I was not able to use matrices of size 30'000x30'000 with CUDA as there was not enough free memory on either GPU available on CUDA01.

Because of the restriction on using shared memory and atomic operations I was unable to come up with a good way of reducing a vector using CUDA. I just used a single thread to sum all of the elements in the vector.

Building and usage:

To build you must specify the data type to use, either float or double. Do this by running either "*make all DATA_TYPE_USED=float*" or "*make all DATA_TYPE_USED=double*".

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.

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:

-n <nrow>: set the number of rows (default 10)
-m <ncol>: set the number of columns (default 10)
-r: seed srand using the current time (default 123456)
-p: 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 8)

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

Recording results:

I created a script called *get_results.sh* that calls the program with various parameters, such as different block sizes, matrix sizes and so on. Each result is saved to a file in the *results/* directory with the parameters as a filename.

I also created a script called *graph.sh* that creates graphs using gnuplot.

Timing on the CPU:

The below tables shows the time taken (in seconds) for each operation using floats and doubles with a 30’000x30’000 matrix

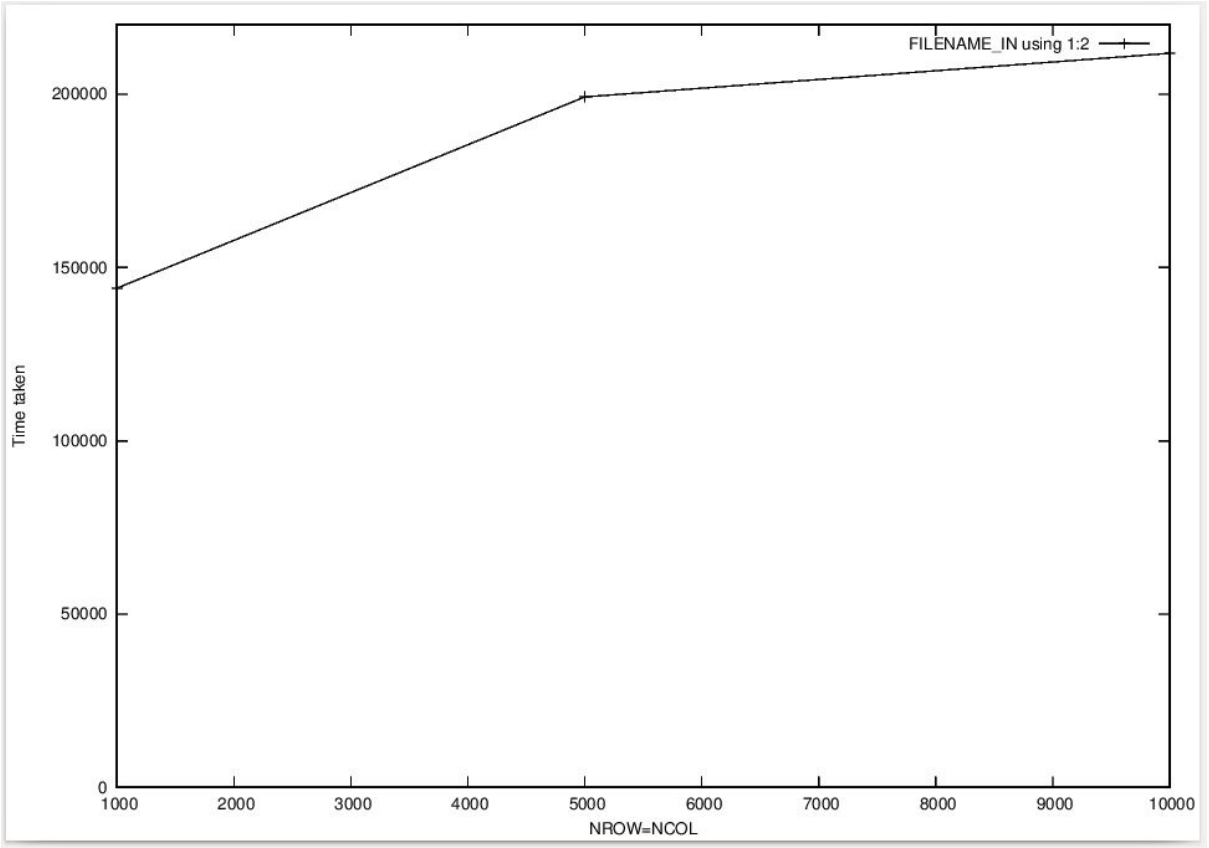
Data type	Summing rows to vector	Summing columns to vector
float	4.3	30.8
double	5.07	31.1

As can be seen summing the rows to a vector is significantly quicker than summing the columns to a vector. This is because the matrix is allocated in a row-major format, so summing the rows into a vector is much more cache friendly than summing columns to a vector.

Interestingly when using doubles the row summing time increases by 16%, while the column summing time taken increases by 0.01%. I think this is because for row summing more pressure is put on the cache leading to worse performance, while the column summing was not using the cache anyway so the increased cache pressure makes no difference.

The cost of copying to the GPU:

I found that there is a significant time cost for copying the matrix to the device. This cost is very high even for small matrices, but I noticed that it increases at a small rate as the matrix size increases. In the below graph you can see that copying a 10’000x10’000 matrix took less than twice as long as copying a 1’000x1’000 matrix, despite there being 100 times as much data to copy. Because of this I would recommend that CUDA be used only when there is a large amount of data to work with, otherwise the high time cost for copying the matrix will outweigh any benefits.



Time taken to copy the data as the data grows.
Original file: graphs/gpu-data-copy-time.eps

CPU vs GPU performance:

The table below shows the time taken to sum the rows into a vector.
As the data size grows the GPU finishes the work faster than the CPU but a large matrix is needed before the cost of the copy to becomes relatively small, as discussed in the previous section.
Time is in microseconds and a block size of 64 was used.

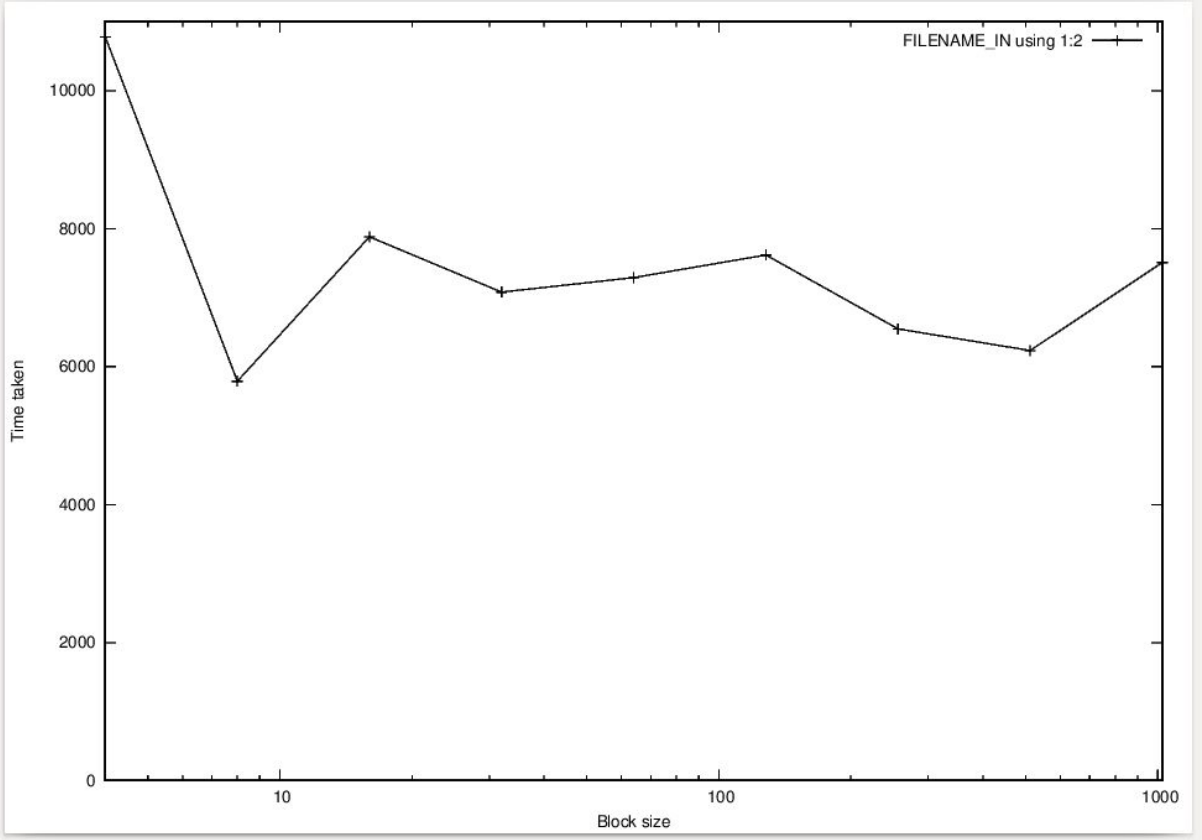
Data size	**CPU time taken**	**GPU time taken (excluding copy)**	**GPU time taken (including copy)**
1’000x1’000	3’948	7’290	134’469
5’000x5’000	97’687	21’831	165’359
10’000x10’000	390’492	49’979	246’988

The effect of increasing block size:

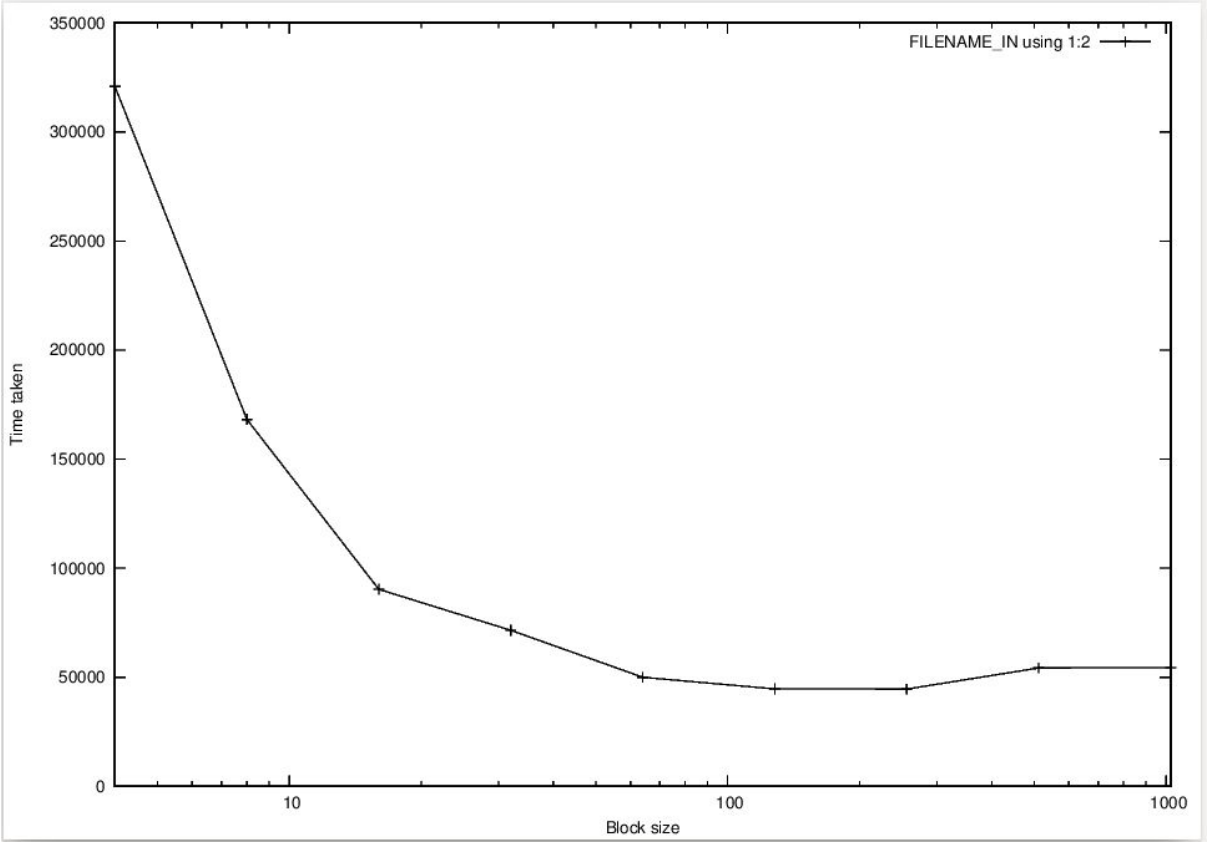
I have included two graphs below. They both show how the time taken to sum all rows into a vector on the GPU. The first graph uses a 1’000x1’000 matrix, while the second graph uses a 10’000x10’000 matrix.

With the 1’000x1’000 matrix we see a small decrease in time taken as the block size increases. However this gain quickly levels off. I believe this is because there is simply not enough data to split between so many threads.

With the 10’000x10’000 matrix we see continuous decreases in the time taken until we go past a block size of 128. With larger block sizes the time taken starts increasing again. I think this because there are too many threads and they are competing for resources. It’s interesting to see that the warp size (32) is not the fastest. Having more threads than the warp size decreases the time taken up until the point mentioned previously. I believe this is because the extra threads can be run while others are waiting for data to be loaded from memory, reducing the idle time. I also believe this configuration scales better with more threads compared to a 1’000x1’000 matrix workload as there is enough work to be distributed to each thread.



With matrix size = 1’000x1’000. Note log scale on x axis
Original file: graphs/gpu-float-size-1000-time-vs-block-size



With matrix size = 10’000x10’000. Note log scale on x axis
Original file: gpu-float-size-10000-time-vs-block-size

Floats and doubles in CUDA:

I expected much worse performance when using doubles on the CUDA device. However performance was quite similar in most cases. I think this is because my code was limited by memory operations - as shared memory was not used the code had to access the slower global memory constantly. This meant that the addition operation was not the bottleneck, so the extra time taken to add doubles did not make a difference.

Precision on CPU compared to CUDA:

I found that the CPU and CUDA versions gave exactly the same answers.

Interestingly on the CPU summing the rows into a vector and then reducing that vector gives a significantly different answer than summing columns into a vector and reducing that vector. In theory this answer should be the same as effectively all elements of the matrix were added together. The difference seems to come from the order of operations being different (by row vs by column), which leads to different results.