Parallel Computing with GPUs

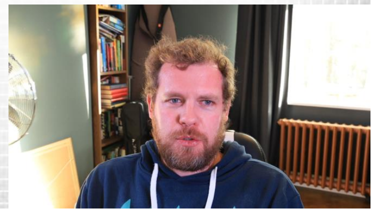
Introduction to CUDA
Part 1 - Programming Model

The University Of Sheffield.

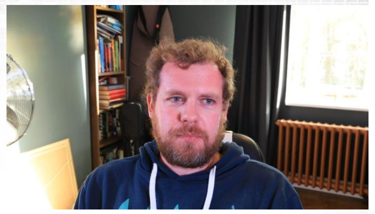
Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

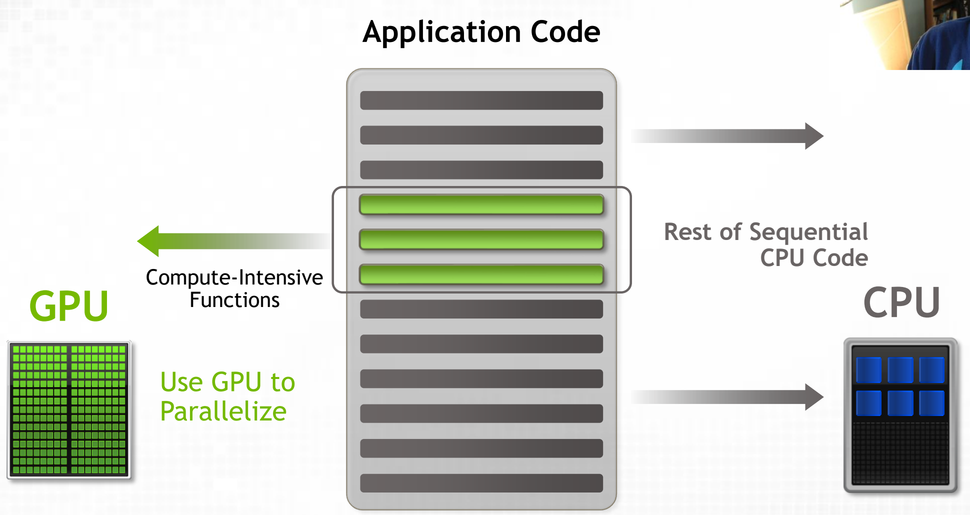
This Lecture (learning objectives)

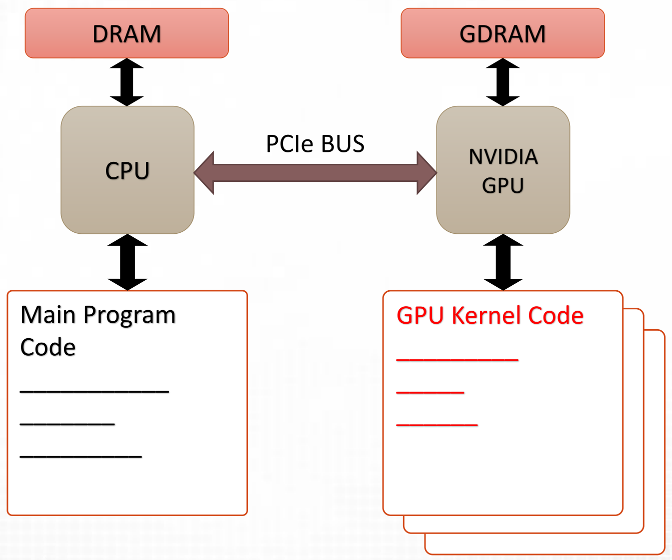
- ❑CUDA Programming Model
  - ❑Present the processing flow for running a GPU program
  - ❑Explain the CUDA software model and its relation to the hardware hierarchy
  - ❑Propose a simple problem which can be implemented on the GPU

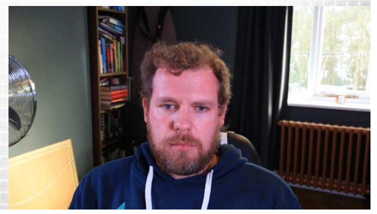


Programming a GPU with CUDA

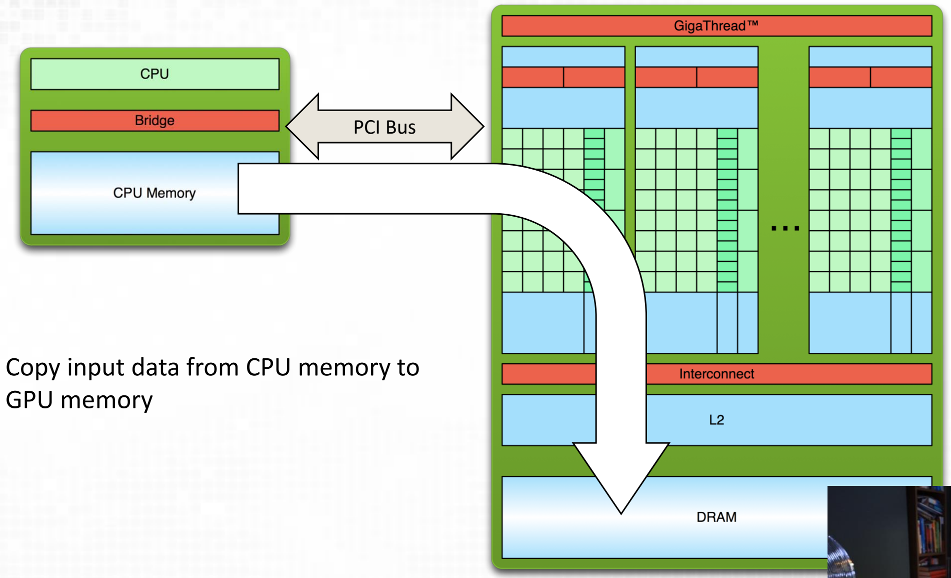






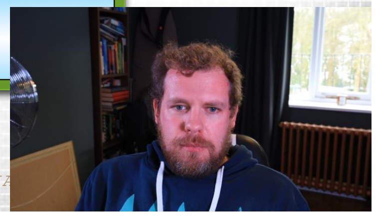


Simple processing flow

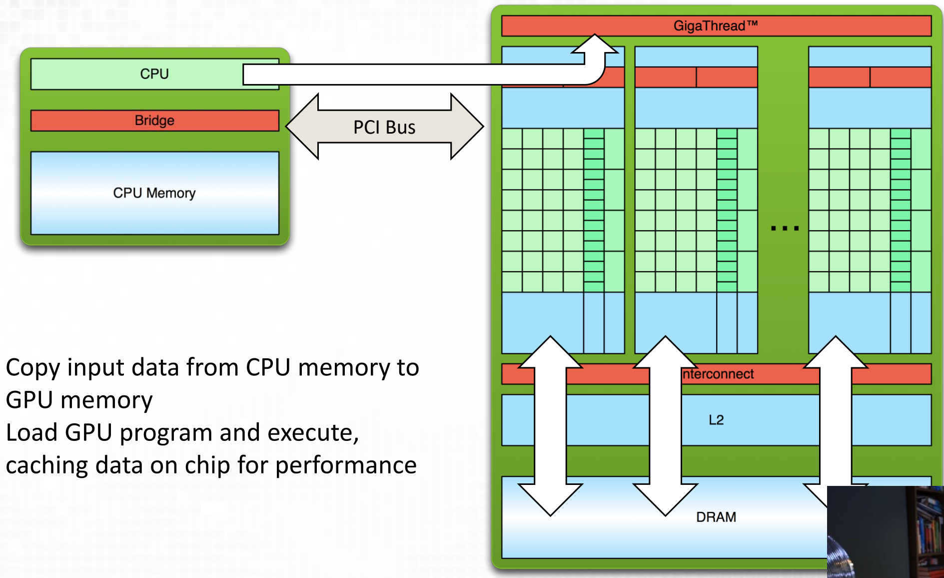


1. Copy input data from CPU memory to GPU memory

Images: Mark Harris (NVIDIA

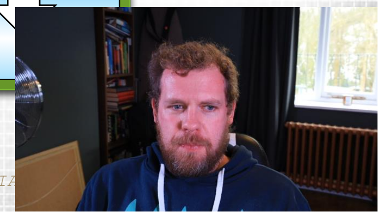


Simple processing flow

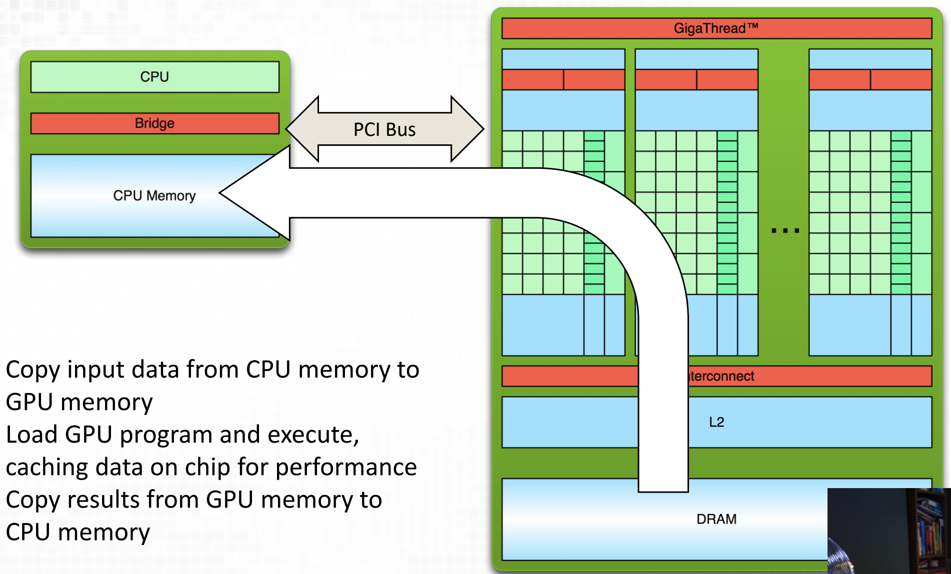


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Images: Mark Harris (NVIDIA

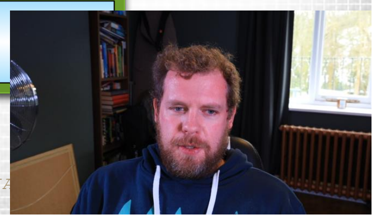


Simple processing flow

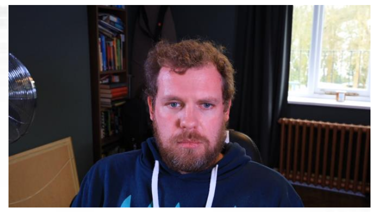


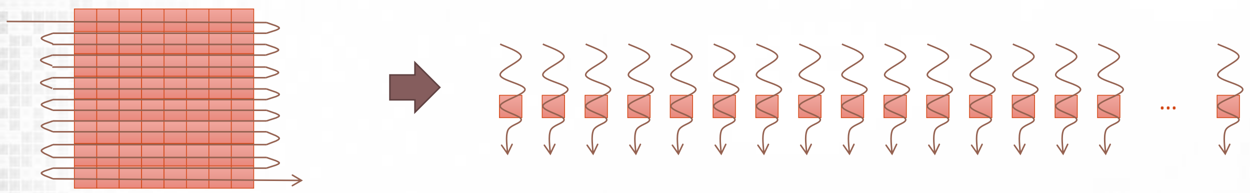
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Images: Mark Harris (NVIDIA



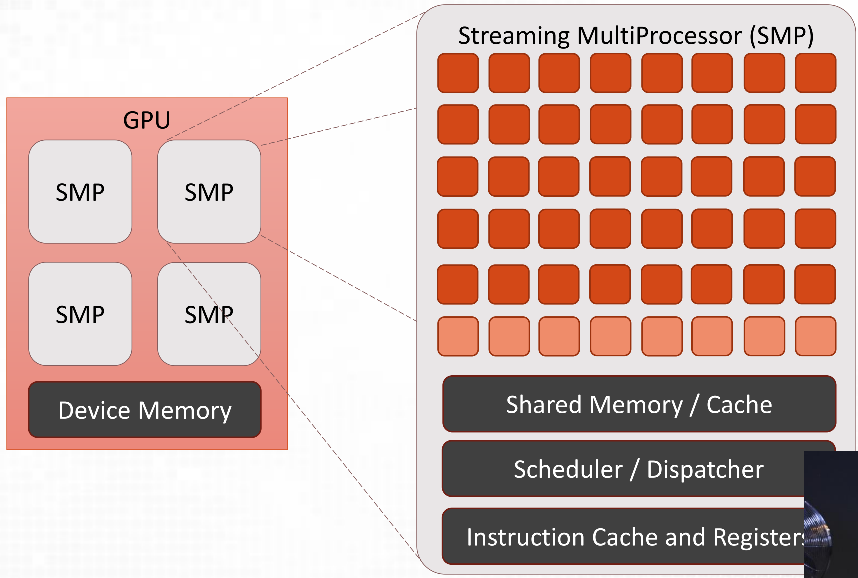
Stream Computing

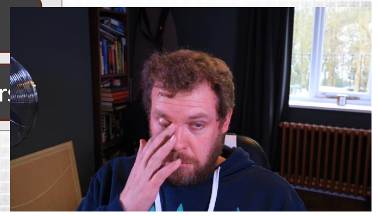




- Data set decomposed into a **stream** of elements
- A single computational function (**kernel**) operates on each element
  - A **thread** is the execution of a kernel on one data element
- Multiple Streaming Multiprocessor cores can operate on multiple elements in parallel
  - Many parallel threads
- Suitable for **Data Parallel** problems

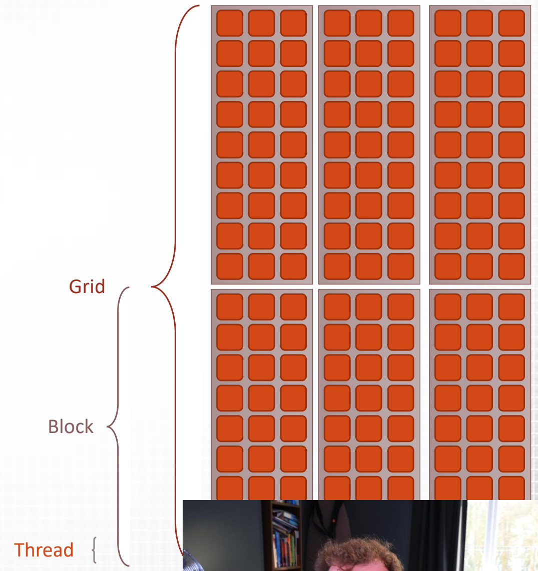
- How does the stream computing principle map to the with the hardware model?

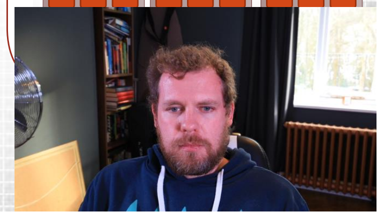




CUDA Software Model

- Hardware abstracted as a **Grid** of **Thread Blocks**
  - Blocks map to SMPs
  - Each thread maps onto a CUDA core
  - Blocks may be 1D, 2D or 3D
- Don't need to know the hardware characteristics
  - Oversubscribe the device
  - Code is portable across different GPU versions





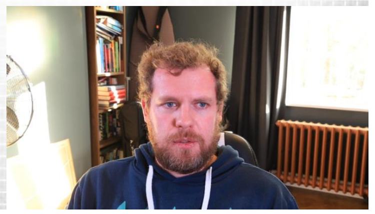
CUDA Vector Types

- CUDA Introduces a new `dim` types. E.g. `dim2`, `dim3`, `dim4`
  - `dim3` contains a collection of three integers (X, Y, Z)

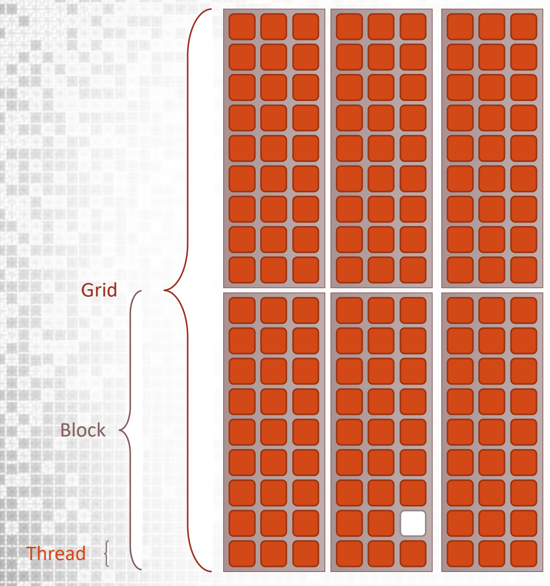
```
dim3 my_xyz (x_value, y_value, z_value);
```

- Values are accessed as members

```
int x = my_xyz.x;
```



Special dim3 Vectors



- `threadIdx`
  - The location of a thread within a block. E.g. (2,1,0)
- `blockIdx`
  - The location of a block within a grid. E.g. (1,0,0)
- `blockDim`
  - The dimensions of the blocks. E.g. (3,9,1)
- `gridDim`
  - The dimensions of the grid. E.g. (3,2,1)

`Idx` values use zero indices, Dim values are a size

Analogy

- ❑Students arrive at halls of residence to check in
  - ❑Rooms are already assigned in order
- ❑Unfortunately admission rates are down!
  - ❑Only half as many rooms as students
  - ❑Each student can be moved from room `i` to room `2i` so that no-one has a neighbour

Serial Solution

- ❑Receptionist performs the following tasks
  1. Asks each student their assigned room number
  2. Works out their new room number
  3. Informs them of their new room number

Parallel Solution

"Everybody check your room number. Multiply it by 2 and go to that room"

Summary

- ❑CUDA Programming Model
  - ❑Present the processing flow for running a GPU program
  - ❑Explain the CUDA software model and its relation to the hardware hierarchy
  - ❑Propose a simple problem which can be implemented on the GPU

- ❑Next Lecture: CUDA Device Code

Acknowledgements and Further Reading

- ❑Some of the content in this lecture material has been provided by;

1. GPUComputing@Sheffield Introduction to CUDA Teaching Material
   - ❑Originally from content provided by Alan Gray at EPCC/NVIDIA
2. NVIDIA Educational Material
   - ❑Specifically Mark Harris's (Introduction to CUDA C)

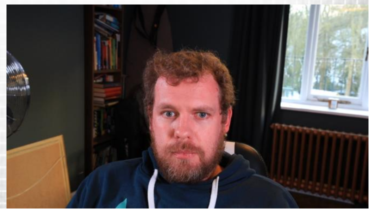
- ❑**Further Reading**
  - ❑Essential Reading: CUDA C Programming Guide
    - ❑http://docs.nvidia.com/cuda/cuda-c-programming-guide/

Parallel Computing with GPUs

Introduction to CUDA Part 2 – Device Code

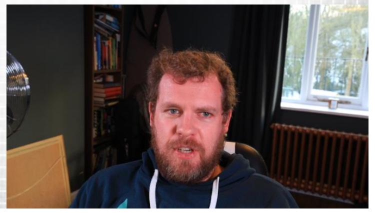
The University Of Sheffield.

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



This Lecture (learning objectives)

- ❑CUDA Device Code
  - ❑Demonstrate a simple CUDA Kernel
  - ❑Explain how the host can configure a grid of thread blocks
  - ❑Identify how the grid block configuration can by utilised by the device



A First CUDA Example

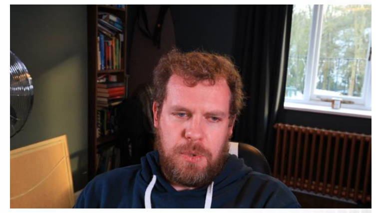
- ❑Serial solution

```
for (i=0;i<N;i++){
   result[i] = 2*i;
}
```

- ❑We can parallelise this by assigning each iteration to a CUDA thread!

CUDA C Example: Device

```
__global__ void myKernel(int *result)
{
   int i = threadIdx.x;
   result[i] = 2*i;
}
```

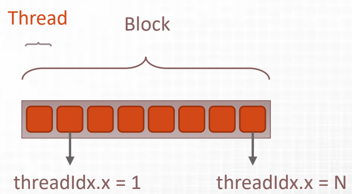


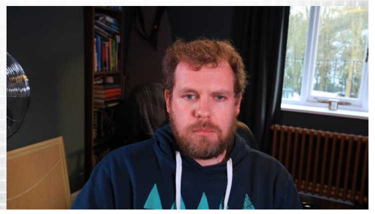
- ❑Replace loop with a “kernel”
  - ❑Use __global__ specifier to indicate it is a CUDA kernel
- ❑Use threadIdx dim variable to get a unique index
  - ❑Assuming for simplicity we have only **one block** which is **1-dimensional**
  - ❑Equivalent to your door number at CUDA Halls of Residence

CUDA C Example: Host

- ❑Call the kernel by using the CUDA kernel launch syntax
  - ❑kernel<<<GRID OF BLOCKS, BLOCK OF THREADS>>>(arguments);

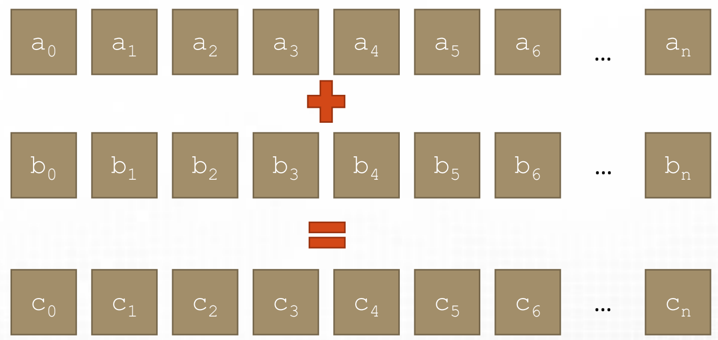
```
dim3 blocksPerGrid(1,1,1);       //use only one block
dim3 threadsPerBlock(N,1,1);     //use N threads in the block

myKernel<<<blocksPerGrid, threadsPerBlock>>>(result);
```





Vector Addition Example

- ❑Consider a more interesting example
  - ❑Vector addition: e.g. a + b = c



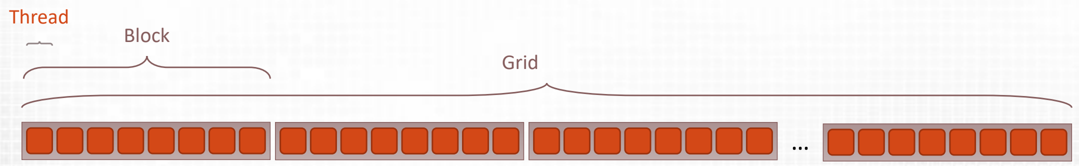
Vector Addition Example

```
//Kernel Code
__global__ void vectorAdd(float *a, float *b, float *c)
{
  int i = threadIdx.x;
  c[i] = a[i] + b[i];
}

//Host Code
...
dim3 blocksPerGrid(1,1,1);
dim3 threadsPerBlock(N,1,1); //single block of threads

vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
```

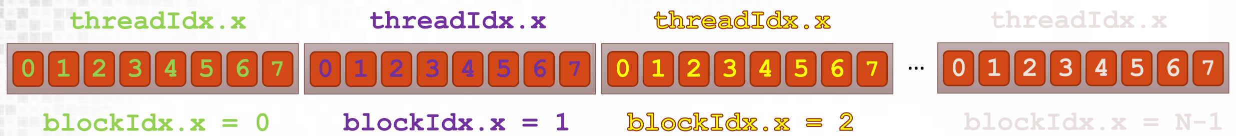
CUDA C Example: Host

- Only one block will give poor performance
  - A block gets allocated to a single SMP!
  - Solution: Use multiple blocks

```
dim3 blocksPerGrid(N/8,1,1);      //assumes 8 divides N exactly
dim3 threadsPerBlock(8,1,1);      //8 threads in the block

myKernel<<<blocksPerGrid, threadsPerBlock>>>(result);
```

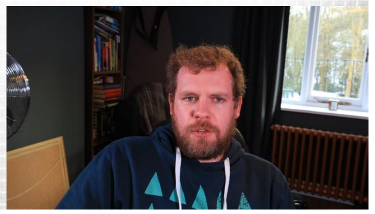


Vector Addition Example



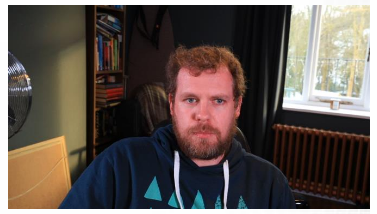

```
//Kernel Code
__global__ void vectorAdd(float *a, float *b, float *c)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}
```

- The integer `i` gives a unique thread Index used to access a unique value from the vectors `a`, `b` and `c`



A note on block sizes

- Thread block sizes can not be larger that 1024
- Max grid size is 2147483647 for 1D
  - Grid y and z dimensions are limited to 65535
- Block size should always be divisible by 32
  - This is the warp size which threads are scheduled
  - Not less than 32 as in our trivial example!
- Varying the block size will result in different performance characteristics
  - Try incrementing by values of 32 and benchmark.
- Calling a kernel with scalar parameters assumes a 1D grid of thread blocks.
  - E.g. `my_kernel<<<8, 128>>>(arguments);`



Device functions

- ❑Kernels are always prefixed with `__global__`
- ❑To call another function from a kernel the function must be a device function (i.e. it must be compiled for the GPU device)
  - ❑A device function must be prefixed with `__device__`
- ❑A device function is not available from the host
  - ❑Unless it is also prefixed with `__host__`

```
int increment(int a){ return a + 1; }                  Host only

__device__ int increment(int a){ return a + 1; }       Device only

__device__ __host__ int increment(int a){ return a + 1; }   Host and device
```

Global functions are always `void` return type

Summary

- ❑CUDA Device Code
  - ❑Demonstrate a simple CUDA Kernel
  - ❑Explain how the host can configure a grid of thread blocks
  - ❑Identify how the grid block configuration can by utilised by the device

- ❑Next Lecture: Host Code and Memory Management

Acknowledgements and Further Reading

- ❑Some of the content in this lecture material has been provided by;

1. GPUComputing@Sheffield Introduction to CUDA Teaching Material
   - ❑Originally from content provided by Alan Gray at EPCC/NVIDIA
2. NVIDIA Educational Material
   - ❑Specifically Mark Harris's (Introduction to CUDA C)

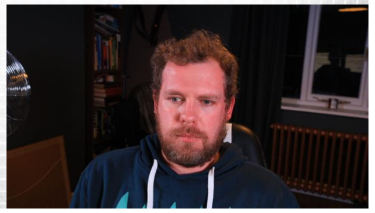
- ❑**Further Reading**
  - ❑Essential Reading: CUDA C Programming Guide
    - ❑http://docs.nvidia.com/cuda/cuda-c-programming-guide/

Parallel Computing with GPUs

Introduction to CUDA
Part 3 - Host Code and Memory Management

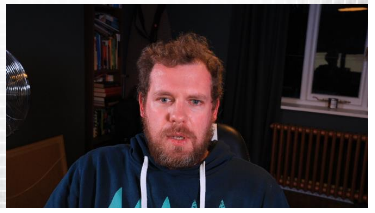
The University Of Sheffield.

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



This Lecture (learning objectives)

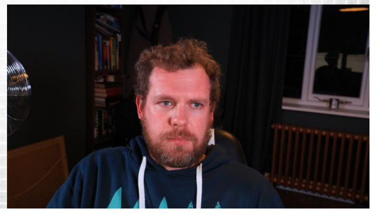
- ❑CUDA Host Code and Memory Management
  - ❑State the methods in which device memory can reserved
  - ❑Demonstrate how host code can be used to move memory to and from the GPU device
  - ❑Present a complete example of a GPU program



Memory Management

- ❑GPU has separate dedicated memory from the host CPU
- ❑Data accessed in kernels must be on GPU memory
  - ❑Data must be copied and transferred
  - ❑A Unified memory approach can be used to do this transparently
- ❑**cudaMalloc()** is used to allocate memory on the GPU
- ❑**cudaFree()** releases memory

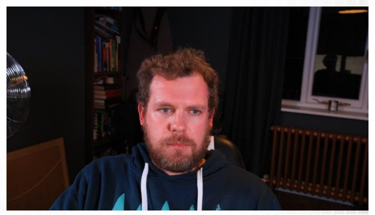
```
float *a;
cudaMalloc(&a, N*sizeof(float));
...
cudaFree(a);
```



Memory Copying

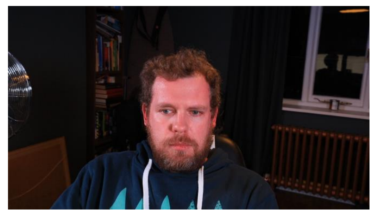
- ❑Once memory has been allocated we need to copy data to it and from it.
- ❑**cudaMemcpy()** transfers memory from the host to device to host and vice versa

```
cudaMemcpy(array_device, array_host,
  N*sizeof(float), cudaMemcpyHostToDevice)

cudaMemcpy(array_host, array_device,
  N*sizeof(float), cudaMemcpyDeviceToHost)
```



- ❑First argument is always the **destination** of transfer
- ❑Transfers are relatively slow and should be minimised where possible

```
#define N 2048
#define THREADS_PER_BLOCK 128

__global__ void vectorAdd(float *a, float *b, float *c) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}

int main(void) {
     float *a, *b, *c;               // host copies of a, b, c
     float *d_a, *d_b, *d_c;         // device copies of a, b, c
     int size = N * sizeof(float);

     cudaMalloc((void **)&d_a, size);
     cudaMalloc((void **)&d_b, size);
     cudaMalloc((void **)&d_c, size);

     a = (float *)malloc(size); random_floats(a, N);
     b = (float *)malloc(size); random_floats(b, N);
     c = (float *)malloc(size);

     cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
     cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

     vectorAdd <<<N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>(d_a, d_b, d_c);

     cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

     free(a); free(b); free(c);
     cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
     return 0;
}
```



Define macros

Define kernel

Define pointer variables

Allocate GPU memory

Allocate host memory and initialise contents

Copy input data to the device

Launch the kernel

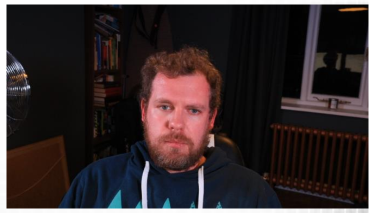
Copy data back to host

Clean up

Statically allocated device memory

- How do we declare a large array on the host without using malloc?
  - Statically allocate using compile time size

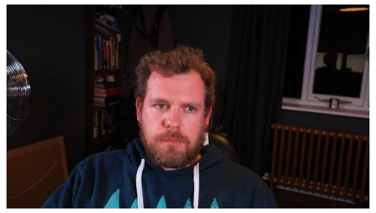
```
int array[N];
```



- We can do the same on the device. i.e.
  - Just like when applied to a function
  - Only available on the device
  - Must use `cudaMemCopyToSymbol()`
  - Must be a global variable

```
__device__ int array[N];
```

```
#define N 2048
#define THREADS_PER_BLOCK 128

__device__ float d_a[N];
__device__ float d_b[N];
__device__ float d_c[N];

__global__ void vectorAdd() {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  d_c[i] = d_a[i] + d_b[i];
}

int main(void) {
    float *a, *b, *c;            // host copies of a, b, c
    int size = N * sizeof(float);

    a = (float *)malloc(size); random_floats(a, N);
    b = (float *)malloc(size); random_floats(b, N);
    c = (float *)malloc(size);

    cudaMemcpyToSymbol(d_a, a, size);
    cudaMemcpyToSymbol(d_b, b, size);

    vectorAdd <<<N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>();

    cudaMemcpyFromSymbol(c, d_c, size);

    free(a); free(b); free(c);
    return 0;
}
```



Define macros

Statically allocate GPU memory

Define kernel

Define pointer variables

Allocate host memory and initialise contents

Copy input data to the device

Launch the kernel

Copy data back to host

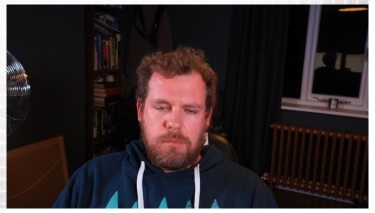
Clean up

Device Synchronisation

- Kernel calls are non-blocking
  - Host continues after kernel launch
  - Overlaps CPU and GPU execution
- **cudaDeviceSynchronise()** call be called from the host to block until GPU kernels have completed

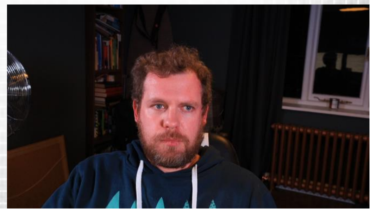
```
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
//do work on host (that doesn't depend on c)
cudaDeviceSynchronise(); //wait for kernel to finish
```

- Standard **cudaMemcpy** calls are blocking
  - Non-blocking variants exist



Summary

- CUDA Host Code and Memory Management
  - State the methods in which device memory can reserved
  - Demonstrate how host code can be used to move memory to and from the GPU device
  - Present a complete example of a GPU program



Acknowledgements and Further Reading

- Some of the content in this lecture material has been provided by;

1. GPUComputing@Sheffield Introduction to CUDA Teaching Material
   - Originally from content provided by Alan Gray at EPCC/NVIDIA
2. NVIDIA Educational Material
   - Specifically Mark Harris's (Introduction to CUDA C)

- **Further Reading**
  - Essential Reading: CUDA C Programming Guide
    - http://docs.nvidia.com/cuda/cuda-c-programming-guide/