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

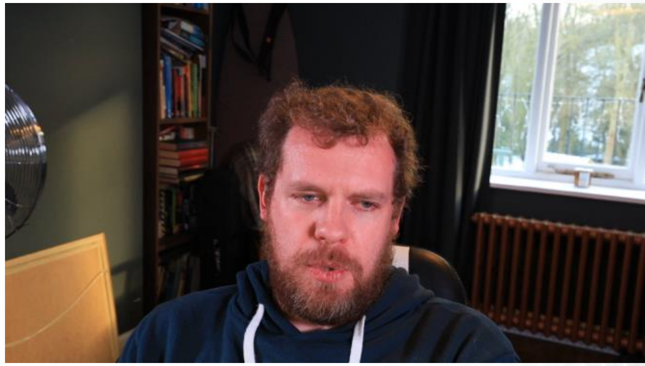
- Serial solution

```
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;
}
```



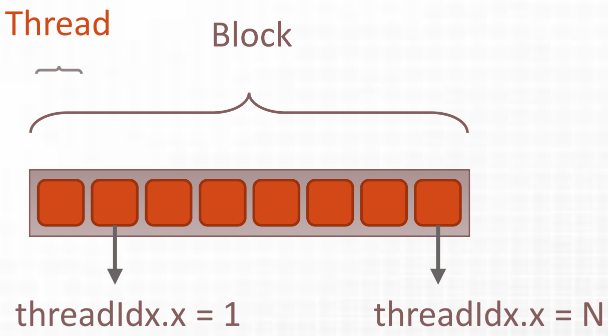
- Replace loop with a "kernel"
  - Use __global__ specifier to indicate it is a CUDA kernel
- Use threadIdx dim variable to get a unique index
  - Assuming for simplicity we have only **one block** which is **1-dimensional**
  - Equivalent to your door number at CUDA Halls of Residence

The University Of Sheffield.

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`

$$
\begin{array}{cccccccc}
a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & \ldots & a_n \\
 & & & & + & & & & \\
b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & \ldots & b_n \\
 & & & & = & & & & \\
c_0 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & \ldots & c_n
\end{array}
$$

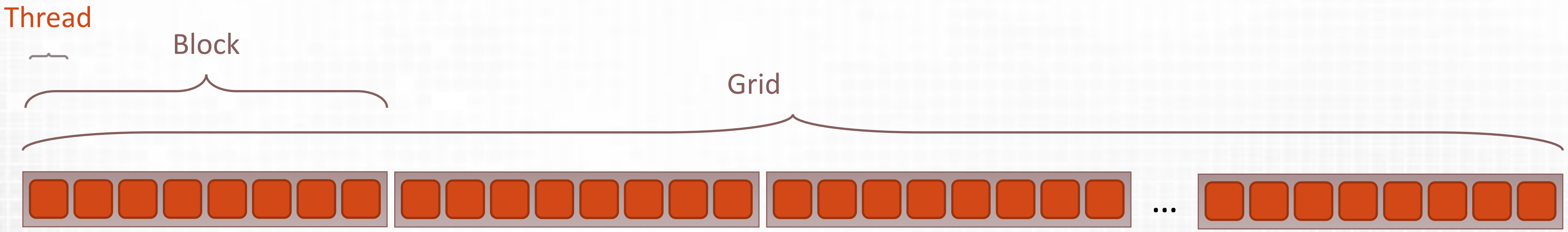
Vector Addition Example

```
//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);
```

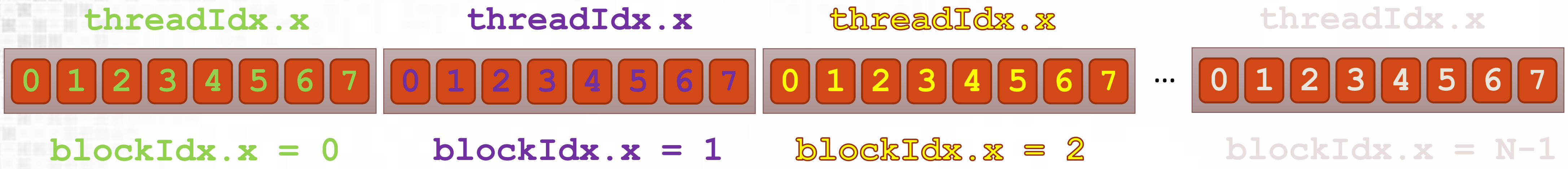
CUDA C Example: Host

- 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/