COMS4040A & COMS7045A: High Performance Computing & Scientific Data Management
Introduction to CUDA C

Hairong Wang

School of Computer Science,
University of the Witwatersrand, Johannesburg

Semester one 2025

Contents

1. Note
2. CUDA Memory
   - Shared Memory
   - Constant Memory
   - Constant Memory Usage Example
   - Texture Memory
   - Texture Memory Usage Example
   - Using 2-Dimensional Texture Memory
   - Texture Object

WITS UNIVERSITY

Outline

1. Note
2. CUDA Memory
   - Shared Memory
   - Constant Memory
   - Constant Memory Usage Example
   - Texture Memory
   - Texture Memory Usage Example
   - Using 2-Dimensional Texture Memory
   - Texture Object

WITS UNIVERSITY

Note

- This lecture slides are about CUDA constant memory and texture memory.
- The slides are based on Chapters 6 and 7 from book [1], which is uploaded earlier.
- In order to learn how to use these two types of CUDA memory, you must work through the two chapters (6 and 7).
- The codes for the two examples are uploaded.
- The lecture video did not cover the last section, which is on texture object. This section is left as self-study.

[1] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming.* NVIDIA, 2010

WITS UNIVERSITY

Outline

1. Note
2. CUDA Memory
   - Shared Memory
   - Constant Memory
   - Constant Memory Usage Example
   - Texture Memory
   - Texture Memory Usage Example
   - Using 2-Dimensional Texture Memory
   - Texture Object

WITS UNIVERSITY

Outline

1. Note
2. CUDA Memory
   - Shared Memory
   - Constant Memory
   - Constant Memory Usage Example
   - Texture Memory
   - Texture Memory Usage Example
   - Using 2-Dimensional Texture Memory
   - Texture Object

Review on Shared Memory

- Load data from device memory to shared memory;
- Synchronize with all the other threads of the block;
- Process the data in shared memory;
- Synchronize again if necessary to make sure that shared memory has been updated with the results;
- Write the results back to device memory.

The throughput of memory accesses by a kernel can vary by an order of magnitude depending on access pattern for each type of memory. Organize memory accesses as optimally as possible.

WITS UNIVERSITY

Outline

1 Note

2 CUDA Memory
- Shared Memory
- Constant Memory
- Constant Memory Usage Example
- Texture Memory
- Texture Memory Usage Example
- Using 2-Dimensional Texture Memory
- Texture Object

WITS UNIVERSITY

Constant Memory

Key properties of constant memory

- The constant memory space resides in device memory and is cached in the L1 caches.
- Constant memory is optimized for read-only broadcast to multiple threads.
- Small size (64KB or so). As fast as registers if all threads in a warp access the same location

Constant Memory

- To declare a section of memory as constant, for example,

  ```
  __constant__ float my_array[1024];
  ```

  or with initial values

  ```
  __constant__ float my_array[1024] = \
      {0.0f, 1.0f, ...};
  ```

- To change the contents of constant memory at runtime, call

  ```
  cudaMemcpyToSymbol(my_array, host_data, \
          sizeof(host_data));
  ```

  prior to kernel launch.

Outline

1. Note
2. CUDA Memory
   - Shared Memory
   - Constant Memory
   - Constant Memory Usage Example
   - Texture Memory
   - Texture Memory Usage Example
   - Using 2-Dimensional Texture Memory
   - Texture Object

WITS UNIVERSITY

Example of Using Constant Memory

Problem: Ray tracing — a simplified model.

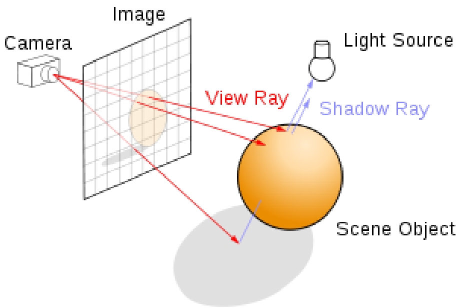

Figure: Build an image using ray tracing

- Shooting a ray from each pixel into the scene.
- Figure out the colour seen by each pixel by tracing a ray from the pixel through the scene until it hits one of the objects in the scene.

Example of Using Constant Memory cont.

- Objects: spheres assigned with different colours.
- Trace the intersection and depth only
- Main computation: Computation of the intersections of the ray with the objects in the scene.
- Ignore optical effects such as reflections, refractions etc.

Ray Tracing Example cont.

```
#define INF 2e10f

struct Sphere {
  float r,b,g;
  float radius;
  float x,y,z;
  __device__ float hit( float ox, float oy, float *n ) {
    float dx = ox - x;
    float dy = oy - y;
    if (dx*dx + dy*dy < radius*radius) {
      float dz = sqrtf( radius*radius - dx*dx - dy*dy );
      *n = dz / sqrtf( radius * radius );
      return -dz + z;
    }
    return INF;
  }
};
```

Ray Tracing Example cont.

```
__global__ void kernel( unsigned char *ptr ) {
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  int offset = x + y * blockDim.x * gridDim.x;
  float ox = (x - DIM/2);
  float oy = (y - DIM/2);
  float r=0, g=0, b=0;
  float minz = INF;
  for(int i=0; i<SPHERES; i++) {
    float n;
    float t = s[i].hit( ox, oy, &n );
    if (t < minz) {
      float fscale = n;
      r = s[i].r * fscale;
      g = s[i].g * fscale;
      b = s[i].b * fscale;
      minz = t;
    }
  }
  ptr[offset*4 + 0] = (int)(r * 255);
  ptr[offset*4 + 1] = (int)(g * 255);
  ptr[offset*4 + 2] = (int)(b * 255);
  ptr[offset*4 + 3] = 255;
}
```

Ray Tracing Example Using Constant Memory

Using global memory variable

```
Sphere *s;
cudaMalloc((void**)&s, sizeof(Sphere) * SPHERES);
......
```

Using constant variable

```
//statically allocate the space in constant memory
__constant__ Sphere s[SPHERES];

//copy from host memory to constant memory
cudaMemcpyToSymbol(s, temp_s, sizeof(Sphere) * SPHERES);
......
```

Ray Tracing Example cont.

- The code for the ray tracing example is given in the folder named `ray`. To compile using `nvcc`, you need to link with `freeglut` and `OpenGL` library as arguments (`-lglut -lGL`) to `nvcc` compiler.

Constant memory cont.

- Constant memory is a form of virtual addressing of global memory
- There is no special reserved constant memory block
- It is cached
- It supports broadcasting a single value to all the elements within a warp
- It is only read-only in respect of GPU's point of view.
- The size of constant memory is restricted, say 64KB.

WITS UNIVERSITY

Outline

1. Note
2. CUDA Memory
   - Shared Memory
   - Constant Memory
   - Constant Memory Usage Example
   - Texture Memory
   - Texture Memory Usage Example
   - Using 2-Dimensional Texture Memory
   - Texture Object

Texture Memory

The texture memory

- resides in device memory;
- is cached in texture cache — can improve performance and reduce memory traffic when reads have certain access patterns.
- Benefit:
  - The texture cache is optimized for 2D spatial locality — so threads of the same warp that read texture addresses that are close together in 2D will achieve best performance.

Texture Fetch

Texture fetch:The process of reading a texture by calling one of the texture functions.

- First parameter: a texture reference (or texture object). A texture reference specifies
  - which part of the texture is fetched;
  - must be bound to the texture memory before being used
  - Its dimensionality, 1D, 2D, or 3D
  - The type of a texture elements (texels) — integer, single-precision floating point, ...
  - The read mode: `cudaReadModeNormalizedFloat` or `cudaReadModeElementType` (default)
  - ......

Texture memory cont.

There are two different APIs to access texture memory:

- The texture reference API — supported on all devices
- The texture object API — supported on newer devices

Texture Memory cont.

- A texture reference is declared at file scope as a variable of type texture:

```
texture<DataType, Type, ReadMode> texRef;
```

  - `DataType` specifies the type of the texel — signed or unsigned 8-, 16-, 32-bit integers, 16-, 32-bit floating points.
  - `Type`: texture dimension, 1 – `cudaTextureType1D` (default), 2 – `cudaTextureType2D`, or 3 – `cudaTextureType3D`.
  - `ReadMode`: specifies the read mode, optional, only affects integer valued textures.
    - `cudaReadModeElementType` (default): no conversion
    - `cudaReadModeNormalizedFloat` — if data type is integer, value returned is mapped to $[-1.0, 1.0]$ for signed; and to $[0.0, 1.0]$ for unsigned.
- A texture reference can only be declared as a static global variable and cannot be passed as an argument to a function.
- The other attributes of a texture reference can be changed at runtime through the host runtime.

Texture Memory cont.

- Before a kernel can use a texture reference to read from texture memory, the texture reference must be bound to a texture. One of such functions:

```
cudaBindTexture(size_t *offset,\
    &texture_reference,\
    &dev_ptr, size_t size)
```

  - binds `size` bytes of texture memory pointed to by `dev_ptr` to `texture_reference`
  - `offset` — optional, byte offset
  - `dev_ptr` — memory area on device
  - `size` — size of the memory pointed to by `dev_ptr`.

Steps for Using Texture Memory

1. Declare the texture memory in CUDA
2. Bind the texture memory to your texture reference in CUDA
3. Read the texture memory from the texture reference in CUDA
4. Unbind the texture memory from your texture reference in CUDA

Texture read and unbind

- Read: `tex1Dfetch(texture_reference, int);`
- Unbind: `cudaUnbindTexture(texture_reference)`

Texture read cont.

- All texture functions except `tex1Dfetch` use floating point indexes into a texture.
- When use unnormalized coordinates, they are in the range $[0, D_{max})$, where D_{max} is the width, height, or depth of the texture.
- For unnormalized indexing, not all the texture features are available.
- Texturing using unnormalized indexes can be used in conjunction with linear filtering and the point filtering (default); as well as limited addressing mode.
- Addressing mode: deals with out of the range texture indexes – 'clamping': clamp to the last in the range element and 'border' mode: returns zero.

Texture read cont.

- Filtering mode: linear filtering mode and point filtering mode.
  - Linear filtering: fetches two neighbouring texture elements, and interpolate between them, weighted by the texture coordinates.
  - Point filtering: Returns one texture element depending on the coordinate.

Outline

1 Note

2 CUDA Memory
- Shared Memory
- Constant Memory
- Constant Memory Usage Example
- Texture Memory
- Texture Memory Usage Example
- Using 2-Dimensional Texture Memory
- Texture Object

WITS UNIVERSITY

Texture Memory Usage Example: Simulating Heat Transfer

Problem: Assume we have a rectangular room, and a few heaters with various fixed temperatures scattered around the room. Construct a simple heat transfer simulation. A simplified heat transfer model:

- The room with a handful of heaters: a rectangular grid with some cells containing heaters with constant temperatures.
- Heat flows between a cell and its neighbours at every time step.
- Compute the new temperature in a grid cell as

$$T_{new} = T_{old} + k \times (T_{top} + T_{bottom} + T_{left} + T_{right} - 4T_{old}) \tag{1}$$

Heat Transfer Example cont.

1. Copy the input cell temperatures to the grid cells.
2. compute the output temperatures based on the update in Equation 1.
3. Swap the input and output buffers in preparation of the next time step.

WITS UNIVERSITY

Heat Transfer Example cont.

Step 2 is most computationally involved

```
__global__ void blend_kernel( float *outSrc, const float *inSrc ){
  // map from threadIdx/BlockIdx to pixel position
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  int offset = x + y * blockDim.x * gridDim.x;
  int left = offset - 1;
  int right = offset + 1;
  if (x == 0) left++;
  if (x == DIM-1) right--;
  int top = offset - DIM;
  int bottom = offset + DIM;
  if (y == 0) top += DIM;
  if (y == DIM-1) bottom -= DIM;
  outSrc[offset] = inSrc[offset] + SPEED * ( inSrc[top] + \
    inSrc[bottom] + inSrc[left] + inSrc[right] - \
    inSrc[offset]*4);
}
```

Setting up the Texture Memory

- Declare the inputs as texture reference

```
texture<float> texConstSrc; //texel type is float
texture<float> texIn;
texture<float> texOut;
```

- Bind the three allocations to the texture references

```
cudaMalloc( (void**)&data.dev_inSrc, imageSize );
cudaMalloc( (void**)&data.dev_outSrc, imageSize );
cudaMalloc( (void**)&data.dev_constSrc, imageSize );
cudaBindTexture( NULL, texConstSrc, data.dev_constSrc, imageSize )
    ;
cudaBindTexture( NULL, texIn, data.dev_inSrc, imageSize );
cudaBindTexture( NULL, texOut, data.dev_outSrc, imageSize );
```

Reading from the Texture Memory

```
template<class DataType>
Type tex1Dfetch( texture<DataType, cudaTextureType1D,
cudaReadModeElementType> texRef, int x);
```

fetches from the region of linear memory bound to the 1D texture reference `texRef` using integer texture coordinate `x`.

Reading from the Texture Memory

```
float t, l, c, r, b;
if (dstOut) {
  t = tex1Dfetch(texIn,top);
  l = tex1Dfetch(texIn,left);
  c = tex1Dfetch(texIn,offset);
  r = tex1Dfetch(texIn,right);
  b = tex1Dfetch(texIn,bottom);
  } else {
  t = tex1Dfetch(texOut,top);
  l = tex1Dfetch(texOut,left);
  c = tex1Dfetch(texOut,offset);
  r = tex1Dfetch(texOut,right);
  b = tex1Dfetch(texOut,bottom);
}
dst[offset] = c + SPEED * (t + b + r + l - 4 * c);
```

WITS UNIVERSITY

Cleaning up the Texture Memory

- Unbind texture references.
- Freeing global memory

```
cudaUnbindTexture( texIn );
cudaUnbindTexture( texOut );
cudaUnbindTexture( texConstSrc );
```

Outline

1 Note

2 CUDA Memory
- Shared Memory
- Constant Memory
- Constant Memory Usage Example
- Texture Memory
- Texture Memory Usage Example
- Using 2-Dimensional Texture Memory
- Texture Object

WITS UNIVERSITY

Using 2-Dimensional Texture Memory

- Declare:

  ```
  texture<float, 2> texIn;
  texture<float, 2> texOut;
  ```

- Bind: To bind a 2D texture reference to linear memory pointed to by `devPtr`:

```
texture<float, 2> texRef;
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float
    >();
size_t offset;
cudaBindTexture2D(&offset, texRef, devPtr, \
    channelDesc, width, height, pitch);
```

WITS UNIVERSITY

2D Texture cont.

- `cudaChannelFormatDesc`: describes the contents of a texture. Specify the number of bits in each member of the texture element, and its type.
- `offset`: number of bytes in offset, `NULL` is often used.
- `pitch`: the actual width of the texture memory, must be aligned with `cudaDeviceProp.texturePitchAlignment`.

WITS UNIVERSITY

2D Texture cont.

- Read: `tex2D(texObj, x, y)` calls – fetches from the region of linear memory specified by the two dimensional texture object `texObj` using texture coordinate `(x,y)`.
- Unbind: `cudaUnbindTexture()`.

Outline

1 Note

2 CUDA Memory
- Shared Memory
- Constant Memory
- Constant Memory Usage Example
- Texture Memory
- Texture Memory Usage Example
- Using 2-Dimensional Texture Memory
- Texture Object

WITS UNIVERSITY

Texture object

- Texture objects are created and specified at runtime.
- Texture references are created at compile time and the texture is specified at runtime by bounding the texture reference to the texture using runtime functions.
- Texture objects don't require binding and unbinding. They can be passed as function arguments, but texture references can't be.

CUDA arrays

- CUDA arrays are opaque memory layouts optimized for texture fetching;
- They can be 1D, 2D, and 3D; each element can have 1, 2, or 4 components that may be signed, unsigned 8-, 16-, or 32-bit integers, 16-, or 32-bit floats;
- CUDA arrays are accessible by kernels through texture fetching.
- It is declared using `cudaArray_t`, e.g., `cudaArray_t cuArray;`
- Allocate space using `cudaMallocArray()`.

CUDA arrays cont.

```
cudaMallocArray(cudaArray_t *array, const cudaChannelDesc *desc, \
  size_t width, size_t height, unsigned int flags)
```

- `array`: pointer to allocated array in device memory
- `desc`: requested channel format
- `width, height`, requested array allocation width and height, respectively.
- `flags`, requested properties of allocated array. If unspecified, `cudaArrayDefault` or 0 values is assumed.

WITS UNIVERSITY

Texture object cont.

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

`x, y, z, w` are the lengths in bits along possible 4 component directions, respectively. These components refer to the 4 vector types, e.g., `char1` has only one component, `char3` has 3 components, etc. `cudaChannelFormatKind` is one of

- `cudaChannelFormatKindSigned`, components are of signed integer type
- `cudaChannelFormatKindUnsigned`, components are of unsigned integer
- `cudaChannelFormatKindFloat`, components are of floating point type.

Texture object cont.

```
cudaMemcpyToArray (cudaArray_t dst, size_t wOffset, size_t hOffset,\
  const void *src, size_t count, cudaMemcpyKind kind)
```

- Copies data between host and device.
- `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 (such as `cudaMemcpyHostToDevice`)

WITS UNIVERSITY

Texture object cont.

```
cudaCreateTextureObject(cudaTextureObject_t *pTexObject,\
  const cudaResourceDesc *pResDesc, const cudaTextureDesc *pTexDesc,\
  const cudaResourceViewDesc *pResViewDesc)
```

Creates a texture object.

- `pTexObject`: texture object to be created
- `pResDesc`: resource descriptor – describes the data to texture from
- `pTexDesc`: describes how the data should be sampled
- `pResViewDesc`: alternative resource descriptor, can be set to `NULL` if there is none.

Texture object cont.

```
cudaMemcpy2DToArray (cudaArray_t dst,size_t wOffset,size_t hOffset,\
  const void *src,size_t spitch,size_t width,size_t height,\
  cudaMemcpyKind kind)
```

- Copies data between host and device.
- `dst` - destination memory address
- `wOffset` - destination starting `x` offset
- `hOffset` - destination starting `y` offset
- `src` - source memory address
- `spitch` - pitch of source memory, which 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. If there is no padding, this is equal to the width (in bytes) of your input matrix point to by `src`.
- `width` - width of matrix to be transferred (columns in bytes)
- `height` - height of matrix to be transferred (rows)
- `kind` - type of transfer

WITS UNIVERSITY

Texture object cont.

For the following two structures:

- `cudaTextureDesc` is a **struct** with a number fields. Among them
  - `addressMode` specifies the addressing mode for each dimension (1D, 2D, or 3D) of the texture data; there are 4 types of addressing mode: `cudaAddressMoseWrap`; `cudaAddressModeClamp`; `cudaAddressModeMirror`; `cudaAddressModeBorder`.
  - `filterMode` specifies the filtering mode to be used when fetching from the texture; there are two types of filtering mode: `cudaFilterModePoint` and `cudaFilterModeLinear`.
  - `readMode` specifies whether integer data should be converted to floating point or not; `cudaReadModeElementType` reads as integer and `cudaReadModeNormalizedFloat` reads as normalized float.
- `cudaResourceDesc` is a **struct** with a number of fields. Among them
  - `resType` specifies the type of resource to texture from.

see documentation [2].

[2] Sec. 6.27, NVIDIA. *CUDA Runtime API*. NVIDIA, 2025. `https://docs.nvidia.com/cuda/cuda-runtime-api/index.html`. Accessed April 2025

Texture object cont.

```
tex2D(cudaTextureObject_t texObj, float x, float y);
```

fetches from the CUDA array or the region of linear memory specified by the 2D texture object `texObj` using texture coordinate `(x,y)`.

Texture object cont.

The code `ex_tex_obj.cu` gives a very simple example of using CUDA array. Extensive comments are given in the code.

References

- References used are in [4, Chapters 6-7], [1], [2], and [3].

References

[1] David B. Kirk and WenMei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, second edition, 2013.

[2] NVIDIA. *CUDA C++ Programming Guide*. NVIDIA, 2025. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. Accessed April 2025.

[3] NVIDIA. *CUDA Runtime API*. NVIDIA, 2025. `https://docs.nvidia.com/cuda/cuda-runtime-api/index.html`. Accessed April 2025.

[4] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. NVIDIA, 2010.

WITS UNIVERSITY