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

Hairong Wang

School of Computer Science,
University of the Witwatersrand, Johannesburg

2019-3-21

WITS UNIVERSITY

Contents

Outline

Note for self study on constant and texture memory

- This lecture slides are to complement for your self study on CUDA constant memory and texture memory.
- The slides are based on Chapters 6 and 7 from the book “Cuda by Example”, which I uploaded in the first lecture of CUDA.
- 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.

Outline

Outline

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.

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.

Outline

Constant Memory

Key properties of constant memory

- The constant memory space resides in device memory and is cached in the constant cache.
- 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(dev_data, host_data, \
        sizeof(host_data));
```

prior to kernel launch.

Outline

Example of Using Constant Memory

Problem: Ray tracing — a simplified model.

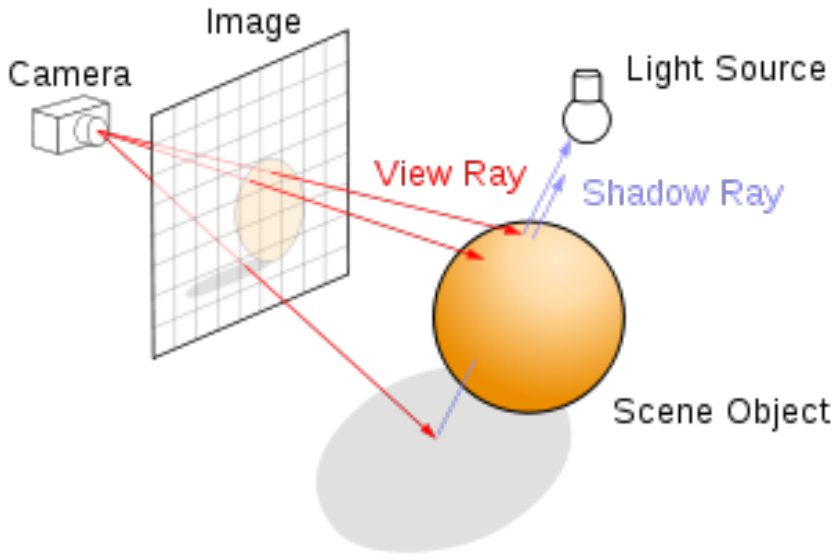

Figure: Build an image using ray tracing

- 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 Contd.

- 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 Contd.

```
1   #define INF 2e10f

3   struct Sphere {
4     float r,b,g;
5     float radius;
6     float x,y,z;
7     __device__ float hit( float ox, float oy, float *n ) {
8       float dx = ox - x;
9       float dy = oy - y;
10      if (dx*dx + dy*dy < radius*radius) {
11        float dz = sqrtf( radius*radius - dx*dx - dy*dy );
12        *n = dz / sqrtf( radius * radius );
13        return -dz + z;
14      }
15      return INF;
16    }
17  };
```

Ray Tracing Example Contd.

```
1  __global__ void kernel( unsigned char *ptr ) {
2  // map from threadIdx/BlockIdx to pixel position
3  int x = threadIdx.x + blockIdx.x * blockDim.x;
4  int y = threadIdx.y + blockIdx.y * blockDim.y;
5  int offset = x + y * blockDim.x * gridDim.x;
6  float ox = (x - DIM/2);
7  float oy = (y - DIM/2);
8  float r=0, g=0, b=0;
9  float minz = INF;
10 for(int i=0; i<SPHERES; i++) {
11   float n;
12   float t = s[i].hit( ox, oy, &n );
13   if (t < minz) {
14     float fscale = n;
15     r = s[i].r * fscale;
16     g = s[i].g * fscale;
17     b = s[i].b * fscale;
18     minz = t;
19   }
20 }
```

Ray Tracing Example Contd.

```
1  ptr[offset*4 + 0] = (int)(r * 255);
2  ptr[offset*4 + 1] = (int)(g * 255);
3  ptr[offset*4 + 2] = (int)(b * 255);
4  ptr[offset*4 + 3] = 255;
```

Ray Tracing Example Using Constant Memory

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


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

8  //copy from host memory to constant memory
9  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.

Outline

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

(We only look at the first one here.)

Texture Memory Contd.

- 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 Contd.

- 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

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

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

Heat Transfer Example Contd.

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.

Heat Transfer Example Contd.

Step 2 is most computationally involved

```
1  __global__ void blend_kernel( float *outSrc, const float *inSrc
       ){
2    // map from threadIdx/BlockIdx to pixel position
3    int x = threadIdx.x + blockIdx.x * blockDim.x;
4    int y = threadIdx.y + blockIdx.y * blockDim.y;
5    int offset = x + y * blockDim.x * gridDim.x;
6    int left = offset - 1;
7    int right = offset + 1;
8    if (x == 0) left++;
9    if (x == DIM-1) right--;
10   int top = offset - DIM;
11   int bottom = offset + DIM;
12   if (y == 0) top += DIM;
13   if (y == DIM-1) bottom -= DIM;
14   outSrc[offset] = inSrc[offset] + SPEED * ( inSrc[top] + \
15     inSrc[bottom] + inSrc[left] + inSrc[right] - \
16     inSrc[offset]*4);
17 }
```

Setting up the Texture Memory

- Declare the inputs as texture reference

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

- Bind the three allocations to the texture references

```
1 cudaMalloc( (void**)&data.dev_inSrc, imageSize );
2 cudaMalloc( (void**)&data.dev_outSrc, imageSize );
3 cudaMalloc( (void**)&data.dev_constSrc, imageSize );
4 cudaBindTexture( NULL, texConstSrc, data.dev_constSrc,
      imageSize );
5 cudaBindTexture( NULL, texIn, data.dev_inSrc, imageSize );
6 cudaBindTexture( NULL, texOut, data.dev_outSrc, imageSize )
      ;
```

Reading from the Texture Memory

```
1 template<class DataType>
2 Type tex1Dfetch( texture<DataType, cudaTextureType1D,
3 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

```
1  float t, l, c, r, b;
2  if (dstOut) {
3    t = tex1Dfetch(texIn,top);
4    l = tex1Dfetch(texIn,left);
5    c = tex1Dfetch(texIn,offset);
6    r = tex1Dfetch(texIn,right);
7    b = tex1Dfetch(texIn,bottom);
8    } else {
9    t = tex1Dfetch(texOut,top);
10   l = tex1Dfetch(texOut,left);
11   c = tex1Dfetch(texOut,offset);
12   r = tex1Dfetch(texOut,right);
13   b = tex1Dfetch(texOut,bottom);
14 }
15 dst[offset] = c + SPEED * (t + b + r + l - 4 * c);
```

Cleaning up the Texture Memory

- Unbind texture references.
- Freeing global memory

```
1  cudaUnbindTexture( texIn );
2  cudaUnbindTexture( texOut );
3  cudaUnbindTexture( texConstSrc );
```

Outline

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`:

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

2D Texture Contd.

- `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`.

2D Texture Contd.

- 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()`.


WITS
UNIVERSITY

References

- CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/
- Programming Massively Parallel Processors: A Hands-on Approach, first edition, by David B. Kirk and Wen-mei W. Hwu. Morgan Kaufmann Publishers Inc., 2010.
- Chapter 6 - 7, CUDA by Example: An Introduction to General-Purpose GPU Programming, 1st edition, Sanders, Jason and Kandrot, Edward, Addison-Wesley Professional, 2010.