GPUs: Practical session 2

1 Product of a matrix and a vector

Start with the code framework `MatrixVectorProduct.cu`. This has had the important parts left out deliberately, so it doesn't compile.

The code constructs an $N \times N$ matrix A and a vector $\mathbf{x}$ of length N and puts them on the device. It then copies a vector $\mathbf{y}$ of length N back from the device and checks that:

$$A\mathbf{x} = \mathbf{y} \tag{1.1}$$

to within a suitable precision.

1.1 Task 1 - First pass

Write in a suitable kernel to perform the correct multiplication. The kernel should take parameters `A, x, y, N`. You may assume that $N \leq 512$. Do not try anything fancy, just write a working kernel. Slow and correct is always better than fast and wrong...

When you have implemented this correctly, the time taken and bandwidth will be displayed.

1.2 Task 2 - Use of shared memory

In Task 1, you probably had a loop that read an element from `x` every iteration. Since all the threads in a block read the same data, it would be better to use all the threads to read `x` from global memory once, and then store it in shared memory for faster access.

Try using this approach to improve the performance of the kernel. If you can't get improved performance using this approach, consider why this might be.

1.3 Task 3 - A different approach

Instead of computing one element of `y` per thread, what happens if you try computing one element per `block`? In this approach, each thread in the block should compute the product of one element of `x` with one of `A`, and then perform the sum of these elements over the thread-block.

Hint: Use shared memory to store the individual element sums initially, and then perform a reduction over all the threads in the block via a tree-based method, ending up with the answer in the first element of the shared-memory block.