10/4/2023

LAB2 – CS 380P – JOHN PETERSON - JLP5729

1. PART 1 -- SEQUENTIAL CPU IMPLEMENTATION

My algorithm uses flat arrays of points, and largely adapts the structure/framework of the Lab 1 assignment (using main.cpp, argparse, helper, etc). I initially chose to use a struct format and 2D arrays for data, but this quickly became unwieldy with accessing specific items, doing math to compare values, and clearing/allocating/deallocating memory so I switched to a flat array format on advice of TAs in the Ed discussion forum.

I utilized Codio for all programming and measurements of performance. As expected, the program runs trivially fast for small entries and has basically a parallel increase in time for increase in data size. A graph is shown below for total runtime vs. total input data (number of points times number of dimensions):

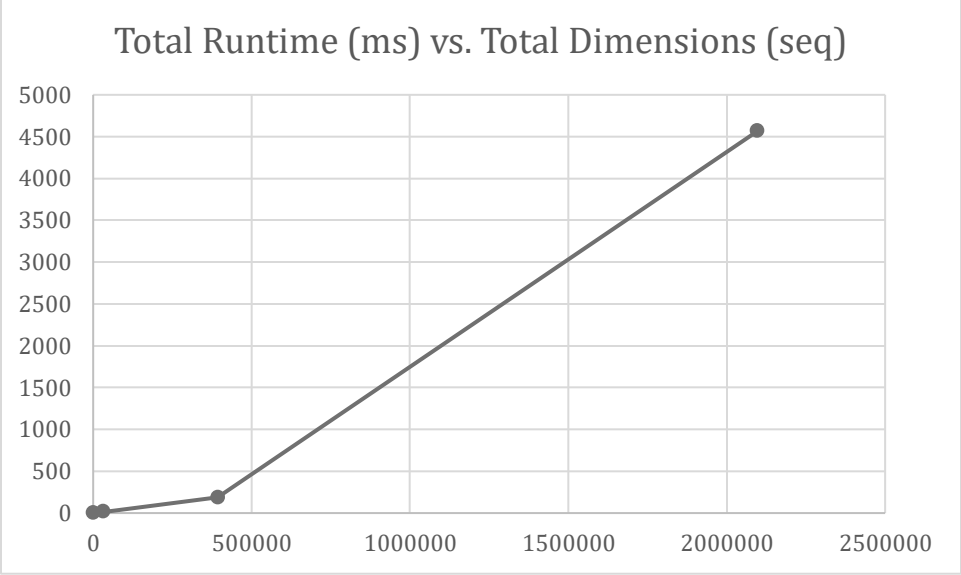

Figure 1: Runtime (total, milliseconds) vs. Total Dimensions of all points time all dimensions for Sequential implementation. Times are only fast for small input sizes.

2. PART 2: CUDA BASIC

The first step in this stage was figuring out the proper syntax and formatting, but the real challenge was thinking in a parallel nature, and just how one could take advantage of the massively parallel hardware in a linear dimension. This was the point at which I decided to switch from 2D arrays to 1D arrays to make the math and indexing easier (and I went back to edit my Part 1 code to match as well). I also encountered some

initial struggles setting up the compiler and file structure; opting for a more consolidated file structure as suggested in the Lab2 guide online.

I also chose to break out more sections of my sequential algorithm to provide more parallelism opportunities, including calculating every distance between every point and every centroid in parallel, and updating the centroid values required splitting out the addition and multiplication steps to avoid data race and dependencies with the math order of operations for averaging.

Finally, debugging also proved challenging as it was difficult to pinpoint with hundreds to thousands of threads and blocks what was actually going on. I made liberal use of print statements in the non-parallel code and thread indexing to make print statements in kernels that wouldn't duplicate hundreds or thousands of times on the console.

This program, as expected, was much slower than sequential (orders of magnitude) on small/trivial inputs. However, it showed a massive speedup over sequential on the two larger test cases, 16384 points with 24 dimensions, and 65536 points with 32 dimensions. The program did not increase at the same rate as sequential due to the parallelism. See a graph below:

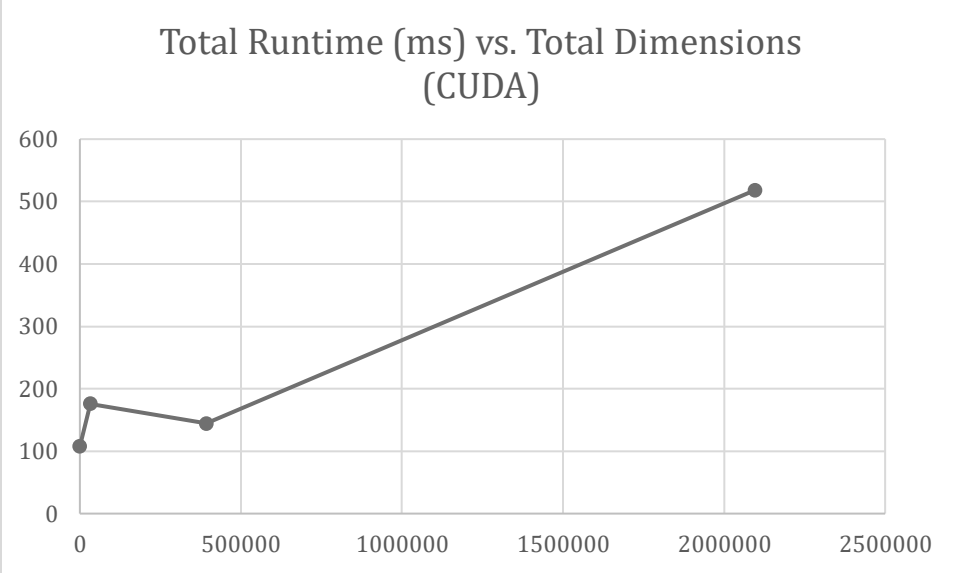

Figure 2: Total Runtime in milliseconds (y-axis) vs. Total Dimensions (number of points times number of dimensions per point) for CUDA Basic

3. PART 3: SHARED MEMORY

For this section, I looked at all my memory accesses and attempted to convert them to local thread-index based shared memory structures. Not all of them were suitable to this conversion; for example, I was not able to copy all possible points into local memory due to the shared memory limit, the size of points and thread blocks, and required data size. I was, however, able to copy centroids, selections of points as necessary to make calculations, and my counts and labels vectors.

This did result in a speed-up for all input samples except the smallest/trivial examples. It was faster on the provided inputs than the CUDA Basic implementation. This makes sense to me, because we are reducing cache conflicts and reads/writes to external buses. We are able to index by thread, copy only the memory necessary to start, and copy out only what has changed or is necessary in one block at the end; this reduces contention and cache conflicts.

There was more overhead required, leading to higher times at low dimension inputs.

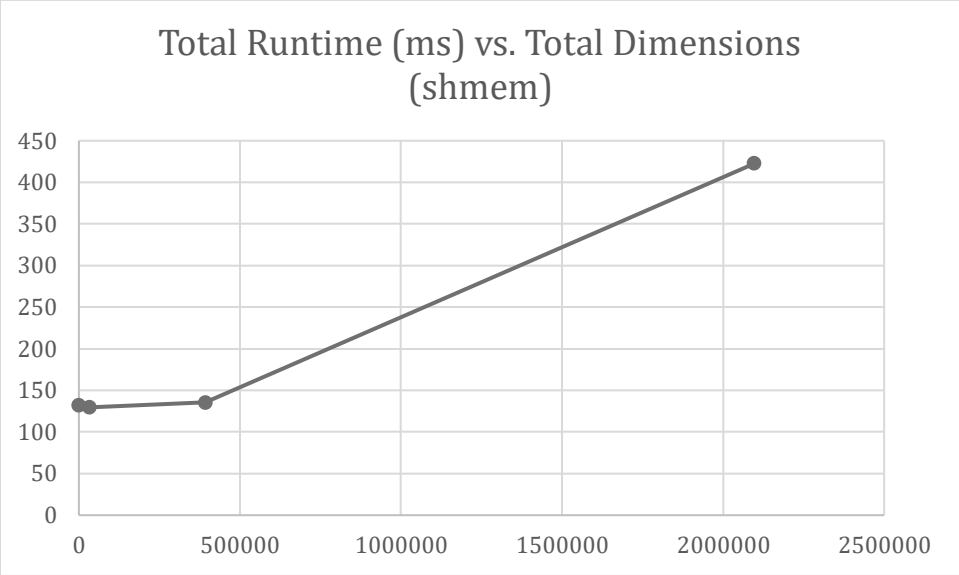

Figure 3: Total Runtime in milliseconds (y-axis) vs. Total Dimensions (number of points times number of dimension per point) for Shared Memory CUDA

4. PART 4: THRUST

This section proved the most challenging for me, primarily due to lack of time. I chose to try to represent each of my functions from the CUDA/Shmem implementations as a combination of functions, functors, and reductions in Thrust. However, I prioritized writing code and ran out of time to debug, so the program unfortunately returns incorrect results for the small/trivial inputs and crashes on larger outputs. Had I had more time I would have gone through functor by functor, line by line, adding print statements and trying different size/shape inputs to determine what was going wrong.

I am optimistic that with more time I could get it working. Additionally, I would have preferred to "start from scratch" and focus on what functions could be best suited for Thrust implementation as part of K-means, instead of "shoehorning" in my previous logic to Thrust implementation. As such, I do not have performance graphs or data to add for this section.

5. ANALYSIS

I performed all coding, measurements, and analysis on Codio. For runtimes, I took five consecutive runs of each input case for each version of the program completed, and averaged them. I have included a chart of the average measurements for each example below:

Version	Input points	Total dims	Runtime (total)	Runtime (per iteration)	Iterations	Runtime (data transfer)	Percentage (data transfer)
Seq	4	8	0.0215162	0.0005592	2	n/a	
Seq	2048	32768	13.04306	0.721744	18	n/a	
Seq	16384	393216	186.5974	7.4623388	25	n/a	
Seq	65536	2097152	4566.672	39.0310148	117	n/a	
CUDA Basic	4	8	107.67246	0.4316152	2	0.0573188	0.054%
CUDA Basic	2048	32768	175.7052	0.716587	18	0.3624172	0.210%
CUDA Basic	16384	393216	144.6092	0.771379	25	1.9994842	1.393%
CUDA Basic	65536	2097152	518.4994	3.2304254	117	8.1218092	1.568%
CUDA Shmem	4	8	131.7852	0.2846378	2	0.0594178	0.045%
CUDA Shmem	2048	32768	129.5854	0.424822	18	0.321142	0.248%
CUDA Shmem	16384	393216	135.6118	0.6941112	25	1.9917382	1.471%
CUDA Shmem	65536	2097152	422.549	2.501495	117	7.7224436	1.878%

Table 4: Performance Times for Program Versions. Large inputs perform better on parallel implementations

While the sequential version performed fastest on the smallest inputs, such as four points with two dimensions and two centroids as well as 2048 points with 16 dimensions and 16 centroids, the Shared Memory performed fastest on all other inputs. CUDA Basic provided a significant speedup over sequential, but Shared Memory was even faster. This aligned with my expectations, as we have learned in lecture the user-managed cache on the CUDA devices provides for much less overhead if configured properly.

In all provided examples, we were able to program one thread to handle one input point and centroid combination (the Codio hardware on the Tesla T4 was able to issue up to 1024 threads per block. However, my implementation could have been improved by further parallelization of the Update Centroids function, which was only parallelized to the number of centroids chosen. I considered adding parallelization of distance function for calculating and updating all dimensions at the same time prior to summing, but this seemed like diminishing returns when I still needed to complete Shared Memory and Thrust. We can see from the chart that even with the parallelized version and large inputs/outputs, a very small portion of the time is spent for data transfer (less than 2% in all cases). This means, that with Ahmdahl's law, we are able to get a parallel speedup on over 98% of the sequential programming costs. Of course, there is overhead to set up the CUDA device and threads, and the data transfer, but we should be able to get close to a 15-16x speedup for an implementation that has 16 centroids.

For the largest example, 65536 points for 2097152 total dimensions, we were able to achieve a 8.8x speedup in overall runtime for CUDA Basic and 10.8x speedup for Shared Memory. Given the overhead of the CUDA devices, this makes sense and while I did not do more in-depth measurements than those in the table above, I would like to investigate further where exactly the overhead plays the largest part, and how that can reduce.

My CUDA Basic implementation is slowest, and I cannot compare it to the Thrust implementation as I did not get it working. As mentioned above, I expected shared memory to be slightly faster, so this matches expectations due to the nature of memory accesses on CUDA devices.

The chart above shows the fraction of end-to-end runtime for each CUDA version, but overall, it was somewhere between 0.05% to 1.878%. This is not a negligible amount, but it is small enough that we can expect large performance boosts from optimally parallelizing the work.

Overall, I spent approximately 50 hours working on the lab. This included 8-10 hours to get familiar with the topics, complete required, optional, and other readings to better understand CUDA, about 12 hours to get my sequential implementation working, approximately 12 hours on CUDA Basic, 6 hours on shared memory, and another 8 on Thrust implantation before running out of time and shifting to the report, on which I spent about two hours.

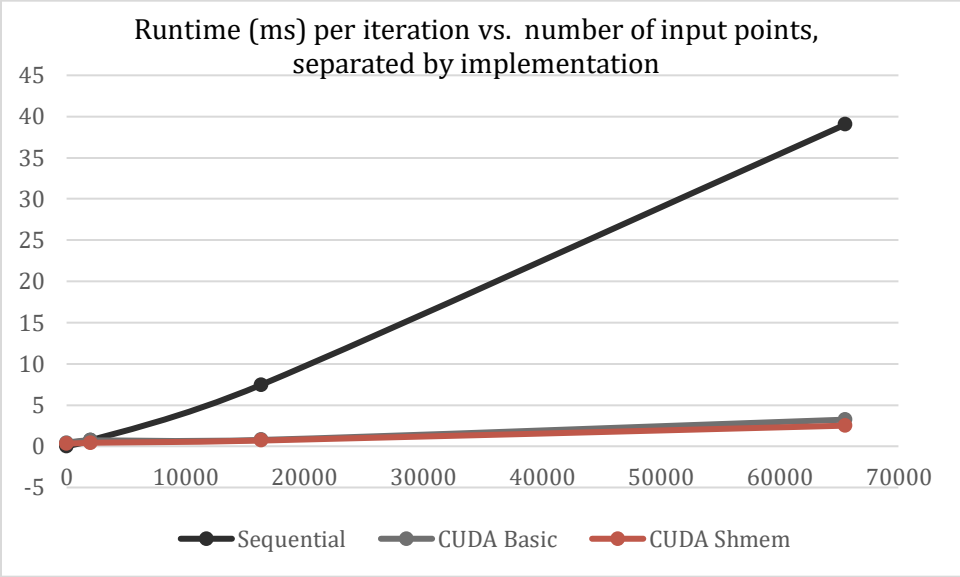

Figure 5: Runtime per iteration in milliseconds for Input Size (number of points on x-axis), CUDA and Sequential. Both CUDA implementations are significantly faster than sequential for large inputs, and Shmem has a slight advantage as input size increases.