5615 CUDA—Assignment 3
Exponential Integral Calculation

Hugh Delaney

May 25, 2021

1 CUDA Implementation

Please see `gpu_funcs.cu`. `make prof1` will run the code on 1 GPU, and `make prof2` will run code on 2 GPUs.

Advanced techniques used:

- Constant/Shared memory—constant memory was used in the first implementations, but it was markedly slower than initializing shared memory with hard coded values. The final implementation uses shared memory as a fast alternative to constant memory.
- Multiple Cards/Streams—Multiple cards are used if you provide the `-s` option. This gives a real boost to performance, especially due to using `cudaMemcpyAsync()`, which allows memory transfer on one card to overlap with computation on the other card. `make run2` will run the code on two cards and report individual timings for each card. Since streams are understood to be launched simultaneously, we can take the overall run time as the greater of the two individual timings.
- Dynamic Parallelism—In `single/gpu_funcs.cu` please see `GPU_exponentialIntegralFloat_4_launch()` and `GPU_exponentialIntegralFloat_4_execute()` (note that these functions are not also included in `double/`). Code can be run dynamically by setting `bool dynamic=true` in the wrapper function `launch_on_one_card()`.

  Approach consists of launching one thread per n value, and precomputing `psi` (if `a<=1.0`), before launching enough threads/blocks to compute the `numberOfSamples` x values from `a` to `b` for that given value of `n`. This approach was markedly slower than normal non-dynamically parallel approach, so is not used in final fast implementation.

Advanced techniques considered:

- Texture memory—Texture memory was considered so that we could index between a and b with floating point indexes. However this was decided against since the overhead of creating and initializing texture memory outweighed the cost of creating x values in kernel, which requires no cudaMemcpys before the kernel can start.

2 Performance

It must first be said that `blockDim.y = 1024/blockDim.x`, so we are not testing for blocks of different sizes, only different shapes.

Max error values were around 1E-7 for single precision and around 1E-15 for double precision.

From the figures we can see that block shape does not make a huge difference in the execution speed of our code. Having said that, having a low `blockDim.x` may give poorer performance than a higher `blockDim.x`. A consistently good block size seems to be around 16x64, 32x32 or 64x16.

Notice that speedups actually decrease as n, m grow. This would seem to indicate that the problem is mostly constrained by the bandwidth of the PCI-Express bus, where for bigger problem sizes, data transfer time increases as a proportion of the overall time. I have designed code so as to minimize this data transfer bottleneck, such as by initializing data without the need for any data transfer, as well as by splitting the data transfer over two cards, so that transfers can happen asynchronously using Cuda streams.

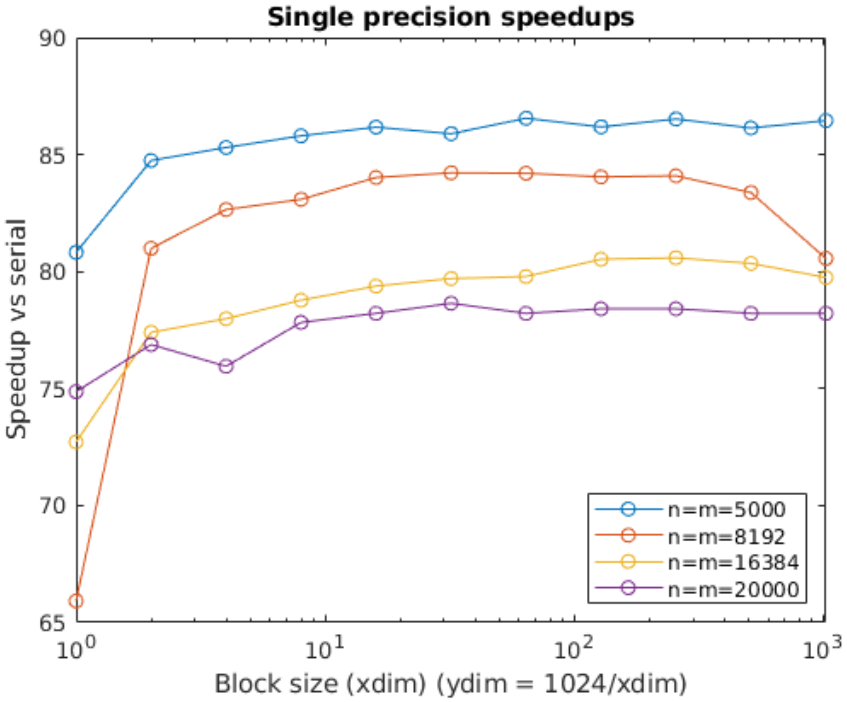

Figure 1: Single Precision Speedups

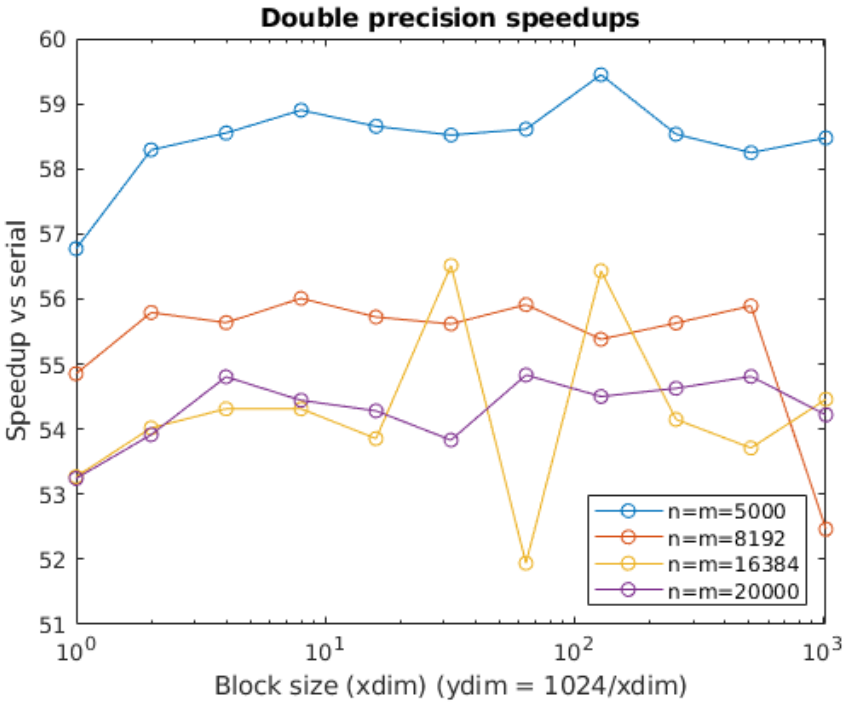

Figure 2: Double Precision Speedups

There seem to be no algorithmic improvements we can make to our code, since our algorithm is trivially parallel, nor any way to make our on card memory any faster. Dynamic parallelism did not offer any benefits (as far as I can see), as the problem is not simplified by recursion.

In order to increase performance even more, more cards would be needed. However, using a single host node would still cause data bottlenecks to the host, so an MPI/Cuda hybrid solution may work well. In this case we ought to use MPI I/O to write our data to file, instead of causing a bottleneck when transferring data to the master node.