# Parallel Computing with GPUs

## CUDA Memory
## Part 1 - Memory Overview

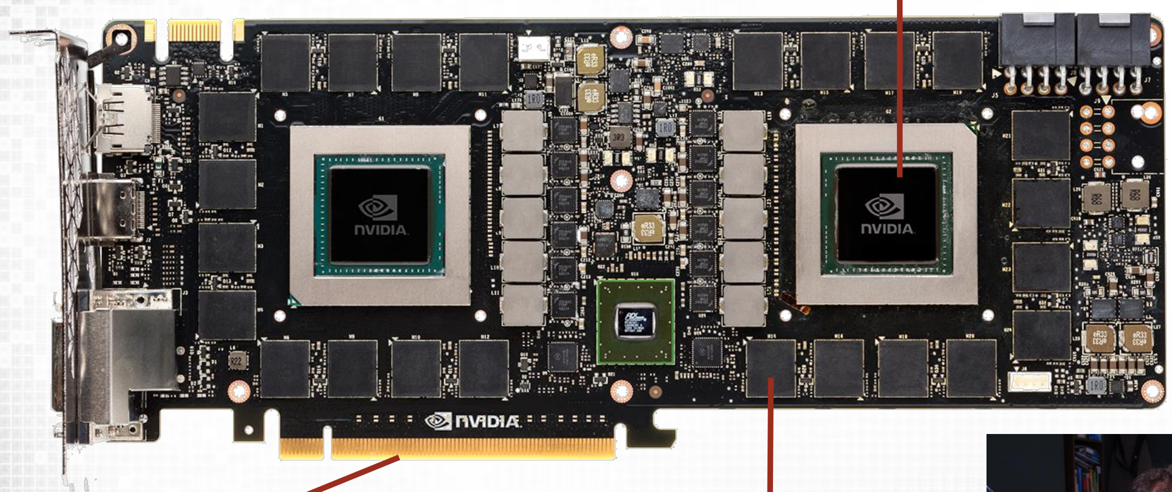
The University Of Sheffield.

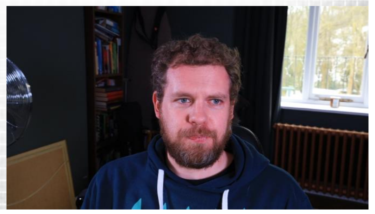
Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

## This Lecture (learning objectives)

- CUDA Memory Overview
  - Present the GPUs memory hierarchy and how this differs between hardware versions
  - Identify where latencies exist memory operations
  - Give an example of how to benchmark a CUDA program

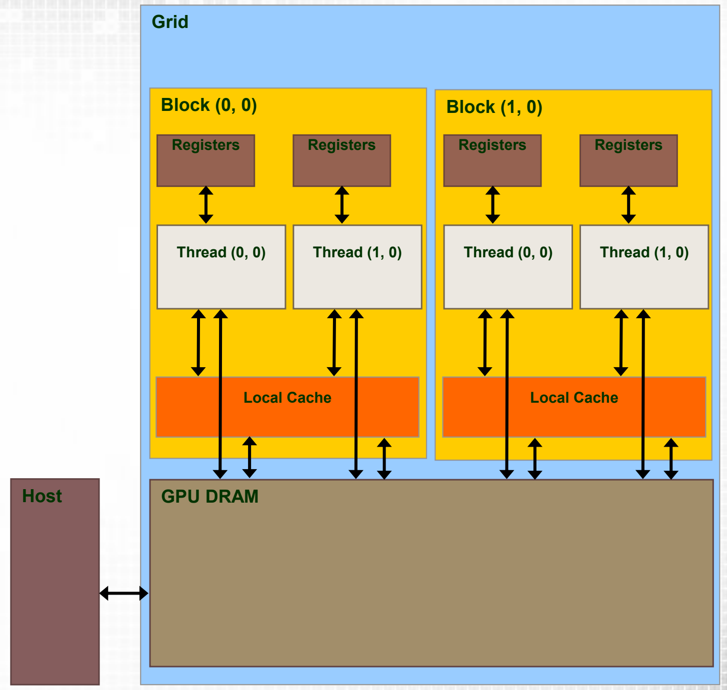
## GPU Memory (GTX Titan Z)

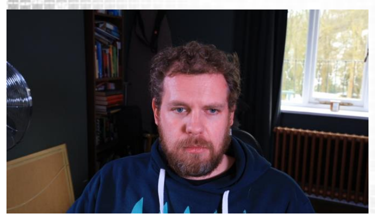


## Simple Memory View

- Threads have access to;
  - **Registers**
    - Read/Write **per thread**
  - **Local memory**
    - Read/Write **per thread**
  - **Local Cache**
    - Read/Write **per block**
  - **Main DRAM Memory**
    - Read/Write **per grid**



## Local Memory

- Local memory (Thread-Local Global Memory)
  - Read/Write per thread
  - Local memory does not physically exist
    - Mapped to reserved area in global memory
    - Usually uses an are of local cache (e.g. L1)
  - Used for variables if you exceed the number of registers available
    - Very bad for performance!
  - Arrays always go in local memory if they are indexed with non constants

```
__global__ void localMemoryExample
(int * input)
{

    int a;
    int b;
    int index;

    int myArray1[4];
    int myArray2[4];
    int myArray3[100];

    index = input[threadIdx.x];
    a = myArray1[0];
    b = myArray2[index];

}
```

non constant index

https://stackoverflow.com/questions/10297067/in-a-cuda-kernel-how-do-i-store-an-array-in-local-thread-memory

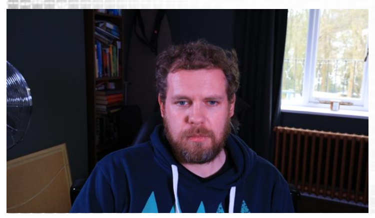
## Memory Latencies

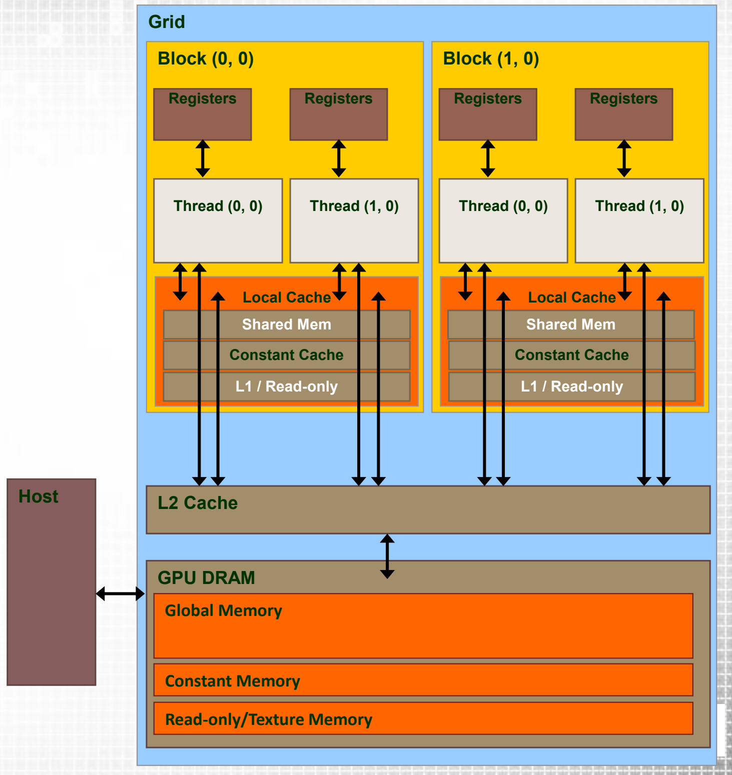
- What is the cost of accessing each area of memory?
  - On chip caches are MUCH lower latency

| | Cost (cycles) |
|---|---|
| Register | 1 |
| Global | 200-800 |
| Shared memory | ~1 |
| L1 | 1 |
| Constant | ~1 (if cached) |
| Read-only (tex) | 1 if cached (same as global if not) |

## Maxwell/Pascal

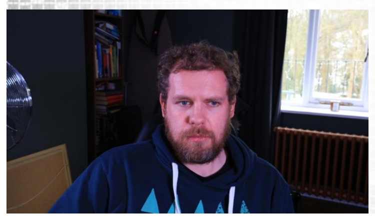
- Each Thread has access to
  - **Registers**
  - **Local memory**
  - Main DRAM Memory via L2 cache
    - Global Memory
      - Can be read via Unified Data Cache
    - Constant Memory
      - Via **L2 cache** and per **block Constant cache**
    - Unified L1/Texture Cache
      - Same cache used for read only or texture reads
    - Dedicated Shared Memory
      - User configurable cache

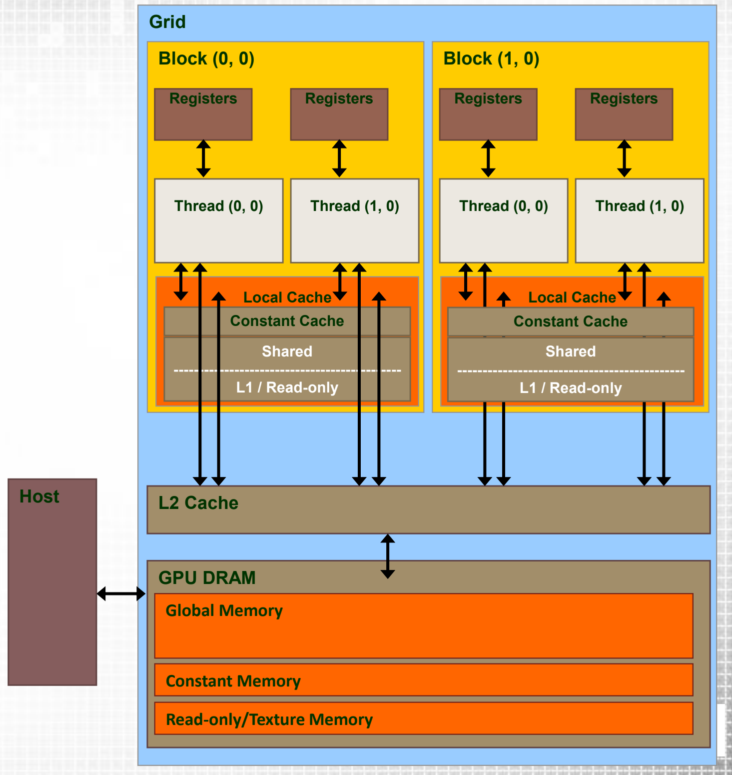





## Volta

- Each Thread has access to
  - Registers
  - Local memory
  - Main DRAM Memory via L2 cache
    - Global Memory
      - Can be read via Unified Data Cache
    - Constant Memory
      - Via L2 cache and per block Constant cache
    - **Unified Shared/L1/Texture Cache**
      - Same cache used for read only or texture reads
      - Amount of shared memory configurable at runtime



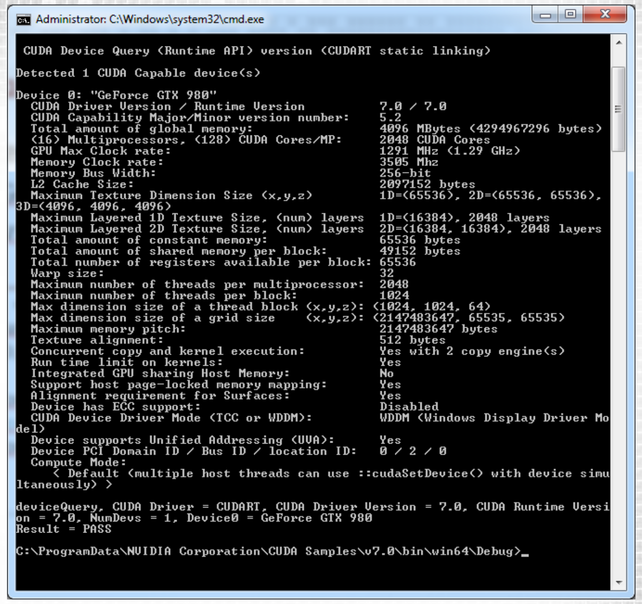


## Cache and Memory Sizes

| | Pascal (P100) GP100 | Volta (V100) GV100 |
|---|---|---|
| **Register File Size** | *256KB per SM* | *256KB per SM* |
| **Shared Memory** | *64KB Dedicated* | *Configurable up to 96KB* |
| **Constant Memory** | *64KB DRAM 8KB Cache per SM* | *64KB DRAM 8KB Cache per SM* |
| **L1/Read Only Memory** | *24KB per SM Dedciated* | *Configurable up to 128KB per SM* |
| **L2 Cache Size** | *4096KB* | *6144KB* |
| **Device Memory** | *16GB* | *16GB* |
| **DRAM Interface** | *4096-bit HBM2* | *4096-bit HBM2* |

https://devblogs.nvidia.com/inside-volta/

## Device Query




- ❑What are the specifics of my GPU?
  - ❑Use `cudaGetDeviceProperties`
  - ❑E.g.
    - ❑`deviceProp.sharedMemPerBlock`
  - ❑CUDA SDK deviceQry example

```
int deviceCount = 0;
cudaGetDeviceCount(&deviceCount);
for (int dev = 0; dev < deviceCount; ++dev)
{
    cudaSetDevice(dev);
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, dev);
    …
}
```

## Performance Measurements

- ❑How can we benchmark our CUDA code?
- ❑Kernel Calls are asynchronous
  - ❑If we use a standard CPU timer it will measure only launch time not execution time
  - ❑We could call `cudaDeviceSynchronise()` but this will stall the entire GPU pipeline
- ❑Alternative: CUDA Events
  - ❑Events are created with `cudaEventCreate()`
  - ❑Timestamps can be set using `cudaEventRecord()`
  - ❑`cudaEventElapsedTime()` sets the time in *ms* between the two events.

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
my_kernel <<<(N /TPB), TPB >>>();
cudaEventRecord(stop);

cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds,
                     start, stop);

cudaEventDestroy(start);
cudaEventDestroy(stop);
```

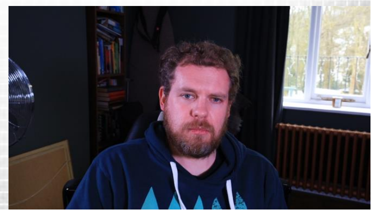
## CUDA qualifiers summary

- ❑Where can a variable be accessed?
  - ❑Is declared inside the kernel?
    - ❑Then the host can not access it
    - ❑Lifespan ends after kernel execution
  - ❑Is declared outside the kernel
    - ❑Then the host can access it (via `cudaMemcpyToSymbol`)

```
__device__ int my_global;


__global__ void my_kernel() {
  int my_local;

  int *ptr1 = &my_global;
  int *ptr2 = &my_local;
}
```

- ❑What about pointers?
  - ❑They can point to anything
  - ❑BUT are <u>not</u> typed on memory space
  - ❑Be careful not to confuse the compiler

```
if (something)
  ptr1 = &my_global;
else
  ptr1 = &my_local;
```



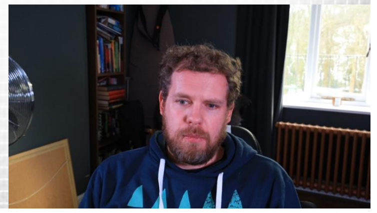
## Summary

- ❑CUDA Memory Overview
  - ❑Present the GPUs memory hierarchy and how this differs between hardware versions
  - ❑Identify where latencies exist memory operations
  - ❑Give an example of how to benchmark a CUDA program
- ❑Next Lecture: Global and Constant Memory

# Parallel Computing with GPUs

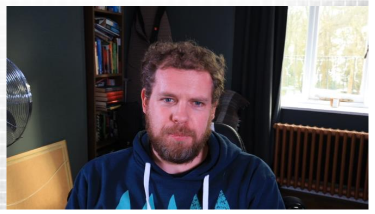
# CUDA Memory Part 2 – Global and Constant Memory

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



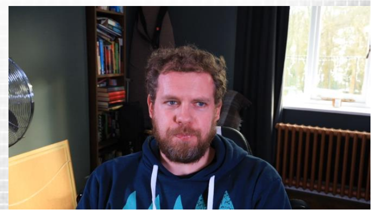
## This Lecture (learning objectives)

- ❑Global and Constant Memory
  - ❑Compare and contrast manual memory movement with Unified Memory
  - ❑Identify the use cases for constant memory
  - ❑Demonstrate an appropriate use of constant memory

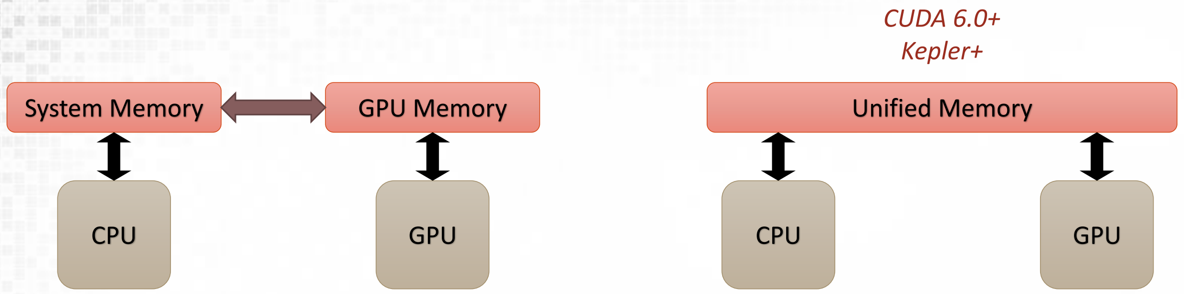


## Dynamic vs Static Global Memory

- ❑In the previous lab we dynamically defined GPU memory
  - ❑Using `cudaMalloc()`
- ❑You can also statically define (and allocate) GPU global memory
  - ❑Using `__device__` qualifier
  - ❑Requires memory copies are performed using `cudaMemcpyToSymbol` or `cudaMemcpyFromSymbol`
  - ❑See example from last weeks lecture
- ❑This is the equivalent of the following in C (host code)
  - ❑`int my_static_array[1024];`
  - ❑`int *my_dynamic_array = (int*) malloc(1024*sizeof(int));`



# Unified Memory




- So far the developer view is that GPU and CPU have separate memory
  - Memory must be explicitly copied
  - Deep copies required for complex data structures
- Unified Memory changes that view

# Unified Memory Example

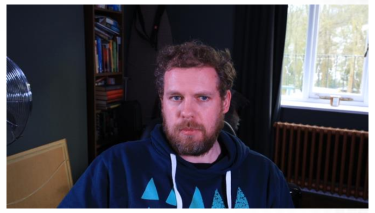
C Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA (6.0+) Code

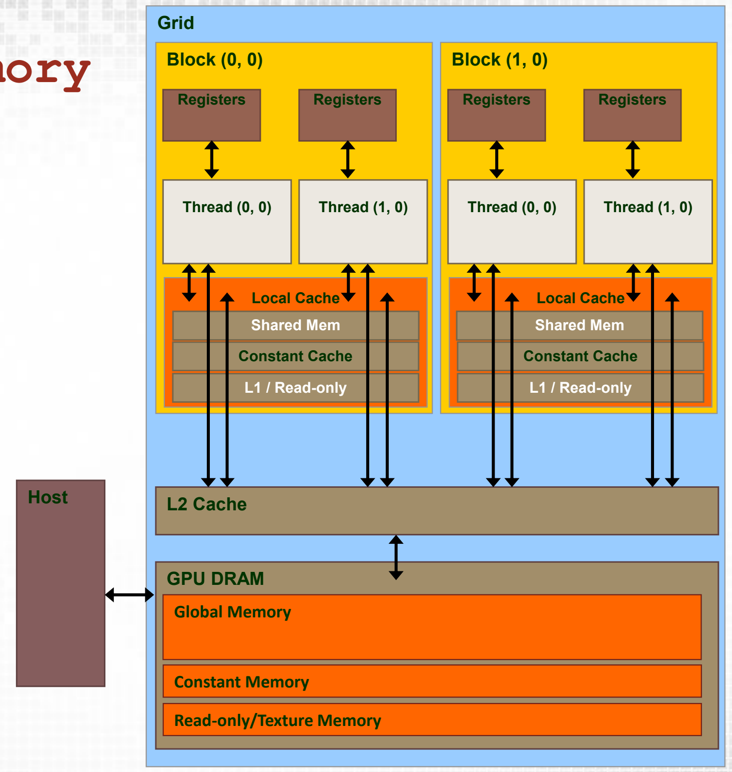
```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  gpu_qsort(data, N, 1, compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

# Implications of CUDA Unified Managed Memory



- Simpler porting of code
- Memory is only *virtually* unified
  - GPU still has discrete memory
  - It still has to be transferred via PCIe (or NVLINK)
- Easier management of data to and from the device
  - Explicit memory movement is not required
  - Similar to the way the OS handles virtual memory
- Issues
  - Requires look ahead and paging to ensure memory is in the correct place (and synchronised)
  - It is not as fast as hand tuned code which has finer explicit control over transfers
- *We will manage memory movement ourselves!*

# Constant Memory




# Constant Memory

- Constant Memory
  - Stored in the devices global memory
  - Read through the per SM constant cache
  - Set at runtime
  - When using correctly only 1/16 of the traffic compared to global loads
- When to use it?
  - When small amounts of data are **read only**
  - When values are **broadcast** to threads in a half warp (of 16 threads)
  - Very fast when cache hit
  - Very slow when no cache hit
- How to use
  - Must be **statically** (compile-time) defined as a symbol using `__co` qualifier
  - Value(s) must be copied using **cudaMemcpytoSymbol**.

# Constant Memory Broadcast

- .... When values are **broadcast** to threads in a half warp (groups of 16 threads)

```
__constant__ int my_const[16];

__global__ void vectorAdd() {
int i = blockIdx.x;

int value = my_const[i % 16];
}
```

```
__constant__ int my_const[16];

__global__ void vectorAdd() {
int i = blockIdx.x * blockDim.x + threadIdx.x;

int value = my_const[i % 16];
}
```

*Which is good use of constant memory?*

# Constant Memory Broadcast

```
__constant__ int my_const[16];

__global__ void constant_test() {
int i = blockIdx.x;

int value = my_const[i % 16];
}
```

```
__constant__ int my_const[16];

__global__ void constant_test() {
int i = blockIdx.x * blockDim.x + threadIdx.x;

int value = my_const[i % 16];
}
```

*Which is good use of constant memory?*

- Best possible use of constant memory
- Every thread in half warp reads the same
  - Index based on `blockIdx`
- No serialisation
  - 1 read request for every thread!
- Other threads in the block will also hit cache

- Worst possible use of constant memory
- Every thread in half warp reads different value
  - Index based on `threadIdx`
- Each access will be serialised
  - 16 different read requests!
- Other threads in the block will likely miss the cache

# Constant Memory

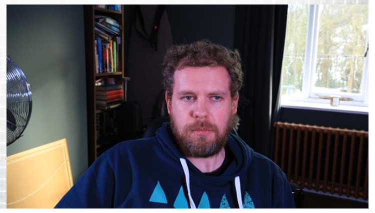
- Question: Should I convert `#define` to constants?
  - E.g. `#define MY_CONST 1234`
- Answer: No
  - Leave alone
  - `#defines` are embed in the code by pre-processors
  - They don't take up registers as they are embed within the instruction space
    - i.e. are replaced with literals by the pre-processor

- Only replace constants that may change at runtime (but not during the GPU programs)

## Summary

- ❑Global and Constant Memory
  - ❑Compare and contrast manual memory movement with Unified Memory
  - ❑Identify the use cases for constant memory
  - ❑Demonstrate an appropriate use of constant memory

- ❑Next Lecture: Read Only and Texture Memory

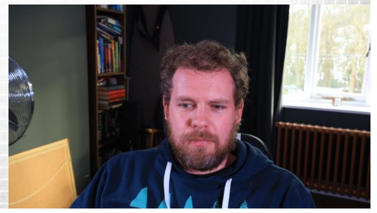


# Parallel Computing with GPUs

## CUDA Memory
## Part 3 – Read Only and Texture Memory

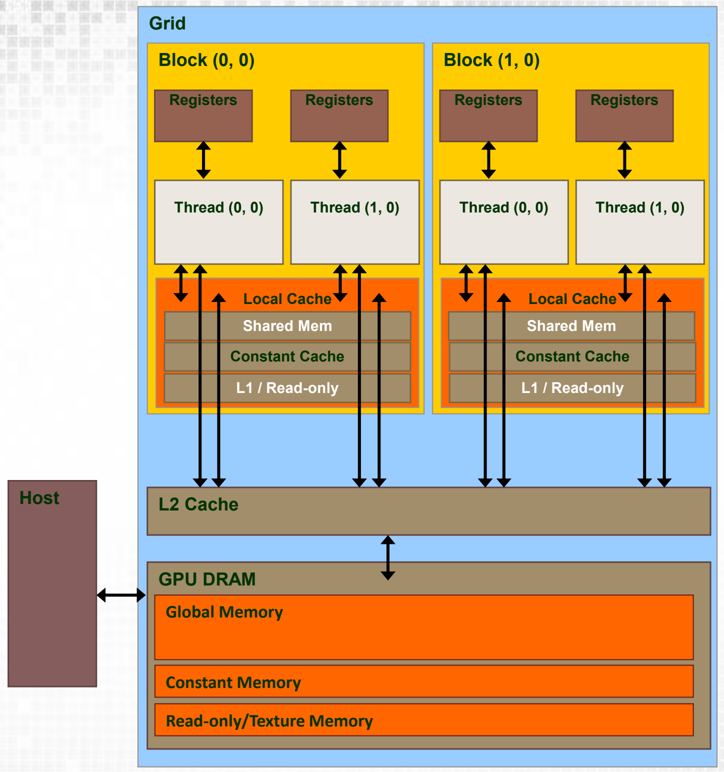
The University Of Sheffield.

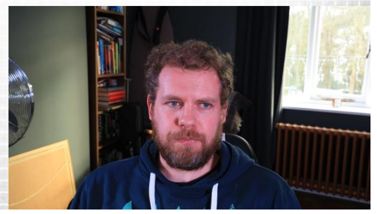
Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/



## 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



# 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);
}
```

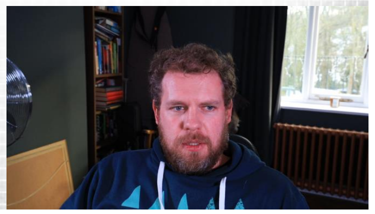
# Texture Memory Binding

- ❑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