Parallel Computing with GPUs: CUDA

Dr Paul Ric

http://paulrichmond.shef.ac COM4521/

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

Previous Lecture and Lab

- We started developing some CUDA programs
- We had to move data from the host to the device memory
- We learnt about mapping problems to grids of thread blocks and how to index data

- ❑Memory Hierarchy Overview
- ❑Global Memory
- ❑Constant Memory
- ❑Texture and Read-onl
- ❑Roundup & Performa

GPU Memory (GTX Titan Z)

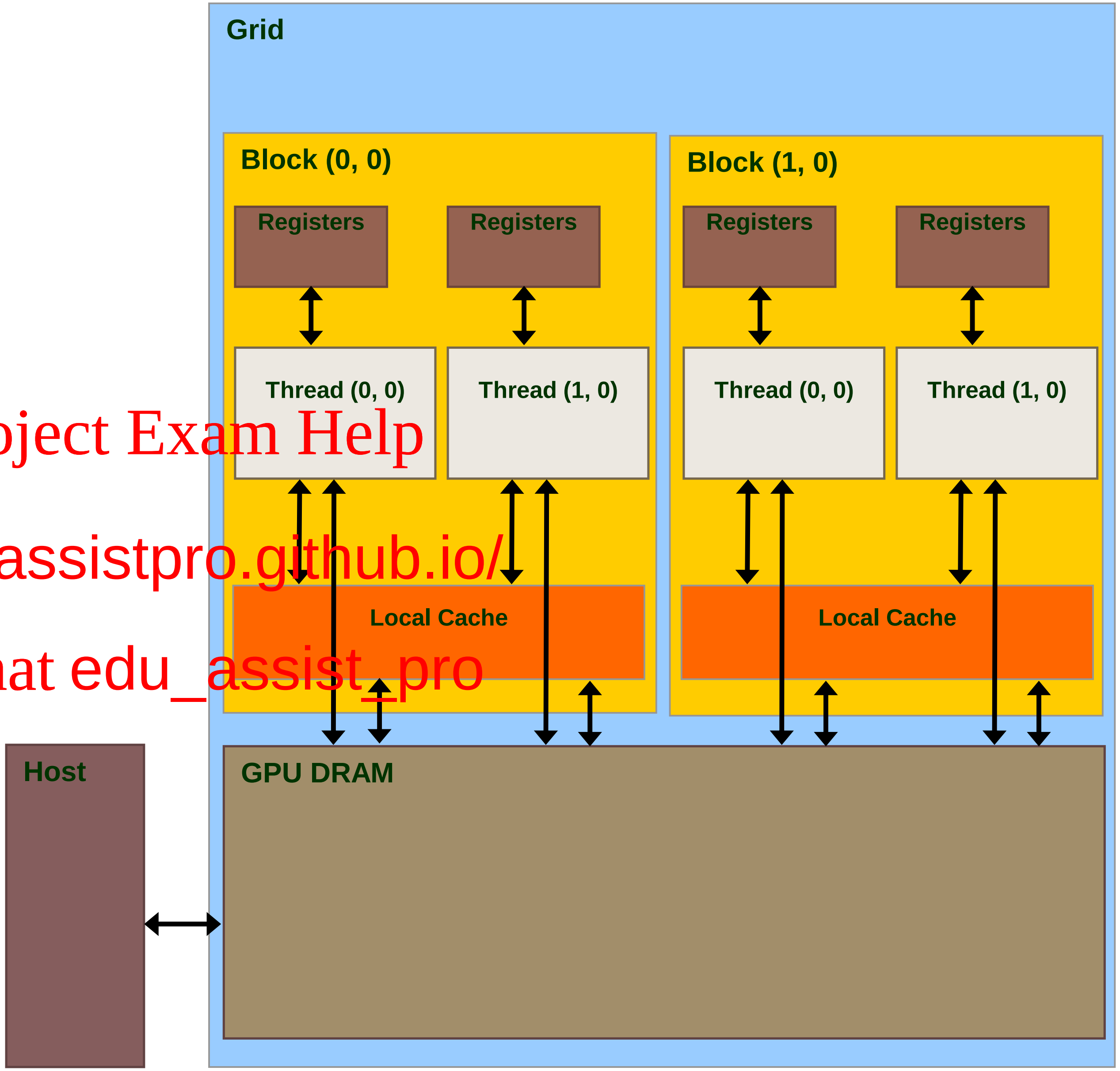
Shared Memory, cache and registers

Host Memory (via PCIe)

GPU DRAM Memory

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 glob
  - 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

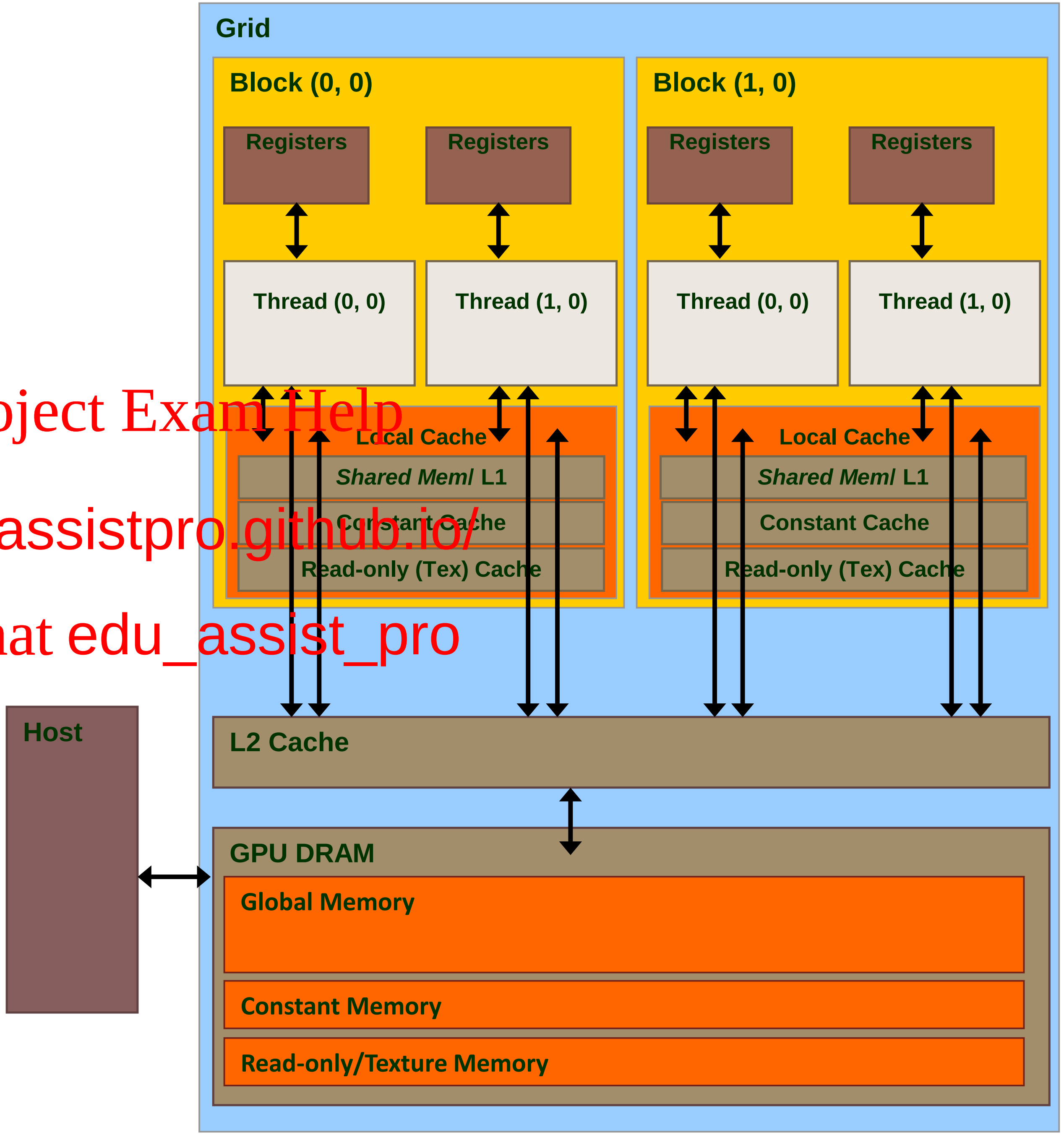
```
__global__ void localMemoryExample
(int * input)
{

    int a;
    int b;
    int index;

              yArray1[4];
              yArray2[4];
               rray3[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 co
        per block **Shared M**
    - 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

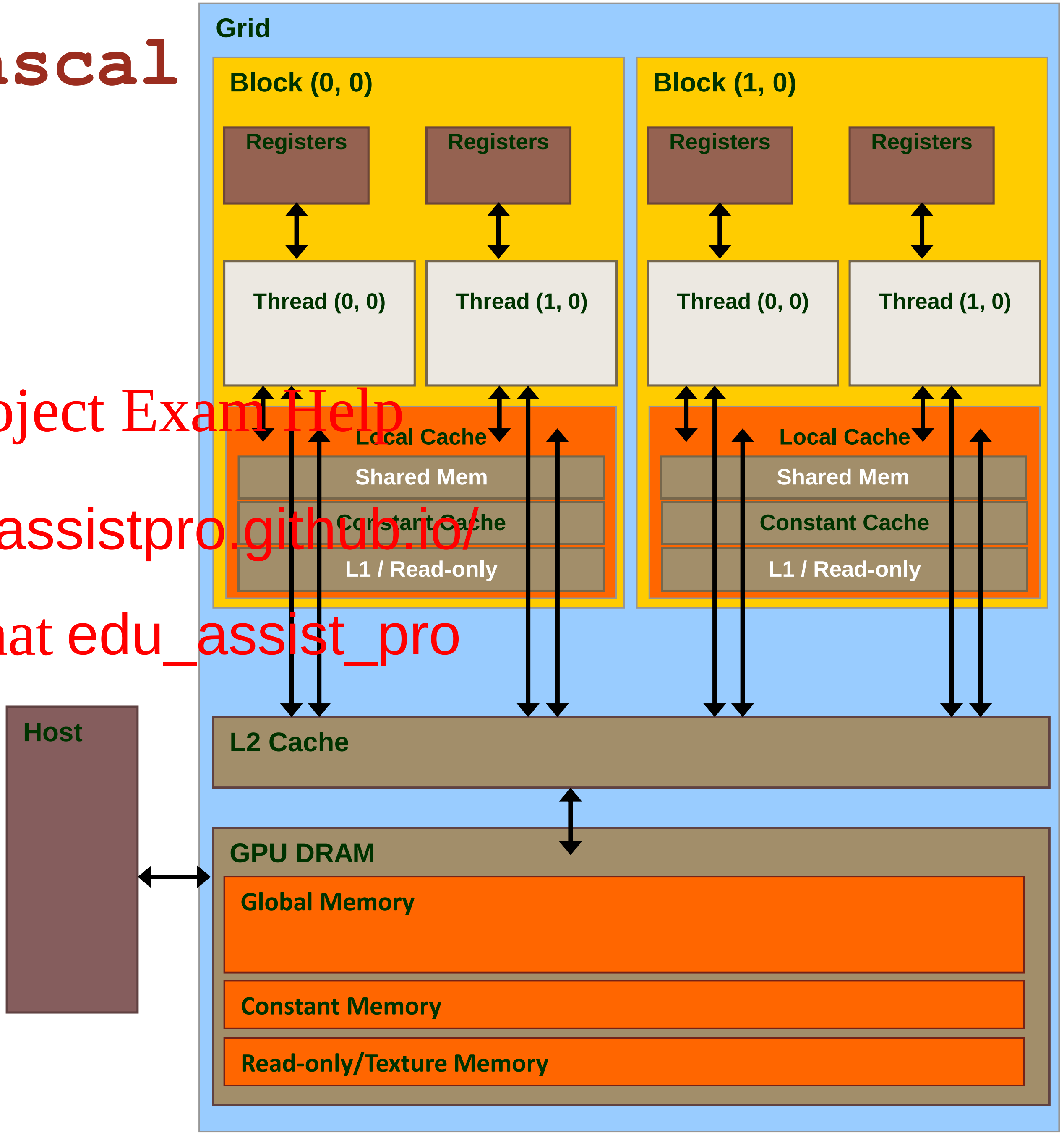
Memory Latencies

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

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

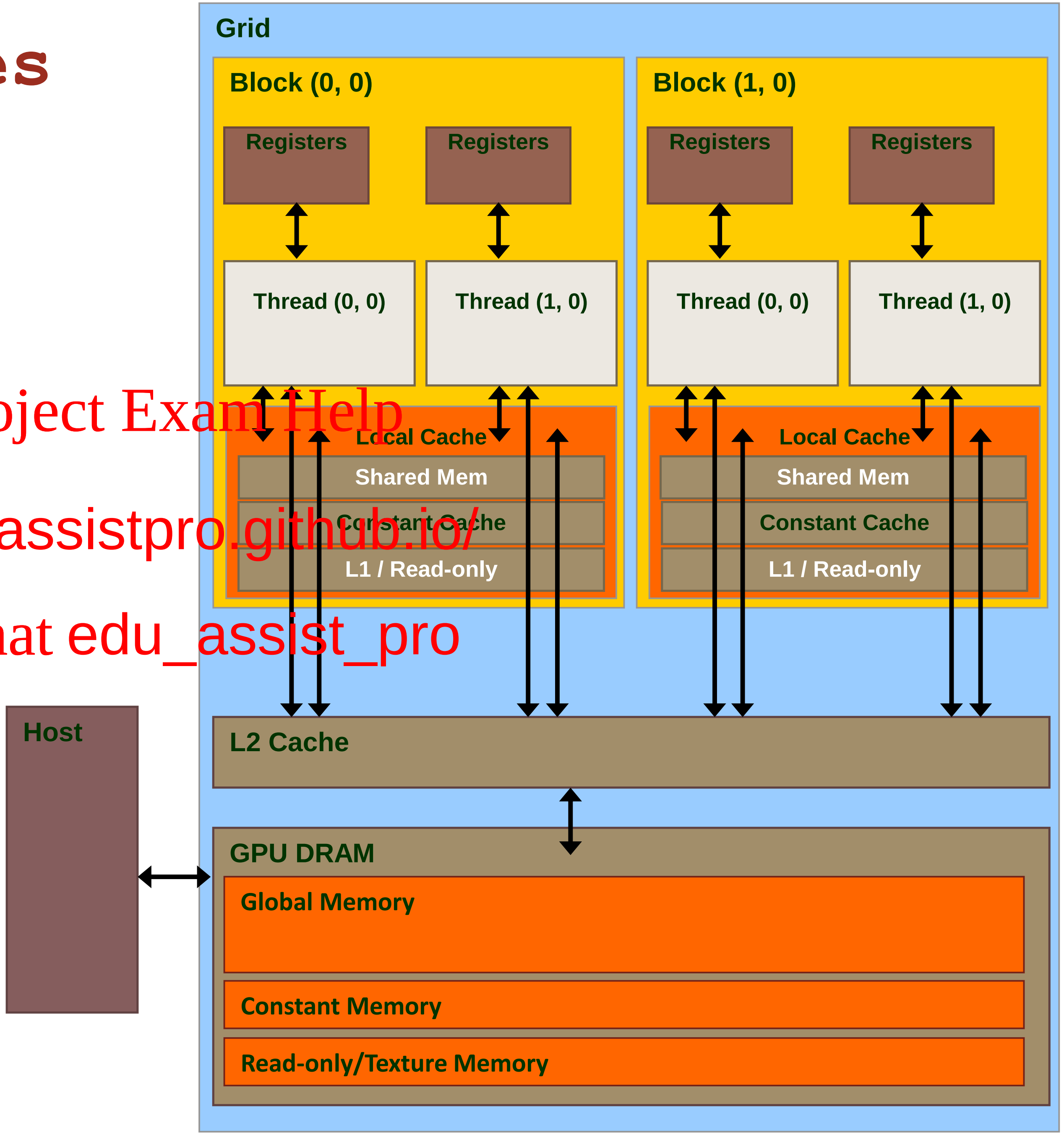
Changes in Maxwell/Pascal

- ❑Shared memory has dedicated Cache
  - ❑No longer shared with L1 as in Fermi and Kepler
- ❑Read-only (texture) ca 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 De*
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


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

- ❑Memory Hierarchy Overview
- ❑Global Memory
- ❑Constant Memory
- ❑Texture and Read-onl
- ❑Roundup & Performa

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 cop `udaMemcpyToSymbol` or `cudaMemcpyFromS`

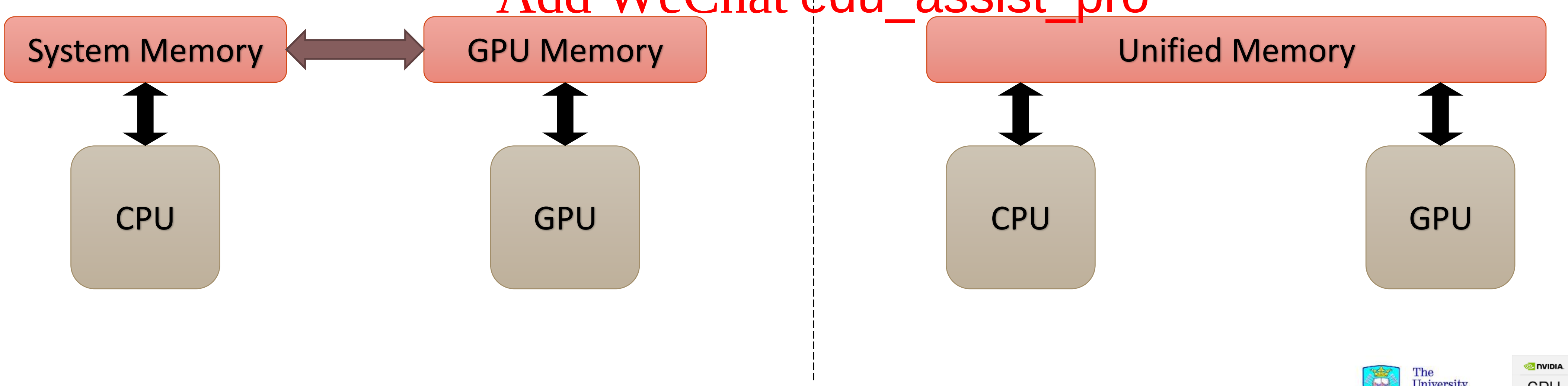
  - See example from last weeks lecture

- This is the difference between the following in C
  - `int my_static_array[1024];`
  - `int *my_dynamic_array = (int*) malloc(1024*sizeof(int));`

Unified Memory

- So far the developer view is that GPU and CPU have separate memory
  - Memory must be explicitly copied
  - Deep copies required es
- Unified Memory chan

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);
}
```

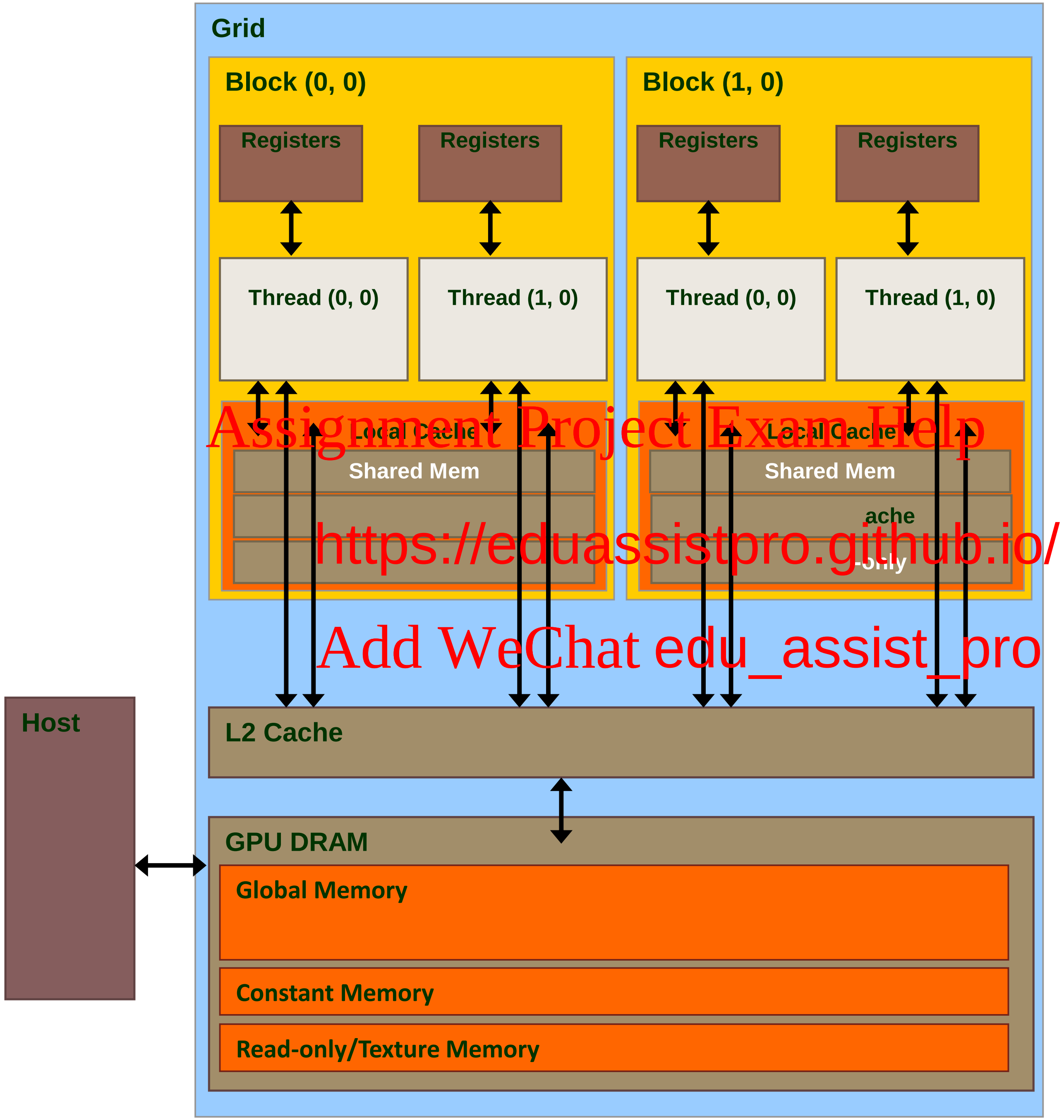
CUDA (6.0+) Code

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

          ata, 1, N, fp);

          (data, N, 1, compare);
          eSynchronize();

  use_data(data);

  free(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 vice
  - Explicit memory movem
  - Similar to the way the OS handles virtual
- 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-onl
- ❑Roundup & Performa


Grid
Block (0, 0)
Block (1, 0)
Registers
Registers
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Thread (0, 0)
Thread (1, 0)
Local Cache
Shared Mem
Local Cache
Shared Mem
ache
-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

  - When values are **broadcast** to threads in f 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;

            my_const[i % 16];
```


Which is good use of cons

The University Of Sheffield.

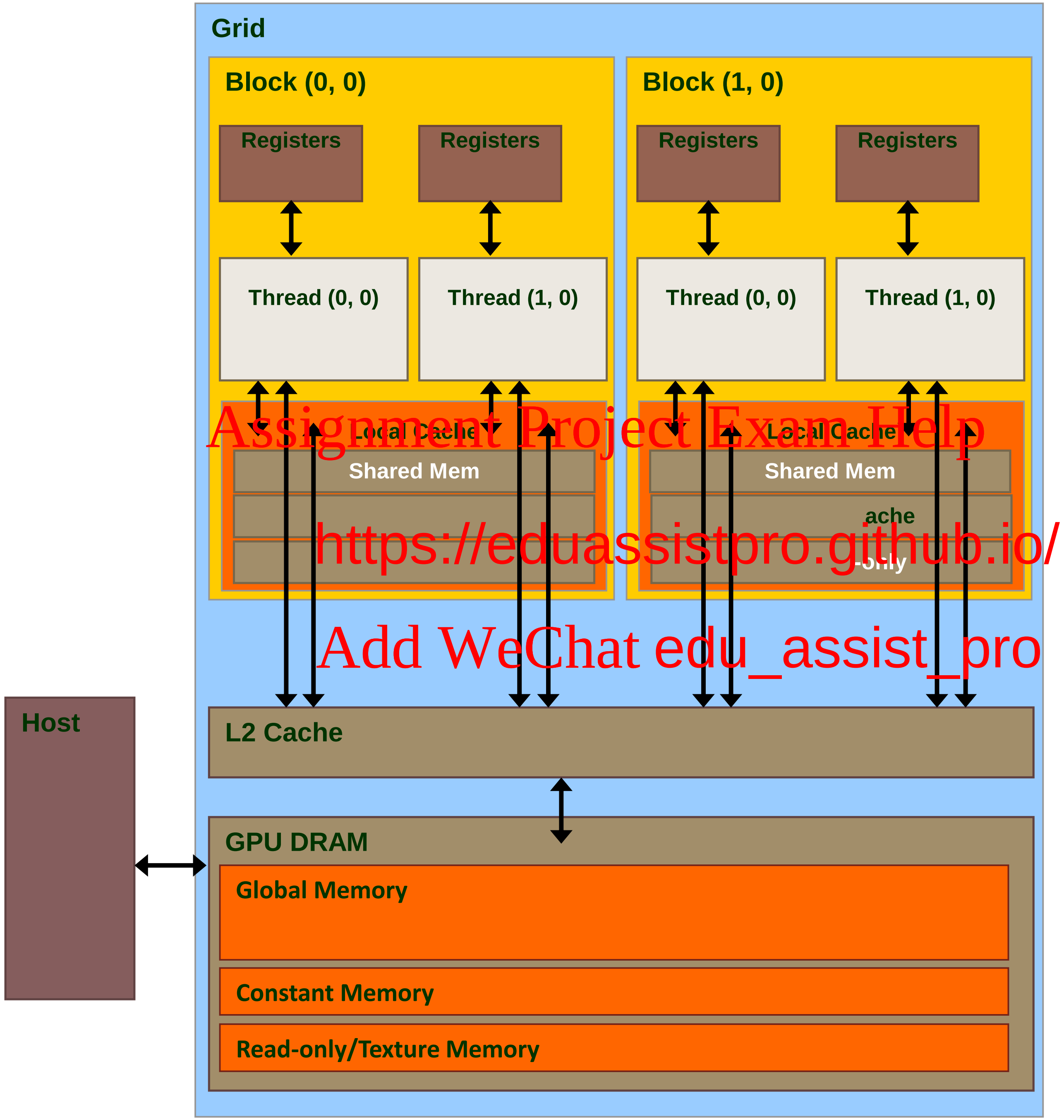
NVIDIA GPU RESEARCH CENTER

Constant Memory

- Question: Should I convert `#define` to constants?
  - E.g. `#define MY_CONST 1234`
- Answer: No
  - Leave alone
  - `#defines` are emb essors
  - They don't take up re within the instruction space
    - i.e. are replaced with literals by the pre-pr

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

- Memory Hierarchy Overview
- Global Memory
- Constant Memory
- Texture and Read-onl
- Roundup & Performa


Grid
Block (0, 0)
Block (1, 0)
Registers
Registers
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Thread (0, 0)
Thread (1, 0)
Local Cache
Shared Mem
Local Cache
Shared Mem
ache
-Only
Host
L2 Cache
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 onl
  - Good for bandwidth li
  - Regular memory accesses with good l 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 *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)
- daTextureType2DLayered (5)
- daTextureTypeCubemap (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 broadc


```
#define N 1024

__global__ void kernel(float cons                __ 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);
}
```

- ❑Memory Hierarchy Overview
- ❑Global Memory
- ❑Constant Memory
- ❑Texture and Read-onl
- ❑Roundup & Performa

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 acce
      `cudaMemcpyToSym`

- 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;

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

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

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 `cudaDeviceSynch` stall the entire GPU pi
- 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 a accessed in broadcast by *nearby* threads
- Read-Only cache is larger than cons but does not have broadcast performance of constant cache
- Kernel variables are not available outside of the kernel

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/
- Jared Hoberock: CUD
  - https://code.google.c /ClassSchedule
- 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

- `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 ture to be filtered. E.g.
  - `cudaFilterModeLi` : Linearly int tween points
  - `cudaFilterModePoint`: Gives the val cific 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