Building and usage:

The program will use floats if you build with "*make all USES_FLOAT=1*".
The program will use doubles if you build with "*make all USES_FLOAT=0*".

Note that if you have previously built with one data type and want to change to the other type used you **must** run "*make clean*" before running the one of the above commands.

All C and CUDA code is separated into different source files, and gcc is used to compile the C code.

The overall time of each major part is recorded. For CUDA parts, the overall time is recorded and individual stages (allocation, copying, execution and so on) and timed using CUDA events.

All parameters are optional. For those that set a value a default value is used if they are not supplied. The following parameters are supported:

-a: compute the average of each row
-n <nrow>: set the number of rows (default 32)
-m <ncol>: set the number of columns (default 32)
-p <niterations>: set the number of iterations (default 10)
-t: print the recorded execution times for each part
-b <nthreads>: set the number of threads per block (default 64)
-v: print the contents of all grids and reduction vectors

The following two parameters are useful for when only the CPU or CUDA timings are wanted:
-c: skip executing the CPU version
-g: skip executing the CUDA version

Naive and fast CUDA versions:

My program executes two different CUDA kernels. One is a naive version that requires the grid to be copied to the device twice and mostly uses global memory. The other is a fast version that I tried to make as fast as possible.

The optimisations used in the fast version are as follows:

1) Before the inner loop is entered the two constant column values and the following three columns are loaded into registers. Then on each inner iteration only one new column value needs to be loaded from memory. This works

because say we compute the new value for grid[i][j], then only grid[i][j+3] has to be loaded to compute the value for grid[i][j+1]. Four of the five previously loaded values in registers can be reused. This greatly reduces the number of memory operations required.

2) The inner loop is unrolled twice. This allows a float2 (or double2) to be used. We can load the next two values at once from memory into the float2/double2, then when the new values have been calculated we can store both of them at once using another float2/double2. This again reduces the number of memory operations. I tried unrolling the loop further, but found there was little benefit in doing so.
3) To avoid if statements needed to handle periodic boundary conditions at the edge of the grid I set the inner loop limit as “ncol - 4” and computed the last four values outside the inner loop. Four values instead of just two are needed outside the loop to prevent out of range memory accesses.
4) Each row computes its own two constant values for the first two columns. This means that there is no need to copy a grid to the device. This mostly helps with smaller grid sizes due to the initial high cost of copying even small amounts of data, but it provides a small speed up even at large grid sizes. Note that a result grid must still be allocated on and copied FROM the device.
5) Division by five is replaced by multiplication by ⅕, where ⅕ is computed once at the start and kept in a register. While I knew that division was more expensive that multiplication, I was surprised by the difference this change made. With n=m=10000 and p=1 I found that this small change reduced execution time by almost 0.8 seconds.

I was unable to find an effective way of using shared/texture/constant memory due to their small sizes.

Reduce kernel:

The reduce kernel is relatively simple. Each thread reduces a single row. It uses a loop that is unrolled four times, and a float 4 is used to load and process four values at once. This results in much less memory accesses. This didn’t give as large of a speed up as I hoped though. Perhaps the CUDA compiler was already applying similar optimisations to reduce the number of memory accesses? Compared to my reduce kernel from the previous assignment it is about 30% faster for large grid sizes.

The effects of block size:

My findings here were identical to what I mentioned in my last report. I found the CUDA versions were fastest with a block size of around 64-256. Any smaller and

each SM was either underutilised or unable to hide memory latency by swapping threads. Any higher and there was too much demand on each SM.

The cost of copying to/from the device:

Again my findings were identical to what was mentioned in the my last report. There is an initial high cost to copy a small amount of data, and this grows at a slow rate. This makes it hard for the CUDA version to be faster than the CPU with smaller data sets. As mentioned above, my fast kernel eliminates the need to copy data to the device, leading to better performance with all data sets, but specifically with small data sets.

Results with floats:

With n=m=15360 and p=100 and a block size of 64:

CPU: Took 903200 milliseconds overall
CUDA NAIVE: Took 342681 milliseconds overall (2.635687x faster than CPU)
CUDA FAST: Took 28268 milliseconds overall (31.951323x faster than CPU)

CPU: Reduce took 1059 milliseconds
CUDA REDUCE: Took 83 milliseconds overall (12.759036x faster than CPU)

As shown above the fast CUDA kernel was just under 32x faster than the CPU version. This was the best result that I was able to get. It also shows that a naive CUDA kernel can be faster than a basic CPU implementation, but careful management of memory accesses is needed to get the most out of CUDA.

Results with doubles:

With n=m=15360 and p=100 and a block size of 64:

CPU: Took 838338 milliseconds overall excluding reduce
CUDA NAIVE: Took 710557 milliseconds overall (1.179832x faster than CPU)
CUDA FAST: Took 35525 milliseconds overall (23.598536x faster than CPU)

CPU: Reduce took 924 milliseconds
CUDA REDUCE: Took 127 milliseconds overall (7.275590x faster than CPU)

In the previous assignment I found that the CUDA performance using floats and doubles was similar most of the time. I believe this was because I was being limited by memory accesses instead of being limited by arithmetic operations.

The fast kernel took 35525 milliseconds overall using doubles, but only took 28268 milliseconds using floats. This is a 25% increase in time taken. This suggests that my kernel is still bound by memory operations, but not as badly as in the last assignment.

However the naive kernel was around twice as slow using doubles compared to using floats. As this kernel should be bound by memory operations I do not know why this happened.

Strangely the CPU was consistently slightly faster when using doubles. I cannot explain this.

Precision on the GPU:

I found that my GPU results matched the CPU results within the tolerance of 1.E-5 with both floats and doubles.