Introduction to Cuda Assignment 3
Cuda Exponential Integral Calculation

Robert Cronin 12313002

May 19, 2017

1 CUDA Implementation

The basic version of my algorithm uses $n * m$ threads, where each thread runs the calculation for a single value of x and n.

I then added functionality to utilise shared memory, cuda streams, multiple cards and dynamic parallelism. These can be used in any combination by using the appropriate flags (see README).

1.1 Shared Memory

The algorithm for each calculation uses many variables, which depending on the card may fill up the registers and spill into global memory which would significantly slow down the program. Therefore, if shared memory is utilised, a 2d array of shared memory is used with 10 memory locations available to each thread.

Depending on whether shared memory is used or not, the cache is configured to favour either registers or shared memory.

1.2 Streams

If streams are used then the quantity of n values to be calculated is distributed between the streams.

1.3 Multiple Cards

On cuda01 we have two cards available to use so we can spread the workload between the two. I used a Master-Slave approach where in this case we have two slaves each using a different card.

The master sends out values of n to the slaves and each slave then computes the next `jump` values of n. The slave then sends a success message and its results after which the master sends a new chunk while there is still computation remaining.

1.4 Dynamic Parallelism

This option runs n threads, one for each n value, each of which spawns a child kernel of m threads to compute each value.

1.5 Combining

Each of these methods can be used on their own or can be combined to use any number of them.

2 Results

For my results, I ran each combination 5 times and chose the best result produced[1]. Similarly I ran it 5 times on the CPU for each grid size to calculate the speed-up.

Full results of timings and speed-ups can be found in `timing.ods`.

For the largest size of 20000$*$20000 I ran every combination of the different methods (for 2, 3 and 10 streams), for various block sizes (32, 64, 128, 256, 512 and 1024).

Some of the key results are shown in Figure 1.

2.1 Best Result - Streams

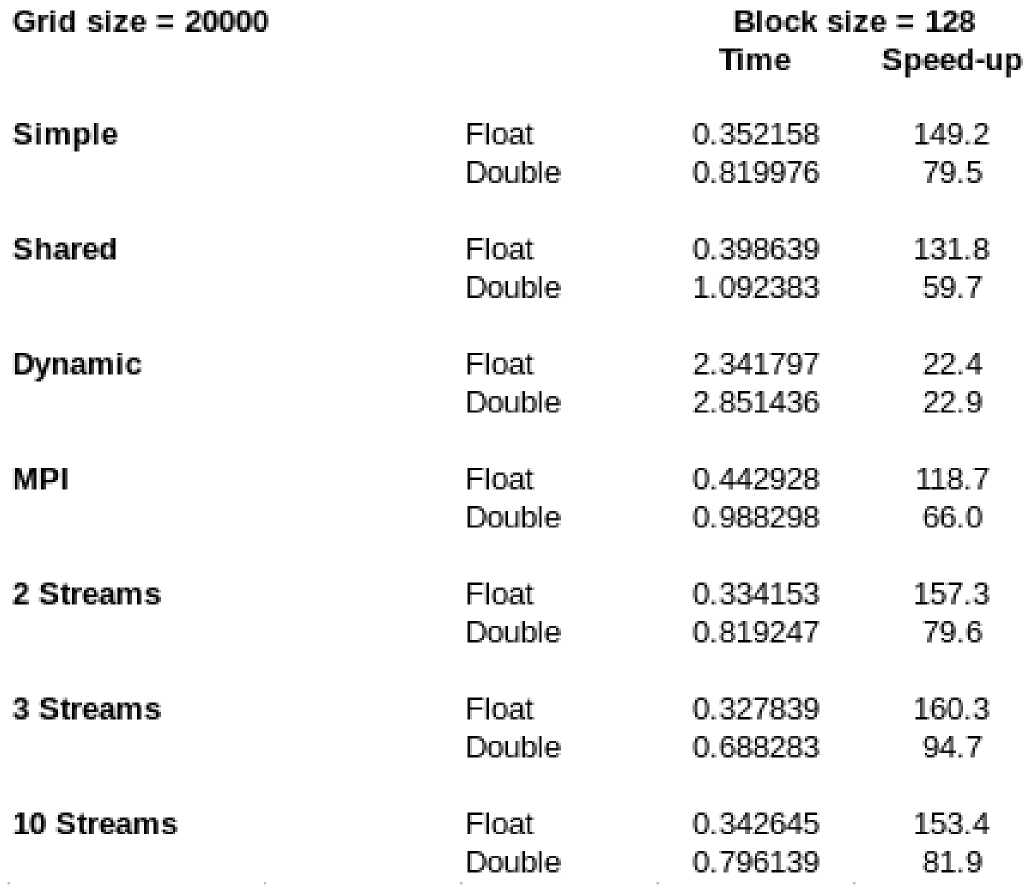
This produced a clear winner of using 3 streams (no shared memory). The best block size for floats was 128 and for doubles was 256 with speed-ups of 160.3x and 94.9x respectively.

The speed-ups for this method are shown in Figure 2. It is clear that the speed-ups increase for larger block sizes up to 128 and then roughly level out.

2.2 Shared Memory

It was interesting to note that using shared memory tended to be slower than just sticking to the registers suggesting that the extra time taken due to

[1]For consistency's sake I ensured that I was the only user running programs at the time



Grid size = 20000		Block size = 128	
		Time	Speed-up
Simple	Float	0.352158	149.2
	Double	0.819976	79.5
Shared	Float	0.398639	131.8
	Double	1.092383	59.7
Dynamic	Float	2.341797	22.4
	Double	2.851436	22.9
MPI	Float	0.442928	118.7
	Double	0.988298	66.0
2 Streams	Float	0.334153	157.3
	Double	0.819247	79.6
3 Streams	Float	0.327839	160.3
	Double	0.688283	94.7
10 Streams	Float	0.342645	153.4
	Double	0.796139	81.9

Figure 1: Some key results for $20000 * 20000$ with block size of 128

variables being demoted to global memory is not significant for this algorithm using the card on cuda01.

2.3 Multiple Cards

Using Multiple cards via MPI proved to be even slower as presumably the overhead of transferring the calculated data between the nodes outweighed the benefits of running on two cards in this case. Perhaps with more than 2 cards this overhead could be overcome to provide much better speed-ups.

2.4 Dynamic Parallelism

It is clear that dynamic parallelism is definitely not suited to this problem as the overhead of spawning thousand of child kernels proved too costly and

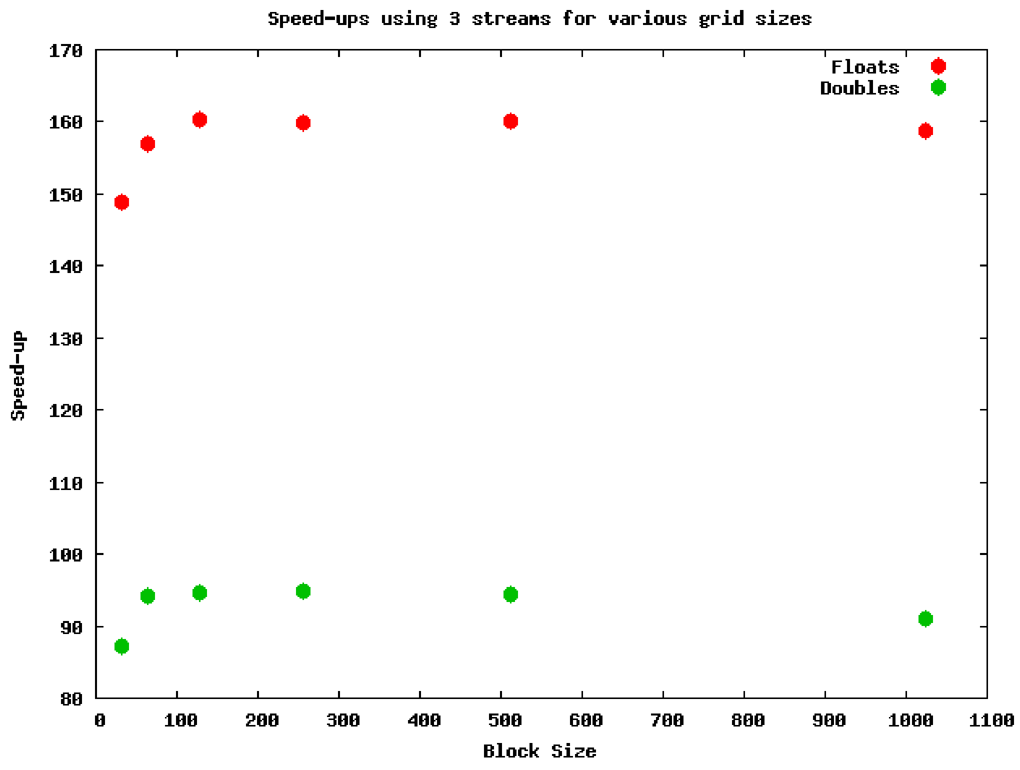


Figure 2: Speed-ups by block-size for using 3 streams

was significantly slower.

2.5 Combinations

Due to quantity of possible combinations I only tested the cases with multiple methods for block size of 128 as it appeared to give the best results on average for the first set of tests.

Due to the fact that of the methods implemented, only one provided a speed-up and the rest were slower, it is not surprising that combinations of the methods did not provide any extra speed-up. Timings for them can be found in `timings.ods`.

2.6 Other Grid Sizes

Tests were also carried out for the methods on their own, with block size of 128, for $n = m = 5000, n = m = 8192$ and $n = m = 16384$ which can also be found in `timing.ods`.

In each case, using 3 streams proved to be the best result for floats, and the best result for all sizes bar 5000 for doubles (where using 10 streams slightly edged it).

We can see in Figure 3 that for floats the speed-up increases as the grid size does as we expect with most parallel problems. It does slightly decrease for doubles but remains roughly constant.

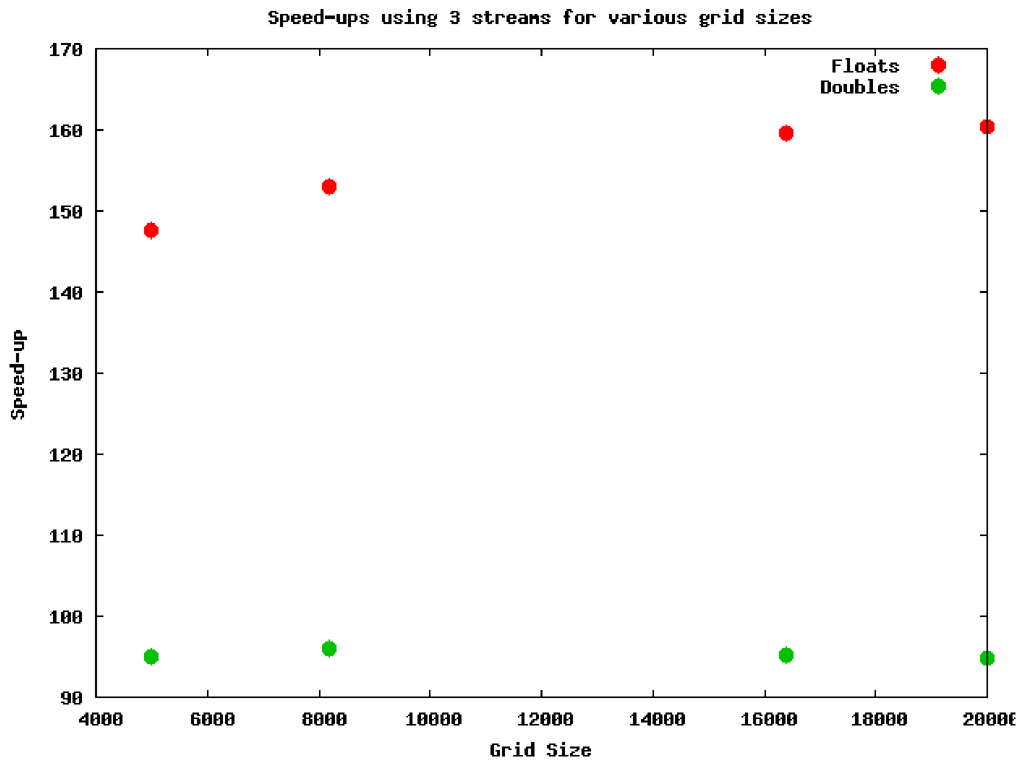


Figure 3: Speed-ups by grid size for using 3 streams

2.7 Other Notes

It is significant to note that the speed-up gained from simply copying the algorithm from the CPU code and running on $n*m$ threads provided a speed up of nearly 150x (compared to the best speed-up achieved of 160x). The time taken to code and test all these other methods provided little benefit for this algorithm but, of course, could provide substantial gains in other cases.