Textures & Surfaces

CUDA Webinar

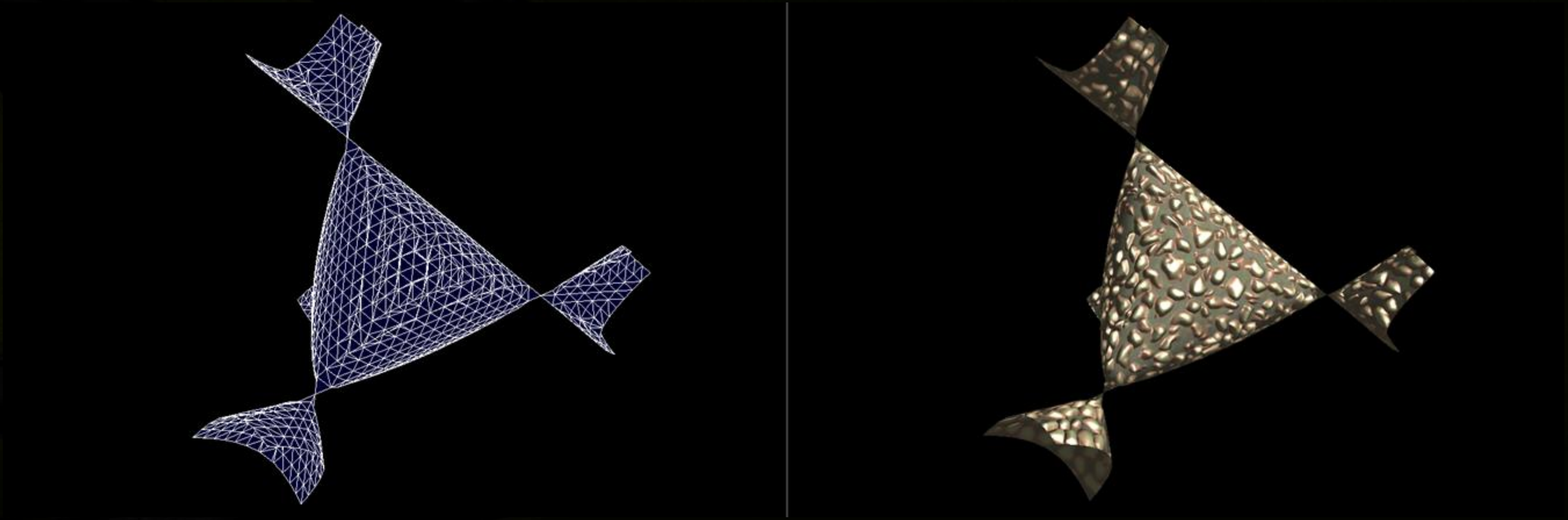
Gernot Ziegler,
Developer Technology (Compute)

Outline

- Intro to Texturing and Texture Unit
- CUDA Array Storage
- Textures in CUDA C (Setup, Binding Modes, Coordinates)
- Texture Data Processing
- Texture Interpolation
- Surfaces
- Layered Textures (CUDA 4.0 Features)
- Usage Advice
- Misc: 16 bit-floating point textures, OpenGL/DirectX Exchange
- Summary, Further Reading and Questions

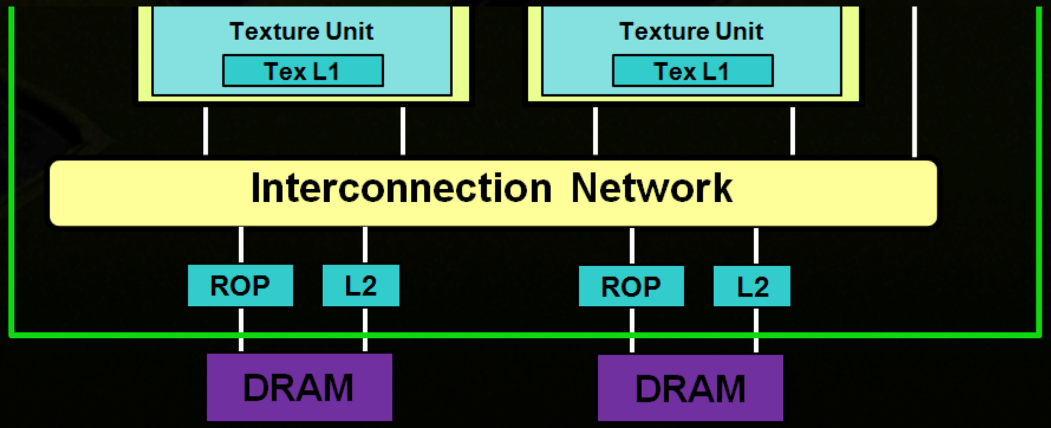
Texturing

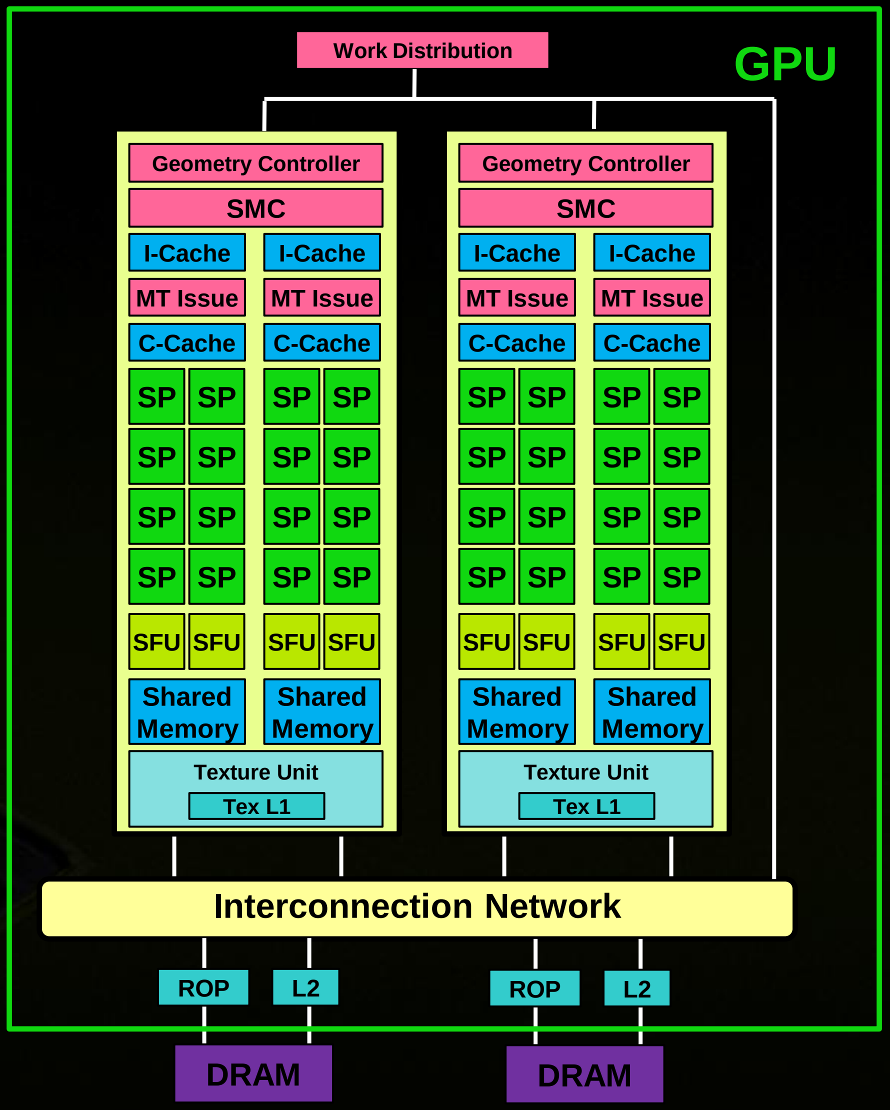
- **Original purpose:**
  - Provide surface coloring for 3D meshes (a "wrapping")
  - 3D mesh has "texture coordinates", hardware looks up 2D color array



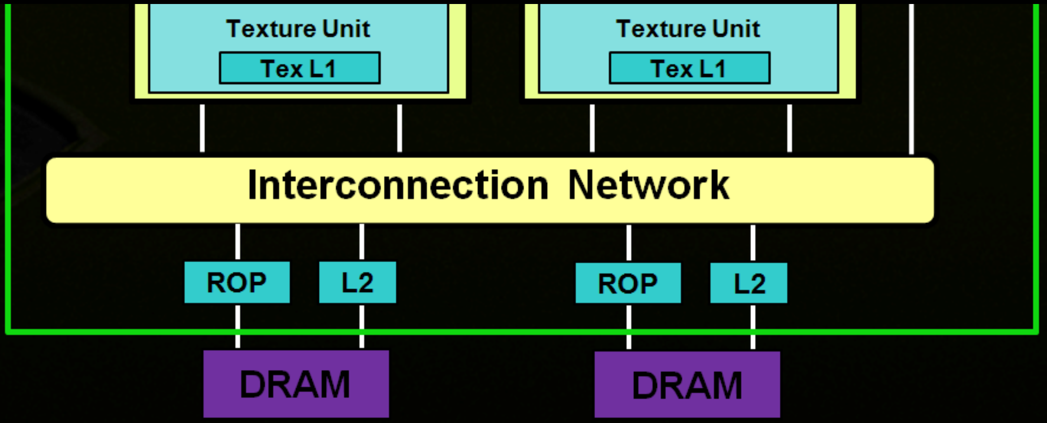
Texture Unit

- **Texture Read:**
  Global memory read via texture hardware path
- **Data reads are cached**
  - Texture Cache (separate from L1)
  - Specialized for 2D/3D spatial locality





Texture Unit



- **Data conversion (integer to float, 16 bit float to 32 bit float)**
- **Data Interpolation (aka Filtering)**
  - Linear / bilinear / trilinear data interpolation in hardware
- **Boundary modes (for “out-of-bounds” addresses)**
  - Addressable in 1D, 2D, or 3D.
  - Coordinate normalization mode (access becomes resolution-independent)
  - Clamp to edge / Clamp to Border color / Repeat / Mirror
- **Works best with CUDA Array as Data Storage**

CUDA Array

- **Opaque object for 1D/2D/3D data storage in global memory**
- **Purpose**
  - Optimal caching for 2D/3D spatial locality
    (for 2D/3D threadblocks accessing in "cloud" pattern)
  - Standard exchange format for OpenGL/DirectX texture exchange
- **Data resides in Global Memory**
  - Host access through special cudaMemcpy operations
  - Device access through texture reads or surface read/write (explained later)

Setup of Textures

- **Host Code**
  - Create Channel Description
    - Used for allocation of CUDA arrays and texture binding
    - Defines number of channels, type and bitness of data stored
    - E.g. 1 x float32, 4 x uchar
  - Declare a texture reference (must currently be at file-scope)
  - Allocate texture data storage (global memory as linear/pitch linear, or CUDA array)
  - Bind texture to its data storage (device pointer / CUDA array)
- **Device Code**
  - Fetch data using texture reference
    - Textures bound to linear memory: `tex1Dfetch(tex, int coord)`
    - Textures bound to pitch linear memory: `tex2D(tex, float2 coord)`
    - Textures bound to CUDA arrays: `tex1D()` `tex2D()` `tex3D()`
    - Layered textures bound to CUDA arrays: `tex1DLayered()` `tex2DLayered()`

Texture binding modes

- **Texture references are bound to device pointer or CUDA Array**
  - Sets the data source for all reads from this texture reference
- **Bind to linear memory (device pointer)**
  - Texture is bound directly to global memory address
  - Large 1D extents (2^27 elements), but integer indexing only
  - Simple, but: No data interpolation, no clamp/repeat addressing modes
- **Bind to pitch linear (device pointer)**
  - Texture is bound directly to global memory address of pitchlinear data
  - 2D indexing (but cache locality still sees pitchlinear mem)
  - Provides data interpolation and clamp/repeat addressing modes
  - SDK: "simplePitchLinearTexture"
- **Bind to CUDA array (handle)**
  - Texture is bound to CUDA array (1D, 2D, or 3D)
  - Float addressing (within array bounds, or normalized bounds)
  - Provides data interpolation and clamp/repeat addressing modes
  - Addressing modes (clamping, repeat)
  - SDK: "simpleTexture", "simpleTexture3D", "simpleTextureDrv"

Linear memory example
(1D texture, simple caching access)

- **Host Code**

```
// global reference (visible for host and device code)
texture<float, cudaTextureType1D, cudaReadModeElementType> linmemTexture;
…
// host code: bind texture reference to linear memory
   (use implicitly created channel description)
cudaBindTexture(NULL, linmemTexture, d_linmemory_ptr,
                cudaCreateChannelDesc<float>(), linmemory_size);
// start kernel that uses texture reference!
```

- **Device Code**

```
float A = tex1Dfetch(linmemTexture, position);
```

CUDA Array example (2D texture interpolation)

Host Code

```
// global declaration of 2D float texture (visible for host and device code)
texture<float, cudaTextureType2D, cudaReadModeElementType> tex;

…
// Create explicit channel description (could use an implicit as well)
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(32, 0, 0, 0, cudaChannelFormatKindFloat);

// Allocate CUDA array in device memory
cudaArray* cuArray;
cudaMallocArray(&cuArray, &channelDesc, width, height);

// Copy some data located at address h_data in host memory into CUDA array
cudaMemcpyToArray(cuArray, 0, 0, h_data, size, cudaMemcpyHostToDevice);

// Set the texture parameters (more sophisticated than a simple linear memory texture)
// boundary handling in x and y-direction!
tex.addressMode[0] = cudaAddressModeWrap;  tex.addressMode[1] = cudaAddressModeWrap;
tex.filterMode = cudaFilterModeLinear;  // linear interpolation
tex.normalized = true; // normalized coordinate bounds [0.0 .. 1.0]

// Bind the array to the texture reference
cudaBindTextureToArray(tex, cuArray, channelDesc);
```

Device Code

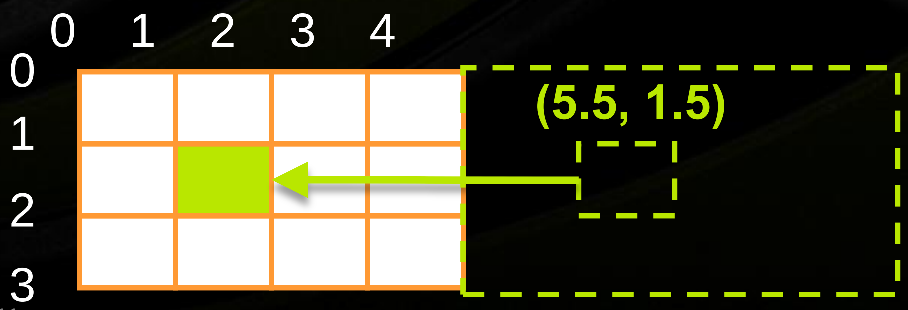
```
float value = tex2D(tex, xpos, ypos);
```

Texture Coordinates

- Texture fetch in device code takes floating point texture coordinates
- Lookup mode and coordinates determine data element fetch from global memory: "Nearest neighbour" mode uses less data than "linear interpolation" mode
- Coordinate bounds can reflect input data dimensions, or be normalized (0.0 .. 1.0)
- Boundary handling in different ways:

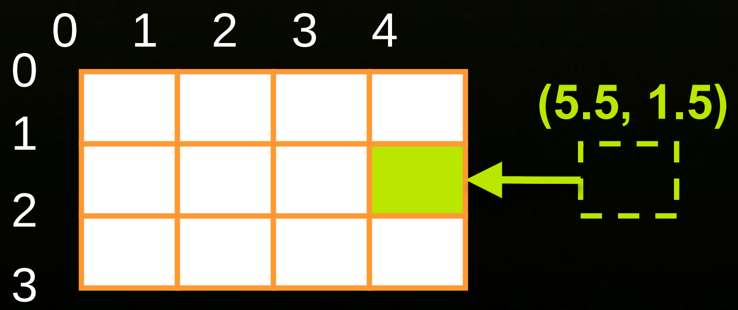
Wrap

- Out-of-bounds coordinate is wrapped (modulo arithmetic)



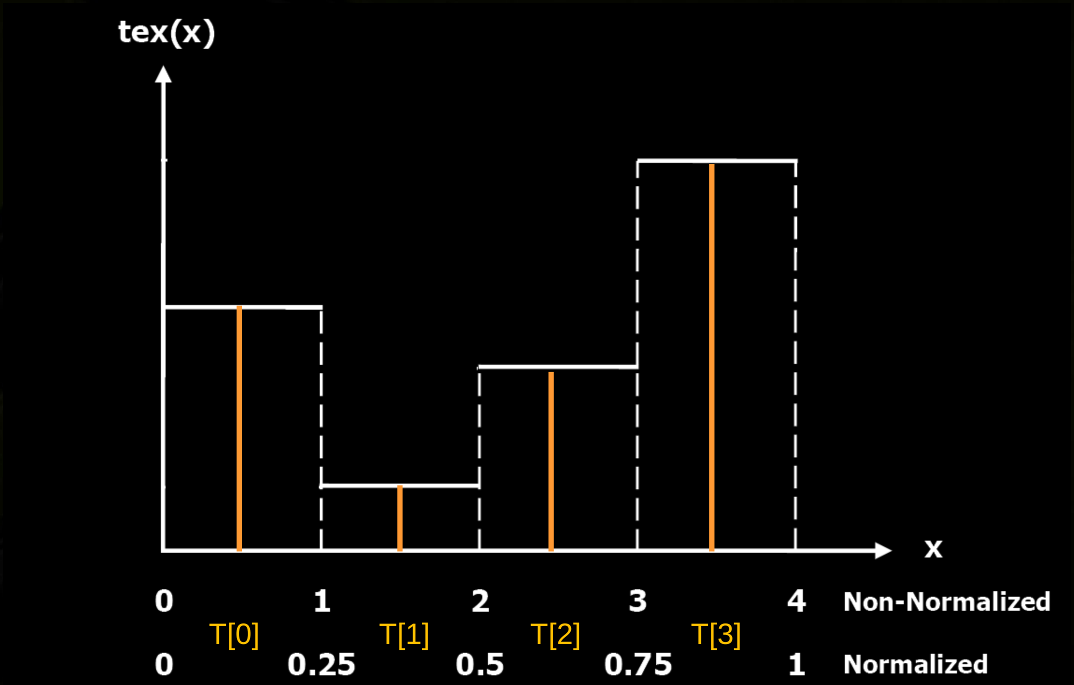
Clamp

- Out-of-bounds coordinate is clamped to closest boundary



Texture Data Processing

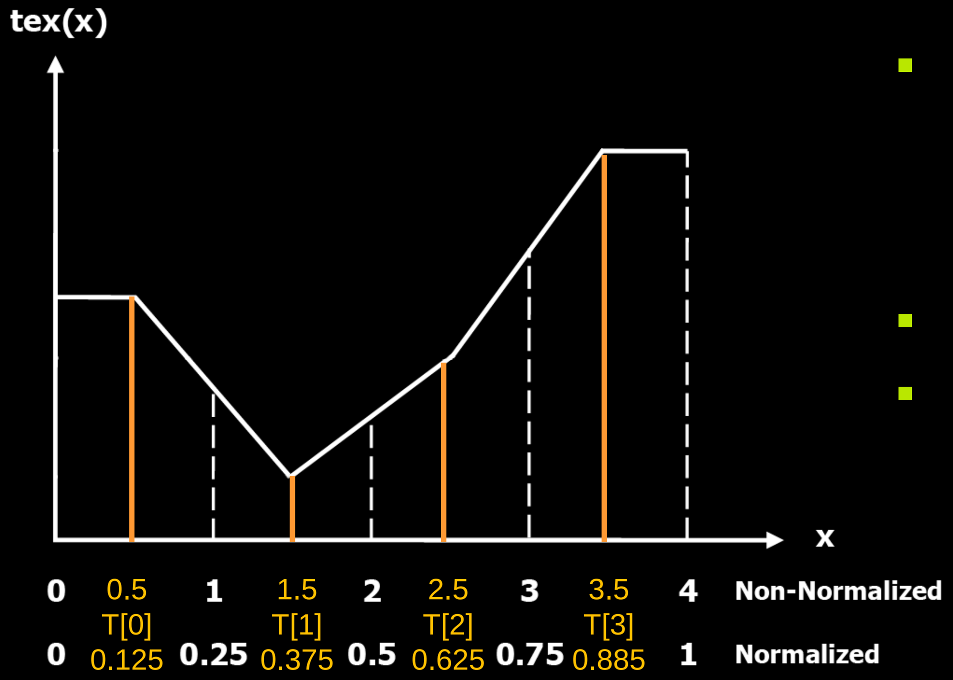
- **Texture unit can convert integer input to floating point output**
  - E.g. 8bit input: uchar4(255, 128, 0, 0) becomes float4(1.0, 0.5, 0.0, 0.0)
- **Coordinate to Data mapping for "Nearest neighbour" mode:**
  - Example: Input data T, four values:



- All input data elements cover equal output ranges
- Details in Programming Guide, Appendix E

Texture Interpolation

- **Texture unit can interpolate between adjacent data elements**
  - Fractional part of texture coordinate becomes interpolation weight
    (Note: Interpolation weight is 8 bit quantized!)
  - Only in float conversion mode, bind to CUDA array or pitchlinear memory



- Warning:
  Input's data values can NOT be read at integer offsets!
- But: Additional GFlops!
- Details in Programming Guide, Appendix E

Surfaces

- Device code can read and write CUDA arrays via Surfaces
  (Programming Guide, Appendix B.9 and SDK "simpleSurfaceWrite")
- Requires Compute Capability 2.0 or higher
- Currently available for 1D and 2D CUDA arrays
  - **Use flag** cudaArraySurfaceLoadStore during CUDA array creation
- Can also bind surface and texture to same CUDA array handle (write-to-texture)
- Surface operations have
  - no interpolation or data conversion
  - but some boundary handling
- Texture cache is not notified of CUDA array modifications!
  - Start new kernel to pick up modifications
- Note: Surface writes take x coordinate in byte size!

Layered Textures

- **Requires Compute Capability 2.0 or higher and CUDA 4.0**
- **3D coordinate, but z dimension is only integer (only xy-interpolation)**
- **Ideal for processing multiple textures with same size/format**
  - Reduced CPU overhead: single binding for entire texture array
  - Large sizes supported on Fermi GPUs with CC >= 2.0 (up to 16k x 16k x 2k)
  - e.g. Medical Imaging, Terrain Rendering (flight simulators), etc.
- **Faster Performance**
  - Faster than 3D Textures: better texture cache performance, since Linear/Bilinear interpolation only within a layer, not across layers
  - Fast interop with OpenGL / Direct3D for each layer
  - No need to create/manage a texture atlas
- **Can be bound to specially created CUDA Arrays**
  - Use **cudaMalloc3DArray() with cudaArrayLayered flag**
- **Details: Programming Guide 4.0, 3.2.10.1.5 Layered Textures**

Usage Advice

- **Texture bound to linear memory (device pointer)**
  - No interpolation!
  - Integer addressing, large extents (2^27 elements)
  - Use if texture cache shall assist L1 cache
- **Texture bound to CUDA arrays (handle)**
  - Use if texture content changes rarely
    (Can still modify content via surface writes or cudaMemcpy)
- **Texture bound to pitch linear memory (device pointer)**
  - Has float/integer addressing, filtering, and clamp/repeat addressing modes
  - Use if conversion to CUDA arrays too tedious (performance / code)
  - Performance caveat: 2D Threadblocks/Warps should only access rows!

16-bit floating point textures

- **GPU supports 16bit floating point format (aka *half*)**
  - Used e.g. for High Definition Color Range in OpenEXR format
  - Specified in IEEE standard 754-2008 as binary2
  - Not native for CPU, but C++ datatype routines are easy to find online
- **Compact representation of floating point data arrays**
  - CUDA arrays can hold 16bit float, use **cudaCreateChannelDescHalf*()**
  - Device code (e.g. for GPU manipulation of pitchlinear memory): `__float2half(float)` and `__half2float(unsigned short)`
- **Texture unit hides 16 bit float handling**
  - Texture lookups convert 16bit half to 32 bit float, can also interpolate!
  - Lookup result is always 32 bit float

Texture exchange with OpenGL/DirectX

- **Interoperability API can bind OpenGL / DirectX context to CUDA C context**
- **Textures/Surfaces from graphics APIs are exported as CUDA Arrays**
  - **Currently available for 2D textures only**
  - Direction flags tell which way data exchange goes from graphics API towards CUDA C (read-only, write-discard, read/write)
  - Host code can then modify textures with cudaArray memcpy
  - Device code can modify textures with surface read/write:
    E.g. while registering an OpenGL texture, use cudaGraphicsGLRegisterImage() with flag cudaGraphicsRegisterFlagsSurfaceLoadStore
- **See Programming Guide 4.0, 3.2.11 Graphics Interoperability**
- **See Reference Manual 4.0, 14.1 Graphics Interoperability**
- **SDK: "postProcessGL", "simpleD3D11Texture" and similar**

Profiler hints

- **Visual Profiler has profiling signals for texture requests and texture cache**
  - Compute Capability < 2.0: texture_cache_hit, texture_cache_miss
    Compute Capability >= 2.0: tex_cache_requests, tex_cache_misses
  - Derived signals:
    Texture cache memory throughput (GB/s), Texture cache hit rate (%)
  - Use these to determine texture cache assistance
- **Visual Profiler can also derive L2 cache requests caused by texture unit**
  - L2 cache texture memory read throughput (GB/s)
  - Compare to global memory throughput to determine how
    L2 cache assists all texture units' caches
- **See Visual Profiler user guide, "Derived Statistic"**

Summary

- **Texturing provides additional performance**
  - Extra cache capacity
  - Linear interpolation of adjacent data in hardware
  - Array boundary handling
  - Integer-to-float conversion, data unpacking
- **Algorithmic design considerations**
  - Texture binding modes (linear memory, pitchlinear memory, CUDA Array)
  - Texture coordinate offsets for correct linear interpolation
  - 8bit weight quantization during linear interpolation
  - Can't flush texture cache during kernel execution
  - 3D: xy-interpolation (layered textures) vs. Trilinear xyz-interpolation (3D textures)

Questions? …
Further reading

- **Textures, Surfaces and CUDA Array creation:**
  Programming Guide, 3.2.10 Texture and Surface Memory
- **Texture lookups in device code:**
  Programming Guide, Appendix B.8
- **Specification of texture interpolation modes and clamping:**
  Programming Guide, Appendix E
- **Surface read/write operations in device code:**
  Programming Guide, Appendix B.9
- **Texture and surface exchange with OpenGL / DirectX:**
  Programming Guide, 3.2.11 Graphics Interoperability
- **Texture usage in applications:**
  Best Practices Guide, 3.2.4 Texture Memory