Practical 3: CUDA

CUDA is Nvidia's GPU programming suite, used for computation on Nvidia (and only Nvidia) graphics cards. GPU processing is a massively parallel paradigm; simple programs are run on many cores simultaneously, and the final result is passed back to the CPU for further processing if required.

In this practical, you will be building up to matrix addition in CUDA from basic principles. You will cover CUDA setup, simple kernel principles, programming for blocks and threads, transferring data between CPU and GPU and working with 2-dimensional CUDA grids or blocks. This practical is based on Sarah Tariq's Introduction to GPU Computing presentation, and the example code is taken from there; if you are struggling, consult these slides – they're available on Blackboard, in the practicals folder.

Hello world

CUDA is installed on all Ubuntu machines in G56. The first step is to activate the CUDA module

```
module add nvidia
```

Once activated, you can compile a CUDA application using

```
nvcc -o output input
```

and then run in the same manner as a normal C/C++ program.

Test the compiler with the following code (save it as a .cu):

```
#include<stdio.h>
__global__ void mykernel(void) {
}
int main(void) {
	mykernel<<<1,1>>>();
	printf("Hello World!\n");
	return 0;
}
```

This creates an empty kernel, runs it on one core and then prints 'Hello world!'

Integer addition

A CUDA program is made of two halves; CPU and GPU code. The GPU kernel is the code that is copied into blocks and threads, and is run on every core of the graphics processor; it is defined in CUDA by `__global__`. The CPU code manages setup, device selection (if more than one GPU is available), memory allocation and data transfer from the CPU to the GPU.

This can be seen in the simple addition code. Consider the following simple kernel:

```
__global__ void add(int *a, int *b, int *c) {
	*c = *a + *b;
}
```

Note how there is no return value (as there is no single execution path to return it to); instead, both the inputs (`a` and `b`) and output (`c`) are passed by reference in the kernel definition. In the client-side code, there are functions that define the areas that these values will be passed from and to; see the next snippet.

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

	// Allocate space for device copies of a, b, c
	cudaMalloc((void **)&d_a, size);
	cudaMalloc((void **)&d_b, size);
	cudaMalloc((void **)&d_c, size);

	// Setup input values
	a = 2;
	b = 7;

	// Copy inputs to device
	cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
	cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

	// Launch add() kernel on GPU
	add<<<1,1>>>(d_a, d_b, d_c);

	// Copy result back to host
	cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

	// Cleanup
	cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
	print("%i",c)
	return 0;
}
```

Pay special attention to the three `cudaMemcpy` functions; these, coupled with the `cudaMalloc/cudaFree` functions, move the data to and the result from the graphic card's memory.

Compile and run the kernel addition code above.

Vector addition

CUDA splits the work it is given into a grid of blocks, with each block consisting of a number of threads. The grid and block dimensions are defined when the kernel function is called;

```
kernelFunc<<<blocksInGrid,threadsPerBlock>>>(args)
```

Consider this modification of the previous kernel. It now supports vector addition using multiple single-thread blocks.

```
__global__ void add(int *a, int *b, int *c) {
      c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

By modifying your `main()` function from the previous section, create CPU-side code that allocates memory, and performs parallel addition of two vectors each 512 members long using the kernel code given above. You will need to increase the space allocated to a, b and c on both the CPU and GPU cards, and modify the kernel so that it creates the appropriate number of blocks. You can populate the vectors with the `random_ints(output, N)` function – for example, `random_ints(foo, 10)` will populate the array `foo` with 10 random integers.

Once that is functioning, alter both your kernel and CPU code so that it first works on threads (instead of blocks), and then an arbitrary number of blocks and threads.

Matrix addition

CUDA grids and blocks do not have to be single-dimensional; they can support 2- and 3D specifications as well. To define your kernel grid or blocks as being executed in a 2D environment, pass a `dim3` structure to `kernel<<<blocks, threads>>>()` as below

```
dim3 foo(10,10);
exampleKernel<<<foo,1>>>() //Creates a 10x10 grid of blocks of 1
thread each
```

Using this code, extend your vector addition program for matrix addition. You will need to change both the kernel and CPU code. It is down to you whether your kernel works with threads, blocks or both.

Further study

If you have reached this point, then try implementing the stencil code in the presentation mentioned in the intro. It covers the main difference between threads and blocks – threads can communicate with each other using shared memory. This will be covered in more detail in next week's lecture, so don't worry if you don't follow it immediately.