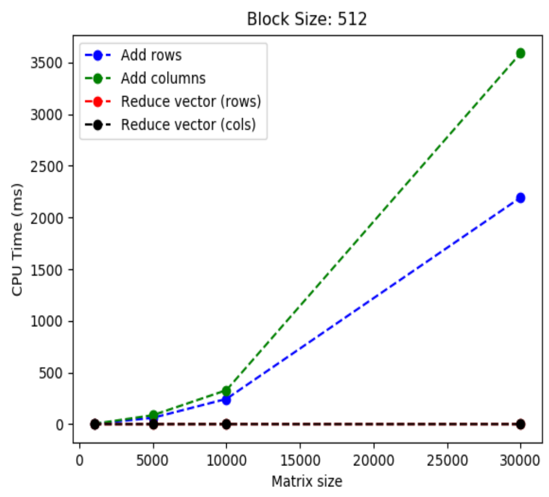
CUDA Programming MA5615, Assignment 1

Shaaz Ahmed, ahmeds1@tcd.ie, 2018-2019

Answers:

1. **Task 1**: The code can be found in `./single_precision/cpu.cu`. Plotting the execution time in ms against the matrix size, we get the following graph (block size is irrelevant).

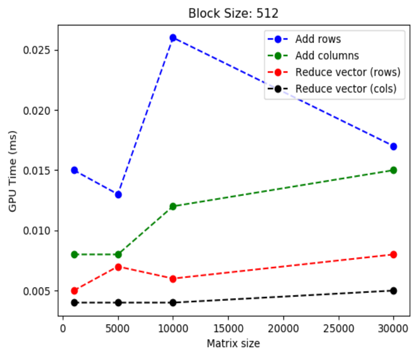


To generate this graph, uncomment the plotting line from ./run.py and run ./run.py in a GPU supported environment

It is evident that for both `add_rows` and the `add_columns` the time taken increases significantly as matrix size increases, while this is much less so for the `reduce_vector`. This is due to the quadratic complexity nature of `add_rows` and `add_columns`, while `reduce_vector`'s running time grows linearly with matrix size.

Also worth noting that `add_rows` is faster than `add_columns` (possibly by a factor) - this is because of the better data locality for `add_rows` as the elements of rows that have to be summed are close to each other in memory.

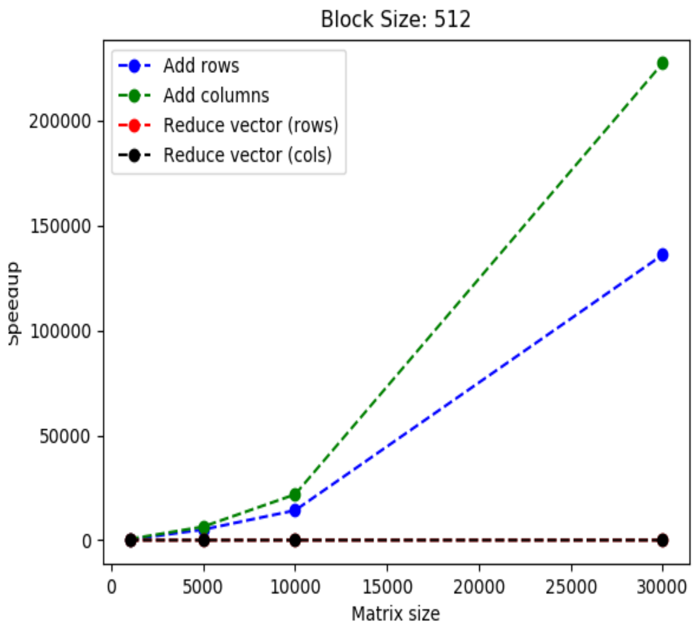
2. **Task 2**: The code can be found in `./single_precision/gpu.cu`. Plotting the execution time in ms against the matrix size for a block size of 512, we get the following graph.



To generate this graph, uncomment the plotting line from ./run.py and run ./run.py in a GPU supported environment

This is clearly orders of magnitude faster and the quadratic operations don't seem to behave like they are quadratic anymore.

3. **Task 3**: The speedup graphs for each block size are in `./single_precision/speedup_graphs`. Plotting the speedup against the matrix size for a block size of 512, we get the following graph:



To generate this graph, uncomment the plotting line from ./run.py and run ./run.py in a GPU supported environment

On the specific system this was run on, running NVIDIA GP107M [GeForce GTX 1050 Ti Mobile], the following are the features of the speedup:

- `reduce_vectors` doesn't have much of a speedup because we've used only GPU thread to sum it up. Since the sum has to be stored in a single variable, only one GPU thread can write to it due to consistency issues, or access to the variable has to be serialized, in which case it's almost equivalent in performance to a single thread writing to it - potentially worse.

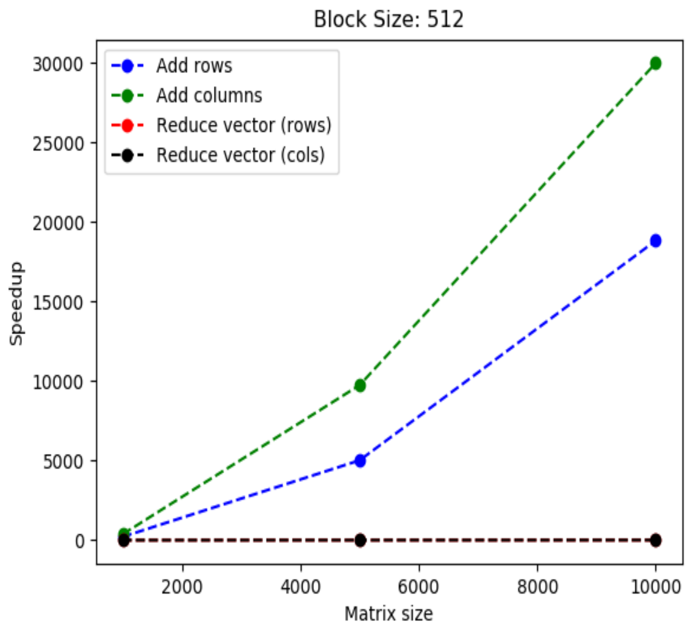
  It is also possible to write this function to use more GPU threads, by splitting the array into evenly-sized sections and summing them separately, using a barrier to synchronize each stage of computation and then further sum their results. This can potentially give a constant factor improvement in performance in the case of large arrays.
- `add_columns` and `add_rows` have very large speedups that gets better with matrix size, but the speedup of `add_rows` is lesser than that of `add_columns` because the CPU version of `add_rows` is slightly faster due to data locality.
- The block size doesn't seem to affect the speedup significantly on the system that this code was run on (which has a maximum of 1024 threads per block dimension allowed).

  Hence, add_rows and add_columns are both worth computing using CUDA, which reduce_vector doesn't have a similar speedup.

4. **Task 4**: The code for double precision and the speedup graphs for each block size are in `./double_precision/`, with a similar code structure. For a block size of 512, the speedup graph is given below.

   The following features are noticeable:

   - The speedup for each of the quadratic operations is much lower than that in the case of single precision arithmetic - nearly an order of magnitude of smaller.



To generate this graph, uncomment the plotting line from ./run.py and run ./run.py in a GPU supported environment

- The block size does affect the speedup - the larger the number of blocks, the higher the speedup, as evident from the graphs.