Techniques used

My code uses both graphics cards available on CUDA01. By default the double kernel will run on the Tesla K40c, while the float kernel will run on the GTX 780. I made this choice as the K40c is more powerful than the GTX 780, so it makes sense for the double kernel to run on it.

Even though it is running on slower hardware the float kernel finishes much more quickly than the double kernel, so one possible improvement would be to run a small part of the double kernel on the GTX 780. This would help even the load, but unfortunately I did not have time to implement it.

Streams and asynchronous transfers were used to allow operations to overlap. This means the kernels can do their work and memory transfers independently.

Constant memory was used to hold a small number of values that were used outside the for loop. I found that using this gave a slight speedup as it lessened the demand for register usage. I tried putting the constants used inside the for loop (such as euler's constant) into constant and shared memory, but I found that this actually reduced performance.

I was unable to find a good way of using shared memory. Anything I tried slowed down the kernels by a slight amount.

A 2D grid was used to distribute the work. Each block takes a number of samples for some N values. With the grid's Y dimension set to 1 each block will process samples for each N, with the grid's Y dimension set to 2 each block will process samples for half of the N values, and so on. By increasing the size of the grid's Y dimension more blocks will be created, helping to increase the occupancy of the device.

The below tables illustrates this grid layout with the number of samples set to 786, the block size is 256 and the grid's Y dimension is 1.

n value / sample	Samples 1 to 256	Samples 257 to 512	Samples 513 to 768
N = 1	Block 0	Block 1	Block 2
N = 2	Block 0	Block 1	Block 2
N = 3	Block 0	Block 1	Block 2
N = 4	Block 0	Block 1	Block 2
N = ...	And so on...		

The next table shows how the above grid changes if just the grid's Y dimension is changed to 2.

n value / sample	Samples 1 to 256	Samples 257 to 512	Samples 513 to 768
N = 1	Block 0	Block 1	Block 2
N = 2	Block 3	Block 4	Block 5
N = 3	Block 0	Block 1	Block 2
N = 4	Block 3	Block 4	Block 5
N = ...	And so on...		

Results

Running on CUDA01 and using both graphics card my best result was with n=m=20’000, a block size of 256 and a grid with the Y dimension set to 128. With these parameters the CUDA version was 55x faster than the CPU version.

Reducing the grid’s Y dimension had a large impact on performance. With the same parameters as above but with a variable Y dimension size I got the following results:

Y dimension size	**Double kernel time taken (milliseconds)**
1	4218
2	2639
4	1849
8	1591
32	1382
128	1330

As the size of the grid’s Y dimension increases the occupancy of the device improves as there are much more threads. The above table shows that increasing the Y dimension size gives huge benefits at first, but the benefits become less and less as it continues to increase.

The below table shows the speedup achieved in the CUDA code as n and m change. The speedup was calculated as the total time taken in the CUDA version (including allocations, transfers, etc) divided by the total time taken by the CPU version. In all runs the block size was 256 and the grid’s Y dimension had a size of 128.

n = m =	**Speedup**

5000	24.6x
8192	37.8x
16384	50.7x
20000	55x

As with my previous assignments I found that a block size of around 128-256 gave the best performance. The below table shows the effect of changing the block size with n=m=20000 and the grid's Y dimension set to 128.

Block size	**Speedup**
32	30x
64	48x
128	55x
256	55x
512	49x
1024	46x