Parallel Computing with GPUs

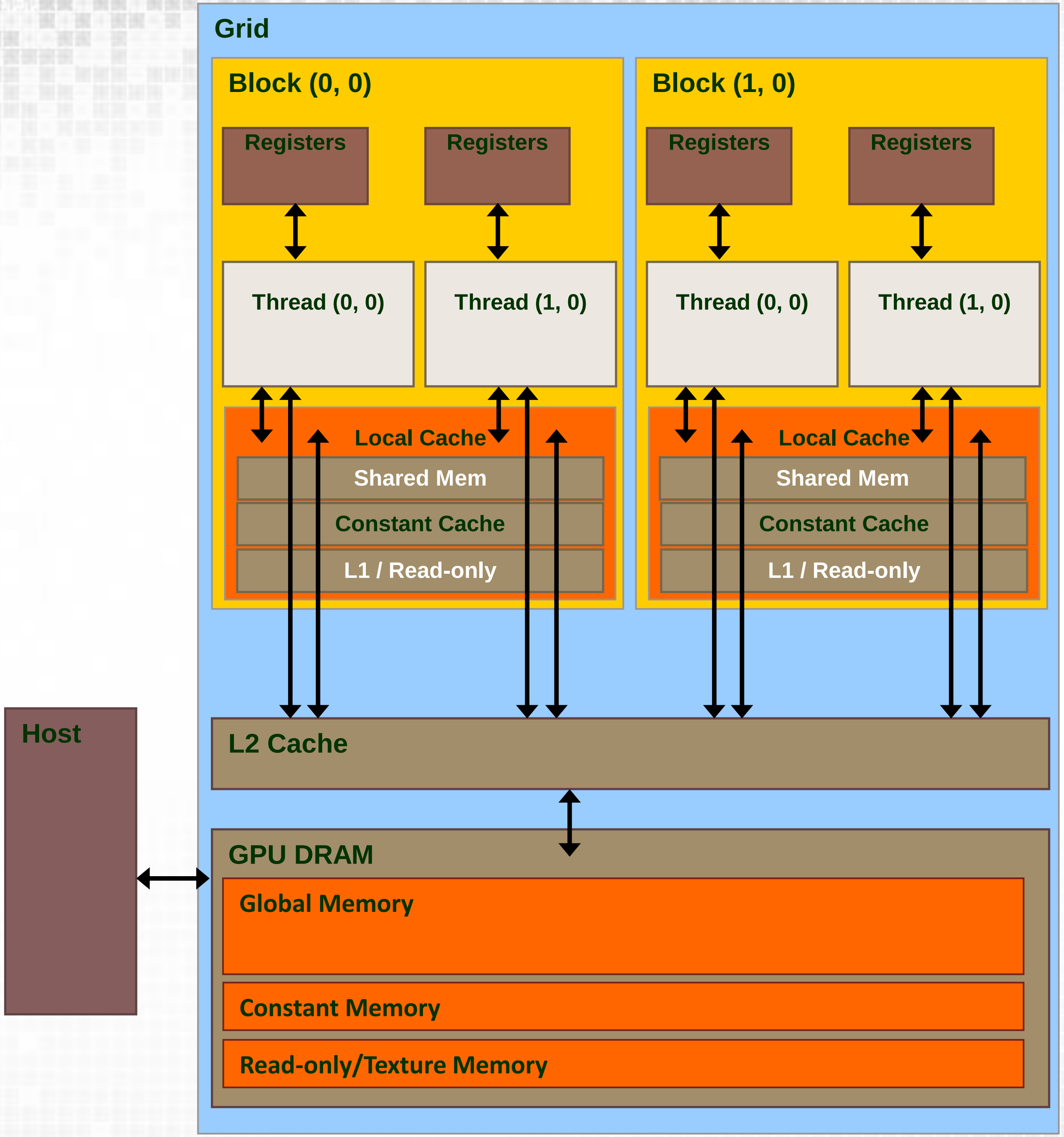
CUDA Memory
Part 3 - Read Only and Texture Memory

The
University
Of
Sheffield.

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

This Lecture (learning objectives)

- ❑Read Only and Texture Memory
  - ❑Identify use cases for read only and texture memory
  - ❑Demonstrate texture memory binding
  - ❑Highlight the simplicity of read on memory usage
  - ❑Extra Material: Demonstrate Bindless Textures



Grid

Block (0, 0)

Registers

Registers

Thread (0, 0)

Thread (1, 0)

Local Cache

Shared Mem

Constant Cache

L1 / Read-only

Block (1, 0)

Registers

Registers

Thread (0, 0)

Thread (1, 0)

Local Cache

Shared Mem

Constant Cache

L1 / Read-only

Host

L2 Cache

GPU DRAM

Global Memory

Constant Memory

Read-only/Texture Memory

Read-only and Texture Memory

- Separate in Kepler but unified with L1 thereafter
    - Same use case but used in different ways

- When to use read-only or texture
    - When data is read only
    - Good for bandwidth limited kernels
    - Regular memory accesses with good locality (think about the way textures are accessed)
    - Texture cache can outperform read only cache for certain scenarios
        - Normalisation/interpolation
        - 2D and 3D loads
    - Read only cache can outperform texture cache
        - Loads of 4 byte values

- Two Methods for utilising Read-only/Texture Memory
    - Bind memory to texture (or use advanced bindless textures in CUDA 5.0+)
    - Hint the compiler to load via read-only cache

Texture Memory Binding

- Known as bound texture (or texture reference method)

```
#define N 1024
texture<float, 1, cudaReadModeElementType> tex;

__global__ void kernel() {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float x = tex1Dfetch(tex, i);
}



int main() {
  float *buffer;
  cudaMalloc(&buffer, N*sizeof(float));
  cudaBindTexture(0, tex, buffer, N*sizeof(float));
  kernel << <grid, block >> >();
  cudaUnbindTexture(tex);
  cudaFree(buffer);
}
```

Texture Memory Binding

- ❑ Known as bound texture (or texture reference method)

```
#define N 1024
texture<float, 1, cudaReadModeElementType> tex;

__global__ void kernel() {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float x = tex1Dfetch(tex, i);
}


int main() {
  float *buffer;
  cudaMalloc(&buffer, N*sizeof(float));
  cudaBindTexture(0, tex, buffer, N*sizeof(float));
  kernel << <grid, block >> >();
  cudaUnbindTexture(tex);
  cudaFree(buffer);
}
```

Must be either;

- ❑ `char`, `short`, `long`, `long long`, `float` or `double`

Vector Equivalents are also permitted e.g.

- ❑ `uchar4`

Texture Memory Binding

- ❑ Known as bound texture (or texture reference method)

```
#define N 1024
texture<float, 1, cudaReadModeElementType> tex;

__global__ void kernel() {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float x = tex1Dfetch(tex, i);
}


int main() {
  float *buffer;
  cudaMalloc(&buffer, N*sizeof(float));
  cudaBindTexture(0, tex, buffer, N*sizeof(float));
  kernel << <grid, block >> >();
  cudaUnbindTexture(tex);
  cudaFree(buffer);
}
```

Dimensionality:

- ❑ `cudaTextureType1D (1)`
- ❑ `cudaTextureType2D (2)`
- ❑ `cudaTextureType3D (3)`
- ❑ `cudaTextureType1DLayered (4)`
- ❑ `cudaTextureType2DLayered (5)`
- ❑ `cudaTextureTypeCubemap (6)`
- ❑ `cudaTextureTypeCubemapLayered (7)`

Texture Memory Binding

❑ Known as bound texture (or texture reference method)

```
#define N 1024
texture<float, 1, cudaReadModeElementType> tex;

__global__ void kernel() {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float x = tex1Dfetch(tex, i);
}



int main() {
  float *buffer;
  cudaMalloc(&buffer, N*sizeof(float));
  cudaBindTexture(0, tex, buffer, N*sizeof(float));
  kernel << <grid, block >> >();
  cudaUnbindTexture(tex);
  cudaFree(buffer);
}
```

Value normalization:

- ❑ cudaReadModeElementType
- ❑ cudaReadModeNormalizedFloat
  - ❑ Normalises values across range

Texture Memory Binding on 2D Arrays

```
#define N 1024
texture<float, 2, cudaReadModeElementType> tex;

__global__ void kernel() {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  float v = tex2D (tex, x, y);
}

int main() {
  float *buffer;
  cudaMalloc(&buffer, W*H*sizeof(float));
  cudaChannelFormatDesc desc = cudaCreateChannelDesc<float>();
  cudaBindTexture2D(0, tex, buffer, desc, W,
                    H, W*sizeof(float));
  kernel << <grid, block >> >();
  cudaUnbindTexture(tex);
  cudaFree(buffer);
}
```

- ❑ Use tex2D rather than tex1Dfetch for CUDA arrays
- ❑ Note that last arg of **cudaBindTexture2D** is pitch
  - ❑ Row size not != total size

Read-only Memory

- No textures required
- Hint to the compiler that the data is read-only without pointer aliasing
  - Using the `const` and `__restrict__` qualifiers
  - Suggests the compiler should use `__ldg` but does not guarantee it
- Not the same as `__constant__`
  - Does not require broadcast reading

```
#define N 1024

__global__ void kernel(float const* __restrict__ buffer) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float x1 = buffer[i];
  float x2 = __ldg(buffer[i]);
}


int main() {
  float *buffer;
  cudaMalloc(&buffer, N*sizeof(float));
  kernel << <grid, block >> >(buffer);
  cudaFree(buffer);
}
```

Probably read through read only cache

Definitely read through read only cache

Summary

❑Read Only and Texture Memory

- ❑Identify use cases for read only and texture memory
- ❑Demonstrate texture memory binding
- ❑Highlight the simplicity of read on memory usage
- ❑Extra Material: Demonstrate Bindless Textures

Acknowledgements and Further Reading

- http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-kepler-texture-objects-improve-performance-and-flexibility/
- Mike Giles (Oxford): Different Memory and Variable Types
  - https://people.maths.ox.ac.uk/gilesm/cuda/
- CUDA Programming Guide
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide/#texture-memory

Bindless Textures (Advanced)

```
#define N 1024

__global__ void kernel(cudaTextureObject_t tex) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float x = tex1Dfetch(tex, i);
}

int main() {
  float *buffer;
  cudaMalloc(&buffer, N*sizeof(float));

  cudaResourceDesc resDesc;
  memset(&resDesc, 0, sizeof(resDesc));
  resDesc.resType = cudaResourceTypeLinear;
  resDesc.res.linear.devPtr = buffer;
  resDesc.res.linear.desc.f = cudaChannelFormatKindFloat;
  resDesc.res.linear.desc.x = 32; // bits per channel
  resDesc.res.linear.sizeInBytes = N*sizeof(float);

  cudaTextureDesc texDesc;
  memset(&texDesc, 0, sizeof(texDesc));
  texDesc.readMode = cudaReadModeElementType;

  cudaTextureObject_t tex;
  cudaCreateTextureObject(&tex, &resDesc, &texDesc, NULL);
  kernel << <grid, block >> >(tex);
  cudaDestroyTextureObject(tex);
  cudaFree(buffer);
}
```

- ❑ Texture Object Approach (Kepler+ and CUDA 5.0+)
- ❑ Textures only need to be created once
  - ❑ No need for binding an unbinding
- ❑ Better performance than binding
  - ❑ Small kernel overhead
- ❑ More details in programming guide
  - ❑ http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#texture-object-api

Address and Filter Modes (Bindless Textures)

- `addressMode`: Dictates what happened when address are out of bounds. E.g.
  - `cudaAddressModeClamp`: in which case addresses out of bounds will be clamped to range
  - `cudaAddressModeWrap`: in which case addressed out of bounds will wrap
- `filterMode`: Allows values read from the texture to be filtered. E.g.
  - `cudaFilterModeLinear`: Linearly interpolates between points
  - `cudaFilterModePoint`: Gives the value at the specific texture point

```
cudaTextureObject_t tex;
cudaCreateTextureObject(&tex, &resDesc, &texDesc, NULL);
tex.addressMode = cudaAddressModeClamp;
```

Bindless Textures

```
texture<float, 1, cudaReadModeElementType> tex;
tex.addressMode = cudaAddressModeClamp;
```

Bound Textures