedArray`
- `cuVDPAUGetDevice`

4.52.2.2 `CUDA_ERROR_invalidHandle`

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.

<table>
<thead>
<tr>
<th>VdpChromaType</th>
<th>arrayIndex</th>
<th>Size</th>
<th>Format</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDP_CHROMA_TYPE_420</td>
<td>0</td>
<td>w x h/2</td>
<td>R8</td>
<td>Top-field luma</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>w x h/2</td>
<td>R8</td>
<td>Bottom-field luma</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>w/2 x h/4</td>
<td>R8G8</td>
<td>Top-field chroma</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>w/2 x h/4</td>
<td>R8G8</td>
<td>Bottom-field chroma</td>
</tr>
<tr>
<td>VDP_CHROMA_TYPE_422</td>
<td>0</td>
<td>w x h/2</td>
<td>R8</td>
<td>Top-field luma</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>w x h/2</td>
<td>R8</td>
<td>Bottom-field luma</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>w/2 x h/2</td>
<td>R8G8</td>
<td>Top-field chroma</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>w/2 x h/2</td>
<td>R8G8</td>
<td>Bottom-field chroma</td>
</tr>
</tbody>
</table>

Parameters:

- `pCudaResource` - Pointer to the returned object handle
- `vdpSurface` - The VdpVideoSurface to be registered
flags - Map flags

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_INVALID_CONTEXT,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:


4.52.2.3 CUresult cuVDPAUCtxCreate (CUcontext * pCtx, unsigned int flags, CUdevice device, VdpDevice vdpDevice, VdpGetProcAddress * vdpGetProcAddress)

Creates a new CUDA context, initializes VDPAU interoperability, and associates the CUDA context with the calling thread. It must be called before performing any other VDPAU interoperability operations. It may fail if the needed VDPAU driver facilities are not available. For usage of the flags parameter, see cuCtxCreate().

Parameters:

pCtx - Returned CUDA context
flags - Options for CUDA context creation
device - Device on which to create the context
vdpDevice - The VdpDevice to interop with
vdpGetProcAddress - VDPAU’s VdpGetProcAddress function pointer

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:


4.52.2.4 CUresult cuVDPAUGetDevice (CUdevice * pDevice, VdpDevice vdpDevice, VdpGetProcAddress * vdpGetProcAddress)

Returns in *pDevice the CUDA device associated with a vdpDevice, if applicable.
Parameters:

- **pDevice** - Device associated with vdpDevice
- **vdpDevice** - A VdpDevice handle
- **vdpGetProcAddress** - VDPAU’s VdpGetProcAddress function pointer

Returns:

- CUDA_SUCCESS
- CUDA_ERROR_DEINITIALIZED
- CUDA_ERROR_NOT_INITIALIZED
- CUDA_ERROR_INVALID_CONTEXT
- CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

Chapter 5

Data Structure Documentation

5.1 CUDA_ARRAY3D_DESCRIPTOR_st Struct Reference

Data Fields

- size_t Depth
- unsigned int Flags
- CUarray_format Format
- size_t Height
- unsigned int NumChannels
- size_t Width

5.1.1 Detailed Description

3D array descriptor

5.1.2 Field Documentation

5.1.2.1 size_t CUDA_ARRAY3D_DESCRIPTOR_st::Depth

Depth of 3D array

5.1.2.2 unsigned int CUDA_ARRAY3D_DESCRIPTOR_st::Flags

Flags

5.1.2.3 CUarray_format CUDA_ARRAY3D_DESCRIPTOR_st::Format

Array format

5.1.2.4 size_t CUDA_ARRAY3D_DESCRIPTOR_st::Height

Height of 3D array
5.1.2.5 unsigned int CUDA_ARRAY3D_DESCRIPTOR_st::NumChannels

Channels per array element

5.1.2.6 size_t CUDA_ARRAY3D_DESCRIPTOR_st::Width

Width of 3D array
5.2 CUDA_ARRAY_DESCRIPTOR_st Struct Reference

Data Fields

- CUarray_format Format
- size_t Height
- unsigned int NumChannels
- size_t Width

5.2.1 Detailed Description

Array descriptor

5.2.2 Field Documentation

5.2.2.1 CUarray_format CUDA_ARRAY_DESCRIPTOR_st::Format

Array format

5.2.2.2 size_t CUDA_ARRAY_DESCRIPTOR_st::Height

Height of array

5.2.2.3 unsigned int CUDA_ARRAY_DESCRIPTOR_st::NumChannels

Channels per array element

5.2.2.4 size_t CUDA_ARRAY_DESCRIPTOR_st::Width

Width of array
5.3 CUDA_MEMCPY2D_st Struct Reference

Data Fields

- CUarray dstArray
- CDeviceptr dstDevice
- void * dstHost
- CUmemorytype dstMemoryType
- size_t dstPitch
- size_t dstXInBytes
- size_t dstY
- size_t Height
- CUarray srcArray
- CDeviceptr srcDevice
- const void * srcHost
- CUmemorytype srcMemoryType
- size_t srcPitch
- size_t srcXInBytes
- size_t srcY
- size_t WidthInBytes

5.3.1 Detailed Description

2D memory copy parameters

5.3.2 Field Documentation

5.3.2.1 CUarray CUDA_MEMCPY2D_st::dstArray

Destination array reference

5.3.2.2 CDeviceptr CUDA_MEMCPY2D_st::dstDevice

Destination device pointer

5.3.2.3 void* CUDA_MEMCPY2D_st::dstHost

Destination host pointer

5.3.2.4 CUmemorytype CUDA_MEMCPY2D_st::dstMemoryType

Destination memory type (host, device, array)

5.3.2.5 size_t CUDA_MEMCPY2D_st::dstPitch

Destination pitch (ignored when dst is array)
5.3.2.6 size_t CUDA_MEMCPY2D_st::dstXInBytes
Destination X in bytes

5.3.2.7 size_t CUDA_MEMCPY2D_st::dstY
Destination Y

5.3.2.8 size_t CUDA_MEMCPY2D_st::Height
Height of 2D memory copy

5.3.2.9 CUarray CUDA_MEMCPY2D_st::srcArray
Source array reference

5.3.2.10 CUdeviceptr CUDA_MEMCPY2D_st::srcDevice
Source device pointer

5.3.2.11 const void* CUDA_MEMCPY2D_st::srcHost
Source host pointer

5.3.2.12 CUMemorytype CUDA_MEMCPY2D_st::srcMemoryType
Source memory type (host, device, array)

5.3.2.13 size_t CUDA_MEMCPY2D_st::srcPitch
Source pitch (ignored when src is array)

5.3.2.14 size_t CUDA_MEMCPY2D_st::srcXInBytes
Source X in bytes

5.3.2.15 size_t CUDA_MEMCPY2D_st::srcY
Source Y

5.3.2.16 size_t CUDA_MEMCPY2D_st::WidthInBytes
Width of 2D memory copy in bytes
5.4 CUDA_MEMCPY3D_PEER_st Struct Reference

Data Fields

- size_t Depth
- CUarray dstArray
- CUcontext dstContext
- CUdeviceptr dstDevice
- size_t dstHeight
- void * dstHost
- size_t dstLOD
- CUmemorytype dstMemoryType
- size_t dstPitch
- size_t dstXInBytes
- size_t dstY
- size_t dstZ
- size_t Height
- CUarray srcArray
- CUcontext srcContext
- CUdeviceptr srcDevice
- size_t srcHeight
- const void * srcHost
- size_t srcLOD
- CUmemorytype srcMemoryType
- size_t srcPitch
- size_t srcXInBytes
- size_t srcY
- size_t srcZ
- size_t WidthInBytes

5.4.1 Detailed Description

3D memory cross-context copy parameters

5.4.2 Field Documentation

5.4.2.1 size_t CUDA_MEMCPY3D_PEER_st::Depth

Depth of 3D memory copy

5.4.2.2 CUarray CUDA_MEMCPY3D_PEER_st::dstArray

Destination array reference

5.4.2.3 CUcontext CUDA_MEMCPY3D_PEER_st::dstContext

Destination context (ignored with dstMemoryType is CU_MEMORYTYPE_ARRAY)
5.4.2.4 CUdeviceptr CUDA_MEMCPY3D_PEER_st::dstDevice
Destination device pointer

5.4.2.5 size_t CUDA_MEMCPY3D_PEER_st::dstHeight
Destination height (ignored when dst is array; may be 0 if Depth==1)

5.4.2.6 void* CUDA_MEMCPY3D_PEER_st::dstHost
Destination host pointer

5.4.2.7 size_t CUDA_MEMCPY3D_PEER_st::dstLOD
Destination LOD

5.4.2.8 CUmemorytype CUDA_MEMCPY3D_PEER_st::dstMemoryType
Destination memory type (host, device, array)

5.4.2.9 size_t CUDA_MEMCPY3D_PEER_st::dstPitch
Destination pitch (ignored when dst is array)

5.4.2.10 size_t CUDA_MEMCPY3D_PEER_st::dstXInBytes
Destination X in bytes

5.4.2.11 size_t CUDA_MEMCPY3D_PEER_st::dstY
Destination Y

5.4.2.12 size_t CUDA_MEMCPY3D_PEER_st::dstZ
Destination Z

5.4.2.13 size_t CUDA_MEMCPY3D_PEER_st::Height
Height of 3D memory copy

5.4.2.14 CUarray CUDA_MEMCPY3D_PEER_st::srcArray
Source array reference

5.4.2.15 CUcontext CUDA_MEMCPY3D_PEER_st::srcContext
Source context (ignored with srcMemoryType is CU_MEMORYTYPE_ARRAY)
5.4.2.16  CUdeviceptr CUDA_MEMCPY3D_PEER_st::srcDevice

Source device pointer

5.4.2.17  size_t CUDA_MEMCPY3D_PEER_st::srcHeight

Source height (ignored when src is array; may be 0 if Depth==1)

5.4.2.18  const void* CUDA_MEMCPY3D_PEER_st::srcHost

Source host pointer

5.4.2.19  size_t CUDA_MEMCPY3D_PEER_st::srcLOD

Source LOD

5.4.2.20  CUmemorytype CUDA_MEMCPY3D_PEER_st::srcMemoryType

Source memory type (host, device, array)

5.4.2.21  size_t CUDA_MEMCPY3D_PEER_st::srcPitch

Source pitch (ignored when src is array)

5.4.2.22  size_t CUDA_MEMCPY3D_PEER_st::srcXInBytes

Source X in bytes

5.4.2.23  size_t CUDA_MEMCPY3D_PEER_st::srcY

Source Y

5.4.2.24  size_t CUDA_MEMCPY3D_PEER_st::srcZ

Source Z

5.4.2.25  size_t CUDA_MEMCPY3D_PEER_st::WidthInBytes

Width of 3D memory copy in bytes
5.5 CUDA_MEMCPY3D_st Struct Reference

Data Fields

- size_t Depth
- CUarray dstArray
- CUdeviceptr dstDevice
- size_t dstHeight
- void * dstHost
- size_t dstLOD
- CUmemorytype dstMemoryType
- size_t dstPitch
- size_t dstXInBytes
- size_t dstY
- size_t dstZ
- size_t Height
- void * reserved0
- void * reserved1
- CUarray srcArray
- CUdeviceptr srcDevice
- size_t srcHeight
- const void * srcHost
- size_t srcLOD
- CUmemorytype srcMemoryType
- size_t srcPitch
- size_t srcXInBytes
- size_t srcY
- size_t srcZ
- size_t WidthInBytes

5.5.1 Detailed Description

3D memory copy parameters

5.5.2 Field Documentation

5.5.2.1 size_t CUDA_MEMCPY3D_st::Depth

Depth of 3D memory copy

5.5.2.2 CUarray CUDA_MEMCPY3D_st::dstArray

Destination array reference

5.5.2.3 CUdeviceptr CUDA_MEMCPY3D_st::dstDevice

Destination device pointer
5.5.2.4  size_t CUDA_MEMCPY3D_st::dstHeight

Destination height (ignored when dst is array; may be 0 if Depth==1)

5.5.2.5  void* CUDA_MEMCPY3D_st::dstHost

Destination host pointer

5.5.2.6  size_t CUDA_MEMCPY3D_st::dstLOD

Destination LOD

5.5.2.7  CUnonhost CUDA_MEMCPY3D_st::dstNonhost

Destination nonhost (host, device, array)

5.5.2.8  size_t CUDA_MEMCPY3D_st::dstPitch

Destination pitch (ignored when dst is array)

5.5.2.9  size_t CUDA_MEMCPY3D_st::dstXInBytes

Destination X in bytes

5.5.2.10  size_t CUDA_MEMCPY3D_st::dstY

Destination Y

5.5.2.11  size_t CUDA_MEMCPY3D_st::dstZ

Destination Z

5.5.2.12  size_t CUDA_MEMCPY3D_st::Height

Height of 3D memory copy

5.5.2.13  void* CUDA_MEMCPY3D_st::reserved0

Must be NULL

5.5.2.14  void* CUDA_MEMCPY3D_st::reserved1

Must be NULL

5.5.2.15  CUnonhost CUDA_MEMCPY3D_st::srcNonhost

Source nonhost (host, device, array)
5.5 CUDA_MEMCPY3D_st Struct Reference

5.5.2.16 CUdeviceptr CUDA_MEMCPY3D_st::srcDevice
Source device pointer

5.5.2.17 size_t CUDA_MEMCPY3D_st::srcHeight
Source height (ignored when src is array; may be 0 if Depth==1)

5.5.2.18 const void* CUDA_MEMCPY3D_st::srcHost
Source host pointer

5.5.2.19 size_t CUDA_MEMCPY3D_st::srcLOD
Source LOD

5.5.2.20 CUMemorytype CUDA_MEMCPY3D_st::srcMemoryType
Source memory type (host, device, array)

5.5.2.21 size_t CUDA_MEMCPY3D_st::srcPitch
Source pitch (ignored when src is array)

5.5.2.22 size_t CUDA_MEMCPY3D_st::srcXInBytes
Source X in bytes

5.5.2.23 size_t CUDA_MEMCPY3D_st::srcY
Source Y

5.5.2.24 size_t CUDA_MEMCPY3D_st::srcZ
Source Z

5.5.2.25 size_t CUDA_MEMCPY3D_st::WidthInBytes
Width of 3D memory copy in bytes
5.6 cudaChannelFormatDesc Struct Reference

Data Fields

- enum cudaChannelFormatKind f
- int w
- int x
- int y
- int z

5.6.1 Detailed Description

CUDA Channel format descriptor

5.6.2 Field Documentation

5.6.2.1 enum cudaChannelFormatKind cudaChannelFormatDesc::f

Channel format kind

5.6.2.2 int cudaChannelFormatDesc::w

w

5.6.2.3 int cudaChannelFormatDesc::x

x

5.6.2.4 int cudaChannelFormatDesc::y

y

5.6.2.5 int cudaChannelFormatDesc::z

z
5.7 \texttt{cudaDeviceProp} Struct Reference

Data Fields

- \texttt{int asyncEngineCount}
- \texttt{int canMapHostMemory}
- \texttt{int clockRate}
- \texttt{int computeMode}
- \texttt{int concurrentKernels}
- \texttt{int deviceOverlap}
- \texttt{int ECCEnabled}
- \texttt{int integrated}
- \texttt{int kernelExecTimeoutEnabled}
- \texttt{int l2CacheSize}
- \texttt{int major}
- \texttt{int maxGridSize [3]}
- \texttt{int maxTexture1D}
- \texttt{int maxTexture1DLayered [2]}
- \texttt{int maxTexture2D [2]}
- \texttt{int maxTexture2DLayered [3]}
- \texttt{int maxTexture3D [3]}
- \texttt{int maxThreadsDim [3]}
- \texttt{int maxThreadsPerBlock}
- \texttt{int maxThreadsPerMultiProcessor}
- \texttt{int memoryBusWidth}
- \texttt{int memoryClockRate}
- \texttt{size_t memPitch}
- \texttt{int minor}
- \texttt{int multiProcessorCount}
- \texttt{char name [256]}
- \texttt{int pciBusID}
- \texttt{int pciDeviceID}
- \texttt{int pciDomainID}
- \texttt{int regsPerBlock}
- \texttt{size_t sharedMemPerBlock}
- \texttt{size_t surfaceAlignment}
- \texttt{int tccDriver}
- \texttt{size_t textureAlignment}
- \texttt{size_t totalConstMem}
- \texttt{size_t totalGlobalMem}
- \texttt{int unifiedAddressing}
- \texttt{int warpSize}

5.7.1 Detailed Description

CUDA device properties
5.7.2 Field Documentation

5.7.2.1 int cudaDeviceProp::asyncEngineCount

Number of asynchronous engines

5.7.2.2 int cudaDeviceProp::canMapHostMemory

Device can map host memory with cudaHostAlloc/cudaHostGetDevicePointer

5.7.2.3 int cudaDeviceProp::clockRate

Clock frequency in kilohertz

5.7.2.4 int cudaDeviceProp::computeMode

Compute mode (See cudaComputeMode)

5.7.2.5 int cudaDeviceProp::concurrentKernels

Device can possibly execute multiple kernels concurrently

5.7.2.6 int cudaDeviceProp::deviceOverlap

Device can concurrently copy memory and execute a kernel. Deprecated. Use instead asyncEngineCount.

5.7.2.7 int cudaDeviceProp::ECCEnabled

Device has ECC support enabled

5.7.2.8 int cudaDeviceProp::integrated

Device is integrated as opposed to discrete

5.7.2.9 int cudaDeviceProp::kernelExecTimeoutEnabled

Specified whether there is a run time limit on kernels

5.7.2.10 int cudaDeviceProp::l2CacheSize

Size of L2 cache in bytes

5.7.2.11 int cudaDeviceProp::major

Major compute capability
5.7.2.12 int cudaDeviceProp::maxGridSize[3]
Maximum size of each dimension of a grid

5.7.2.13 int cudaDeviceProp::maxTexture1D
Maximum 1D texture size

5.7.2.14 int cudaDeviceProp::maxTexture1DLayered[2]
Maximum 1D layered texture dimensions

5.7.2.15 int cudaDeviceProp::maxTexture2D[2]
Maximum 2D texture dimensions

5.7.2.16 int cudaDeviceProp::maxTexture2DLayered[3]
Maximum 2D layered texture dimensions

5.7.2.17 int cudaDeviceProp::maxTexture3D[3]
Maximum 3D texture dimensions

5.7.2.18 int cudaDeviceProp::maxThreadsDim[3]
Maximum size of each dimension of a block

5.7.2.19 int cudaDeviceProp::maxThreadsPerBlock
Maximum number of threads per block

5.7.2.20 int cudaDeviceProp::maxThreadsPerMultiProcessor
Maximum resident threads per multiprocessor

5.7.2.21 int cudaDeviceProp::memoryBusWidth
Global memory bus width in bits

5.7.2.22 int cudaDeviceProp::memoryClockRate
Peak memory clock frequency in kilohertz

5.7.2.23 size_t cudaDeviceProp::memPitch
Maximum pitch in bytes allowed by memory copies

Generated for NVIDIA CUDA Library by Doxygen
5.7.2.24  int cudaDeviceProp::minor

Minor compute capability

5.7.2.25  int cudaDeviceProp::multiProcessorCount

Number of multiprocessors on device

5.7.2.26  char cudaDeviceProp::name[256]

ASCII string identifying device

5.7.2.27  int cudaDeviceProp::pciBusID

PCI bus ID of the device

5.7.2.28  int cudaDeviceProp::pciDeviceID

PCI device ID of the device

5.7.2.29  int cudaDeviceProp::pciDomainID

PCI domain ID of the device

5.7.2.30  int cudaDeviceProp::regsPerBlock

32-bit registers available per block

5.7.2.31  size_t cudaDeviceProp::sharedMemPerBlock

Shared memory available per block in bytes

5.7.2.32  size_t cudaDeviceProp::surfaceAlignment

Alignment requirements for surfaces

5.7.2.33  int cudaDeviceProp::tccDriver

1 if device is a Tesla device using TCC driver, 0 otherwise

5.7.2.34  size_t cudaDeviceProp::textureAlignment

Alignment requirement for textures

5.7.2.35  size_t cudaDeviceProp::totalConstMem

Constant memory available on device in bytes
5.7.2.36  size_t cudaDeviceProp::totalGlobalMem

Global memory available on device in bytes

5.7.2.37  int cudaDeviceProp::unifiedAddressing

Device shares a unified address space with the host

5.7.2.38  int cudaDeviceProp::warpSize

Warp size in threads
5.8   cudaExtent Struct Reference

Data Fields

- size_t depth
- size_t height
- size_t width

5.8.1  Detailed Description

CUDA extent

See also:
  make_cudaExtent

5.8.2  Field Documentation

5.8.2.1  size_t cudaExtent::depth

Depth in elements

5.8.2.2  size_t cudaExtent::height

Height in elements

5.8.2.3  size_t cudaExtent::width

Width in elements when referring to array memory, in bytes when referring to linear memory
5.9 cudaFuncAttributes Struct Reference

Data Fields

- int binaryVersion
- size_t constSizeBytes
- size_t localSizeBytes
- int maxThreadsPerBlock
- int numRegs
- int ptxVersion
- size_t sharedSizeBytes

5.9.1 Detailed Description

CUDA function attributes

5.9.2 Field Documentation

5.9.2.1 int cudaFuncAttributes::binaryVersion

The binary architecture version for which the function was compiled. This value is the major binary version * 10 +
the minor binary version, so a binary version 1.3 function would return the value 13.

5.9.2.2 size_t cudaFuncAttributes::constSizeBytes

The size in bytes of user-allocated constant memory required by this function.

5.9.2.3 size_t cudaFuncAttributes::localSizeBytes

The size in bytes of local memory used by each thread of this function.

5.9.2.4 int cudaFuncAttributes::maxThreadsPerBlock

The maximum number of threads per block, beyond which a launch of the function would fail. This number depends
on both the function and the device on which the function is currently loaded.

5.9.2.5 int cudaFuncAttributes::numRegs

The number of registers used by each thread of this function.

5.9.2.6 int cudaFuncAttributes::ptxVersion

The PTX virtual architecture version for which the function was compiled. This value is the major PTX version * 10
+ the minor PTX version, so a PTX version 1.3 function would return the value 13.
5.9.2.7  size_t cudaFuncAttributes::sharedSizeBytes

The size in bytes of statically-allocated shared memory per block required by this function. This does not include
dynamically-allocated shared memory requested by the user at runtime.
5.10 cudaMemcpy3DParms Struct Reference

Data Fields

- struct cudaArray * dstArray
- struct cudaMemcpyKind kind
- struct cudaArray * srcArray
- struct cudaMemcpyKind kind

5.10.1 Detailed Description

CUDA 3D memory copying parameters

5.10.2 Field Documentation

5.10.2.1 struct cudaArray* cudaMemcpy3DParms::dstArray [read]

Destination memory address

5.10.2.2 struct cudaMemcpyKind cudaMemcpy3DParms::dstPos [read]

Destination position offset

5.10.2.3 struct cudaMemcpyKind cudaMemcpy3DParms::srcArray [read]

Source memory address

5.10.2.4 struct cudaMemcpyKind cudaMemcpy3DParms::srcPos [read]

Source position offset
5.10.2.8 struct cudaPitchedPtr cudaMemcpy3DParms::srcPtr [read]

Pitched source memory address
5.11 cudaMemcpy3DPeerParms Struct Reference

Data Fields

- struct cudaArray * dstArray
- int dstDevice
- struct cudaMemcpy3DPeerParms::dstArray [read]

Destination memory address

- int cudaMemcpy3DPeerParms::dstDevice

Destination device

- struct cudaMemcpy3DPeerParms::dstPos [read]

Destination position offset

- struct cudaMemcpy3DPeerParms::dstPtr [read]

Pitched destination memory address

- struct cudaMemcpy3DPeerParms::extent [read]

Requested memory copy size

- struct cudaMemcpy3DPeerParms::srcArray [read]

Source memory address

- int cudaMemcpy3DPeerParms::srcDevice

Source device

5.11.1 Detailed Description

CUDA 3D cross-device memory copying parameters

5.11.2 Field Documentation

5.11.2.1 struct cudaArray * cudaMemcpy3DPeerParms::dstArray [read]

Destination memory address

5.11.2.2 int cudaMemcpy3DPeerParms::dstDevice

Destination device

5.11.2.3 struct cudaMemcpy3DPeerParms::dstPos [read]

Destination position offset

5.11.2.4 struct cudaMemcpy3DPeerParms::dstPtr [read]

Pitched destination memory address

5.11.2.5 struct cudaMemcpy3DPeerParms::extent [read]

Requested memory copy size

5.11.2.6 struct cudaArray * cudaMemcpy3DPeerParms::srcArray [read]

Source memory address

5.11.2.7 int cudaMemcpy3DPeerParms::srcDevice

Source device
5.11.2.8  **struct cudaMemcpy3DPeerParms::srcPos**  [read]

Source position offset

5.11.2.9  **struct cudaMemcpy3DPeerParms::srcPtr**  [read]

Pitched source memory address
5.12 cudaPitchedPtr Struct Reference

Data Fields

• size_t pitch
• void * ptr
• size_t xsize
• size_t ysize

5.12.1 Detailed Description

CUDA Pitched memory pointer

See also:

make_cudaPitchedPtr

5.12.2 Field Documentation

5.12.2.1 size_t cudaPitchedPtr::pitch

Pitch of allocated memory in bytes

5.12.2.2 void* cudaPitchedPtr::ptr

Pointer to allocated memory

5.12.2.3 size_t cudaPitchedPtr::xsize

Logical width of allocation in elements

5.12.2.4 size_t cudaPitchedPtr::ysize

Logical height of allocation in elements
5.13 `cudaPointerAttributes Struct Reference`

**Data Fields**

- `int device`
- `void * devicePointer`
- `void * hostPointer`
- `enum cudaMemcpyType memoryType`

5.13.1 **Detailed Description**

CUDA pointer attributes

5.13.2 **Field Documentation**

5.13.2.1 `int cudaPointerAttributes::device`

The device against which the memory was allocated or registered. If the memory type is `cudaMemoryTypeDevice` then this identifies the device on which the memory referred physically resides. If the memory type is `cudaMemoryTypeHost` then this identifies the device which was current when the memory was allocated or registered (and if that device is deinitialized then this allocation will vanish with that device’s state).

5.13.2.2 `void * cudaPointerAttributes::devicePointer`

The address which may be dereferenced on the current device to access the memory or NULL if no such address exists.

5.13.2.3 `void * cudaPointerAttributes::hostPointer`

The address which may be dereferenced on the host to access the memory or NULL if no such address exists.

5.13.2.4 `enum cudaMemcpyType cudaPointerAttributes::memoryType`

The physical location of the memory, `cudaMemoryTypeHost` or `cudaMemoryTypeDevice`. 

Generated for NVIDIA CUDA Library by Doxygen
5.14 cudaPos Struct Reference

Data Fields

- size_t x
- size_t y
- size_t z

5.14.1 Detailed Description

CUDA 3D position

See also:

make_cudaPos

5.14.2 Field Documentation

5.14.2.1 size_t cudaPos::x

x

5.14.2.2 size_t cudaPos::y

y

5.14.2.3 size_t cudaPos::z

z
5.15 CUdevprop_st Struct Reference

Data Fields

- int clockRate
- int maxGridSize [3]
- int maxThreadsDim [3]
- int maxThreadsPerBlock
- int memPitch
- int regsPerBlock
- int sharedMemPerBlock
- int SIMDWidth
- int textureAlign
- int totalConstantMemory

5.15.1 Detailed Description

Legacy device properties

5.15.2 Field Documentation

5.15.2.1 int CUdevprop_st::clockRate

Clock frequency in kilohertz

5.15.2.2 int CUdevprop_st::maxGridSize[3]

Maximum size of each dimension of a grid

5.15.2.3 int CUdevprop_st::maxThreadsDim[3]

Maximum size of each dimension of a block

5.15.2.4 int CUdevprop_st::maxThreadsPerBlock

Maximum number of threads per block

5.15.2.5 int CUdevprop_st::memPitch

Maximum pitch in bytes allowed by memory copies

5.15.2.6 int CUdevprop_st::regsPerBlock

32-bit registers available per block

5.15.2.7 int CUdevprop_st::sharedMemPerBlock

Shared memory available per block in bytes
5.15.2.8  int CUdevprop_st::SIMDWidth

Warp size in threads

5.15.2.9  int CUdevprop_st::textureAlign

Alignment requirement for textures

5.15.2.10 int CUdevprop_st::totalConstantMemory

Constant memory available on device in bytes
5.16 surfaceReference Struct Reference

Data Fields

- struct cudaChannelFormatDesc channelDesc

5.16.1 Detailed Description

CUDA Surface reference

5.16.2 Field Documentation

5.16.2.1 struct cudaChannelFormatDesc surfaceReference::channelDesc [read]

Channel descriptor for surface reference
5.17 textureReference Struct Reference

Data Fields

- enum cudaTextureAddressMode addressMode[3]
- struct cudaChannelFormatDesc channelDesc
- enum cudaTextureFilterMode filterMode
- int normalized
- int sRGB

5.17.1 Detailed Description

CUDA texture reference

5.17.2 Field Documentation

5.17.2.1 enum cudaTextureAddressMode textureReference::addressMode[3]

Texture address mode for up to 3 dimensions

5.17.2.2 struct cudaChannelFormatDesc textureReference::channelDesc

Channel descriptor for the texture reference

5.17.2.3 enum cudaTextureFilterMode textureReference::filterMode

Texture filter mode

5.17.2.4 int textureReference::normalized

Indicates whether texture reads are normalized or not

5.17.2.5 int textureReference::sRGB

Perform sRGB->linear conversion during texture read
Index

addressMode
  textureReference, 383
asyncEngineCount
  cudaDeviceProp, 366

binaryVersion
  cudaFuncAttributes, 371

C++ API Routines, 115
canMapHostMemory
  cudaDeviceProp, 366
channelDesc
  surfaceReference, 382
  textureReference, 383
clockRate
  cudaDeviceProp, 366
  CUdevprop_st, 380
computeMode
  cudaDeviceProp, 366
concurrentKernels
  cudaDeviceProp, 366
constSizeBytes
  cudaFuncAttributes, 371
Context Management, 197
CU_AD_FORMAT_FLOAT
  CUDA_TYPES, 179
CU_AD_FORMAT_HALF
  CUDA_TYPES, 179
CU_AD_FORMAT_SIGNED_INT16
  CUDA_TYPES, 179
CU_AD_FORMAT_SIGNED_INT32
  CUDA_TYPES, 179
CU_AD_FORMAT_SIGNED_INT8
  CUDA_TYPES, 179
CU_AD_FORMAT_UNSIGNED_INT16
  CUDA_TYPES, 179
CU_AD_FORMAT_UNSIGNED_INT32
  CUDA_TYPES, 179
CU_AD_FORMAT_UNSIGNED_INT8
  CUDA_TYPES, 179
CU_COMPUTEMODE_DEFAULT
  CUDA_TYPES, 179
CU_COMPUTEMODE_EXCLUSIVE
  CUDA_TYPES, 179
CU_COMPUTEMODE_EXCLUSIVE_PROCESS
  CUDA_TYPES, 179
CU_COMPUTEMODE_PROHIBITED
  CUDA_TYPES, 179
CU_CTX_BLOCKING_SYNC
  CUDA_TYPES, 180
CU_CTX_LMEM_RESIZE_TO_MAX
  CUDA_TYPES, 180
CU_CTX_MAP_HOST
  CUDA_TYPES, 180
CU_CTX_SCHED_AUTO
  CUDA_TYPES, 180
CU_CTX_SCHED_BLOCKING_SYNC
  CUDA_TYPES, 180
CU_CTX_SCHED_SPIN
  CUDA_TYPES, 180
CU_CTX_SCHED_YIELD
  CUDA_TYPES, 180
CU_CUBEMAP_FACE_NEGATIVE_X
  CUDA_TYPES, 179
CU_CUBEMAP_FACE_NEGATIVE_Y
  CUDA_TYPES, 179
CU_CUBEMAP_FACE_NEGATIVE_Z
  CUDA_TYPES, 179
CU_CUBEMAP_FACE_POSITIVE_X
  CUDA_TYPES, 179
CU_CUBEMAP_FACE_POSITIVE_Y
  CUDA_TYPES, 179
CU_CUBEMAP_FACE_POSITIVE_Z
  CUDA_TYPES, 179
CU_D3D10_DEVICE_LIST_ALL
  CUDA_D3D10, 329
CU_D3D10_DEVICE_LIST_CURRENT_FRAME
  CUDA_D3D10, 329
CU_D3D10_DEVICE_LIST_NEXT_FRAME
  CUDA_D3D10, 329
CU_D3D11_DEVICE_LIST_ALL
  CUDA_D3D11, 344
CU_D3D11_DEVICE_LIST_CURRENT_FRAME
  CUDA_D3D11, 344
CU_D3D11_DEVICE_LIST_NEXT_FRAME
  CUDA_D3D11, 344
CU_D3D9_DEVICE_LIST_ALL
  CUDA_D3D9, 314
CU_D3D9_DEVICE_LIST_CURRENT_FRAME
  CUDA_D3D9, 314
CU_D3D9_DEVICE_LIST_NEXT_FRAME
CUDA_D3D9, 314
CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_CLOCK_RATE
CUDA_TYPES, 182
CU_DEVICE_ATTRIBUTE_COMPUTE_MODE
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_ECC_ENABLED
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_GPU_OVERLAP
CUDA_TYPES, 182
CU_DEVICE_ATTRIBUTE_INTEGRATED
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_L2_CACHE_SIZE
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X
CUDA_TYPES, 182
CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y
CUDA_TYPES, 182
CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z
CUDA_TYPES, 182
CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X
CUDA_TYPES, 182
CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y
CUDA_TYPES, 182
CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z
CUDA_TYPES, 182
CU_DEVICE_ATTRIBUTE_MAX_PITCH
CUDA_TYPES, 182
CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK
CUDA_TYPES, 182
CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK
CUDA_TYPES, 182
CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK
CUDA_TYPES, 182
CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR
CUDA_TYPES, 182
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_HEIGHT
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_NUMSLICES
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_WIDTH
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_HEIGHT
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_WIDTH
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT
CUDA_TYPES, 182
CU_DEVICE_ATTRIBUTE_PCI_BUS_ID
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_PCI_DEVICE_ID
CUDA_TYPES, 183
CU_DEVICE_ATTRIBUTE_PCI_DOMAIN_ID
CUDA_TYPES, 183
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_TYPES, 183</td>
<td></td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_REGISTERS_PER_BLOCK</td>
<td>CUDA_TYPES, 182</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_SHARED_MEMORY_PER_BLOCK</td>
<td>CUDA_TYPES, 182</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_SURFACE_ALIGNMENT</td>
<td>CUDA_TYPES, 183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_TCC_DRIVER</td>
<td>CUDA_TYPES, 183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT</td>
<td>CUDA_TYPES, 182</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY</td>
<td>CUDA_TYPES, 182</td>
</tr>
<tr>
<td>CU_FUNC_ATTRIBUTE_BINARY_VERSION</td>
<td>CUDA_TYPES, 184</td>
</tr>
<tr>
<td>CU_FUNC_ATTRIBUTE_CONST_SIZE_BYTES</td>
<td>CUDA_TYPES, 184</td>
</tr>
<tr>
<td>CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES</td>
<td>CUDA_TYPES, 184</td>
</tr>
<tr>
<td>CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK</td>
<td>CUDA_TYPES, 184</td>
</tr>
<tr>
<td>CU_FUNC_ATTRIBUTE_NUM_REGS</td>
<td>CUDA_TYPES, 184</td>
</tr>
<tr>
<td>CU_FUNC_ATTRIBUTE_PTX_VERSION</td>
<td>CUDA_TYPES, 184</td>
</tr>
<tr>
<td>CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES</td>
<td>CUDA_TYPES, 184</td>
</tr>
<tr>
<td>CU_FUNC_CACHE_PREFER_L1</td>
<td>CUDA_TYPES, 184</td>
</tr>
<tr>
<td>CU_FUNC_CACHE_PREFER_NONE</td>
<td>CUDA_TYPES, 184</td>
</tr>
<tr>
<td>CU_FUNC_CACHE_PREFER_SHARED</td>
<td>CUDA_TYPES, 184</td>
</tr>
<tr>
<td>CU_JIT_ERROR_LOG_BUFFER</td>
<td>CUDA_TYPES, 185</td>
</tr>
<tr>
<td>CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES</td>
<td>CUDA_TYPES, 185</td>
</tr>
<tr>
<td>CU_JIT_MAX_REGISTERS</td>
<td>CUDA_TYPES, 185</td>
</tr>
<tr>
<td>CU_JIT_OPTIMIZATION_LEVEL</td>
<td>CUDA_TYPES, 185</td>
</tr>
<tr>
<td>CU_JIT_THREAD'S_PER_BLOCK</td>
<td>CUDA_TYPES, 185</td>
</tr>
<tr>
<td>CU_JIT_WALL_TIME</td>
<td>CUDA_TYPES, 185</td>
</tr>
<tr>
<td>CU_JIT_INFO_LOG_BUFFER</td>
<td>CUDA_TYPES, 185</td>
</tr>
<tr>
<td>CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES</td>
<td>CUDA_TYPES, 185</td>
</tr>
<tr>
<td>CU_JIT_MAX_REGISTERS</td>
<td>CUDA_TYPES, 185</td>
</tr>
<tr>
<td>CU_JIT_OPTIMIZATION_LEVEL</td>
<td>CUDA_TYPES, 185</td>
</tr>
<tr>
<td>CU_JIT_THREADS_PER_BLOCK</td>
<td>CUDA_TYPES, 185</td>
</tr>
<tr>
<td>CU_JIT_WALL_TIME</td>
<td>CUDA_TYPES, 185</td>
</tr>
<tr>
<td>CU_JIT_FALLBACK_STRATEGY</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CUDA_MEMORYTYPE_ARRAY</td>
<td>CUDA_TYPES, 187</td>
</tr>
<tr>
<td>CUDA_MEMORYTYPE_DEVICE</td>
<td>CUDA_TYPES, 187</td>
</tr>
<tr>
<td>CUDA_MEMORYTYPE_HOST</td>
<td>CUDA_TYPES, 187</td>
</tr>
<tr>
<td>CUDA_MEMORYTYPE_UNIFIED</td>
<td>CUDA_TYPES, 187</td>
</tr>
<tr>
<td>CU_POINTER_ATTRIBUTE_CONTEXT</td>
<td>CUDA_TYPES, 187</td>
</tr>
<tr>
<td>CU_POINTER_ATTRIBUTE_DEVICE_POINTER</td>
<td>CUDA_TYPES, 187</td>
</tr>
<tr>
<td>CU_POINTER_ATTRIBUTE_HOST_POINTER</td>
<td>CUDA_TYPES, 187</td>
</tr>
<tr>
<td>CU_POINTER_ATTRIBUTE_MEMORY_TYPE</td>
<td>CUDA_TYPES, 187</td>
</tr>
<tr>
<td>CU_POINTER_ATTRIBUTE_CONTEXT</td>
<td>CUDA_TYPES, 187</td>
</tr>
<tr>
<td>CU_POINTER_ATTRIBUTE_DEVICE_POINTER</td>
<td>CUDA_TYPES, 187</td>
</tr>
<tr>
<td>CU_POINTER_ATTRIBUTE_HOST_POINTER</td>
<td>CUDA_TYPES, 187</td>
</tr>
<tr>
<td>CU_POINTER_ATTRIBUTE_MEMORY_TYPE</td>
<td>CUDA_TYPES, 187</td>
</tr>
<tr>
<td>CU_PREFER_BINARY</td>
<td>CUDA_TYPES, 185</td>
</tr>
<tr>
<td>CU_PREFER_PTX</td>
<td>CUDA_TYPES, 185</td>
</tr>
<tr>
<td>CU_TARGET_COMPUTE_10</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CU_TARGET_COMPUTE_11</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CU_TARGET_COMPUTE_12</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CU_TARGET_COMPUTE_13</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CU_TARGET_COMPUTE_20</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CU_TARGET_COMPUTE_21</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CU_TARGET_COMPUTE_22</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CU_TARGET_COMPUTE_23</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CU_TARGET_COMPUTE_24</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CU_TARGET_COMPUTE_25</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CU_TARGET_COMPUTE_26</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CU_TARGET_COMPUTE_27</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CU_TARGET_COMPUTE_28</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CU_TARGET_COMPUTE_29</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CU_TARGET_COMPUTE_30</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CU_TARGET_COMPUTE_31</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CU_TARGET_COMPUTE_32</td>
<td>CUDA_TYPES, 186</td>
</tr>
<tr>
<td>CU_TR_ADDRESS_MODE_BORDER</td>
<td>CUDA_TYPES, 186</td>
</tr>
</tbody>
</table>
CU_TR_ADDRESS_MODE_CLAMP: CUDA_TYPES, 178
CU_LAUNCH_PARAM_BUFFER_POINTER: CUDA_TYPES, 174
CU_LAUNCH_PARAM_BUFFER_SIZE: CUDA_TYPES, 174
CU_LAUNCH_PARAM_END: CUDA_TYPES, 174
CU_PARAM_TR_DEFAULT: CUDA_TYPES, 174
CU_TR_FILTER_MODE_LINEAR: CUDA_TYPES, 184
CU_TR_FILTER_MODE_POINT: CUDA_TYPES, 184
CU_MEMHOSTALLOC_DEVICEMAP: CUDA_TYPES, 174
CU_MEMHOSTALLOC_PORTABLE: CUDA_TYPES, 174
CU_MEMHOSTALLOC_WRITECOMBINED: CUDA_TYPES, 174
CU_MEMHOSTREGISTER_DEVICEMAP: CUDA_TYPES, 174
CU_MEMHOSTREGISTER_PORTABLE: CUDA_TYPES, 174
CU_MEMHOSTREGISTER_WRITECOMBINED: CUDA_TYPES, 174
CU_TRSA_OVERRIDE_FORMAT: CUDA_TYPES, 174
CU_TRSF_NORMALIZED_COORDINATES: CUDA_TYPES, 175
CU_TRSF_READ_AS_INTEGER: CUDA_TYPES, 175
CU_TRSF_SRGB: CUDA_TYPES, 175
CUaddress_mode: CUDA_TYPES, 175
CUaddress_mode_enum: CUDA_TYPES, 178
CUarray: CUDA_TYPES, 175
cuArrayCreate: CUDA_MEM, 220
cuArrayDestroy: CUDA_MEM, 222
cuArrayGetDescriptor: CUDA_MEM, 222
CUcomputemode: CUDA_TYPES, 176
CUcomputemode_enum: CUDA_TYPES, 179
CUcontext: CUDA_TYPES, 176
CUctx_flags: CUDA_TYPES, 176
CUctx_flags_enum: CUDA_TYPES, 179
cuCtxAttach: CUDA_CTX, 200
cuCtxCreate: CUDA_CTX, 199
cuCtxCreateOnDevice: CUDA_CTX, 201
cuCtxDestroy: CUDA_CTX, 198
cuCtxDetach: CUDA_CTX, 199
cuCtxDisablePeerAccess: CUDA_PEER_ACCESS, 297
cuCtxEnablePeerAccess: CUDA_PEER_ACCESS, 297
cuCtxGetApiVersion: CUDA_CTX, 199
cuCtxGetCacheConfig: CUDA_CTX, 200
cuCtxGetCurrent: CUDA_CTX, 200
cuCtxGetDevice: CUDA_CTX, 201
cuCtxGetLimit: CUDA_CTX, 201
cuCtxPopCurrent: CUDA_CTX, 202
cuCtxPushCurrent: CUDA_CTX, 202
cuCtxSetCacheConfig: CUDA_CTX, 203
cuCtxSetCurrent: CUDA_CTX, 203
cuCtxSetLimit: CUDA_CTX, 204
cuCtxSynchronize: CUDA_CTX, 205
cuD3D10CtxCreate: CUDA_D3D10, 329
cuD3D10CtxCreateOnDevice: CUDA_D3D10, 329
CU_D3D9_DEVICE_LIST_ALL, 314
CU_D3D9_DEVICE_LIST_CURRENT_FRAME, 314
CU_D3D9_DEVICE_LIST_NEXT_FRAME, 314
CUDA_ERROR_ALREADY_ACQUIRED
CUDA_TYPES, 181
CUDA_ERROR_ALREADY_MAPPED
CUDA_TYPES, 181
CUDA_ERROR_ARRAY_IS_MAPPED
CUDA_TYPES, 181
CUDA_ERROR_CONTEXT_ALREADY_CURRENT
CUDA_TYPES, 180
CUDA_ERROR_CONTEXT_ALREADY_IN_USE
CUDA_TYPES, 181
CUDA_ERROR_CONTEXT_IS_DESTROYED
CUDA_TYPES, 182
CUDA_ERROR_DEINITIALIZED
CUDA_TYPES, 180
CUDA_ERROR_ECC_UNCORRECTABLE
CUDA_TYPES, 181
CUDA_ERROR_FILE_NOT_FOUND
CUDA_TYPES, 181
CUDA_ERROR_INVALID_CONTEXT
CUDA_TYPES, 180
CUDA_ERROR_INVALID_DEVICE
CUDA_TYPES, 180
CUDA_ERROR_INVALID_HANDLE
CUDA_TYPES, 181
CUDA_ERROR_INVALID_IMAGE
CUDA_TYPES, 180
CUDA_ERROR_INVALID_SOURCE
CUDA_TYPES, 181
CUDA_ERROR_INVALID_VALUE
CUDA_TYPES, 180
CUDA_ERROR_LAUNCH_FAILED
CUDA_TYPES, 181
CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING
CUDA_TYPES, 182
CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES
CUDA_TYPES, 181
CUDA_ERROR_LAUNCH_TIMEOUT
CUDA_TYPES, 181
CUDA_ERROR_MAP_FAILED
CUDA_TYPES, 181
CUDA_ERROR_NO_BINARY_FOR_GPU
CUDA_TYPES, 181
CUDA_ERROR_NO_DEVICE
CUDA_TYPES, 180
CUDA_ERROR_NOT_FOUND
CUDA_TYPES, 181
CUDA_ERROR_NOT_INITIALIZED
CUDA_TYPES, 180
CUDA_ERROR_NOT_MAPPED
CUDA_TYPES, 181
CUDA_ERROR_NOT_MAPPED_AS_ARRAY
CUDA_TYPES, 181
CUDA_ERROR_NOT_MAPPED_AS_POINTER
CUDA_TYPES, 181
CUDA_ERROR_NOT_READY
CUDA_TYPES, 181
CUDA_ERROR_OPERATING_SYSTEM
CUDA_TYPES, 181
CUDA_ERROR_OUT_OF_MEMORY
CUDA_TYPES, 180
CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED
CUDA_TYPES, 182
CUDA_ERROR_PEER_ACCESS_NOT_ENABLED
CUDA_TYPES, 182
CUDA_ERROR_PRIMARY_CONTEXT_ACTIVE
CUDA_TYPES, 182
CUDA_ERROR_PROFILER_ALREADY_STARTED
CUDA_TYPES, 180
CUDA_ERROR_PROFILER_ALREADY_STOPPED
CUDA_TYPES, 180
CUDA_ERROR_PROFILER_DISABLED
CUDA_TYPES, 180
CUDA_ERROR_PROFILER_NOT_INITIALIZED
CUDA_TYPES, 180
CUDA_ERROR_SHARED_OBJECT_INIT_FAILED
CUDA_TYPES, 181
CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND
CUDA_TYPES, 181
CUDA_ERROR_UNKNOWN
CUDA_TYPES, 182
CUDA_ERROR_UNMAP_FAILED
CUDA_TYPES, 181
CUDA_ERROR_UNSUPPORTED_LIMIT
CUDA_TYPES, 181
CUDA_SUCCESS
CUDA_TYPES, 181
CUDA_TYPES
CU_AD_FORMAT_FLOAT, 179
CU_AD_FORMAT_HALF, 179
CU_AD_FORMAT_SIGNED_INT16, 179
CU_AD_FORMAT_SIGNED_INT32, 179
CU_AD_FORMAT_SIGNED_INT8, 179
CU_AD_FORMAT_UNSIGNED_INT16, 179
CU_AD_FORMAT_UNSIGNED_INT32, 179
CU_AD_FORMAT_UNSIGNED_INT8, 179
CU_COMPUTEMODE_DEFAULT, 179
CU_COMPUTEMODE_EXCLUSIVE, 179
CU_COMPUTEMODE_EXCLUSIVE_PROCESS, 179
CU_COMPUTEMODE_PROHIBITED, 179
CU_CTX_BLOCKING_SYNC, 180
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU_CTX_LMEM_RESIZE_TO_MAX</td>
<td>180</td>
</tr>
<tr>
<td>CU_CTX_MAP_HOST</td>
<td>180</td>
</tr>
<tr>
<td>CU_CTX_SCHED_AUTO</td>
<td>180</td>
</tr>
<tr>
<td>CU_CTX_SCHED_BLOCKING_SYNC</td>
<td>180</td>
</tr>
<tr>
<td>CU_CTX_SCHED_SPIN</td>
<td>180</td>
</tr>
<tr>
<td>CU_CTX_SCHED_YIELD</td>
<td>180</td>
</tr>
<tr>
<td>CU_CUBEMAP_FACE_NEGATIVE_X</td>
<td>179</td>
</tr>
<tr>
<td>CU_CUBEMAP_FACE_NEGATIVE_Y</td>
<td>179</td>
</tr>
<tr>
<td>CU_CUBEMAP_FACE_NEGATIVE_Z</td>
<td>179</td>
</tr>
<tr>
<td>CU_CUBEMAP_FACE_POSITIVE_X</td>
<td>179</td>
</tr>
<tr>
<td>CU_CUBEMAP_FACE_POSITIVE_Y</td>
<td>179</td>
</tr>
<tr>
<td>CU_CUBEMAP_FACE_POSITIVE_Z</td>
<td>179</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_CLOCK_RATE</td>
<td>182</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_COMPUTE_MODE</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_ECC_ENABLED</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_GPU_OVERLAP</td>
<td>182</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_INTEGRATED</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_L2_CACHE_SIZE</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X</td>
<td>182</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y</td>
<td>182</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z</td>
<td>182</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X</td>
<td>182</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y</td>
<td>182</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z</td>
<td>182</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_PITCH</td>
<td>182</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK</td>
<td>182</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK</td>
<td>182</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK</td>
<td>182</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_HEIGHT</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_NUMSLICES</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_WIDTH</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_HEIGHT</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT</td>
<td>183</td>
</tr>
<tr>
<td>CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH</td>
<td>183</td>
</tr>
<tr>
<td>CUDEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE</td>
<td>183</td>
</tr>
<tr>
<td>CUDEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT</td>
<td>182</td>
</tr>
<tr>
<td>CUDEVICE_ATTRIBUTE_PCI_BUS_ID</td>
<td>183</td>
</tr>
<tr>
<td>CUDEVICE_ATTRIBUTE_PCI_DEVICE_ID</td>
<td>183</td>
</tr>
<tr>
<td>CUDEVICE_ATTRIBUTE_PCI_DOMAIN_ID</td>
<td>183</td>
</tr>
<tr>
<td>CUDEVICE_ATTRIBUTE_PIXELFORMAT</td>
<td>183</td>
</tr>
<tr>
<td>CUDEVICE_ATTRIBUTE_REGISTERS_PER_BLOCK</td>
<td>182</td>
</tr>
<tr>
<td>CUDEVICE_ATTRIBUTE_SHARED_MEMORY_PER_BLOCK</td>
<td>182</td>
</tr>
<tr>
<td>CUDEVICE_ATTRIBUTE_SURFACE_ALIGNMENT</td>
<td>183</td>
</tr>
<tr>
<td>CUDEVICE_ATTRIBUTE_TCC_DRIVER</td>
<td>183</td>
</tr>
<tr>
<td>CUDEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT</td>
<td>182</td>
</tr>
<tr>
<td>CUDEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY</td>
<td>182</td>
</tr>
<tr>
<td>CUDEVICE_ATTRIBUTE_UNIFIED_ADDRESSING</td>
<td>183</td>
</tr>
<tr>
<td>CUDEVICE_ATTRIBUTE_WARP_SIZE</td>
<td>182</td>
</tr>
<tr>
<td>CU_EVENT_BLOCKING_SYNC</td>
<td>184</td>
</tr>
<tr>
<td>CU_EVENT_DEFAULT</td>
<td>184</td>
</tr>
<tr>
<td>CU_EVENT_DISABLE_TIMING</td>
<td>184</td>
</tr>
</tbody>
</table>
CU_FUNC_ATTRIBUTECONST_SIZE_BYTES, 184
CU_FUNC_ATTRIBUTELOCAL_SIZE_BYTES, 184
CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK, 184
CU_FUNC_ATTRIBUTE_NUM_REGS, 184
CU_FUNC_ATTRIBUTE_PTX_VERSION, 184
CU_FUNC_ATTRIBUTE_SHARED_SIZEBYTES, 184
CU_FUNC_CACHEPREFER_L1, 184
CU_FUNC_CACHEPREFER_NONE, 184
CU_FUNC_CACHEPREFER_SHARED, 184
CU_JIT_ERROR_LOG_BUFFER, 185
CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES, 186
CU_JIT_FALLBACK_STRATEGY, 186
CU_JIT_INFO_LOG_BUFFER, 185
CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES, 185
CU_JIT_MAX_REGISTERS, 185
CU_JIT_OPTIMIZATION_LEVEL, 186
CU_JIT_TARGET, 186
CU_JIT_TARGET_FROM_CUCONTEXT, 186
CU_JIT_THREADS_PER_BLOCK, 185
CU_JIT_WALL_TIME, 185
CU_LIMIT_MALLOC_HEAP_SIZE, 186
CU_MEMORYTYPE_ARRAY, 187
CU_MEMORYTYPE_DEVICE, 187
CU_MEMORYTYPE_HOST, 187
CU_MEMORYTYPE_UNIFIED, 187
CU_POINTER_ATTRIBUTE_CONTEXT, 187
CU_POINTER_ATTRIBUTE_DEVICE, 187
CU_POINTER_ATTRIBUTE_HOST_POINTER, 187
CU_POINTER_ATTRIBUTE_MEMORY_TYPE, 187
CU_PREFER_BINARY, 185
CU_PREFER_PTX, 185
CU_TARGET_COMPUTE_10, 186
CU_TARGET_COMPUTE_11, 186
CU_TARGET_COMPUTE_12, 186
CU_TARGET_COMPUTE_13, 186
CU_TARGET_COMPUTE_20, 186
CU_TARGET_COMPUTE_21, 186
CU_TR_ADDRESS_MODE_BORDER, 178
CU_TR_ADDRESS_MODE_CLAMP, 178
CU_TR_ADDRESS_MODE_MIRROR, 178
CU_TR_ADDRESS_MODE_WRAP, 178
CU_TR_FILTER_MODE_LINEAR, 184
CU_TR_FILTER_MODE_POINT, 184
CUDA_ERROR_ALREADY_ACQUIRED, 181
CUDA_ERROR_ALREADY_MAPPED, 181
CUDA_ERROR_ARRAY_IS_MAPPED, 181
CUDA_ERROR_CONTEXT_ALREADY_CURRENT, 180
CUDA_ERROR_CONTEXT_ALREADY_IN_USE, 181
CUDA_ERROR_CONTEXT_IS_DESTROYED, 182
CUDA_ERROR_DEINITIALIZED, 180
CUDA_ERROR_ECC_UNCORRECTABLE, 181
CUDA_ERROR_FILE_NOT_FOUND, 181
CUDA_ERROR_INVALID_CONTEXT, 180
CUDA_ERROR_INVALID_DEVICE, 180
CUDA_ERROR_INVALID_HANDLE, 181
CUDA_ERROR_INVALID_IMAGE, 180
CUDA_ERROR_INVALID_SOURCE, 181
CUDA_ERROR_NOT_READY, 181
CUDA_ERROR_NOT_INITIALIZED, 180
CUDA_ERROR_NOT_MAPPED, 181
CUDA_ERROR_NOT_MAPPED_AS_ARRAY, 181
CUDA_ERROR_NOT_MAPPED_AS_POINTER, 181
CUDA_ERROR_NOT_MAPPED_AS_SHARED_MEMORY, 181
CUDA_ERROR_PRIMARY_CONTEXT_ACTIVE, 182
CUDA_ERROR_PROFILER_ALREADY_STARTED, 180
CUDA_ERROR_PROFILER_NOT_INITIALIZED, 180
CUDA_ERROR_SHARED_OBJECT_INIT_FAILED, 181
CUDA_ERROR_SHARED_OBJECT_LINKING_FAILED, 181
CUDA_ERROR_SHARED_OBJECT漕过CIG failures, 181
CUDA_ERROR_SHARED_OBJECT漕过CIG failures, 181
CUDA_ERROR_SHARED_OBJECT漕过CIG failures, 181
CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND, 181
CUDA_ERROR_UNKNOWN, 182
CUDA_ERROR_UNMAP_FAILED, 181
CUDA_ERROR_UNSUPPORTED_LIMIT, 181
CUDA_SUCCESS, 180
CUDA_ARRAY3D_2DARRAY
CUDA_TYPES, 175
CUDA_ARRAY3D_DESCRIPTOR
CUDA_TYPES, 176
CUDA_ARRAY3D_DESCRIPTOR_st, 353
  Depth, 353
  Format, 353
  Height, 353
  NumChannels, 353
  Width, 354
CUDA_ARRAY3D_LAYERED
CUDA_TYPES, 175
CUDA_ARRAY3D_SURFACE_LDST
CUDA_TYPES, 175
CUDA_ARRAY_DESCRIPTOR
CUDA_TYPES, 176
CUDA_ARRAY_DESCRIPTOR_st, 355
  Format, 355
  Height, 355
  NumChannels, 355
  Width, 355
CUDA_CTX
  cuCtxCreate, 198
  cuCtxDestroy, 199
  cuCtxGetApiVersion, 199
  cuCtxGetCacheConfig, 200
  cuCtxGetCurrent, 200
  cuCtxGetDevice, 201
  cuCtxGetLimit, 201
  cuCtxPopCurrent, 202
  cuCtxPushCurrent, 202
  cuCtxSetCacheConfig, 203
  cuCtxSetCurrent, 203
  cuCtxSetLimit, 204
  cuCtxSynchronize, 205
CUDA_CTX_DEPRECATED
  cuCtxAttach, 206
  cuCtxDetach, 206
CUDA_D3D10
  cuD3D10CtxCreate, 329
  cuD3D10CtxCreateOnDevice, 329
  cuD3D10DeviceList, 329
  cuD3D10DeviceList_enum, 329
  cuD3D10GetDevice, 330
  cuD3D10GetDevices, 330
  cuD3D10GetDirect3DDevice, 331
  cuGraphicsD3D10RegisterResource, 331
CUDA_D3D10_DEPRECATED
  cuD3D10map_flags, 335
  cuD3D10map_flags_enum, 335
  cuD3D10MapResources, 335
  cuD3D10register_flags, 335
  cuD3D10register_flags_enum, 335
  cuD3D10RegisterResource, 336
  cuD3D10ResourceGetMappedArray, 337
  cuD3D10ResourceGetMappedPitch, 338
  cuD3D10ResourceGetMappedPointer, 338
  cuD3D10ResourceGetMappedSize, 339
  cuD3D10ResourceGetSurfaceDimensions, 340
  cuD3D10ResourceSetMapFlags, 340
  cuD3D10UnmapResources, 341
  cuD3D10UnregisterResource, 342
CUDA_D3D11
  cuD3D11CtxCreate, 344
  cuD3D11CtxCreateOnDevice, 344
  cuD3D11DeviceList, 343
  cuD3D11DeviceList_enum, 344
  cuD3D11GetDevice, 345
  cuD3D11GetDevices, 345
  cuD3D11GetDirect3DDevice, 346
  cuGraphicsD3D11RegisterResource, 346
CUDA_D3D9
  cuD3D9CtxCreate, 314
  cuD3D9CtxCreateOnDevice, 314
  cuD3D9DeviceList, 314
  cuD3D9DeviceList_enum, 314
  cuD3D9GetDevice, 315
  cuD3D9GetDevices, 315
  cuD3D9GetDirect3DDevice, 316
  cuGraphicsD3D9RegisterResource, 316
CUDA_D3D9_DEPRECATED
  cuD3D9map_flags, 320
  cuD3D9map_flags_enum, 320
  cuD3D9MapResources, 320
  cuD3D9register_flags, 320
  cuD3D9register_flags_enum, 320
  cuD3D9RegisterResource, 321
  cuD3D9ResourceGetMappedArray, 322
  cuD3D9ResourceGetMappedPitch, 323
  cuD3D9ResourceGetMappedPointer, 324
  cuD3D9ResourceGetMappedSize, 324
  cuD3D9ResourceGetSurfaceDimensions, 325
  cuD3D9ResourceSetMapFlags, 326
  cuD3D9UnmapResources, 326
  cuD3D9UnregisterResource, 327
CUDA_DEVICE
  cuDeviceComputeCapability, 190
  cuDeviceGet, 191
  cuDeviceGetAttribute, 191
  cuDeviceGetCount, 193
  cuDeviceGetName, 194
dstHost, 356
dstMemoryType, 356
dstPitch, 356
dstXInBytes, 356
dstY, 357
Height, 357
srcArray, 357
srcDevice, 357
srcHost, 357
srcMemoryType, 357
srcPitch, 357
srcXInBytes, 357
srcY, 357
WidthInBytes, 357
CUDA_MEMCPY3D
CUDA_TYPES, 176
CUDA_MEMCPY3D_PEER
CUDA_TYPES, 176
CUDA_MEMCPY3D_PEER_st, 358
Depth, 358
dstContext, 358
dstDevice, 358
dstHeight, 359
dstHost, 359
dstLOD, 359
dstMemoryType, 359
dstPitch, 359
dstXInBytes, 359
dstY, 359
dstZ, 359
Height, 359
srcArray, 359
srcContext, 359
srcDevice, 359
srcHeight, 360
srcHost, 360
dstLOD, 360
dstMemoryType, 360
srcPitch, 360
srcXInBytes, 360
srcY, 360
srcZ, 360
WidthInBytes, 360
CUDA_MEMCPY3D_st, 361
Depth, 361
dstArray, 361
dstDevice, 361
dstHeight, 361
dstHost, 362
dstLOD, 362
dstMemoryType, 362
srcPitch, 362
dstXInBytes, 362
dstY, 362
dstZ, 362
Height, 362
reserved0, 362
reserved1, 362
srcHost, 363
dstLOD, 363
srcMemoryType, 363
srcPitch, 363
srcXInBytes, 363
srcY, 363
srcZ, 363
WidthInBytes, 363
CUDA_MODULE
cuModuleGetFunction, 208
cuModuleGetGlobal, 209
cuModuleGetSurfRef, 209
cuModuleGetTexRef, 210
cuModuleLoad, 210
cuModuleLoadData, 211
cuModuleLoadDataEx, 211
cuModuleLoadFatBinary, 213
cuModuleUnload, 213
CUDA_PEER_ACCESS
cuCtxDisablePeerAccess, 297
cuCtxEnablePeerAccess, 297
cuDeviceCanAccessPeer, 298
CUDA_STREAM
cuStreamCreate, 267
cuStreamDestroy, 267
cuStreamQuery, 268
cuStreamSynchronize, 268
cuStreamWaitEvent, 269
CUDA_SURFREF
cuSurfRefGetArray, 295
cuSurfRefSetArray, 295
CUDA_TEXREF
cuTexRefGetAddress, 286
cuTexRefGetAddressMode, 286
cuTexRefGetArray, 286
cuTexRefGetFilterMode, 287
cuTexRefGetFlags, 287
cuTexRefGetFormat, 288
Generated for NVIDIA CUDA Library by Doxygen
<table>
<thead>
<tr>
<th>Function/Enum</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuTexRefCreate</td>
<td>293</td>
</tr>
<tr>
<td>cuTexRefDestroy</td>
<td>293</td>
</tr>
<tr>
<td>CUDA_TYPES</td>
<td></td>
</tr>
<tr>
<td>CU_LAUNCH_PARAM_BUFFER_POINTER</td>
<td>174</td>
</tr>
<tr>
<td>CU_LAUNCH_PARAM_BUFFER_SIZE</td>
<td>174</td>
</tr>
<tr>
<td>CU_LAUNCH_PARAM_END</td>
<td>174</td>
</tr>
<tr>
<td>CU_MEMHOSTALLOC_DEVICEMAP</td>
<td>174</td>
</tr>
<tr>
<td>CU_MEMHOSTALLOC_PORTABLE</td>
<td>174</td>
</tr>
<tr>
<td>CU_MEMHOSTALLOC_WRITECOMBINED</td>
<td>174</td>
</tr>
<tr>
<td>CU_MEMHOSTREGISTER_DEVICEMAP</td>
<td>174</td>
</tr>
<tr>
<td>CU_MEMHOSTREGISTER_PORTABLE</td>
<td>174</td>
</tr>
<tr>
<td>CU_PARAM_TR_DEFAULT</td>
<td>174</td>
</tr>
<tr>
<td>CU_TRSA_OVERRIDE_FORMAT</td>
<td>174</td>
</tr>
<tr>
<td>CU_TRSF_NORMALIZED_COORDINATES</td>
<td>175</td>
</tr>
<tr>
<td>CU_TRSF_READ_AS_INTEGER</td>
<td>175</td>
</tr>
<tr>
<td>CU_TRSF_SRGB</td>
<td>175</td>
</tr>
<tr>
<td>CUAddress_mode</td>
<td>175</td>
</tr>
<tr>
<td>CUaddress_mode_enum</td>
<td>178</td>
</tr>
<tr>
<td>CUAarray</td>
<td>175</td>
</tr>
<tr>
<td>CUAarray_cubemap_face</td>
<td>175</td>
</tr>
<tr>
<td>CUAarray_cubemap_face_enum</td>
<td>178</td>
</tr>
<tr>
<td>CUAarray_format</td>
<td>175</td>
</tr>
<tr>
<td>CUAarray_format_enum</td>
<td>179</td>
</tr>
<tr>
<td>CUcomputemode</td>
<td>176</td>
</tr>
<tr>
<td>CUcomputemode_enum</td>
<td>176</td>
</tr>
<tr>
<td>CUcontext</td>
<td>176</td>
</tr>
<tr>
<td>CUcontext_flags</td>
<td>176</td>
</tr>
<tr>
<td>CUcontext Flags</td>
<td>176</td>
</tr>
<tr>
<td>CUCtx_flags</td>
<td>176</td>
</tr>
<tr>
<td>CUCtx_flags_enum</td>
<td>179</td>
</tr>
<tr>
<td>CUDA_ARRAY3D_2DARRAY</td>
<td>175</td>
</tr>
<tr>
<td>CUDA_ARRAY3D_DESCRIPTOR</td>
<td>176</td>
</tr>
<tr>
<td>CUDA_ARRAY3D_LAYERED</td>
<td>175</td>
</tr>
<tr>
<td>CUDA_ARRAY3D_SURFACE_LDST</td>
<td>175</td>
</tr>
<tr>
<td>CUDA_ARRAY_DESCRIPTOR</td>
<td>176</td>
</tr>
<tr>
<td>CUDA_MEMCPY2D</td>
<td>176</td>
</tr>
<tr>
<td>CUDA_MEMCPY3D</td>
<td>176</td>
</tr>
<tr>
<td>CUDA_MEMCPY3D_PEER</td>
<td>176</td>
</tr>
<tr>
<td>CUDA_VERSION</td>
<td>175</td>
</tr>
<tr>
<td>cudaError_enum</td>
<td>180</td>
</tr>
<tr>
<td>Cudevice</td>
<td>176</td>
</tr>
<tr>
<td>Cudevice_attribute</td>
<td>176</td>
</tr>
<tr>
<td>Cudevice_attribute_enum</td>
<td>182</td>
</tr>
<tr>
<td>Cudevicecctor</td>
<td>176</td>
</tr>
<tr>
<td>Cudevprop</td>
<td>176</td>
</tr>
<tr>
<td>Cuevent</td>
<td>177</td>
</tr>
<tr>
<td>Cuevent_flags</td>
<td>177</td>
</tr>
<tr>
<td>Cuevent_flags_enum</td>
<td>183</td>
</tr>
<tr>
<td>Cufilter_mode</td>
<td>177</td>
</tr>
<tr>
<td>Cufilter_mode_enum</td>
<td>184</td>
</tr>
<tr>
<td>Cufunc_cache</td>
<td>177</td>
</tr>
<tr>
<td>Cufunc_cache_enum</td>
<td>184</td>
</tr>
<tr>
<td>Cufunction</td>
<td>177</td>
</tr>
<tr>
<td>Cufunction_attribute</td>
<td>177</td>
</tr>
<tr>
<td>Cufunction_attribute_enum</td>
<td>184</td>
</tr>
<tr>
<td>CUgraphicsMapResourceFlags</td>
<td>177</td>
</tr>
<tr>
<td>CUgraphicsMapResourceFlags_enum</td>
<td>185</td>
</tr>
<tr>
<td>CUgraphicsRegisterFlags</td>
<td>177</td>
</tr>
<tr>
<td>CUgraphicsRegisterFlags_enum</td>
<td>185</td>
</tr>
<tr>
<td>CUjit_fallback</td>
<td>177</td>
</tr>
<tr>
<td>CUjit_fallback_enum</td>
<td>185</td>
</tr>
<tr>
<td>CUjit_option</td>
<td>177</td>
</tr>
<tr>
<td>CUjit_option_enum</td>
<td>185</td>
</tr>
<tr>
<td>CUjit_target</td>
<td>177</td>
</tr>
<tr>
<td>CUjit_target_enum</td>
<td>186</td>
</tr>
<tr>
<td>CUlimit</td>
<td>178</td>
</tr>
<tr>
<td>CUlimit_enum</td>
<td>186</td>
</tr>
<tr>
<td>CUmemorytype</td>
<td>178</td>
</tr>
<tr>
<td>CUmemorytype_enum</td>
<td>186</td>
</tr>
<tr>
<td>CUmodule</td>
<td>178</td>
</tr>
<tr>
<td>CUpointer_attribute</td>
<td>178</td>
</tr>
<tr>
<td>CUpointer_attribute_enum</td>
<td>187</td>
</tr>
<tr>
<td>CUresult</td>
<td>178</td>
</tr>
<tr>
<td>CUSTream</td>
<td>178</td>
</tr>
<tr>
<td>CUSurffref</td>
<td>178</td>
</tr>
<tr>
<td>CUtexref</td>
<td>178</td>
</tr>
<tr>
<td>CUDA_UNIFIED</td>
<td></td>
</tr>
<tr>
<td>cuPointerGetAttribute</td>
<td>264</td>
</tr>
<tr>
<td>CUDA_VDPAU</td>
<td></td>
</tr>
<tr>
<td>cuGraphicsVDPAURegisterOutputSurface</td>
<td>349</td>
</tr>
<tr>
<td>cuGraphicsVDPAURegisterVideoSurface</td>
<td>350</td>
</tr>
<tr>
<td>cuVDPAUCtxCreate</td>
<td>351</td>
</tr>
<tr>
<td>cuVDPAUGetDevice</td>
<td>351</td>
</tr>
<tr>
<td>CUDA_VERSION</td>
<td></td>
</tr>
<tr>
<td>CUDA_TYPES</td>
<td>175</td>
</tr>
<tr>
<td>cuDriverGetVersion</td>
<td>189</td>
</tr>
<tr>
<td>cudaAddressModeBorder</td>
<td></td>
</tr>
<tr>
<td>CUDART_TYPES</td>
<td>166</td>
</tr>
<tr>
<td>cudaAddressModeClamp</td>
<td></td>
</tr>
<tr>
<td>CUDART_TYPES</td>
<td>166</td>
</tr>
<tr>
<td>cudaAddressModeMirror</td>
<td></td>
</tr>
<tr>
<td>CUDART_TYPES</td>
<td>166</td>
</tr>
<tr>
<td>cudaAddressModeWrap</td>
<td></td>
</tr>
<tr>
<td>CUDART_TYPES</td>
<td>166</td>
</tr>
<tr>
<td>cudaArrayDefault</td>
<td></td>
</tr>
<tr>
<td>CUDART_TYPES</td>
<td>157</td>
</tr>
<tr>
<td>cudaArrayLayered</td>
<td></td>
</tr>
<tr>
<td>CUDART_TYPES</td>
<td>157</td>
</tr>
<tr>
<td>cudaArraySurfaceLoadStore</td>
<td></td>
</tr>
<tr>
<td>CUDART_TYPES</td>
<td>157</td>
</tr>
<tr>
<td>cudaBindSurfaceToArray</td>
<td></td>
</tr>
<tr>
<td>CUDART_HIGHLEVEL</td>
<td>116</td>
</tr>
<tr>
<td>CUDA_SURFACE</td>
<td>112</td>
</tr>
<tr>
<td>cudaBindTexture</td>
<td></td>
</tr>
<tr>
<td>CUDART_HIGHLEVEL</td>
<td>117</td>
</tr>
<tr>
<td>CUDART_SURFACE</td>
<td>112</td>
</tr>
<tr>
<td>cudaBindTexture2D</td>
<td></td>
</tr>
<tr>
<td>CUDART_HIGHLEVEL</td>
<td>118</td>
</tr>
<tr>
<td>CUDART_TEXTURE</td>
<td>107</td>
</tr>
<tr>
<td>Function</td>
<td>Category</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>cudaDevicePropDontCare</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaDeviceReset</td>
<td>CUDA_DEVICE</td>
</tr>
<tr>
<td>cudaDeviceScheduleAuto</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaDeviceScheduleBlockingSync</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaDeviceScheduleSpin</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaDeviceScheduleYield</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaDeviceSetCacheConfig</td>
<td>CUDA_DEVICE</td>
</tr>
<tr>
<td>cudaDeviceSetLimit</td>
<td>CUDA_DEVICE</td>
</tr>
<tr>
<td>cudaDeviceSynchronize</td>
<td>CUDA_DEVICE</td>
</tr>
<tr>
<td>cudaDriverGetVersion</td>
<td>CUDA__VERSION</td>
</tr>
<tr>
<td>cudaError</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaError_enum</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaError_t</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorAddressOfConstant</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorApiFailureBase</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorCudartUnloading</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorDeviceAlreadyInUse</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorDevicesUnavailable</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorDuplicateSurfaceName</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorDuplicateTextureName</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorDuplicateVariableName</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorECCUncorrectable</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorIncompatibleDriverContext</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInitializationError</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInitializationError</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInsufficientDriver</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidChannelDescriptor</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidConfiguration</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidDevice</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidDeviceFunction</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidDevicePointer</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidFilterSetting</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidHostPointer</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidKernelImage</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidMemcpyDirection</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidMemcpypDirection</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidNormSetting</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidPitchValue</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidResourceHandle</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidSurface</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidSymbol</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidTexture</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidTextureBinding</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorInvalidValue</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorLaunchFailure</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorLaunchOutOfResources</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorLaunchTimeout</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorMapBufferObjectFailed</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorMemoryAllocation</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorMemoryValueTooLarge</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorMissingConfiguration</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorMixedDeviceExecution</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorNoDevice</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorNoKernelImageForDevice</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorNotReady</td>
<td>CUDA_TYPES</td>
</tr>
<tr>
<td>cudaErrorNotYetImplemented</td>
<td>CUDA_TYPES</td>
</tr>
</tbody>
</table>

Generated for NVIDIA CUDA Library by Doxygen
cudaErrorPeerAccessAlreadyEnabled  
  CUDART_TYPES, 163
cudaErrorPeerAccessNotEnabled  
  CUDART_TYPES, 163
cudaErrorPriorLaunchFailure  
  CUDART_TYPES, 160
cudaErrorProfilerAlreadyStarted  
  CUDART_TYPES, 163
cudaErrorProfilerAlreadyStopped  
  CUDART_TYPES, 163
cudaErrorProfilerDisabled  
  CUDART_TYPES, 163
cudaErrorProfilerNotInitialized  
  CUDART_TYPES, 163
cudaErrorSetOnActiveProcess  
  CUDART_TYPES, 162
cudaErrorSharedObjectInitFailed  
  CUDART_TYPES, 162
cudaErrorSharedObjectSymbolNotFound  
  CUDART_TYPES, 162
cudaErrorStartupFailure  
  CUDART_TYPES, 163
cudaErrorSynchronizationError  
  CUDART_TYPES, 161
cudaErrorTextureFetchFailed  
  CUDART_TYPES, 161
cudaErrorTextureNotBound  
  CUDART_TYPES, 161
cudaErrorUnknown  
  CUDART_TYPES, 162
cudaErrorUnmapBufferObjectFailed  
  CUDART_TYPES, 161
cudaErrorUnsupportedLimit  
  CUDART_TYPES, 162
cudaEvent_t  
  CUDART_TYPES, 159
cudaEventBlockingSync  
  CUDART_TYPES, 158
cudaEventCreate  
  CUDART_EVENT, 33
  CUDART_HIGHLEVEL, 121
cudaEventCreateWithFlags  
  CUDART_EVENT, 33
cudaEventDefault  
  CUDART_TYPES, 158
cudaEventDestroy  
  CUDART_EVENT, 34
cudaEventDisableTiming  
  CUDART_TYPES, 158
cudaEventElapsedTime  
  CUDART_EVENT, 33
cudaEventQuery  
  CUDART_EVENT, 35
cudaEventRecord  
  CUDART_EVENT, 35
cudaEventSynchronize  
  CUDART_EVENT, 36
cudaExtent  
  depth, 370
  height, 370
  width, 370
cudaFilterModeLinear  
  CUDART_TYPES, 166
cudaFilterModePoint  
  CUDART_TYPES, 166
cudaFormatModeAuto  
  CUDART_TYPES, 165
cudaFormatModeForced  
  CUDART_TYPES, 165
cudaFree  
  CUDART_MEMORY, 44
cudaFreeArray  
  CUDART_MEMORY, 45
cudaFreeHost  
  CUDART_MEMORY, 45
cudaFuncAttributes  
  binaryVersion, 371
  constSizeBytes, 371
  localSizeBytes, 371
  maxThreadsPerBlock, 371
  numRegs, 371
  ptxVersion, 371
  sharedSizeBytes, 371
cudaFuncCache  
  CUDART_TYPES, 163
cudaFuncCachePreferL1  
  CUDART_TYPES, 163
cudaFuncCachePreferNone  
  CUDART_TYPES, 163
cudaFuncCachePreferShared  
  CUDART_TYPES, 163
cudaFuncGetAttributes  
  CUDART_EXECUTION, 38
  CUDART_HIGHLEVEL, 122
cudaFuncSetCacheConfig  
  CUDART_EXECUTION, 38
  CUDART_HIGHLEVEL, 123
cudaGetChannelDesc  
  CUDART_TEXTURE, 110
cudaGetDevice  
  CUDART_DEVICE, 17
cudaGetDeviceCount  
  CUDART_DEVICE, 17
cudaGetDeviceProperties  
  CUDART_DEVICE, 18
cudaGetLastError  
  CUDART_ERROR, 28
<table>
<thead>
<tr>
<th>Function</th>
<th>Category</th>
<th>Line Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDART_ERROR</td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>cudaGetSurfaceReference</td>
<td>CUDART_SURFACE</td>
<td>112</td>
</tr>
<tr>
<td>CUDART_HIGHLEVEL</td>
<td></td>
<td>123</td>
</tr>
<tr>
<td>CUDART_MEMORY</td>
<td></td>
<td>45</td>
</tr>
<tr>
<td>cudaGetSymbolAddress</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDART_HIGHLEVEL</td>
<td></td>
<td>124</td>
</tr>
<tr>
<td>CUDART_MEMORY</td>
<td></td>
<td>46</td>
</tr>
<tr>
<td>cudaGetSymbolSize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDART_HIGHLEVEL</td>
<td></td>
<td>124</td>
</tr>
<tr>
<td>CUDART_TEXTURE</td>
<td></td>
<td>110</td>
</tr>
<tr>
<td>cudaGetTextureReference</td>
<td></td>
<td>111</td>
</tr>
<tr>
<td>cudaGLMapBufferObject</td>
<td>CUDART_OPENGL</td>
<td>149</td>
</tr>
<tr>
<td>cudaGLMapBufferObjectAsync</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGLMapFlags</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGLMapFlagsNone</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGLMapFlagsReadOnly</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGLMapFlagsWriteDiscard</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGLRegisterBufferObject</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGLSetBufferObjectMapFlags</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGLUnmapBufferObject</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGLUnmapBufferObjectAsync</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGLUnregisterBufferObject</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGraphicsCubeFace</td>
<td></td>
<td>163</td>
</tr>
<tr>
<td>cudaGraphicsCubeFaceNegativeX</td>
<td></td>
<td>164</td>
</tr>
<tr>
<td>cudaGraphicsCubeFaceNegativeY</td>
<td></td>
<td>164</td>
</tr>
<tr>
<td>cudaGraphicsCubeFaceNegativeZ</td>
<td></td>
<td>164</td>
</tr>
<tr>
<td>cudaGraphicsCubeFacePositiveX</td>
<td></td>
<td>164</td>
</tr>
<tr>
<td>cudaGraphicsCubeFacePositiveY</td>
<td></td>
<td>164</td>
</tr>
<tr>
<td>cudaGraphicsCubeFacePositiveZ</td>
<td></td>
<td>164</td>
</tr>
<tr>
<td>cudaGraphicsD3D9RegisterResource</td>
<td></td>
<td>87</td>
</tr>
<tr>
<td>cudaGraphicsGLRegisterBuffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGraphicsGLRegisterImage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGraphicsMapFlags</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGraphicsMapFlagsNone</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGraphicsMapFlagsReadOnly</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGraphicsMapFlagsWriteDiscard</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGraphicsResource_t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGraphicsResourceGetMappedPointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGraphicsResourceSetMapFlags</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGraphicsResourceUnregisterResource</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaGraphicsVDPAURegisterOutputSurface</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>cudaGraphicsVDPAURegisterVideoSurface</td>
<td></td>
<td>101</td>
</tr>
<tr>
<td>cudaHostAlloc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaHostAllocDefault</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaHostAllocMapped</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaHostAllocPortable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaHostAllocWriteCombined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaHostGetDevicePointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaHostMemory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaHostMemoryDefault</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaHostMemoryMapped</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaHostMemoryPortable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaHostMemoryWriteCombined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaHostMemoryGetDevicePointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generated for NVIDIA CUDA Library by Doxygen</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
cudaHostGetFlags
    CUDART_MEMORY, 48
cudaHostRegister
    CUDART_MEMORY, 48
cudaHostRegisterDefault
    CUDART_TYPES, 158
cudaHostRegisterMapped
    CUDART_TYPES, 158
cudaHostRegisterPortable
    CUDART_TYPES, 158
cudaHostUnregister
    CUDART_MEMORY, 49
cudaLaunch
    CUDART_EXECUTION, 39
    CUDART_HIGHLEVEL, 125
cudaLimit
    CUDART_TYPES, 164
cudaLimitMallocHeapSize
    CUDART_TYPES, 164
cudaLimitPrintfFifoSize
    CUDART_TYPES, 164
cudaLimitStackSize
    CUDART_TYPES, 164
cudaMalloc
    CUDART_MEMORY, 49
cudaMalloc3D
    CUDART_MEMORY, 50
cudaMalloc3DArray
    CUDART_MEMORY, 50
cudaMallocArray
    CUDART_MEMORY, 52
cudaMallocHost
    CUDART_HIGHLEVEL, 125
    CUDART_MEMORY, 52
cudaMallocPitch
    CUDART_MEMORY, 53
cudaMemcpy
    CUDART_MEMORY, 54
cudaMemcpy2D
    CUDART_MEMORY, 54
cudaMemcpy2DArrayToArray
    CUDART_MEMORY, 55
cudaMemcpy2DAsync
    CUDART_MEMORY, 56
cudaMemcpy2DFromArray
    CUDART_MEMORY, 56
cudaMemcpy2DFromArrayAsync
    CUDART_MEMORY, 57
cudaMemcpy2DToArray
    CUDART_MEMORY, 58
cudaMemcpy2DToArrayAsync
    CUDART_MEMORY, 59
cudaMemcpy3D
    CUDART_MEMORY, 60
cudaMemcpy3DAsync
    CUDART_MEMORY, 61
cudaMemcpy3DParms
    dstArray, 373
dstPos, 373
dstPtr, 373
extent, 373
kind, 373
srcArray, 373
srcPos, 373
srcPtr, 373
cudaMemcpy3DPeer
    CUDART_MEMORY, 62
cudaMemcpy3DPeerAsync
    CUDART_MEMORY, 63
cudaMemcpy3DPeerParms
    dstArray, 375
dstDevice, 375
dstPos, 375
dstPtr, 375
extent, 375
srcArray, 375
srcDevice, 375
srcPos, 375
srcPtr, 376
cudaMemcpyArrayToArray
    CUDART_MEMORY, 63
cudaMemcpyAsync
    CUDART_MEMORY, 64
cudaMemcpyDefault
    CUDART_TYPES, 165
cudaMemcpyDeviceToDevice
    CUDART_TYPES, 165
cudaMemcpyDeviceToHost
    CUDART_TYPES, 165
cudaMemcpyFromArray
    CUDART_MEMORY, 65
cudaMemcpyFromArrayAsync
    CUDART_MEMORY, 65
cudaMemcpyFromSymbol
    CUDART_MEMORY, 65
cudaMemcpyFromSymbolAsync
    CUDART_MEMORY, 66
cudaMemcpyHostToDevice
    CUDART_TYPES, 165
cudaMemcpyHostToHost
    CUDART_TYPES, 165
cudaMemcpyKind
    CUDART_TYPES, 164
cudaMemcpyPeer
    CUDART_MEMORY, 67
cudaMemcpyPeerAsync
    CUDART_MEMORY, 68
cudaMemcpyToArray
    CUDART_MEMORY, 69
CUDART_MEMORY, 68
cudaMemcpyToArrayAsync
CUDART_MEMORY, 69
cudaMemcpyToSymbol
CUDART_MEMORY, 70
cudaMemcpyToSymbolAsync
CUDART_MEMORY, 70
cudaMemGetInfo
CUDART_MEMORY, 71
cudaMemoryType
CUDART_TYPES, 165
cudaMemoryTypeDevice
CUDART_TYPES, 165
cudaMemoryTypeHost
CUDART_TYPES, 165
cudaMemset
CUDART_MEMORY, 71
cudaMemset2D
CUDART_MEMORY, 72
cudaMemset2DAsync
CUDART_MEMORY, 72
cudaMemset3D
CUDART_MEMORY, 73
cudaMemset3DAsync
CUDART_MEMORY, 74
cudaMemsetAsync
CUDART_MEMORY, 74
cudaOutputMode
CUDART_TYPES, 165
cudaOutputMode_t
CUDART_TYPES, 159
cudaPeekAtLastError
CUDART_ERROR, 29
cudaPeerAccessDefault
CUDART_TYPES, 159
cudaPitchedPtr, 377
pitch, 377
ptr, 377
xsize, 377
ysize, 377
cudaPointerAttributes, 378
device, 378
devicePointer, 378
hostPointer, 378
memoryType, 378
cudaPointerGetAttributes
CUDART_UNIFIED, 78
cudaPos, 379
x, 379
y, 379
z, 379
cudaReadModeElementType
CUDART_TYPES, 166
cudaReadModeNormalizedFloat
CUDART_TYPES, 166
CUDART
CUDART_VERSION, 12
CUDART_D3D10
cudaD3D10DeviceListAll, 90
cudaD3D10DeviceListCurrentFrame, 90
cudaD3D10DeviceListNextFrame, 90
CUDART_D3D10_DEPRECATED
cudaD3D10MapFlagsNone, 140
cudaD3D10MapFlagsReadOnly, 140
cudaD3D10MapFlagsWriteDiscard, 140
cudaD3D10RegisterFlagsArray, 140
cudaD3D10RegisterFlagsNone, 140
CUDART_D3D11
cudaD3D11DeviceListAll, 95
cudaD3D11DeviceListCurrentFrame, 95
cudaD3D11DeviceListNextFrame, 95
CUDART_D3D9
cudaD3D9DeviceListAll, 85
cudaD3D9DeviceListCurrentFrame, 85
cudaD3D9DeviceListNextFrame, 85
CUDART_D3D9_DEPRECATED
cudaD3D9MapFlagsNone, 131
cudaD3D9MapFlagsReadOnly, 131
cudaD3D9MapFlagsWriteDiscard, 131
cudaD3D9RegisterFlagsArray, 131
cudaD3D9RegisterFlagsNone, 131
CUDART_OPENGL_DEPRECATED
cudaGLMapFlagsNone, 148
cudaGLMapFlagsReadOnly, 148
cudaGLMapFlagsWriteDiscard, 148
CUDART_TYPES
cudaAddressModeBorder, 166
cudaAddressModeClamp, 166
cudaAddressModeMirror, 166
cudaAddressModeWrap, 166
cudaBoundaryModeClamp, 165
cudaBoundaryModeTrap, 165
cudaBoundaryModeZero, 165
cudaChannelFormatKindFloat, 159
cudaChannelFormatKindNone, 159
cudaChannelFormatKindSigned, 159
cudaChannelFormatKindUnsigned, 159
cudaComputeModeDefault, 160
cudaComputeModeExclusive, 160
cudaComputeModeExclusiveProcess, 160
cudaComputeModeProhibited, 160
cudaErrorAddressOfConstant, 161
cudaErrorApiFailureBase, 163
cudaErrorCudartUnloading, 162
cudaErrorDeviceAlreadyInUse, 163
cudaErrorDevicesUnavailable, 163
cudaErrorDuplicateSurfaceName, 162
cudaErrorDuplicateTextureName, 162

Generated for NVIDIA CUDA Library by Doxygen
<table>
<thead>
<tr>
<th>Category</th>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CUDART_ERROR cudaGetErrorString, cudaGetLastError, cudaPeekAtLastError</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>CUDART_EXECUTION cudaConfigureCall, cudaFuncGetAttributes, cudaFuncSetCacheConfig, cudaLaunch, cudaSetDoubleForDevice, cudaSetDoubleForHost, cudaSetupArgument</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>CUDART_MEMORY cudaFree, cudaFreeArray, cudaFreeHost, cudaGetSymbolAddress, cudaGetSymbolSize, cudaHostAlloc, cudaHostGetDevicePointer, cudaHostGetFlags, cudaHostRegister, cudaHostUnregister, cudaMalloc, cudaMalloc3D, cudaMalloc3DArray, cudaMallocArray, cudaMallocHost, cudaMemcpy, cudaMemcpy2D, cudaMemcpy2DAsync, cudaMemcpy2DFromArray, cudaMemcpy2DFromArrayAsync, cudaMemcpy2DToArray, cudaMemcpy2DToArrayAsync</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>CUDART_EVENT cudaEventCreate, cudaEventCreateWithFlags, cudaEventDestroy, cudaEventElapsedTime, cudaEventQuery, cudaEventRecord, cudaEventSynchronize</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>CUDART_HIGHLEVEL cudaBindSurfaceToArray, cudaBindTexture, cudaBindTexture2D, cudaBindTextureToArray, cudaCreateChannelDesc, cudaEventCreate, cudaFuncGetAttributes, cudaFuncSetCacheConfig, cudaGetSymbolAddress, cudaGetSymbolMap, cudaGetSymbolSize, cudaGetTextureAlignmentOffset, cudaLaunch, cudaMallocHost, cudaSetupArgument, cudaUnbindTexture</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>CUDART_MEMORY cudaFree, cudaFreeArray, cudaFreeHost, cudaGetSymbolAddress, cudaGetSymbolSize, cudaHostAlloc, cudaHostGetDevicePointer, cudaHostGetFlags, cudaHostRegister, cudaHostUnregister, cudaMalloc, cudaMalloc3D, cudaMalloc3DArray, cudaMallocArray, cudaMallocHost, cudaMemcpy, cudaMemcpy2D, cudaMemcpy2DAsync, cudaMemcpy2DFromArray, cudaMemcpy2DFromArrayAsync, cudaMemcpy2DToArray, cudaMemcpy2DToArrayAsync</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>CUDART_EVENT cudaEventCreate, cudaEventCreateWithFlags, cudaEventDestroy, cudaEventElapsedTime, cudaEventQuery, cudaEventRecord, cudaEventSynchronize</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>CUDART_MEMORY cudaFree, cudaFreeArray, cudaFreeHost, cudaGetSymbolAddress, cudaGetSymbolSize, cudaHostAlloc, cudaHostGetDevicePointer, cudaHostGetFlags, cudaHostRegister, cudaHostUnregister, cudaMalloc, cudaMalloc3D, cudaMalloc3DArray, cudaMallocArray, cudaMallocHost, cudaMemcpy, cudaMemcpy2D, cudaMemcpy2DAsync, cudaMemcpy2DFromArray, cudaMemcpy2DFromArrayAsync, cudaMemcpy2DToArray, cudaMemcpy2DToArrayAsync</td>
<td>44</td>
</tr>
</tbody>
</table>

Generated for NVIDIA CUDA Library by Doxygen
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>cudaMemcpy3D</td>
<td>60</td>
</tr>
<tr>
<td>cudaMemcpy3DAsync</td>
<td>61</td>
</tr>
<tr>
<td>cudaMemcpy3DPeer</td>
<td>62</td>
</tr>
<tr>
<td>cudaMemcpy3DPeerAsync</td>
<td>63</td>
</tr>
<tr>
<td>cudaMemcpyArrayToArray</td>
<td>63</td>
</tr>
<tr>
<td>cudaMemcpyAsync</td>
<td>64</td>
</tr>
<tr>
<td>cudaMemcpyFromArray</td>
<td>65</td>
</tr>
<tr>
<td>cudaMemcpyFromArrayAsync</td>
<td>65</td>
</tr>
<tr>
<td>cudaMemcpyFromSymbol</td>
<td>66</td>
</tr>
<tr>
<td>cudaMemcpyFromSymbolAsync</td>
<td>67</td>
</tr>
<tr>
<td>cudaMemcpyPeer</td>
<td>67</td>
</tr>
<tr>
<td>cudaMemcpyPeerAsync</td>
<td>68</td>
</tr>
<tr>
<td>cudaMemcpyToSymbol</td>
<td>69</td>
</tr>
<tr>
<td>cudaMemcpyToSymbolAsync</td>
<td>70</td>
</tr>
<tr>
<td>cudaMemcpyToSymbolAsync</td>
<td>71</td>
</tr>
<tr>
<td>cudaMemcpyToSymbolAsync</td>
<td>72</td>
</tr>
<tr>
<td>cudaMemcpyToSymbolAsync</td>
<td>73</td>
</tr>
<tr>
<td>cudaMemcpyToSymbolAsync</td>
<td>74</td>
</tr>
<tr>
<td>make_cudExtent</td>
<td>75</td>
</tr>
<tr>
<td>make_cudapiPitchedPtr</td>
<td>75</td>
</tr>
<tr>
<td>make_cudapiPos</td>
<td>76</td>
</tr>
<tr>
<td>CUDART_OPENGL</td>
<td></td>
</tr>
<tr>
<td>cudaGraphicsGLRegisterBuffer</td>
<td>82</td>
</tr>
<tr>
<td>cudaGraphicsGLRegisterImage</td>
<td>83</td>
</tr>
<tr>
<td>cudaWGLGetDevice</td>
<td>83</td>
</tr>
<tr>
<td>CUDART_OPENGL_DEPRECATED</td>
<td></td>
</tr>
<tr>
<td>cudaGLMapBufferObject</td>
<td>149</td>
</tr>
<tr>
<td>cudaGLMapBufferObjectAsync</td>
<td>149</td>
</tr>
<tr>
<td>cudaGLMapFlags</td>
<td>148</td>
</tr>
<tr>
<td>cudaGLRegisterBufferObject</td>
<td>150</td>
</tr>
<tr>
<td>cudaGLSetBufferObjectMapFlags</td>
<td>150</td>
</tr>
<tr>
<td>cudaGLUnmapBufferObject</td>
<td>151</td>
</tr>
<tr>
<td>cudaGLUnmapBufferObjectAsync</td>
<td>151</td>
</tr>
<tr>
<td>cudaGLUnregisterBufferObject</td>
<td>152</td>
</tr>
<tr>
<td>CUDART_PEER</td>
<td></td>
</tr>
<tr>
<td>cudaDeviceCanAccessPeer</td>
<td>80</td>
</tr>
<tr>
<td>cudaDeviceDisablePeerAccess</td>
<td>80</td>
</tr>
<tr>
<td>cudaDeviceEnablePeerAccess</td>
<td>81</td>
</tr>
<tr>
<td>CUDART_STREAM</td>
<td></td>
</tr>
<tr>
<td>cudaStreamCreate</td>
<td>30</td>
</tr>
<tr>
<td>cudaStreamDestroy</td>
<td>30</td>
</tr>
<tr>
<td>cudaStreamQuery</td>
<td>31</td>
</tr>
<tr>
<td>cudaStreamSynchronize</td>
<td>31</td>
</tr>
<tr>
<td>cudaStreamWaitEvent</td>
<td>31</td>
</tr>
<tr>
<td>CUDART_SURFACE</td>
<td></td>
</tr>
<tr>
<td>cudaBindSurfaceToArray</td>
<td>112</td>
</tr>
<tr>
<td>cudaGetSurfaceReference</td>
<td>112</td>
</tr>
<tr>
<td>CUDART_TEXTURE</td>
<td></td>
</tr>
<tr>
<td>cudaBindTexture</td>
<td>107</td>
</tr>
<tr>
<td>cudaMemcpy2D</td>
<td>108</td>
</tr>
<tr>
<td>cudaMemcpyTextureToArray</td>
<td>109</td>
</tr>
<tr>
<td>cudaMemcpyTextureToArrayAsync</td>
<td>109</td>
</tr>
<tr>
<td>createChannelDesc</td>
<td>109</td>
</tr>
<tr>
<td>getChannelDesc</td>
<td>110</td>
</tr>
<tr>
<td>getTextureAlignmentOffset</td>
<td>110</td>
</tr>
<tr>
<td>getTextureReference</td>
<td>111</td>
</tr>
<tr>
<td>unbindTexture</td>
<td>111</td>
</tr>
<tr>
<td>CUDART_THREAD_DEPRECATED</td>
<td></td>
</tr>
<tr>
<td>cudaThreadExit</td>
<td>23</td>
</tr>
<tr>
<td>cudaThreadGetCacheConfig</td>
<td>24</td>
</tr>
<tr>
<td>cudaThreadGetLimit</td>
<td>24</td>
</tr>
<tr>
<td>cudaThreadSetCacheConfig</td>
<td>25</td>
</tr>
<tr>
<td>cudaThreadSetLimit</td>
<td>26</td>
</tr>
<tr>
<td>cudaThreadSynchronize</td>
<td>26</td>
</tr>
<tr>
<td>CUDART_TYPES</td>
<td></td>
</tr>
<tr>
<td>cudaArrayDefault</td>
<td>157</td>
</tr>
<tr>
<td>cudaArrayLayered</td>
<td>157</td>
</tr>
<tr>
<td>cudaArraySurfaceLoadStore</td>
<td>157</td>
</tr>
<tr>
<td>cudaChannelFormatKind</td>
<td>157</td>
</tr>
<tr>
<td>cudaComputeMode</td>
<td>159</td>
</tr>
<tr>
<td>cudaDeviceBlockingSync</td>
<td>157</td>
</tr>
<tr>
<td>cudaDeviceLmemResizeToMax</td>
<td>157</td>
</tr>
<tr>
<td>cudaDeviceMapHost</td>
<td>157</td>
</tr>
<tr>
<td>cudaDeviceMask</td>
<td>157</td>
</tr>
<tr>
<td>cudaDevicePropDontCare</td>
<td>157</td>
</tr>
<tr>
<td>cudaDeviceScheduleAuto</td>
<td>157</td>
</tr>
<tr>
<td>cudaDeviceScheduleBlockingSync</td>
<td>157</td>
</tr>
<tr>
<td>cudaDeviceScheduleSpin</td>
<td>157</td>
</tr>
<tr>
<td>cudaDeviceScheduleYield</td>
<td>158</td>
</tr>
<tr>
<td>cudaError</td>
<td>159</td>
</tr>
<tr>
<td>cudaError_t</td>
<td>159</td>
</tr>
<tr>
<td>cudaEvent_t</td>
<td>159</td>
</tr>
<tr>
<td>cudaEventBlockingSync</td>
<td>158</td>
</tr>
<tr>
<td>cudaEventDefault</td>
<td>158</td>
</tr>
<tr>
<td>cudaEventDisableTiming</td>
<td>158</td>
</tr>
<tr>
<td>cudaFuncCache</td>
<td>163</td>
</tr>
<tr>
<td>cudaGraphicsCubeFace</td>
<td>163</td>
</tr>
<tr>
<td>cudaGraphicsMapFlags</td>
<td>163</td>
</tr>
<tr>
<td>cudaGraphicsRegisterFlags</td>
<td>164</td>
</tr>
<tr>
<td>cudaGraphicsResource_t</td>
<td>159</td>
</tr>
<tr>
<td>cudaHostAllocDefault</td>
<td>158</td>
</tr>
<tr>
<td>cudaHostAllocMapped</td>
<td>158</td>
</tr>
<tr>
<td>cudaHostAllocPortable</td>
<td>158</td>
</tr>
<tr>
<td>cudaHostAllocWriteCombined</td>
<td>158</td>
</tr>
<tr>
<td>cudaHostRegisterDefault</td>
<td>158</td>
</tr>
<tr>
<td>cudaHostRegisterMapped</td>
<td>158</td>
</tr>
<tr>
<td>cudaHostRegisterPortable</td>
<td>158</td>
</tr>
<tr>
<td>cudaLimit</td>
<td>164</td>
</tr>
<tr>
<td>cudaMemcpyKind</td>
<td>164</td>
</tr>
<tr>
<td>cudaMemoryType</td>
<td>165</td>
</tr>
<tr>
<td>cudaOutputMode</td>
<td>165</td>
</tr>
<tr>
<td>cudaOutputMode_t</td>
<td>159</td>
</tr>
<tr>
<td>cudaPeerAccessDefault</td>
<td>159</td>
</tr>
<tr>
<td>cudaStream_t</td>
<td>159</td>
</tr>
</tbody>
</table>
memPitch, 380
gsPerBlock, 380
sharedMemPerBlock, 380
SIMDWidth, 380
textureAlign, 381
totalConstantMemory, 381
cuDriverGetVersion
 CUDA_VERSION, 189
CUevent
 CUDA_TYPE, 177
CUevent_flags
 CUDA_TYPE, 177
CUevent_flags_enum
 CUDA_TYPE, 183
cuEventCreate
 CUDA_EVENT, 270
cuEventDestroy
 CUDA_EVENT, 271
cuEventElapsedTime
 CUDA_EVENT, 271
cuEventQuery
 CUDA_EVENT, 272
cuEventRecord
 CUDA_EVENT, 272
cuEventSynchronize
 CUDA_EVENT, 273
CUfilter_mode
 CUDA_TYPE, 177
CUfilter_mode_enum
 CUDA_TYPE, 184
CUfunc_cache
 CUDA_TYPE, 177
CUfunc_cache_enum
 CUDA_TYPE, 184
cuFuncGetAttribute
 CUDA_EXEC, 274
cuFuncSetBlockShape
 CUDA_EXEC_DEPRECATED, 278
cuFuncSetCacheConfig
 CUDA_EXEC, 275
cuFuncSetSharedSize
 CUDA_EXEC_DEPRECATED, 279
CUfunction
 CUDA_TYPE, 177
CUfunction_attribute
 CUDA_TYPE, 177
CUfunction_attribute_enum
 CUDA_TYPE, 184
cuGLCtxCreate
 CUDA_GL, 304
cuGLInit
 CUDA_GL_DEPRECATED, 308
CUGLmap_flags
 CUDA_GL_DEPRECATED, 307
cuGLMapBufferObject
 CUDA_GL_DEPRECATED, 308
cuGLMapBufferObjectAsync
 CUDA_GL_DEPRECATED, 309
cuGLRegisterBufferObject
 CUDA_GL_DEPRECATED, 309
cuGLSetBufferObjectMapFlags
 CUDA_GL_DEPRECATED, 310
cuGLUnmapBufferObject
 CUDA_GL_DEPRECATED, 310
cuGLUnmapBufferObjectAsync
 CUDA_GL_DEPRECATED, 311
cuGLUnregisterBufferObject
 CUDA_GL_DEPRECATED, 311
cuGraphicsD3D10RegisterResource
 CUDA_D3D10, 331
cuGraphicsD3D11RegisterResource
 CUDA_D3D11, 346
cuGraphicsD3D9RegisterResource
 CUDA_D3D9, 316
cuGraphicsGLRegisterBuffer
 CUDA_GL, 305
cuGraphicsGLRegisterImage
 CUDA_GL, 305
CUgraphicsMapResourceFlags
 CUDA_TYPE, 177
CUgraphicsMapResourceFlags_enum
 CUDA_TYPE, 185
CUgraphicsResource
 CUDA_TYPE, 177
CUgraphicsResourceGetMappedPointer
 CUDA_GRAPHICS, 300
cuGraphicsResourceSetMapFlags
 CUDA_GRAPHICS, 300
cuGraphicsSubResourceGetMappedArray
 CUDA_GRAPHICS, 301
cuGraphicsUnmapResources
 CUDA_GRAPHICS, 302
cuGraphicsUnregisterResource
 CUDA_GRAPHICS, 302
cuGraphicsVDPAURegisterOutputSurface
 CUDA_VDPAU, 349
cuGraphicsVDPAURegisterVideoSurface
 CUDA_VDPAU, 350
cuInit
 CUDA_INITIALIZE, 188
CUjit_fallback
   CUDA_TYPES, 177
CUjit_fallback_enum
   CUDA_TYPES, 185
CUjit_option
   CUDA_TYPES, 177
CUjit_option_enum
   CUDA_TYPES, 185
CUjit_target
   CUDA_TYPES, 177
CUjit_target_enum
   CUDA_TYPES, 186
cuLaunch
   CUDA_EXEC_DEPRECATED, 279
cuLaunchGrid
   CUDA_EXEC_DEPRECATED, 280
cuLaunchGridAsync
   CUDA_EXEC_DEPRECATED, 280
cuLaunchKernel
   CUDA_EXEC, 276
CUlimit
   CUDA_TYPES, 178
CUlimit_enum
   CUDA_TYPES, 186
cuMemAlloc
   CUDA_MEM, 223
cuMemAllocHost
   CUDA_MEM, 224
cuMemAllocPitch
   CUDA_MEM, 224
cuMemcpy
   CUDA_MEM, 225
cuMemcpy2D
   CUDA_MEM, 226
cuMemcpy2DAsync
   CUDA_MEM, 228
cuMemcpy2DUnaligned
   CUDA_MEM, 231
cuMemcpy3D
   CUDA_MEM, 233
cuMemcpy3DAsync
   CUDA_MEM, 235
cuMemcpy3DPeer
   CUDA_MEM, 238
cuMemcpy3DPeerAsync
   CUDA_MEM, 239
cuMemcpyAsync
   CUDA_MEM, 239
cuMemcpyAtoA
   CUDA_MEM, 240
cuMemcpyAtoD
   CUDA_MEM, 240
cuMemcpyAtoH
   CUDA_MEM, 241
cuMemcpyAtoHAsync
   CUDA_MEM, 244
cuMemcpyDtoA
   CUDA_MEM, 242
cuMemcpyDtoD
   CUDA_MEM, 243
cuMemcpyDtoDAsync
   CUDA_MEM, 243
cuMemcpyDtoH
   CUDA_MEM, 244
cuMemcpyDtoHAsync
   CUDA_MEM, 244
cuMemcpyHtoA
   CUDA_MEM, 245
cuMemcpyHtoAAsync
   CUDA_MEM, 246
cuMemcpyHtoD
   CUDA_MEM, 246
cuMemcpyHtoDAsync
   CUDA_MEM, 247
cuMemcpyPeer
   CUDA_MEM, 248
cuMemcpyPeerAsync
   CUDA_MEM, 248
cuMemFree
   CUDA_MEM, 249
cuMemFreeHost
   CUDA_MEM, 249
cuMemGetAddressRange
   CUDA_MEM, 250
cuMemGetInfo
   CUDA_MEM, 250
cuMemHostAlloc
   CUDA_MEM, 251
cuMemHostGetDevicePointer
   CUDA_MEM, 252
cuMemHostGetFlags
   CUDA_MEM, 253
cuMemHostRegister
   CUDA_MEM, 253
cuMemHostUnregister
   CUDA_MEM, 254
CUmemorytype
   CUDA_TYPES, 178
CUmemorytype_enum
   CUDA_TYPES, 186
cuMemsetD16
   CUDA_MEM, 255
cuMemsetD16Async
   CUDA_MEM, 255
cuMemsetD2D16
   CUDA_MEM, 256
cuMemsetD2D16Async
   CUDA_MEM, 257
cuWGLGetDevice
cudadevice

Data types used by CUDA driver, 168
Data types used by CUDA Runtime, 153
Depth
CUAD_ARRAY3D_DESCRIPTOR_st, 353
CUAD_MEMCPY3D_PEER_st, 358
CUAD_MEMCPY3D_st, 361
depth

cudaExtent, 370
device

cudadeviceAttributes, 378
Device Management, 13, 190
deviceOverlap
cudadeviceProp, 366
devicePointer

cudadeviceAttributes, 378
Direct3D 10 Interoperability, 90, 328
Direct3D 11 Interoperability, 95, 343
Direct3D 9 Interoperability, 85, 313
dstArray

CUAD_MEMCPY2D_st, 356
CUAD_MEMCPY3D_PEER_st, 358
CUAD_MEMCPY3D_st, 361
cudamemcpy3DParms, 373
cudamemcpy3DPeerParms, 375
dstContext

CUAD_MEMCPY3D_PEER_st, 358
dstDevice

CUAD_MEMCPY2D_st, 356
CUAD_MEMCPY3D_PEER_st, 358
CUAD_MEMCPY3D_st, 361
cudamemcpy3DParms, 373
cudamemcpy3DPeerParms, 375
dstHeight

CUAD_MEMCPY3D_PEER_st, 359
CUAD_MEMCPY3D_st, 361
dstHost

CUAD_MEMCPY2D_st, 356
CUAD_MEMCPY3D_PEER_st, 359
CUAD_MEMCPY3D_st, 362
dstLOD

CUAD_MEMCPY3D_PEER_st, 359
CUAD_MEMCPY3D_st, 362
dstMemoryType

CUAD_MEMCPY2D_st, 356
CUAD_MEMCPY3D_PEER_st, 359
CUAD_MEMCPY3D_st, 362
dstPitch

CUAD_MEMCPY2D_st, 356
CUAD_MEMCPY3D_PEER_st, 359
CUAD_MEMCPY3D_st, 362
dstPos

cudamemcpy3DParms, 373
cudamemcpy3DPeerParms, 375
dstPtr

cudamemcpy3DParms, 373
cudamemcpy3DPeerParms, 375
dstXInBytes

CUAD_MEMCPY2D_st, 356
CUAD_MEMCPY3D_PEER_st, 359
CUAD_MEMCPY3D_st, 362
dstY

CUAD_MEMCPY2D_st, 357
CUAD_MEMCPY3D_PEER_st, 359
CUAD_MEMCPY3D_st, 362
dstZ

CUAD_MEMCPY3D_PEER_st, 359
CUAD_MEMCPY3D_st, 362
ECCEnabled

cudadeviceProp, 366
Error Handling, 28
Event Management, 33, 270
Execution Control, 37, 274
extent


cudamemcpy3DParms, 373
cudamemcpy3DPeerParms, 375
f

cudachannelFormatDesc, 364
filterMode

textureReference, 383
Flags

CUAD_ARRAY3D_DESCRIPTOR_st, 353
Format

CUAD_ARRAY3D_DESCRIPTOR_st, 353
CUAD_ARRAY_DESCRIPTOR_st, 355
Graphics Interoperability, 103, 299
Height

CUAD_ARRAY3D_DESCRIPTOR_st, 353
CUAD_ARRAY_DESCRIPTOR_st, 355
CUAD_MEMCPY2D_st, 357
CUAD_MEMCPY3D_PEER_st, 359
CUAD_MEMCPY3D_st, 362
height

cudaExtent, 370
hostPointer

cudadeviceAttributes, 378
Initialization, 188
integrated

cudadeviceProp, 366
Interactions with the CUDA Driver API, 128
kernelExecTimeoutEnabled

cudadeviceProp, 366
kind
  cudaMemcpy3DParms, 373
l2CacheSize
  cudaMemcpy3DParms, 373
localSizeBytes
  cudaMemcpy3DParms, 373
major
  cudaMemcpy3DParms, 373
make_cudaExtent
  CUDART_MEMORY, 75
make_cudaPitchedPtr
  CUDART_MEMORY, 75
make_cudaPos
  CUDART_MEMORY, 76
maxGridSize
  cudaMemcpy3DParms, 373
  cudaMemcpy3DPeerParms, 375
maxTexture1D
  cudaMemcpy3DParms, 373
maxTexture1DLayered
  cudaMemcpy3DParms, 373
maxTexture2D
  cudaMemcpy3DParms, 373
maxTexture2DLayered
  cudaMemcpy3DParms, 373
maxThreadsDim
  cudaMemcpy3DParms, 373
maxThreadsPerBlock
  cudaMemcpy3DParms, 373
  cudaMemcpy3DPeerParms, 375
Memory Management, 41, 215
memoryBusWidth
  cudaMemcpy3DParms, 373
memoryClockRate
  cudaMemcpy3DParms, 373
memoryType
  cudaMemcpy3DParms, 373
memPitch
  cudaMemcpy3DParms, 373
minor
  cudaMemcpy3DParms, 373
Module Management, 208
multiProcessorCount
  cudaMemcpy3DParms, 373
name
  cudaMemcpy3DParms, 373
normalized
  textureReference, 383
NumChannels
  CUDA_ARRAY3D_DESCRIPTOR_st, 353
  CUDA_ARRAY_DESCRIPTOR_st, 355
numRegs
  cudaMemcpy3DParms, 373
OpenGL Interoperability, 82, 304
pciBusID
  cudaMemcpy3DParms, 373
pciDeviceID
  cudaMemcpy3DParms, 373
pciDomainID
  cudaMemcpy3DParms, 373
Peer Context Memory Access, 297
Peer Device Memory Access, 80
pitch
  cudaMemcpy3DParms, 373
ptr
  cudaMemcpy3DParms, 373
ptxVersion
  cudaMemcpy3DParms, 373
regsPerBlock
  cudaMemcpy3DParms, 373
  cudaMemcpy3DPeerParms, 375
reserved0
  cudaMemcpy3DParms, 373
  cudaMemcpy3DPeerParms, 375
reserved1
  cudaMemcpy3DParms, 373
  cudaMemcpy3DPeerParms, 375
sharedMemPerBlock
  cudaMemcpy3DParms, 373
  cudaMemcpy3DPeerParms, 375
sharedSizeBytes
  cudaMemcpy3DParms, 373
SIMDWidth
  cudaMemcpy3DParms, 373
srcArray
  cudaMemcpy3DParms, 373
srcContext
  cudaMemcpy3DParms, 373
srcDevice
  cudaMemcpy3DParms, 373
srcHeight
  cudaMemcpy3DParms, 373
CUDA_MEMCPY3D_PEER_st, 360
CUDA_MEMCPY3D_st, 363

srcHost
CUDA_MEMCPY2D_st, 357
CUDA_MEMCPY3D_PEER_st, 360
CUDA_MEMCPY3D_st, 363

srcLOD
CUDA_MEMCPY3D_PEER_st, 360
CUDA_MEMCPY3D_st, 363

srcMemoryType
CUDA_MEMCPY2D_st, 357
CUDA_MEMCPY3D_PEER_st, 360
CUDA_MEMCPY3D_st, 363

srcPitch
CUDA_MEMCPY2D_st, 357
CUDA_MEMCPY3D_PEER_st, 360
CUDA_MEMCPY3D_st, 363

srcPos
cudaMemcpy3DParms, 373
cudaMemcpy3DPeerParms, 375

cudaMemcp3DParms, 373
cudaMemcpy3DPeerParms, 376

srcPtr
cudaMemcpy3DParms, 373
cudaMemcpy3DPeerParms, 376

srcXInBytes
CUDA_MEMCPY2D_st, 357
CUDA_MEMCPY3D_PEER_st, 360
CUDA_MEMCPY3D_st, 363

srcY
CUDA_MEMCPY2D_st, 357
CUDA_MEMCPY3D_PEER_st, 360
CUDA_MEMCPY3D_st, 363

cudaChannelFormatDesc, 364
cudaPos, 379

cudaChannelFormatDesc, 364
cudaPos, 379

cudaChannelFormatDesc, 364
cudaPos, 379

cudaChannelFormatDesc, 364
cudaPos, 379

cudaChannelFormatDesc, 364
cudaPos, 379

cudaChannelFormatDesc, 364
cudaPos, 379
Notice
ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks
NVIDIA, the NVIDIA logo, GeForce, Tesla, and Quadro are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright
© 2007-2011 NVIDIA Corporation. All rights reserved.