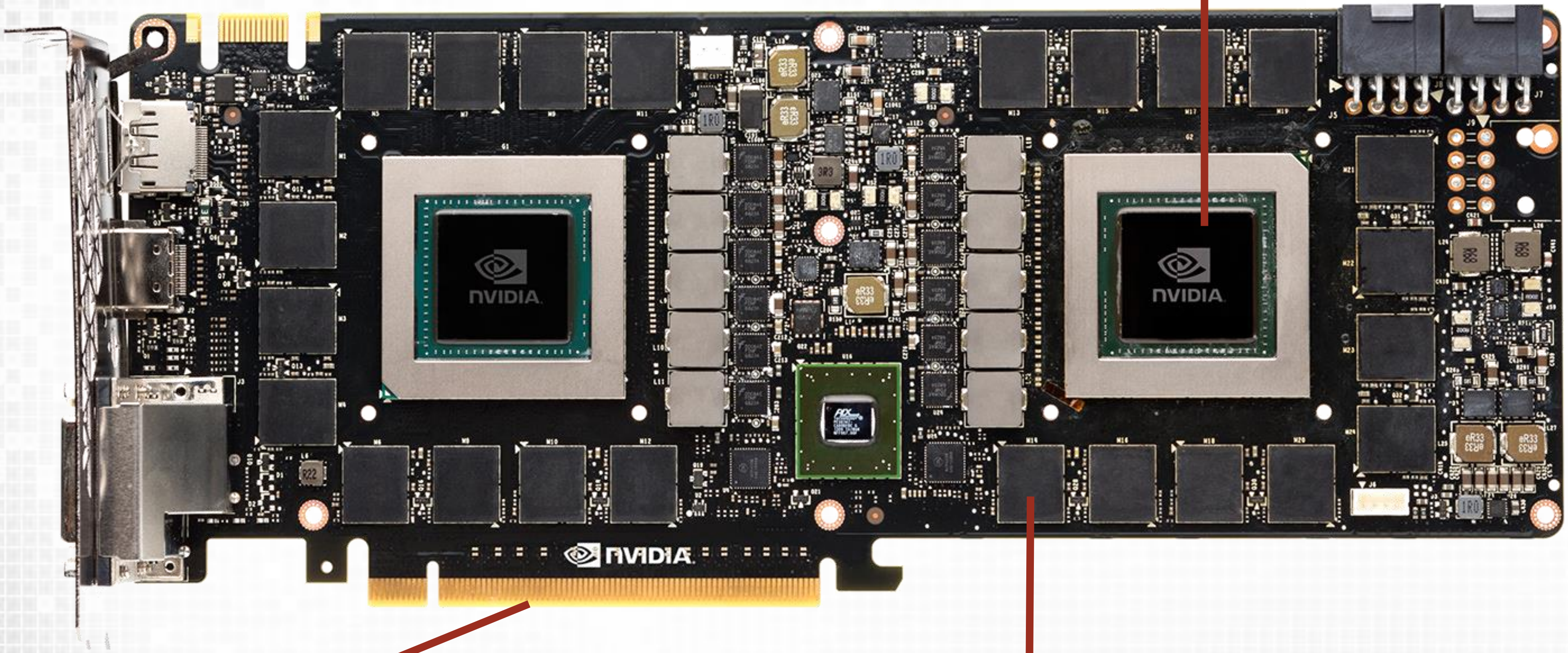
Parallel Computing with GPUs: CUDA Memory

Dr Mozhgan Kabiri Chimeh
http://mkchimeh.staff.shef.ac.uk/teaching/COM4521

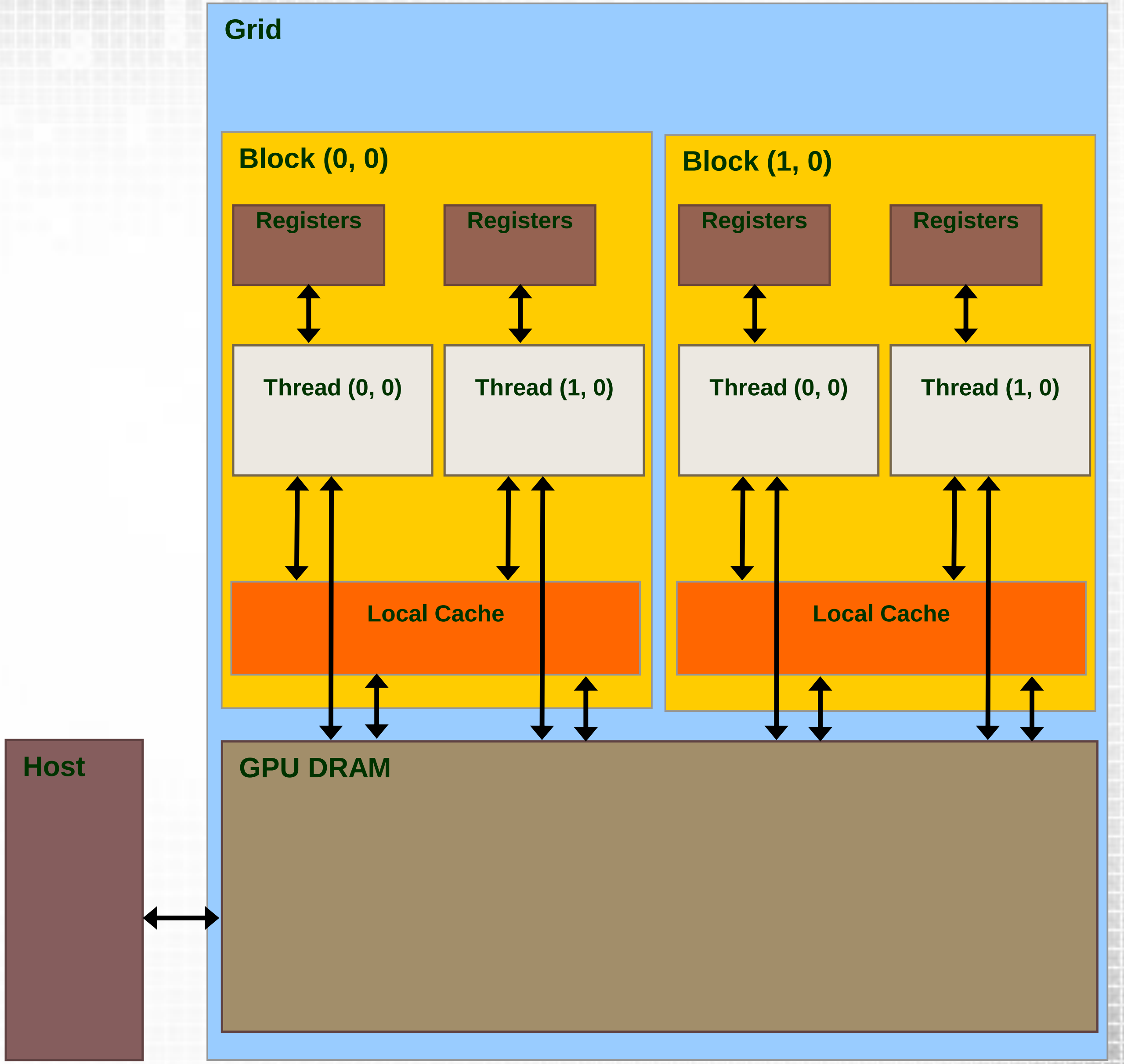
- Memory Hierarchy Overview
- Global Memory
- Constant Memory
- Texture and Read-only Memory
- Roundup & Performance Timing

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
  - Does not physically exist (reserved area in global memory)
  - Cached locally
  - Used for variables if you exceed the number of registers available
    - Very bad for perf!
  - Arrays go in local memory if they are indexed with non constants

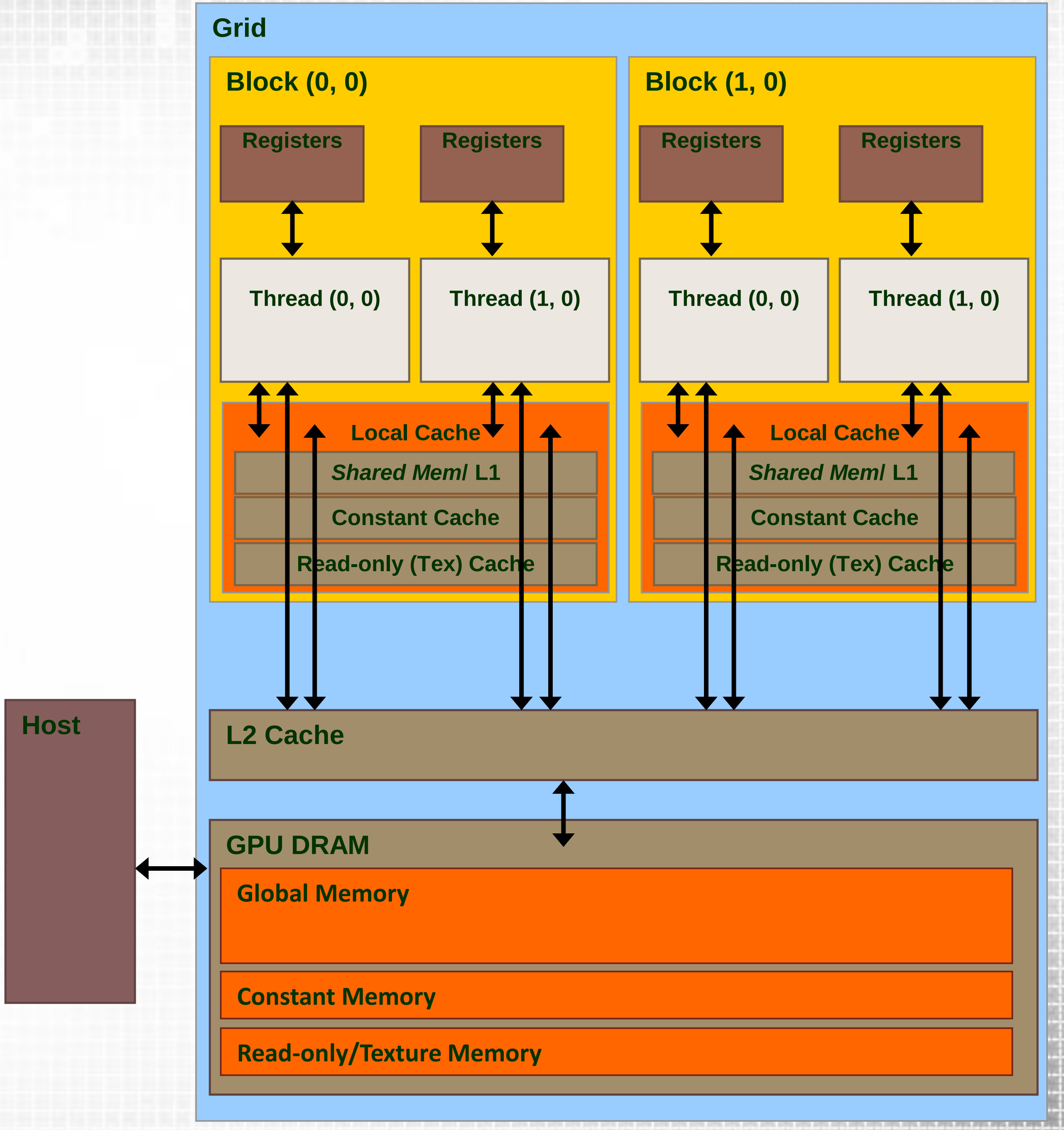
```
__global__ void localMemoryExample
(int * input)
{

    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

Kepler Memory View

- ❑Each Thread has access to
  - ❑**Registers**
  - ❑**Local memory**
  - ❑Main DRAM Memory via cache
    - ❑Global Memory
      - ❑Via **L2 cache** and configurable per block **Shared Memory cache**
    - ❑Constant Memory
      - ❑Via **L2 cache** and per **block Constant cache**
    - ❑Read-only/Texture Memory
      - ❑Via **L2 cache** and per block **Read-only cache**



Kepler and Fermi

Memory Latencies

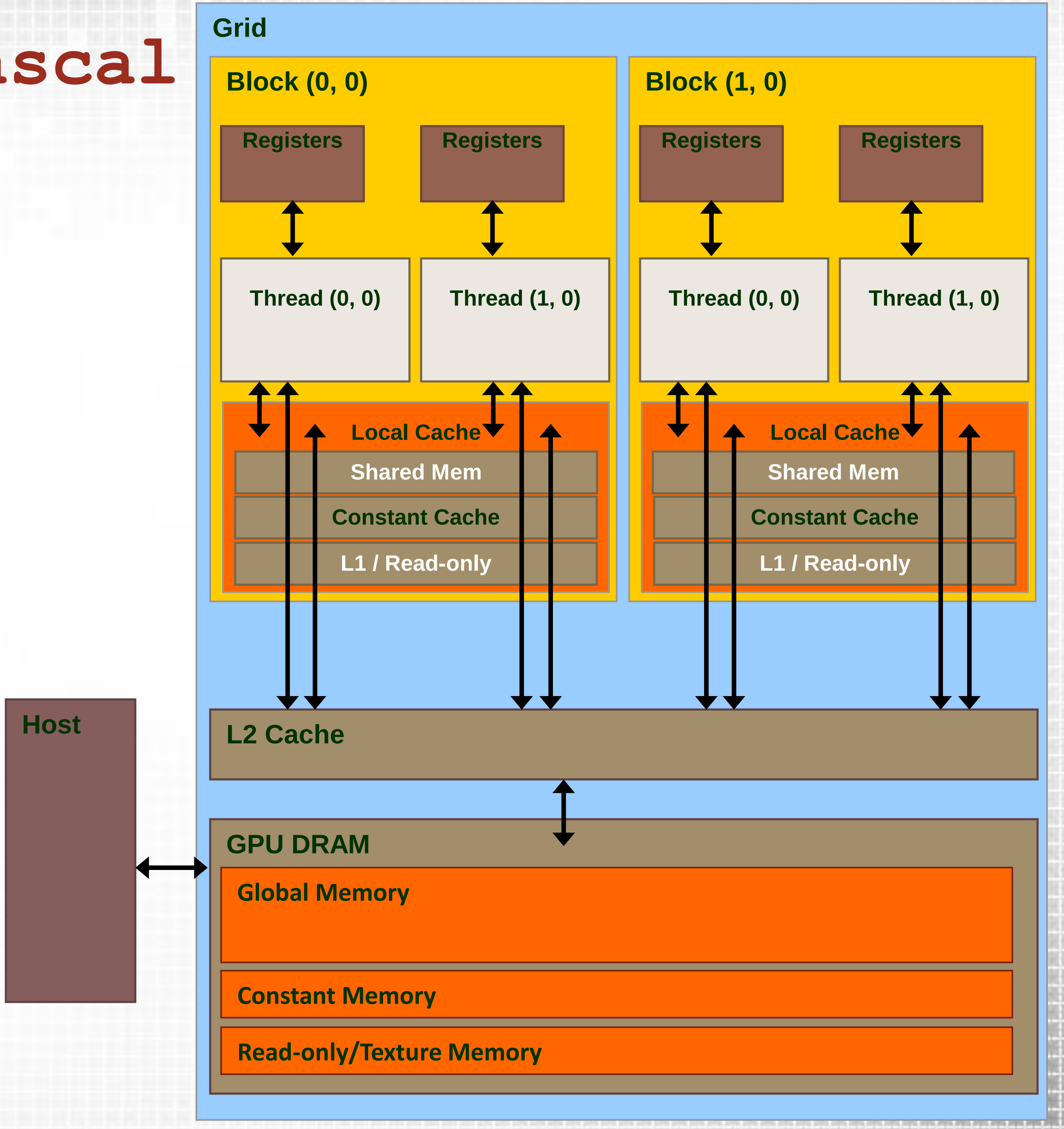
- What is the cost of accessing each area of memory?
  - On chip caches are MUCH lower latency

	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)

The University Of Sheffield. NVIDIA GPU RESEARCH CENTER

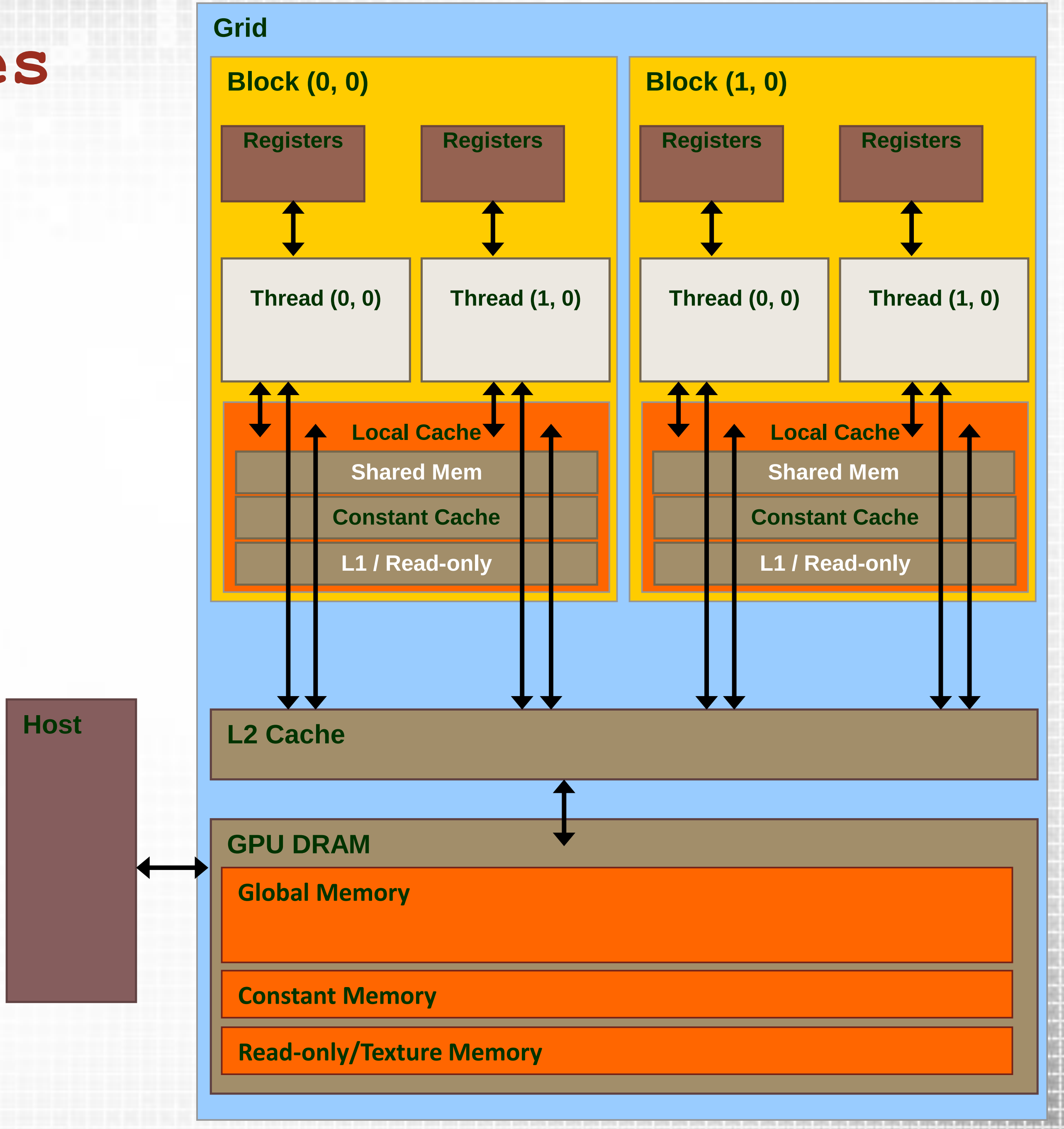
Changes in Maxwell/Pascal

- Shared memory has dedicated Cache
  - No longer shared with L1 as in Fermi and Kepler
- Read-only (texture) cache unified with L1



Cache and Memory Sizes

	Kepler	Maxwell
Registers	*64k 32 bit registers per SM*	*64k 32 bit registers per SM*
Max Registers / thread	*63*	*255*
Shared Memory	*16KB / 48KB Configurable SM and L1*	*64KB Dedicated*
Constant Memory	*64KB DRAM 8KB Cache per SM*	*64KB DRAM 8KB Cache per SM*
Read Only Memory	*48KB per SM*	*48KB per SM Shared with L1*
Device Memory	*Varying 12GB Max*	*Varying 12GB Max*



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);
    …
}
```

Administrator: C:\Windows\system32\cmd.exe

```
 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 980"
  CUDA Driver Version / Runtime Version          7.0 / 7.0
  CUDA Capability Major/Minor version number:    5.2
  Total amount of global memory:                 4096 MBytes (4294967296 bytes)
  (16) Multiprocessors, (128) CUDA Cores/MP:     2048 CUDA Cores
  GPU Max Clock rate:                            1291 MHz (1.29 GHz)
  Memory Clock rate:                             3505 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536),
3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Mo
del)
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 2 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simu
ltaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 7.0, CUDA Runtime Versi
on = 7.0, NumDevs = 1, Device0 = GeForce GTX 980
Result = PASS

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v7.0\bin\win64\Debug>_
```

- ❑ Memory Hierarchy Overview
- ❑ Global Memory
- ❑ Constant Memory
- ❑ Texture and Read-only Memory
- ❑ Roundup & Performance Timing

Dynamic vs Static Global Memory

- In the previous lecture we learned about 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 difference between the following in C
  - `int my_static_array[1024];`
  - `int *my_dynamic_array = (int*) malloc(1024*sizeof(int));`

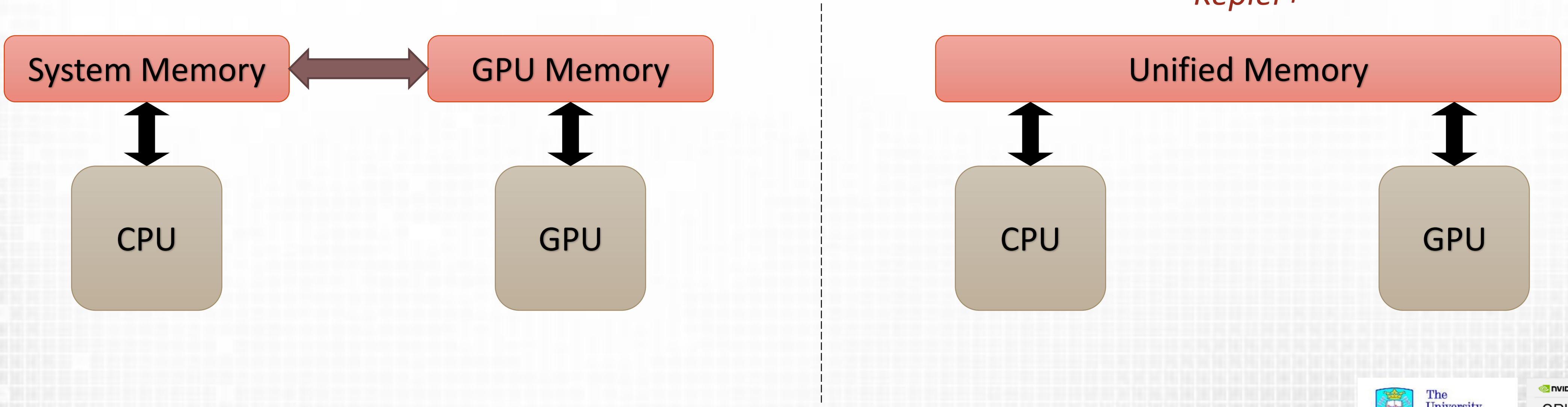
The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

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

CUDA 6.0+
Kepler+



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);
}
```

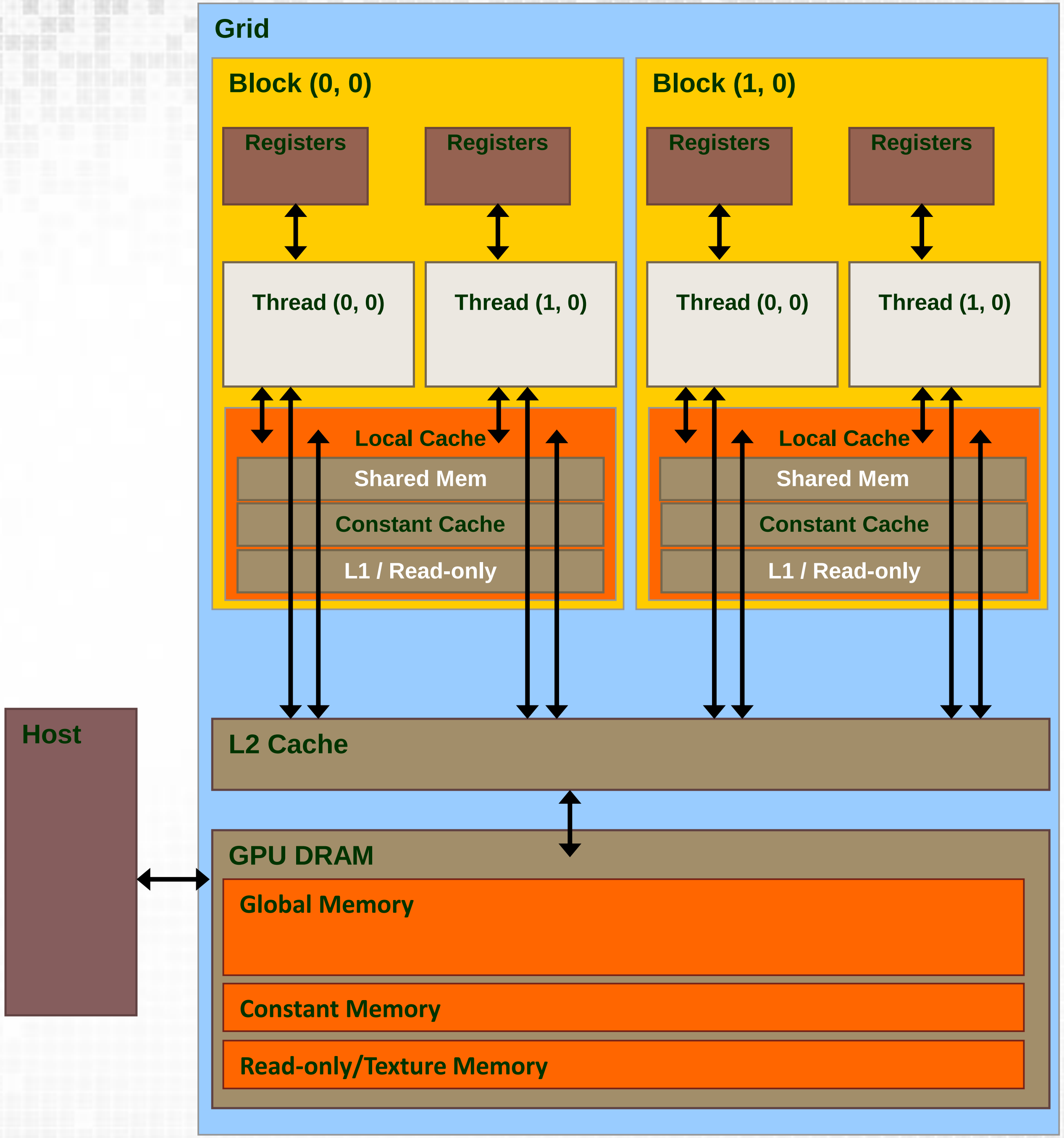
CUDA (6.0+) Code

```
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!*

- Memory Hierarchy Overview
- Global Memory
- Constant Memory
- Texture and Read-only Memory
- Roundup & Performance Timing


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

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 `__constant__` 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

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

```
__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)

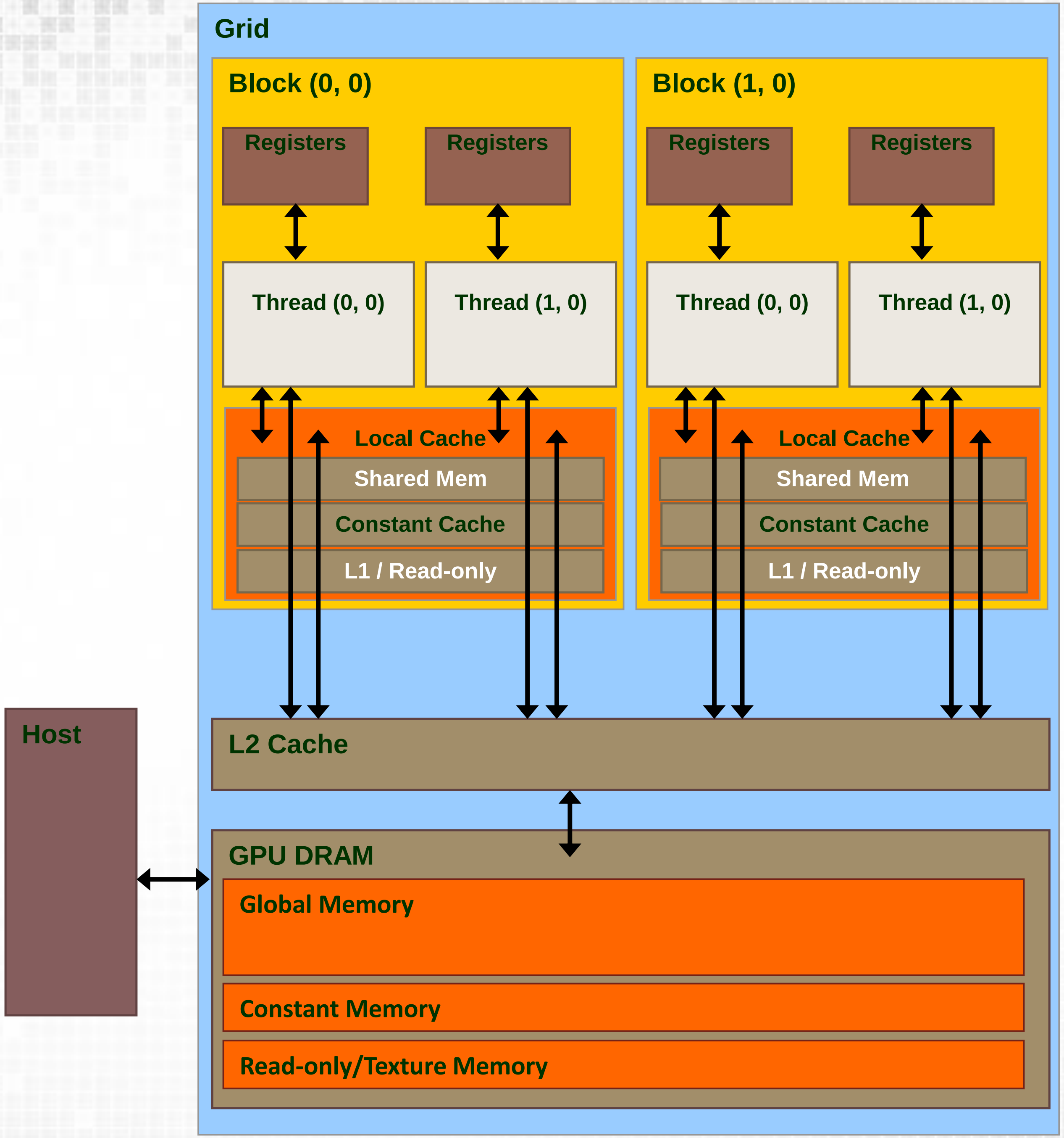
The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

- ❑Memory Hierarchy Overview
- ❑Global Memory
- ❑Constant Memory
- ❑Texture and Read-only Memory
- ❑Roundup & Performance Timing

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER


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
L2 Cache
Host
GPU DRAM
Global Memory
Constant Memory
Read-only/Texture Memory

Read-only and Texture Memory

- Separate in Kepler but unified 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)

- 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 *d_buffer;
  cudaMalloc(&d_buffer, N*sizeof(float));
  cudaBindTexture(0, tex, d_buffer, N*sizeof(float));
  kernel << <grid, block >> >();
  cudaUnbindTexture(tex);
  cudaFree(d_buffer);
}
```

The University Of Sheffield. NVIDIA GPU RESEARCH CENTER

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 x = __ldg(buffer[i]);
}


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

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

- ❑Memory Hierarchy Overview
- ❑Global Memory
- ❑Constant Memory
- ❑Texture and Read-only Memory
- ❑Roundup & Performance Timing

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`)

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

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

Remember!

`const int *p != int * const p`

```
__device__ int my_global;
__constant__ int my_constant;

__global__ void my_kernel() {
  int my_local;

  int *ptr1 = &my_global;
  int *ptr2 = &my_local;
  const int *ptr3 = &my_constant;
}
```

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);
```

Summary

- The CUDA Memory Hierarchy varies between hardware generations
- Utilisation of local caches can have a big impact on the expected performance (1 cycle vs. 100s)
- Global memory can be declared statically or dynamically
- Constant cache good for small read only data accessed in broadcast by *nearby* threads
- Read-Only cache is larger than constant cache but does not have broadcast performance of constant cache
- Kernel variables are not available outside of the kernel

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

Mole Quiz – March 5th

- **NO lectures for next 2 weeks**
- **Labs will be as usual**
- **2 Weeks from NOW (17:00) will be a MOLE quiz.**
  - **Where**? Alfred Denny PC Room (ADB-A04)
  - **When**? March 5th
  - **How Long**: 45 mins (25 Questions)
  - **What**? Everything up to the end of last weeks lectures…

- E.g.

```
int a[5] = {1,2,3,4,5};
x = &a[3];
```

  - What is `x`?
    1. a pointer to an integer with value of 3
    2. a pointer to an integer with value of 4
    3. a pointer to an integer with a value of the address of the third element of `a`
    4. an integer with a value of 4