CUDA
Assignment 1

Hugh Delaney

March 25, 2021

The code for questions 1-3 can be found in the directory `single`, whereas the code for section 4 can be found in the directory `double`.

1 CPU Calculation

Please see `cpu_funcs.cu` for implementation. `./matrix` accepts parameters:

- `-n [num_rows]` specify number of rows
- `-m [num_cols]` specify number of cols
- `-b [block_size]` specify block size.
- `-t` display the time?
- `-r` seed with random value?

2 Parallel Implementation

Please see `gpu_funcs.cu` for kernels. Use `-t` when running `./matrix` to display the CPU vs GPU times and speedups.

`vector_reduction_GPU` contains two internal kernel calls to `reduce0_GPU` and `reduce1_GPU`.

3 Performance Improvement

3.1 Rowsum Speedup vs CPU

Block size	n,m = 1000	n,m = 5000	n,m = 10000	n,m = 25000
4	291	968	2023	4847
8	339	972	2024	4650
16	291	970	1937	5058
32	316	1011	2023	4851
64	315	969	2023	5056
128	291	973	2023	5061
256	299	1013	2114	4848
512	274	1057	1942	4845
1024	274	1015	1939	4846

We are clearly getting a massive speedup over the CPU when doing this kind of operation. The CPU is already quite good at doing rowsums in C due to contiguous data access (since C is row-major), however the incredible parallelism given by CUDA cannot be beaten. As is to be expected, the speedup increases as $m, n \to 25000$, since the proportion of time that data is being transmitted becomes a smaller fraction of the entire operation.

It is also worth noting that the GPU time (see `output/*.txt`) is almost identical for each operation regardless of the matrix dimension. This goes to show that most of the work is merely going to and from the GPU in the first place.

It seems that anything from 16-256 could be chosen as optimal block size. There is some random variation in performance at times (perhaps due to other jobs on the machine) so I think any of these values could be called optimal. They're all pretty good!

3.2 Column Sum Speedup vs CPU

Block size	n,m = 1000	n,m = 5000	n,m = 10000	n,m = 25000
4	1757	11852	19568	65376
8	2181	9368	3764	65684
16	1610	11718	24468	65421
32	1668	11755	24805	52240
64	1630	9522	19810	65891
128	1652	11825	24696	65435
256	1652	11838	19945	66331
512	1658	15682	24600	65529
1024	1650	9444	24628	65865

We get an even better speedup for column sums when compared with the CPU. Not only is this due to the fact that the CPU is worse at column sums than rowsums, but also because we are able to do CUDA column sums an order of magnitude faster than CUDA rowsums (see `output/*.txt`). I'm not sure why this should be the case, if anything we would again assume that column-major C would make CUDA also slightly better at column sums than rowsums, or even almost the same if there was some fancy compiler trickery at work which made the memory access per thread more contiguous.

There seems to be more spread in terms of which block sizes perform well. The smaller block sizes seem to do well with smaller matrix dimensions, whereas bigger block sizes seem to do well for bigger matrix dimensions. For balance we will choose a middle block size (somewhat arbitrarily, since there must be some noise in the data that is causing any outlier values). We will choose 128 as our optimal block size for part 4.

3.3 Reduction Speedup vs CPU

Block size	n,m = 1000	n,m = 5000	n,m = 10000	n,m = 25000
4	0.002	0.002	0.001	0.001
8	0.002	0.002	0.002	0.001
16	0.001	0.002	0.002	0.0008
32	0.002	0.002	0.002	0.002
64	0.002	0.002	0.002	0.002
128	0.002	0.002	0.002	0.001
256	0.002	0.002	0.002	0.002
512	0.001	0.002	0.002	0.002
1024	0.001	0.001	0.001	0.001

Reduction doesn't work well with CUDA! This is due to the fact that we need to combine all data into a single place, which is not easily achieved in CUDA, due to its low arithmetic intensity and also since it requires some global syncs (which I achieved using multiple kernel invocations). If we were able to use the shared memory then maybe the performance would be somewhat improved, but as it stands this is not a task that suits CUDA.

If we were to get an efficient algorithm working using the shared memory and some other fancy optimisation tricks (as discussed in Mark Harris's CUDA reduction walkthrough—which incidentally only ever compares CUDA reductions to other CUDA reductions, not to CPU reduction speeds), then maybe we would be able to beat CPU reduction speeds. But at this point it must be asked: is it worthwhile to spend lots of time sweating over a fiddly algorithm that might only barely beat a CPU performance-wise? The answer is almost always *no*. The best approach for an assignment like this would be to do the matrix operations on the GPU and then offload the reductions to the CPU. Hybrid solutions are always best.

4 Double Precision Testing

Using blocksize 128 we can compare the performance between single and double precision. Please see `double/output/` for data. Run `make test` when in `double` to generate values into `double/output/`.

4.1 Matrix Rowsum Speedup `float` vs `double`

Precision	n,m = 1000	n,m = 5000	n,m = 10000	n,m = 25000
Single	291	973	2023	5061
Double	224	1022	2044	4909

4.2 Matrix Colsum Speedup `float` vs `double`

Precision	n,m = 1000	n,m = 5000	n,m = 10000	n,m = 25000
Single	1652	11825	24696	65435
Double	2158	13148	21866	74403

4.3 Colsum Errors `float` vs `double`

Precision	n,m = 1000	n,m = 5000	n,m = 10000	n,m = 25000
Single	7×10^{-7}	1×10^{-6}	8×10^{-7}	8×10^{-7}
Double	1×10^{-15}	1×10^{-15}	5×10^{-16}	5×10^{-16}

We can see that our code performs just as well if not *better* than our single precision code. This is most unexpected. So why do we use `float`s instead of `double`s, in general? Presumably this is because on most GPUs we will be limited in our calculations by the amount of memory we can have on the GPU at any particular time. Using `float`s allows us to work on twice as many values than if we are using `double`s, so the data throughput can be greater, if we are indeed limited by memory capacity. For this problem, however, where we are not limited by the GPU's memory, it may be preferable to use `double`s. This is because we get the exact same performance, and our error gets better by 8-10 orders of magnitude.