Concurrency Assignment #3: k Means Lab

Ryan O'Leary

March 30, 2021

1 Overview

This lab took me about **34 hours** to complete. Around 20 hours was spent coding the different implementations of k-means as well as debugging. Approximately another 7 hours was spent testing my implementations through both black-box and white-box testing to ensure that the final centroids were correct. Finally, the remaining 7 hours was spent on the report and analyzing the timing data that I had collected.

2 Methodology

I collected my results by running trials on the UTCS host pedagogical-1 which operates on x86-64 architecture with a Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz version processor. There are 64 processing units and the OS Version is Ubuntu 18.04.5 LTS. For the GPU, nvidia-smi returned a Nvidia driver version of 450.102.04 and CUDA version of 11.0. The GPU being used was a Matrox G200eW3 Graphics Controller (rev 04). For each of my implementations, I took repeated samples using the given input data and averaged the results to determine end-to-end run-time and speedup between solutions.

To compile my code, I included a Makefile with the build statements specified in the instructions, with the addition of lboost_program_options since my implementation uses the boost library to parse command line arguments. Additionally, I added the optional flag -a followed by a String to specify the answer txt file such that I could test my output against the given test output.

3 Sequential Algorithm

For my baseline sequential implementation, I chose to implement Lloyd's Algorithm as was given in the example psuedocode. To do this, I first had to read in the coordinate values from the input file, which I chose to store in an array of "Points", a struct I defined which held cluster ID, an array of floats representing coordinate dimensions, and the minimum distance to the nearest centroid. Next, I defined the set of initial K centroids by randomly selecting points using the given seed. With the initial set of centroids chosen, the algorithm loops and respectively re-clusters points and recomputes centroids until the centroids have "converged", or no longer update by a significant amount (specif iced by the threshold value) after each re-clustering. Re-clustering simply involved computing the nearest centroid to every point and assigning that point to its "cluster" using its centroid ID (i.e. the labeling step). Distance between two points was calculated by implementing the Euclidean distance formula for N dimensions (loop N times and sum the squared differences of each corresponding dimension for the two points, then square root the sum). Recomputing the centroids required calculating the mean coordinate value of each cluster as well as the cluster sizes so that each centroid Point could be set to the mean of their respective clusters.

3.1 Time Complexity

- assigning centroids: O(N points * number clusters * dimensions)
- computing sums: O(N points * dimensions)
- recomputing centroids: O(number clusters)

3.2 Space Complexity

- O(N * dimensions) for the N-Point array
- O(number clusters * dimensions) for the array holding the centroids and the arrays of cluster dimension sums and cluster sizes

3.3 Timing

Baseline for speedup calculations, trials taken with threshold value of 1e-10:

* time per iteration includes time taken to pick the initial centroids

* end to end runtime includes file I/O and the output of results

* timing values were averaged over 10 trials

N = 2048

- time per iteration = 3.702 ms (averaged over 10 iterations)
- end to end runtime = 43.680 ms

N = 16384

- time per iteration = 44.394 ms (averaged over 8 iterations)
- end to end runtime = 429.006 ms

N = 65536

- time per iteration = 225.185 ms (averaged over 6 iterations)
- end to end runtime = 1727.991 ms

For the three main steps of computation, we saw average time spent per iteration of:

Assigning Points to Centroids:

- **N = 2048**: 3.14621 ms
- **N = 16384**: 34.55488 ms
- **N = 65536**: 181.44518 ms

Summing Cluster Dimensions/Size Values:

- **N = 2048**: 0.62051 ms
- **N = 16384**: 7.11845 ms
- **N = 65536**: 37.01238 ms

Computing New Centroids:

- **N = 2048**: 0.01806 ms
- **N = 16384**: 0.02732 ms
- **N = 65536**: 0.03609 ms

From this data, we can draw a number of conclusions. First, it's easy to see that our end-to-end runtime is dominated by the computation steps of our algorithm, with file I/O making up a rather small percentage of our total runtime. Thus, by parallelizing these steps of computation we can expect to see significant speedup. Second, assigning points to clusters took by far the most time per iteration of the three major steps, with summing dimension/cluster values also taking a significant amount of time and the step of computing new centroids taking a negligible amount of time comparatively. These comparative timing values are about what we'd expect to see given the respective time complexities of these steps.

4 Basic CUDA

I implemented K-Means in CUDA by moving certain sections of computation from my sequential implementation of Lloyd's Algorithm to be ran on the GPU. Specifically, I created three new kernels to respectively handle assign points to "clusters" based on the nearest centroid, sum the coordinate dimensions of points in a cluster as well as the sizes of clusters, and finally to compute the mean values of each cluster and reassign centroids.

4.1 Labeling Points

Point "labels", or the centroid they were associated with, was represented by assigning the cluster ID value held by each Point to the desired centroid ID. To implement this step in CUDA, I created a kernel dedicated to assigning each point to a centroid. This kernel took the array of Points as well as the array of centroids as arguments. I setup my kernel initialization such that each thread block would hold 1024 (the maximum allowed) threads and that there would be (N Points + 1023) / 1024 thread blocks. This ensured that there would be N Points threads running on this kernel, and that the point ID being worked on could be computed with blockIdx.x * blockDim.x + threadIdx.x. As a result, the work done within the kernel was simply to loop over each centroid and compute the one closest to the Point being worked on, assigning its ID to the cluster ID value within the Point. Additional overhead required of this step (as opposed to the sequential version) involved mallocing device memory for the point and centroid arrays as well as data transfer between host and device code before assigning the nearest centroids.

4.2 Computing New Centroids

To compute new centroids, I moved the steps of summing the cluster dimensions/sizes and setting the old centroids to the mean value of each cluster to two separate kernels. The first, sumDimensionsKernel, was initialized with 1024 threads and (N Points + 1023) / 1024 thread blocks such that each thread (out of the total) represented one point. The kernel code then simply involved using atomicAdd to add the dimensions of the given point to the desired index of the sumDimensions array (2D-array representing the summed coordinate values of all the points associated with each centroid) and to increment the desired index of the array holding cluster sizes. The second kernel, updateCentroidsKernel, took the previously computed sumDimensions and clusterSizes arrays as arguments. I initialized this kernel with 1 thread block (since it was guaranteed that the number of clusters would be less than 1024) and centroids threads. Since each thread represented a single centroid, it was simple to compute the new centroid coordinate values by dividing the value of each index in sumDimensions by the cluster size value held in clusterSizes and passing them to the centroid with atomicExch. Overhead associated with these two steps was the time it took to pass data between host and device memory (done before the call to sumDimensionsKernel to update the point assignments, after sumDimensionsKernel such that we could compute the means in updateCentroidsKernel, and after updateCentroidsKernel to retrieve the new centroid values), and mallocing memory on the device for the sumDimensions and clusterSizes arrays.

4.3 Convergence

This step occured in the sumDimensions kernel, and was implemented by assigning a Boolean to true if any of the newly computed centroid dimensions differed from their previous value by a set threshold. Since this was done while the new centroids were being computed, it required no additional time complexity. If the value of the Boolean was ever not equal to true upon completion of device code, the while loop in host code would end and our final centroids would be output.

4.4 Timing

Used for speedup calculations, trials taken with threshold value of 1e-5:

* same command line arguments used as sequential trials

* timing values taken using cudaEvents as opposed to chrono for the sequential version

N = 2048

- time per iteration = 2.209 ms (averaged over 9 iterations)
- end to end runtime = 28.181 ms

N = 16384

- time per iteration = 16.178 ms (averaged over 8 iterations)
- end to end runtime = 204.859 ms

N = 65536

- time per iteration = 94.139 ms (averaged over 6 iterations)
- end to end runtime = 932.637 ms

For the three main steps of computation, we saw average time spent per iteration of:

Assigning Points to Centroids:

- **N = 2048**: 0.349337 ms
- **N = 16384**: 0.498912 ms
- **N = 65536**: 1.324261 ms

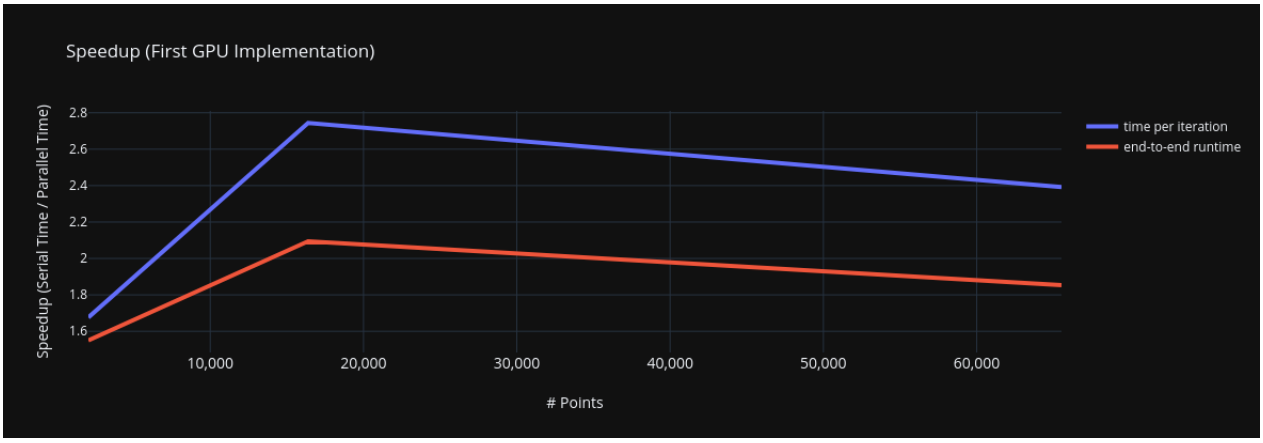
Summing Cluster Dimensions/Size Values:

- **N = 2048**: 0.055012 ms
- **N = 16384**: 0.184043 ms
- **N = 65536**: 0.655317 ms

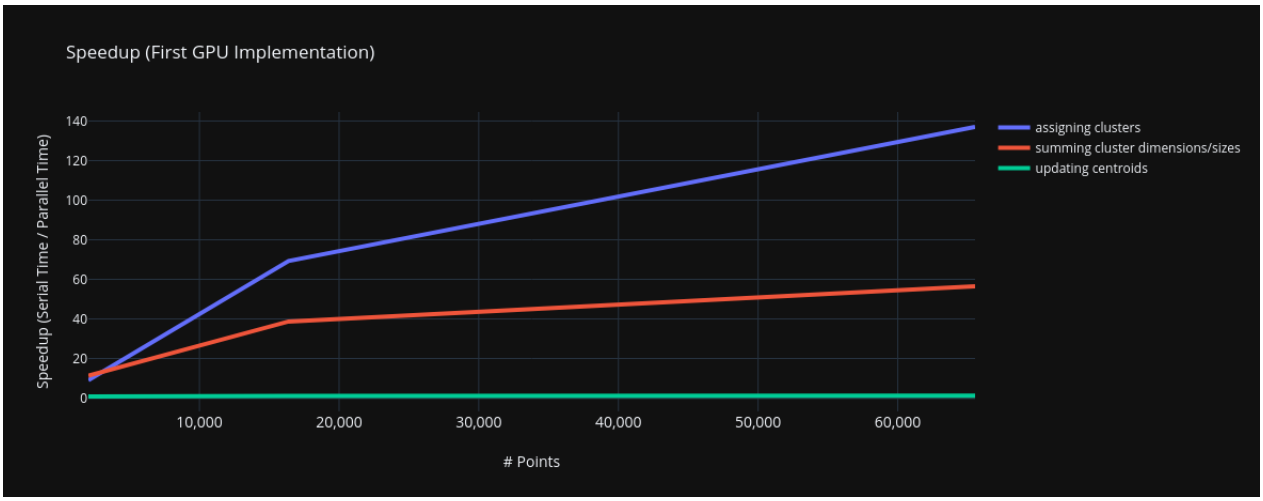
Computing New Centroids:

- **N = 2048**: 0.022368 ms
- **N = 16384**: 0.027001 ms
- **N = 65536**: 0.030704 ms

4.5 Results and Discussion



As seen in the graph, this implementation saw significant speedup over the sequential K-Means solution, with respective time-per-iteration speedups of 1.676x, 2.744x, and 2.392x and end-to-end runtime speedups of 1.550x, 2.094x, 1.853x for the 2048, 16384, and 65536 N Points data sets.



The above graph breaks down the respective speedups of the three parallelized steps in the algorithm. For the assigning points to clusters step, we were able to see massive speedups of 9.006x, 69.260x, and 137.016x for the 2048, 16384, and 65536 N Points data sets. Next, for the summing point dimensions/sizes of clusters step, we saw significant speedups of 11.280x, 38.679x, and 56.480x for the 2048, 16384, and 65536 N Points data sets. With these speedup values, we were nearly able to achieve ideal speedup for summing dimensions/cluster sizes for large data sets and we actually recorded super linear speedup for the assigning points step (likely as a result of data being stored in the cache and our less than ideal sequential implementation). Finally, for the updating centroids steps, we saw negligible speedups (and even slow-downs) of 0.807x, 1.0118x, and 1.175x for the 2048, 16384, and 65536 N Points data sets. Looking at the time complexities of these respective steps in the sequential solution, the speedup values are about what we'd expect to see, with the large big O values of steps that iterated through all N points being greatly reduced by parallelization. Conversely, for the updating centroid step, with a comparatively small big O value, the overhead of copying memory to the GPU and initializing the kernel minimize the amount of speedup we are able to see.

5 Shared Memory CUDA

5.1 Implementation

To use shared memory, I altered my first CUDA implementation by creating a new kernel, updateCentroidsSharedMem, that handled the steps of summing cluster dimensions/sizes and updating centroids. I initialized updateCentroidsSharedMem with 1024 threads and (N Points + 1023) / 1024 thread blocks such that each thread represented one point. Using the index of the given point, updateCentroidsSharedMem then dynamically created a shared array to store all the associated points with one thread block. Each thread could then store its associated point into the shared point array. Once this was done and all threads were synced, I then had thread 0 in each thread block iterate over its shared array and help sum the associated dimension and size values for each point's cluster. After the threads were once again synced, the first num clusters threads updated the centroid values and passed it back to global memory. Thus, the only work shared among all threads was storing Point data in the shared array, while (N Points + 1023) / 1024 threads computed the sum dimensions and cluster size values and num clusters threads updated the centroids.

5.2 Timing

Used for speedup calculations, trials taken with threshold value of 1e-5:

N = 2048

- time per iteration = 3.216 ms (averaged over 9 iterations)
- end to end runtime = 35.449 ms

N = 16384

- time per iteration = 19.8503 ms (averaged over 7 iterations)
- end to end runtime = 209.372 ms

N = 65536

- time per iteration = 25.710 ms (averaged over 26 iterations)
- end to end runtime = 1034.1636 ms

For the three main steps of computation, we saw average time spent per iteration of:

Assigning Points to Centroids:

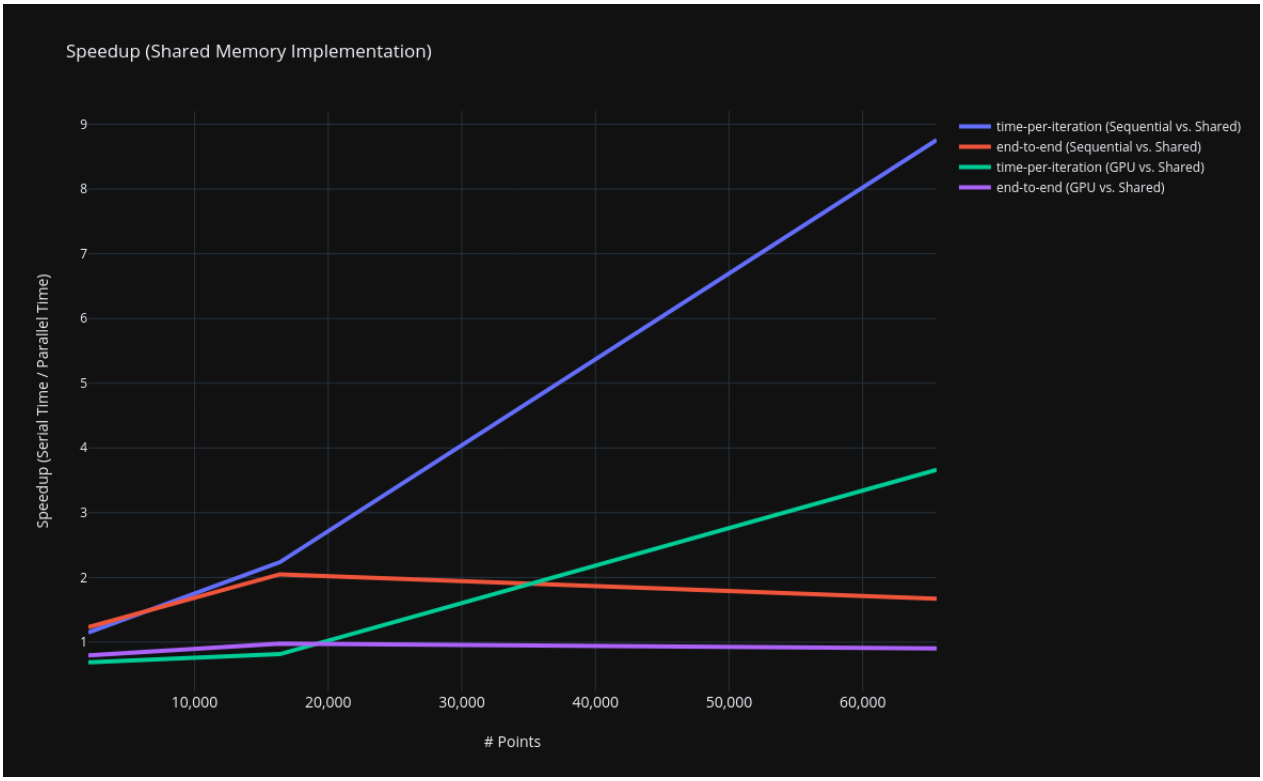
- **N = 2048**: 0.347947 ms
- **N = 16384**: 0.528590 ms
- **N = 65536**: 1.127774 ms

Summing Cluster Dimensions/Size Values and Updating Centroids (combined in one kernel):

- **N = 2048**: 1.066684 ms
- **N = 16384**: 1.447255 ms
- **N = 65536**: 1.793479 ms

5.3 Results and Discussion

By using shared memory to hold the point data, which had the largest amount of reads on it for each iteration, I was able to see significant speedup since reads on shared data with CUDA are much faster than those on global memory.



As seen in the graph, this implementation saw significant speedup over the sequential K-Means solution, with respective time-per-iteration speedups of 1.151x, 2.236x, and 8.759x and end-to-end runtime speedups of 1.232x, 2.049x, and 1.671x for the 2048, 16384, and 65536 N Points data sets. Compared to the Basic CUDA implementation, the shared memory implementation saw time-per-iteration speedups of 0.687x, 0.815x, and 3.662x and end-to-end runtime speedups of 0.795x, 0.978x, 0.902x for the 2048, 16384, and 65536 N Points data sets. For the first two data sets, the Basic CUDA implementation is actually faster, with lower end-to-end and time-per-iteration times. However, we can see the end-to-end speedup approach 1 as the size of the data set increases, and the time-per-iteration was significantly faster with shared memory for the largest data size. It's important to note that the shared memory implementation took more iterations to converge, and there was generally a higher degree of non-determinism in the number of iterations as well as execution time for the shared memory trials. I believe the first CUDA implementation was faster most of the time because it had a higher degree of parallelism as compared to the shared memory implementation. For the first GPU implementation, N points threads compute not only the new labels, but also the cluster dimension sums. Additionally, num clusters threads compute the new centroid values. Comparatively, for the sum dimensions step in the shared implementation, only (N Points + 1023) / 1024 threads sum the cluster dimensions/sizes. Furthermore, the additional overhead of loading data to/from shared memory decreases the speedup we might see between these two implementations for smaller data sets. Thus, the first CUDA implementation was generally faster, but the shared memory implementation had the potential to be faster with larger sets of points or perhaps if we increased the degree of parallelism as well as the amount of data that we load into shared memory (since we currently only load the point array).

6 K-means++

6.1 Implementation

This implementation was identical to the first GPU implementation with the exception of the assigning initial centroids step. I assigned the initial centroids for K-means++ following the psuedocode given in the instructions and by computing "D-Values" in a new CUDA kernel. To compute D-values, I created a new

kernel wrapper to start the kernel, computeDValue, with 1024 threads per block and (Points + 1023) / 1024 thread blocks, such that there would be N points total threads. Inside the computeDValue kernel, I used the computed index of the given point to find the distance to that point's closest centroid. This involved simply looping through all the initialized centroids and finding the distance to each centroid, replacing the D-value of the point if the newly computed minimum distance was less than the current D-value. This CUDA step was called on each iteration of the while loop to compute new D values for each point until num clusters centroids had been chosen. Additional overhead associated with this implementation (as opposed to the first GPU implementation) included mallocing host and device centroid arrays on each iteration of the while loop (since our centroid array is being built with each iteration) as well as cudaMemcpy calls to transfer the new centroid arrays from host to device and the newly computed D-Values from device to host.

6.2 Timing

Used for speedup calculations, trials taken with threshold value of 1e-5:

N = 2048

- time per iteration = 18.938959 ms (averaged over 2 iterations)
- end to end runtime = 44.680672 ms

N = 16384

- time per iteration = 131.889313 ms (averaged over 2 iterations)
- end to end runtime = 335.333344 ms

N = 65536

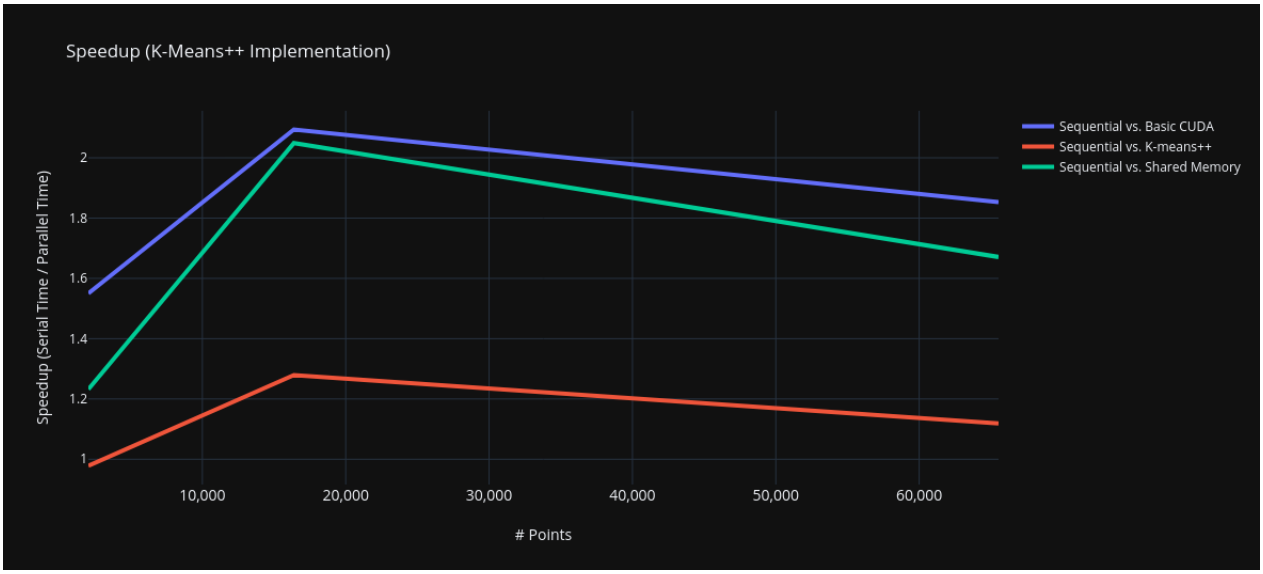
- time per iteration = 589.468262 ms (averaged over 2 iterations)
- end to end runtime = 1543.709595 ms

For a K value of 16, the time to initialize centroids for the following N point arrays was:

- **N = 2048**: 18.511295 ms
- **N = 16384**: 134.162338 ms
- **N = 65536**: 542.453979 ms

6.3 Results and Discussion

The K-means++ initialization step added quite a bit of additional overhead as compared to the initial GPU implementation but decreased the number of needed iterations for the centroids to converge as a result of the more accurate centroid initialization. The actual time per iteration (not including centroid initialization) is about the same as the first GPU implementation despite the end-to-end runtime being higher, however we are still able to see speedup over the sequential implementation when looking at the end-to-end runtime comparisons for larger data sets (didn't include time per iteration speedup on the graph because it's dominated by centroid initialization). Furthermore, the time spent in the actual algorithm was lower for K-Means++ than either of the two other parallel implementations, however the time saved as a result of fewer iterations was overshadowed by the large amount of time taken to initialize the centroids.



This graph displays speedup for end-to-end runtime of the three different parallel implementations vs. the sequential solution. It's apparent that the K-Means++ solution displays speedup over the sequential solution, but is far slower than both the Basic CUDA and Shared Memory implementations. However, it's likely that K-Means++ could be the fastest implementation if we found a way around the large amount of data transfer overhead which dominates its runtime, such as using the thrust library or host pinned memory. The K-means++ solution would also likely be much more competitive for speedup over larger data sets than those used for the trials, since the initial overhead would amortize out with the time saved through fewer iterations to converge.

7 Analysis

7.1 Best Implementation and Best-Case Performance

The fastest implementation is the first GPU solution. This matched my expectations since this implementation had the highest degree of parallelism while minimizing the associated overhead. For much larger data sets, I believe it's likely that the K-Means++ and Shared Memory implementations would see higher speedup in end-to-end run times, however for the N points that I tested with the overhead associated with storing/retrieving shared memory and the high asymptotic running time of K-Means initialization increase the execution time by too much to see speedup over the first GPU implementation (although the trends on the speedup graphs indicate they could surpass it).

The number of processing contexts supported by the hardware is 64, which means at any given moment only 64 threads could be executing in parallel (for each step except updating centroids we have N points total threads which is far greater than 64). For our parallel implementation of Lloyd's algorithm, we'd achieve ideal speedup when our speedup is equal to 64 (or at least a speedup factor of 64 for the assigning points to clusters and summing cluster dimensions/sizes steps), or the number of available processing contexts. Therefore, the best-case performance the parallel implementations could hope for is end-to-end run times of 0.6825 ms, 6.703 ms, and 26.999 ms for the 2048, 16384, and 65536 N Points data sets. This is significantly far off the results of each of the solutions, including the fastest GPU implementation, for a number of reasons. First, a significant percent of the end-to-end execution time is made up of file I/O and other code segments like setting/filling up arrays, etc. that are executed serially. The only parallelized portions of the code are the three main steps of computation, which for the CUDA implementations require a large amount of data transfers/working with host and device memory before, in-between, and after each step. As a result, our actual speedup is bounded by a large amount of serial code. Additionally, it's necessary to call cudaDeviceSynchronize between the steps of computation and before copying data back from the device

(i.e. our algorithm must follow a serial ordering on each iteration), which can be very slow as it forces the host code to wait for the kernels to finish execution of every thread. As a result, although we do achieve significant speedup with our parallel implementations we are not able to achieve linear speedup.

7.2 Slowest Implementation

The slowest parallel implementation was the K-Means++ solution, however, it was also the implementation that required the least amount of iterations to arrive at the correct answer. This matched my expectations because of the large amount of overhead that was associated with choosing the initial centroids for this implementation. The while loop within K-Means initialization iterates at least K times to add each centroid, and on each iteration it's necessary to compute the new D-Values of all N points. The large amount of data transfers (centroid array to device, new D-values back to host on each iteration) was ultimately almost as much as the time spent in data transfers during the actual algorithm. Thus, while the K-Means++ implementation required less time to arrive at the correct centroids (since it needed to iterate very few times), it was the slowest implementation since its end-to-end runtime was dominated by centroid initialization and data transfer.

7.3 Data Transfer

Timing values for end-to-end runtime spent in data transfer:

Basic CUDA Implementation:

- **N = 2048**: 15.976 ms - 56.7% of end-to-end runtime
- **N = 16384**: 131.540 ms - 64.2% of end-to-end runtime
- **N = 65536**: 532.238 ms - 57.1% of end-to-end runtime

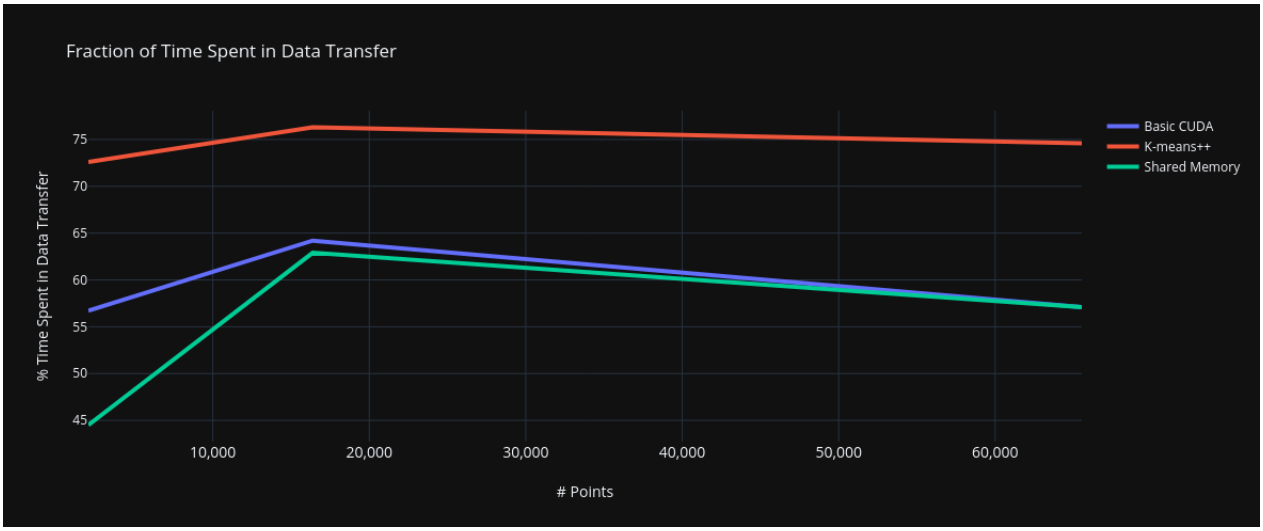
CUDA with Shared Memory Implementation:

- **N = 2048**: 15.768126 ms - 44.5% of end-to-end runtime
- **N = 16384**: 131.660889 ms - 62.9% of end-to-end runtime
- **N = 65536**: 590.895325 ms - 57.1% of end-to-end runtime

K-Means++ Implementation:

- **N = 2048**: 32.421 ms (16.0107 ms in centroid initialization) - 72.6% of end-to-end runtime
- **N = 16384**: 255.797 ms (122.954 ms in centroid initialization) - 76.3% of end-to-end runtime
- **N = 65536**: 1152.181 ms (554.801 ms in centroid initialization) - 74.6% of end-to-end runtime

Thus, we can see from the high percentages that data transfer largely dominated the end-to-end runtime of all the CUDA implementations. It made up the highest fraction of end-to-end runtime for the K-Means implementation, leading to K-Means becoming the slowest solution. Conversely, the Shared memory implementation had the lowest fraction of data transfers as a result of combining the steps of summing cluster dimensions/sizes and updating centroids into one kernel, removing the data transfer to/from host memory in between these steps.



8 References

Here are some helpful websites and resources that I referenced during the project:

- https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/
- https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/
- https://developer.nvidia.com/blog/how-query-device-properties-and-handle-errors-cuda-cc/
- https://docs.nvidia.com/cuda/thrust/index.html