CS 378H Lab 3

Siddharth Sundaram

March 3, 2025

1 Introduction

In this lab, I wrote a C++/CUDA program to implement the k-means clustering algorithm. K-means is a popular unsupervised machine learning algorithm that finds natural clusters among unlabeled data and iteratively converges on an approximately optimal clustering.

First, the algorithm defines k initial centroids that are chosen at random among the data points. Then in every iteration of k-means, each point is assigned to the cluster that corresponds with the closest centroid to the point, and the centroid's position is updated to be the mean of all the points assigned to its cluster. The algorithm converges and terminates when all centroids have remained in the same position as the previous iteration.

While this can be done sequentially with a C++ program, large datasets will take a very long time to converge due to the sheer number of computations per data point (and per centroid). Fortunately, these computations don't have any dependencies on other data points, so they can be performed in parallel to increase efficiency on large datasets. This is where CUDA is utilized: k-means++ centroid initialization[1], cluster assignment, centroid updating, and convergence are all parts of the k-means algorithm that can be parallelized.

This report will detail the performance benefits of 3 parallel implementations of k-means compared to the sequential implementation of k-means: Basic CUDA, Shared Memory, and k-means++ centroid initialization. The performance comparisons are measured in speedup, which is the time ratio of the sequential implementation to the parallel implementation. As a side note, the implementations of k-means allow for disappearing centroids, so if a particular input configuration causes one or more centroids to have zero points in their clusters, the output will show such centroids to be at position 0.0 across all dimensions.

[1] k-means++ is an algorithm that aims to converge in fewer iterations by choosing initial centroids more methodically than at random. After the first centroid is chosen at random, the remaining centroids are chosen iteratively based on a probability distribution made from their distance to the closest centroid. The farther a point is from its closest centroid, the more likely it is to be chosen as the next centroid.

2 Hardware/OS Details

2.1 GPU

Device Name	Quadro RTX 6000
Architecture	Turing
Total Global Memory	22691 MB
Compute Capability	7.5
Multiprocessors	72
Max Threads / Block	1024
Max Threads / Multiprocessor	1024
Max Threads Dim	(1024, 1024, 64)
Max Grid Size	(2147483647, 65535, 65535)
Shared Memory / Block	48 KB
Threads / Warp	32
Max Blocks / Multiprocessor	16
Max Active Warps / Multiprocessor	32

Table 1: GPU Hardware Details

2.2 CPU

Model Name	Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz
Architecture	x86_64
Sockets	2
Cores / Socket	16
Threads / Core	2
L1 Data Cache	1 MiB
L1 Instruction Cache	1 MiB
L2 Cache	32 MiB
L3 Cache	44 MiB

Table 2: CPU Hardware Details

2.3 OS

Version	Ubuntu 20.04.6 LTS

Table 3: OS Version Details

3 Performance Analysis

3.1 Speedup Visualizations

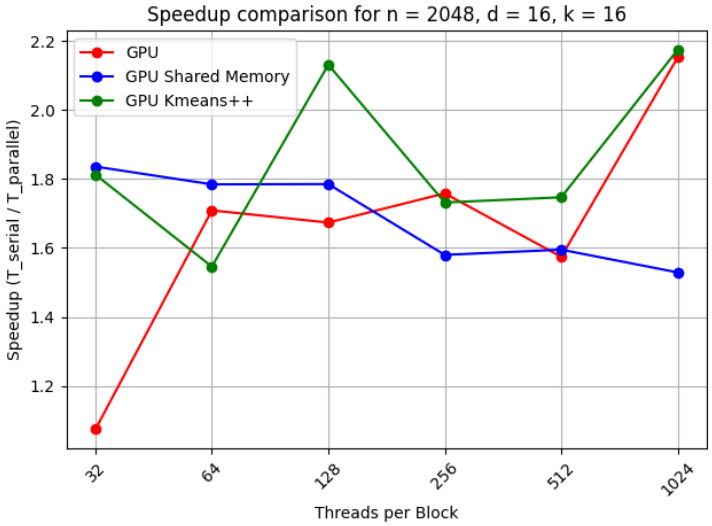

Figure 1: End-to-end speedup graph for small test input

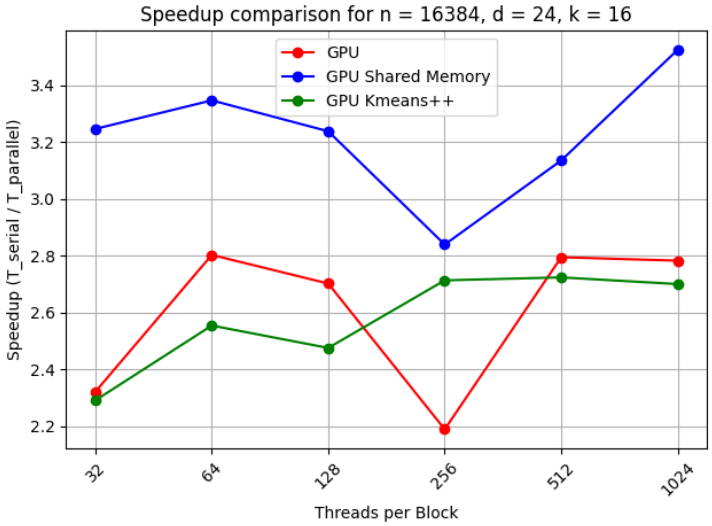

Figure 2: End-to-end speedup graph for medium test input

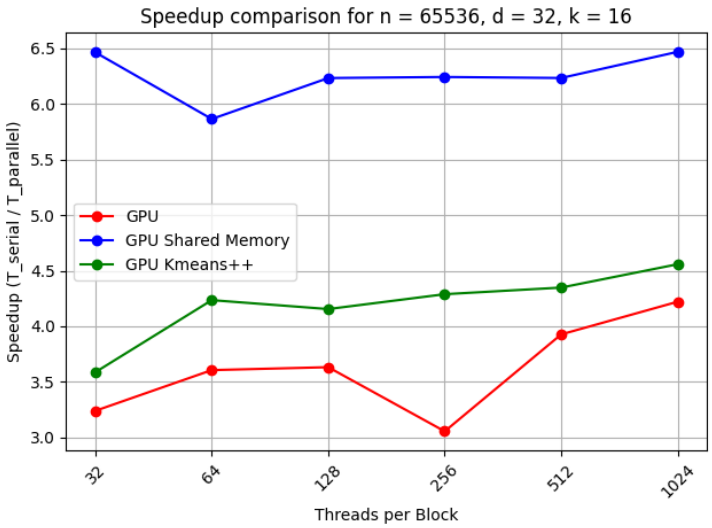

Figure 3: End-to-end speedup graph for large test input

To clear up any potential confusion about the legend in the above figures, "GPU" refers to the basic CUDA implementation, "GPU Shared Memory" refers to the shared memory CUDA implementation, and "GPU Kmeans++" refers to the CUDA k-means++ implementation that also uses the basic CUDA implementation for the k-means algorithm.

3.2 Fastest Implementation

According to the above figures (1-3), the CUDA shared memory implementation is the fastest implementation, and it scales better than the other implementations with larger input sizes. The shared memory implementation uses shared memory in the kernel that assigns data points to clusters and the kernel that computes the new centroid positions. Centroid position and cluster size data are placed in shared memory since these kernels have several reads and writes to centroid data.

During cluster assignment, each thread (1 for each data point) reads the position data of each centroid; if there are k centroids, d dimensions, and n data points, the kernel will have on the order of kdn global memory accesses. With the centroid position data in shared memory, these are now kdn memory accesses to shared memory (data guaranteed to be in the GPU's L1 cache equivalent), making this kernel much faster.

When computing new centroids, each thread (one per data point) adds to the count of its cluster and adds its corresponding data point's position to the new

centroid's position sum. Once all data points' positions in the cluster have been summed, the sum is divided by the cluster size (averaged). This would be done with on the order of dn memory accesses to global memory. Since the shared memory implementation loads centroid data into shared memory, there are now dn memory accesses to shared memory, and correctness is enforced with atomic addition. Once all threads have contributed to their centroid's sum and cluster's size, a reduction is performed to sum each block's shared memory centroid data into global memory. Once in global memory, the centroid position is averaged to yield the final result.

This behavior matches my expectations. The shared memory implementation is designed to be faster than the basic CUDA implementation, as well as the k-means++ implementation. Had the k-means++ implementation used shared memory in the k-means part of the algorithm, I predict it would have been faster upon successfully converging earlier.

3.3 Best-Case Performance Speedup

Based on the hardware details of the GPU used for this lab, the maximum number of threads that can run concurrently is shown below:

$$\frac{\#Threads}{Warp} \cdot \frac{\#MaxActiveWarps}{SM} \cdot \frac{\#SMs}{GPU} = MaxActiveThreads$$

$$32 \cdot 32 \cdot 72 = 73728 \text{ Active Threads}$$

The above equation assumes 1024 threads are being scheduled to each multiprocessor. Another hardware limitation to consider is the max number of blocks per multiprocessor, which is 16 in this case. Since the max number of threads per block is 1024, each block can have anywhere from 64 to 1024 threads (ideally powers of two) and theoretically make full use of each multiprocessor.

When comparing the GPU parallel implementations with the CPU serial implementation that uses 1 core and 1 thread, the best-case theoretical speedup, assuming all GPU threads are doing useful work and there are no bottlenecks with host-device data transfer, etc., is $\frac{73728}{1} = 73728$. This also assumes that the entire algorithm can be parallelized, memory copying is costless, memory accesses have no latency, GPU and CPU clock speeds are the same, there are exactly 73728 points, and syncing threads for the shared memory implementation has no latency. Given this, the calculated theoretical speedup is speculative at best. The actual best-case speedup my program has produced is 129.27, since just the sequential algorithm (excluding input parsing, etc.) took 3772.66 ms and just the CUDA shared memory algorithm took 29.1846 ms on the large test input ($\frac{3772.66}{29.1846} = 129.27$). Compared to the theoretical best-case speedup, it is orders of magnitude lower ($\frac{129.27}{73728} = 0.00175$). The basic CUDA algorithm had a best-case speedup of 8.974 and the k-means++ algorithm had a best-case speedup of 11.487 (both much lower likely due to extremely latent GPU

global memory accesses). The observed best-case performances being much lower than the theoretical is because of costs associated with memory allocation, data transfer, latent memory accesses, synchronization, differing CPU and GPU clock speeds, and inevitable serialization of parts of the algorithm.

3.4 Slowest Implementation

According to the above figures (1-3), the basic CUDA implementation is the slowest implementation. For the small and medium test inputs, the basic CUDA implementation performs on par with the CUDA k-means++ implementation, but it performs worse than both other implementations for the large test input.

This behavior matches my expectations. The shared memory CUDA implementation is supposed to be a direct improvement on the basic CUDA implementation by optimizing memory accesses in kernels, so it makes sense that the basic CUDA implementation performs worse than the shared memory CUDA implementation. The k-means++ implementation is supposed to result in faster convergence by optimizing centroid initialization, so it makes sense that the basic CUDA implementation performs slightly worse than the k-means++ implementation when the latter actually results in earlier convergence. I predict that the basic CUDA implementation would have performed better than the k-means++ implementation if the actual k-means part of it was sequential.

Therefore, it makes sense that the basic CUDA implementation is the slowest of the parallel implementations, which is what the empirical data suggests.

3.5 Data Transfer

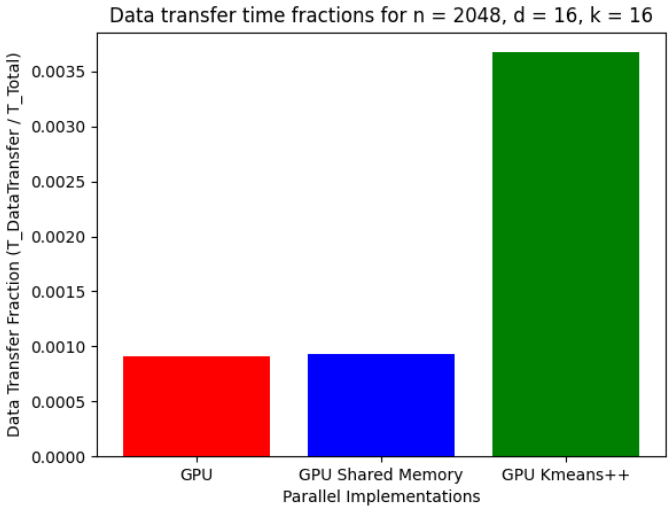

Figure 4: Fraction of data transfer time for small test input

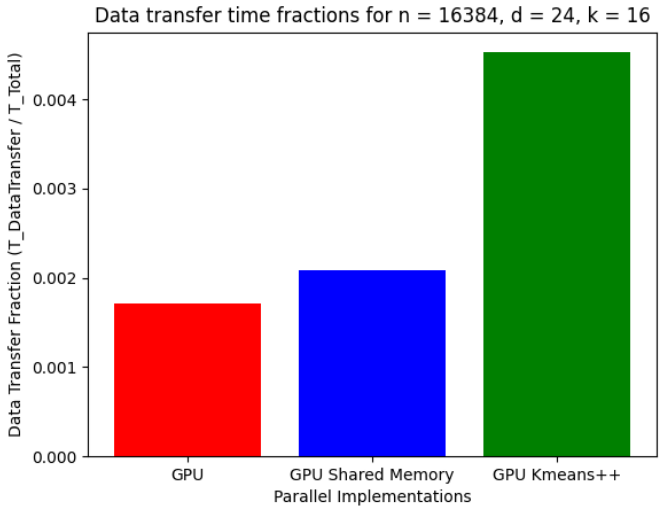

Figure 5: Fraction of data transfer time for medium test input

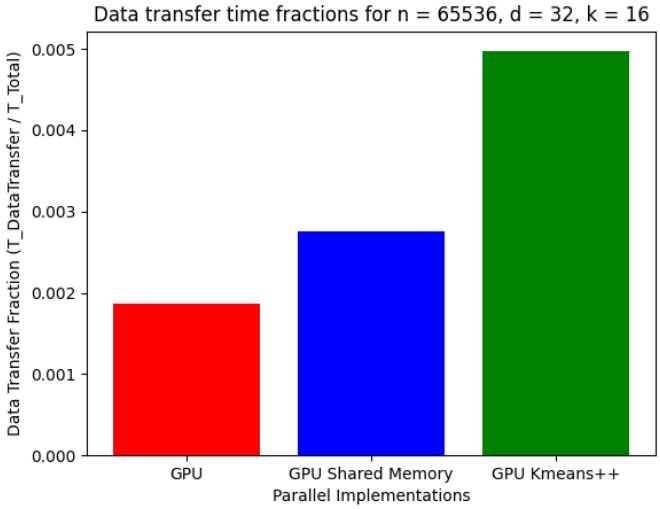

Figure 6: Fraction of data transfer time for large test input

As seen in the above figures (4-6), the fraction of end-to-end time spent in data transfer is quite small, though the fraction does seem to increase with the size of the test input. With all test inputs, the GPU K-means++ implementation has the largest fraction of time spent in data transfer, with the largest fraction being around 0.005 for the large test input. These relatively small fractions show that the program is behaving ideally, since data transfer can be a bottleneck in speedup if it happens very frequently and redundantly. I tried to write the program in such a way that minimizes data transfers between the host and the device.

4 Conclusion

This lab exhibited the quintessential use case of GPU programming: machine learning algorithms. This has been my favorite lab so far, in large part due to actual speedup > 1, but also because learning about and programming in CUDA has been so fun. I spent about 40 hours on this lab.