Kevin Cho
A20393339

CUDA

Algorithm

The algorithm for the original serial matrix normalization was approximately 3N^2. The new algorithm in my cuda implementation is (3N^2)/ dB_numThreads. Instead of a single thread finding the mean, standard deviation, and doing the normalization calculation for all N, I split the workload N by the number of dB_numThreads specified by a user. Every thread still finds the mean, stand deviation, and does the normalization calculation, but they work on a different column of the matrix based on their threadIdx.x or their thread id. Thus, the bigger N becomes, the better the performance of CUDA.

Iterations

In another iteration, I tried using shared variables in my kernel function such as row, mu, and sigma. My idea behind this was since all the variables were shared in the same thread block, the program would run faster since on-chip shared memory had low latency. It ended up not really changing my runtimes, and it also affected the correctness of my program.

Efficiency

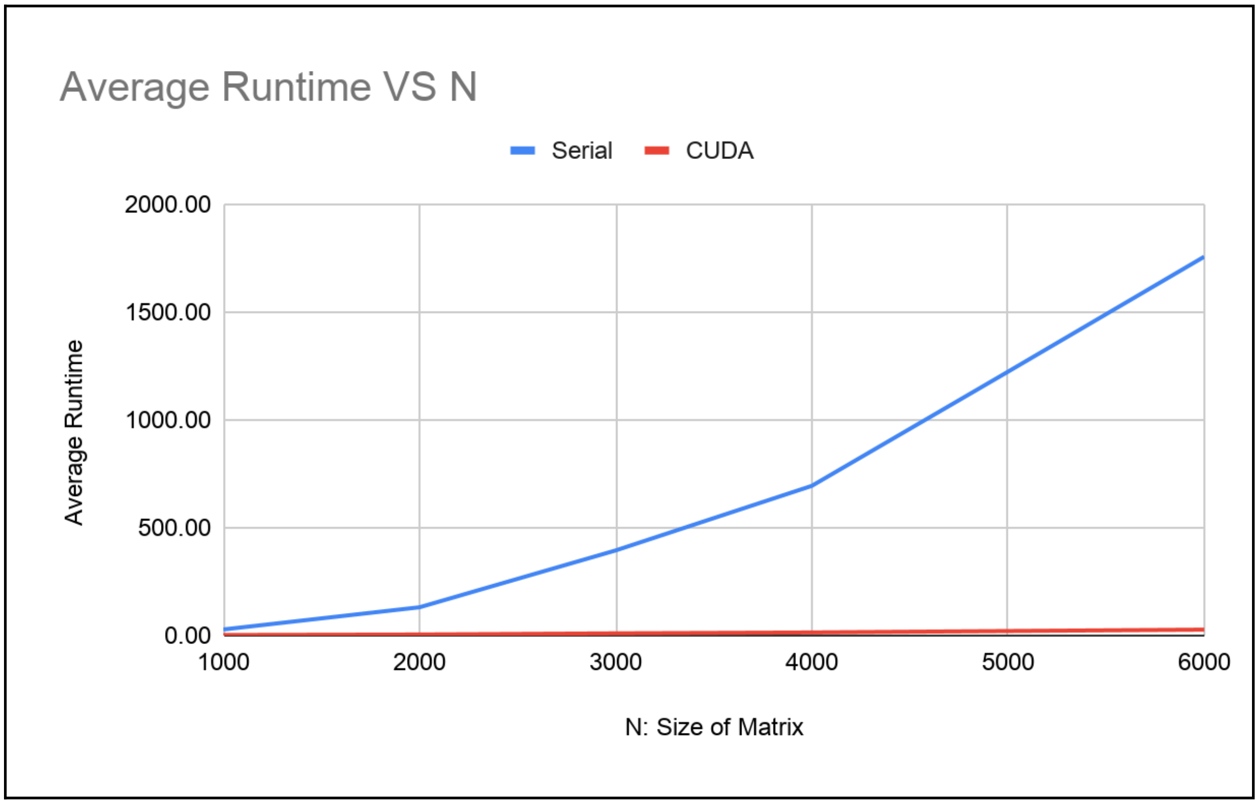
The runtime in GPU for my CUDA program was very fast and efficient compared to the serial code. For total runtime, there was slight overhead by the cudaMallocs I had for A_d and B_d. The cudaMallocs added a slight .05 to .10 ms runtime length to my program, but the overhead was well worth it. The total runtime speed for CUDA was still faster than the serial code for N > 1000 in all cases.

Correctness

I know my solution is correct because the kernel function iterates through N using blockIdx.x * blockDim.x + threadIdx.x which ensures there is no overlap in any of the data. Each thread basically gets its own unique set of columns to work on. Also in the README, I ran some correctness checks on small matrices to ensure that the output was the same for serial and CUDA.

Performance Analysis

Data can be found in README.

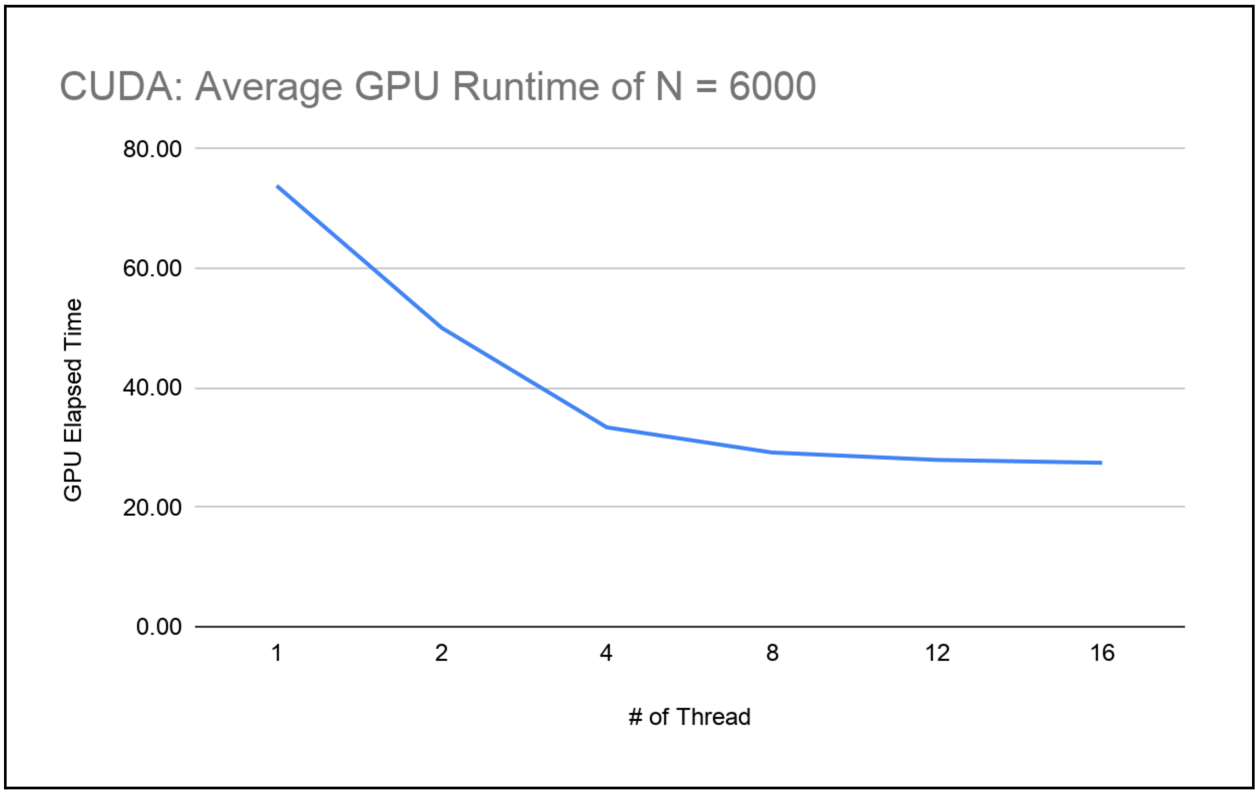


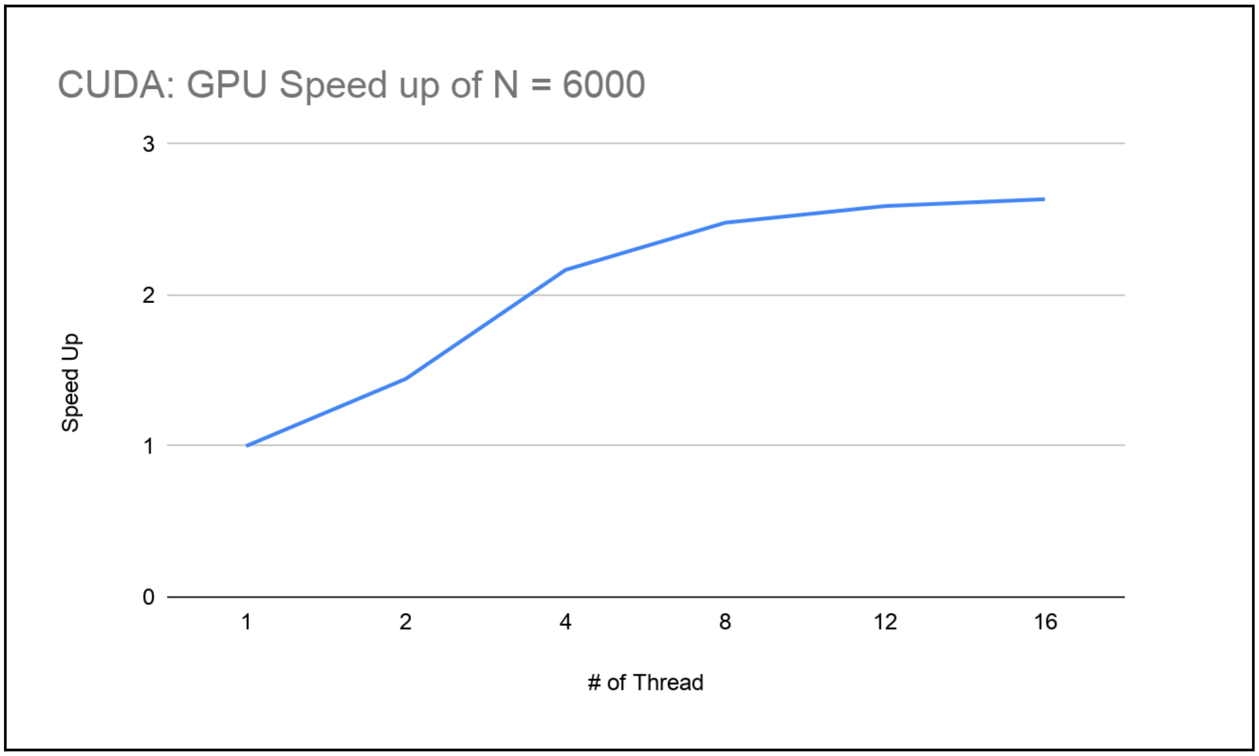
Serial	Runtime 1	Runtime 2	Runtime 3	Average Run
1000	28.87	28.80	27.48	28.39
2000	138.41	131.28	124.00	131.23
3000	382.98	382.88	418.39	394.75
4000	691.17	691.97	698.99	694.04
5000	1238.57	1215.84	1215.33	1223.25
6000	1766.95	1755.04	1746.42	1756.14

CUDA	Runtime 1	Runtime 2	Runtime 3	Average Run
1000	2.88	3.06	2.01	2.65
2000	4.78	5.51	4.73	5.01
3000	10.35	11.44	9.17	10.32
4000	13.79	16.06	13.62	14.49
5000	21.14	19.37	22.94	21.15
6000	25.75	31.21	25.72	27.56

In all cases of N > 1000, CUDA vastly outperforms the serial code in terms of scalability.

Below are graphs comparing CUDA to itself using different # of threads on dG_numBlocks = N





Cuda	Runtime 1	Runtime 2	Runtime 3	Average Run	Speed up
1	70.47	75.04	76.09	73.86	1
2	51.10	50.99	48.10	50.06	1.45
4	32.83	32.19	35.16	33.39	2.17
8	28.90	29.55	29.10	29.18	2.48
12	27.91	27.96	27.95	27.94	2.59
16	27.44	27.79	27.16	27.46	2.64

The CUDA code scales sub optimally but scales nonetheless. The speedup is linear up to 8 threads where it immediately plateau. A larger N size would probably work more efficiently with my CUDA code.