CUDA
Assignment 1

Hugh Delaney

March 25, 2021

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	4826	20144	120456
8	294	5040	19297	125733
16	339	4830	19308	120458
32	316	4839	19296	115626
64	327	4831	20141	120557
128	342	4835	19303	120450
256	274	4824	20167	120445
512	274	4838	19309	115690
1024	274	5042	19322	120480

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 opration.

It seems that 16 is a close to optimal block size, since although it is not the fastest for $n, m = 10000$, it is consistently one of the fastest for each matrix dimension.

3.2 Column Sum Speedup vs CPU

Block size	n,m = 1000	n,m = 5000	n,m = 10000	n,m = 25000
4	1653	51842	279702	3697568
8	1626	64977	276609	3694547
16	2206	86626	280472	3886223
32	1651	64723	277459	3758644
64	1633	65055	281323	3772076
128	1629	65184	275030	3739942
256	1648	64428	275669	2954227
512	2182	51512	277657	3715496
256	1635	65210	277535	3033259

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.

Once again 16 seems like a nice block size. We will use this 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	0.001	0.0005	0.0003
8	0.002	0.002	0.0011	0.0005
16	0.002	0.002	0.002	0.0008
32	0.002	0.002	0.002	0.001
64	0.002	0.002	0.002	0.001
128	0.002	0.002	0.002	0.001
256	0.002	0.002	0.002	0.001
512	0.002	0.002	0.002	0.001
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