K-Means Clustering with OpenMP and CUDA Optimization

Final Project report for APMA 2822B

Xinrui Wang and Haiting Huang

Brown University

December 22, 2023

Contents

1. Introduction

K-Means clustering is a popular and efficient unsupervised classification algorithm. The algorithm aims to divide “n” data points into “k” different clusters according to their distances to each other. The algorithm operates iteratively and has three key phases: centroids initialization, cluster assignments, and centroid updates.

Centroids Initialization: “k” data points are chosen at first to serve as the first cluster centers during the algorithm’s beginning phase. There are multiple ways of initializing the centroids, such as random initialization and the more sophisticated K-Means++ initialization. In order to allow more consistent and efficient convergence, we will focus on the K-Means++ initialization, which ensures that the initial centroids are spread out evenly.

Cluster Assignment: During the assignment phase, each data point in the input dataset is assigned to their closest cluster center. There are multiple ways of calculating the distance between the data points and each centroid, such as Euclidean distance and Manhattan distance. In this project, we will focus on the most commonly used Euclidean Distance.

Centroids Update: During the centroids update phase, cluster centers are recalculated using the average of each point assigned to each cluster during the previous process. The result is used as the new centroids for the next iteration.

The algorithm repeats the cluster assignment and centroids update phases until either a predefined number of iterations is achieved or the cluster centers stop changing a lot after an iteration, signaling convergence.

As the algorithm contains multiple independent computation tasks, such as distance calculation, data assignment, and data averaging, it has huge potential for optimization through parallelization. Two prominent tools, OpenMP and CUDA, offer effective parallelization strategies here.

OpenMP for Cluster Assignment and Centroid Update: OpenMP excels in utilizing multi-core CPU architectures by distributing the workload among individual cores. This optimization shines during the update phase, where multiple threads can simultaneously recalculate cluster centroids by taking the average across clustered data points. It also excels during the assignment phase, where data points assignment can be processed in parallel by multiple threads to be assigned to their respective new clusters according to distance calculated.

CUDA for Distance Calculation: CUDA harnesses the immense parallel processing capabilities of GPUs, making it particularly advantageous for distance computations, especially when we have a large number of data points.

In this report, we will implement an efficient K-means clustering algorithm using C++, optimized using CUDA and OpenMP. We will implement a sophisticated centroid initialization algorithm called K-Means++. We will examine a roofline analysis of our algorithm and study the CUDA optimization using the Nvidia NSight profiler. Lastly, we will compare the performance of the algorithm with a serial implementation of K-Means clustering.

2. Method

2.1 Centroids Initialization

There are multiple methods of centroids initialization, with the easiest one being random initialization - each starting centroid is chosen at random from the dataset. This strategy is straightforward yet it has some significant disadvantages. The algorithm's convergence of the final clusters may be impacted by the randomness if the initial centroids are distributed unevenly. It can occasionally lead to the algorithm converging to less-than-ideal solutions, or needing many iterations to obtain good clustering.

K-Means++ is an initialization algorithm that takes a more sophisticated approach when selecting the initial centroids. By choosing centroids in a manner that disperses them more evenly, it lessens the probability of erroneous classification due to the influence of bad random initializations. In general, this approach produces more dependable and consistent outcomes than random initialization. This is the reason why we chose K-Means++ in our implementation.

K-Means++ adopts the following steps to choose the initial centroids. Firstly, it randomly selects the first centroid. The second centroid is selected through a probability distribution, which the probability of being selected is dependent on the distance between this point and the first centroid. The farther away the point is from the first centroid, the greater the probability of being selected as the second centroid. The selection of subsequent points is based on a probability proportional to the squared distance to its nearest current centroid. We repeat this process until we have exactly ‘k’ centroids.

We utilize OpenMP to parallelize the calculation of the distances across data points. This allows more efficient search for the next centroid by distributing the workload across multiple CPU threads. The “#pragma omp parallel for” directive instructs the compiler to distribute the distance computation across multiple threads. Each thread will then calculate the distance for a subset of data points, thereby speeding up the process. To avoid overwriting the minDistance variable, which could happen due to concurrent access by multiple threads, the optimization is maintained inside the inner loop over the variable “j.” Each thread retains its own local minimum distance calculation and ensures that the selection of the next centroid is based on the correct computation of distances from existing centroids.

2.2 Distance Calculation

The distance calculation algorithm is implemented using CUDA, enabling efficient parallel computation on GPUs. The core function, ‘calculateDistancesBatched,’ is defined as a global CUDA kernel. This function computes the distances between data points and a set of centroids.

The algorithm operates in batches to efficiently process large datasets. Each thread in the GPU handles a batch of data points. The batch size is dynamically determined based on the total number of data points

(n) and the number of CUDA streams (numStreams) being used. This approach ensures that the batch size is optimized for both the data size and the parallel processing capacity of the GPU, balancing the workload across multiple streams for efficient processing. The batch processing is controlled by the 'batchIndex,' which is calculated from the block and thread indices (blockIdx.x * blockDim.x + threadIdx.x). The start and end indices for each batch (start and end) determine the subset of data points that a particular thread will process.

Inside the kernel, two nested loops iterate over the rows (data points) and columns (centers) within the batch. The innermost loop calculates the squared Euclidean distance between a data point and a center. This involves iterating over each dimension (dimension), computing the difference between the corresponding components of a data point and a center, squaring this difference, and summing these squares to obtain the total squared distance (sum). Finally, the square root of this sum is computed to get the Euclidean distance, which is then stored in the distance result array.

This algorithm effectively utilizes the parallel processing capabilities of GPUs by distributing the distance computation workload across multiple threads. Each thread computes distances for a subset of data points, enabling large-scale distance calculations to be performed efficiently. The calculated distance is then copied back to the host for cluster assignments and centroids to update.

2.3 Cluster Assignment and Centroids Update

The cluster assignment phase is parallelized using OpenMP, with each thread processing a portion of the data. The closest centroid for each data point is determined by comparing distances computed during the last phase using CUDA. For each data point, the algorithm finds the centroid with the minimum distance and assigns the data point to that centroid. This assignment is stored in 'belongClass,' and the count of data points assigned to each centroid is tracked in counts. Additionally, we store the cumulative sum of the coordinates of the data points assigned to each centroid in 'centroidSums.'

The centroid update phase involves recalculating the position of each centroid based on the new assignments. This is done by averaging the coordinates of all data points assigned to each centroid. For each centroid, the algorithm divides the cumulative sum of the coordinates (stored in 'centroidSums') by the number of points assigned to that centroid (stored in 'counts'). This calculation results in the new position of the centroid, which is stored in 'centerNext.'

Following the update of all centroids, the algorithm checks for convergence. Convergence is determined by comparing the updated centroid positions with their positions in the previous iteration. If the change in position for all centroids is below a certain threshold (0.0001), the algorithm is deemed to have converged. Otherwise, it continues the iteration until it either reaches convergence or reaches a maximum number of iterations (1000). At the end of each iteration, the algorithm resets the centroid sums and counts for the next iteration.

2.4 Roofline Model Analysis

We will divide our algorithm into 3 components to perform roofline model analysis: centroid initialization, distance calculation, and centroid update. We will compute the arithmetic intensity of each component, which can be calculated using the following equation:

$$Arithmetic\ Intensity\ = \frac{Total\ FLOPs}{Total\ Bytes\ Transferred} \quad (1)$$

We will vary the number of data points - N, dataDimension, and the number of centroids - k. We will record the time, the number of iterations it took to converge, and calculate the arithmetic intensity for each combination of N, dataDimension, and k.

Centroid Initialization:

FLOPs: The first centroid is chosen randomly without computation. For subsequent centroids, we calculate the distance from each data point to each already chosen centroid, followed by a probability computation for selection. Assuming the worst case, where each data point is compared with all previously selected centroids, the FLOPs per data point would be up to $2 \cdot dataDimension \cdot (i - 1)$ for i centroids already chosen. There is also one division operation per data point to normalize the distances for probability calculations. Therefore, for each centroid, the worst case FLOPs is $2N \cdot dataDimension \cdot (k - 1) + N.$

Bytes Transferred: The initial random centroid selection requires reading one data point from memory. For each subsequent centroid, all the data points are read again to calculate new distances, and the minimum distances are updated. Therefore, using float32, the number of bytes transferred during the centroid initialization process can be calculated as $4N \cdot dataDimension \cdot k$ for reading data points and $4N \cdot k$ for updating the minimum distance array in memory.

Distance Calculation:

FLOPs: For each data point, the FLOPS for each centroid is calculated as $2N \cdot dataDimension$ operations (subtract and square for each dimension, and a square root per centroid). This is done for all N points and k centroids. We also have a square root operation per centroid. Therefore, for each iteration, the final FLOPs is calculated as $2N \cdot dataDimension \cdot k + k.$

Bytes Transferred: Each point and centroid coordinates are read from memory, and the result is written back. This involves reading $4(N + k) \cdot dataDimension$ and $4N \cdot k$ for writing the distance array back.

Centroid Update:

FLOPs: For each centroid, it involves averaging the points assigned to it, which is dataDimension additions and a division for each of the k centroids. Therefore, the FLOPs per iteration is $(dataDimension + 1) \cdot k.$

Bytes Transferred: During this process, we read the sum of the data points, and the new position of the centroids is written back. Therefore, the total number of bytes transferred per iteration is calculated as $2 \cdot 4k \cdot dataDimension.$

The FLOPs and Bytes Transferred calculated above for distance calculation and centroid update are per-iteration FLOPs and Bytes Transferred. To calculate the final result, we would need to multiply with the number of iterations the algorithm took to converge. We will graph the performance (TFLOPs/s) against the computed arithmetic intensity (TFLOP/Byte) along with the memory and compute bounds.

We will use the Oscar VM for the computation task, with the Intel(R) Xeon(R) Platinum 8268 CPU @ 2.90GHz and the 24GB Quadro RTX 6000 GPU. The memory bound is estimated at 150 GB/s for the CPU and at 672 GB/s for the GPU. To calculate the peak performance ceiling, we use the following equation:

$$Performance = Number\ of\ Cores \cdot Max\ Clock\ Speed \cdot FLOPs\ per\ Cycle \cdot Number\ of\ Threads\ per\ Cores$$

This leads to an estimated peak performance of 160 GFlops for the CPU. For the GPU, we estimate a 16.3 TFlops performance ceiling.

2.5 Comparison with Serial Implementation

To measure the effect of the optimization, we implement a serial implementation of the K-Means algorithm. Specifically, in our optimized implementation, we use OpenMP for centroid initialization, cluster assignment, and centroid updates. We also use CUDA for distance calculation before the cluster assignment. In our serial implementation, all the above processes are implemented serially using C++.

We structure our experiment to evaluate the performance difference between an optimized and an unoptimized algorithm across varying dataset sizes and numbers of clusters. We conducted a series of tests using datasets with a range of M (number of data points), N (feature dimensionality), and k (number of clusters) values. Each dataset was processed by both the optimized (indicated by '1') and unoptimized (indicated by '0') versions of the algorithm, and the execution time in milliseconds was recorded. The datasets were created randomly using Python.

For the configuration of the dataset, we range M from 100 to 1,000,000 and N from 10 to 10,000. The number of clusters (k) was fixed at two distinct values: 2 and 10. However, it is important to note that we faced constraints due to the computational limits of the Oscar system. Particularly when dealing with exceptionally large combinations of M and N—such as 100,000 by 10,000 or 1,000,000 by 10,000—the execution time exceeded Oscar's permissible time thresholds. Therefore, we made the decision to exclude these overly extensive data configurations from our test suite.

3. Result

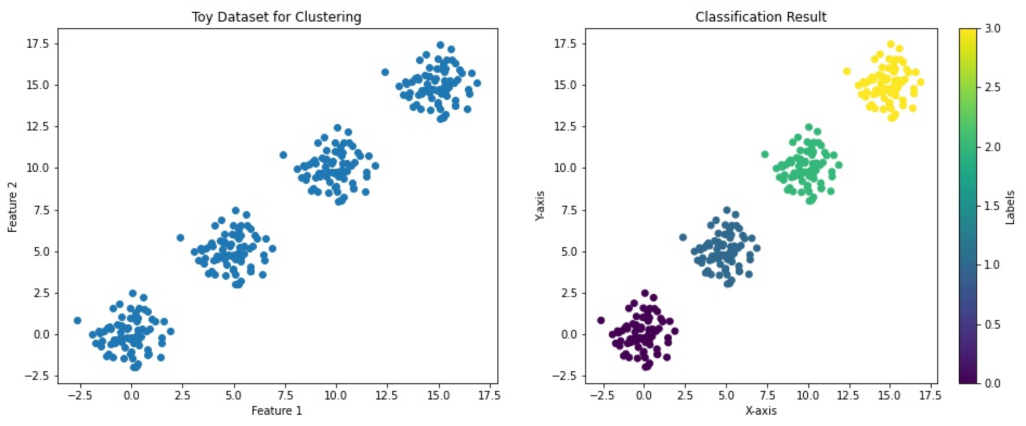

Figure 1. Clustering Results from Toy Data

This left graph shows a toy dataset that we will use as the input to generate four clusters. It is easy for humans to identify that the data can be divided into 4 groups. To check if the algorithm is correct or not, we need to use our algorithm to generate the clustering assignment.

The right graph shows the output of our algorithm when the toy input data is used. The right sidebar indicates different labels assigned to each cluster. As we have set k equal to 4, we have 4 labels. Different colors indicate different cluster assignments computed by our algorithm.

Centroids	x	y
1	2.38	5.82
2	14.84	15.40
3	10.81	8.77
4	0.82	-1.22

Table 1. Centroid Initialization for 2-D Toy Dataset

Table 1 shows the centroid initialization for the 2-D toy dataset in Figure 1. using the K-Means++ algorithm.

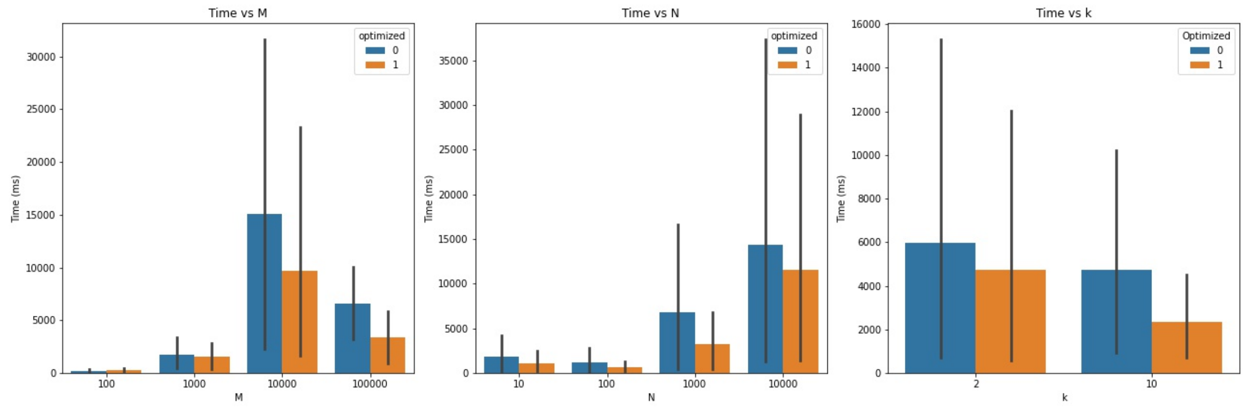

Figure 3. TIme Difference Comparison between Optimized vs. Unoptimized Algorithm

The graph shows three bar charts comparing the execution time of the optimized algorithm and the unoptimized, serial algorithm in milliseconds (ms) against three different parameters: M, N, and k. Each chart has two sets of bars representing a non-optimized version (indicated as "0") and an optimized version (indicated as "1") of the K-Means algorithm.

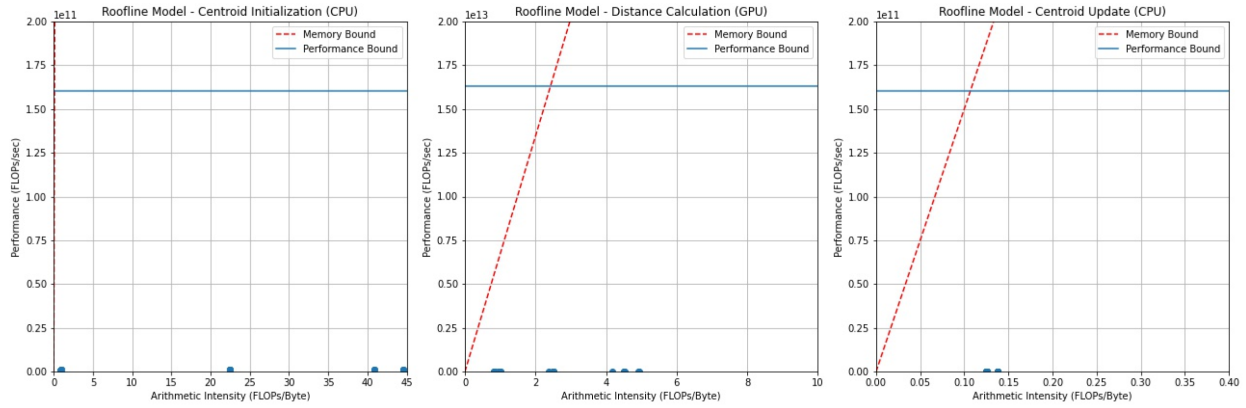

Figure 4. Roofline Analysis

Figure 4 shows the roofline analysis for three different components of the algorithm - Centroid Initialization, Distance Computation, and Centroid Update. Each graph shows a compute and memory bound of the CPU/GPU used as well as the scatter plot of the performance measurement when different inputs of N, dataDimension, and k are used.

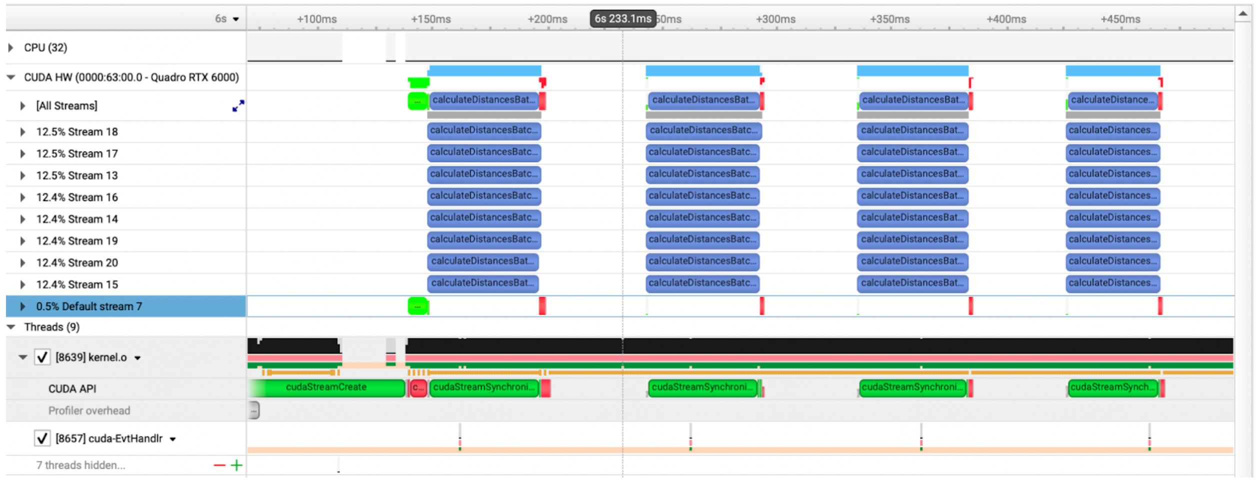

Figure 5. NSight Profiler Analysis

The graph showcases the timeline of various operations executed during the running of our algorithm with inputs configured with M = 1,000,000, N = 10, and k = 2 parameters. We use 8 streams in our computation. The blue blocks are labeled “calculateDistancesBatched,” which corresponds to the execution of the “calculateDistancesBatched” function. These blue blocks indicate the periods during which the GPU is actively computing the distances between data points and cluster centroids, utilizing CUDA for parallel processing.

4. Discussion

From Figure 1 and Figure 3, we see that our algorithm works correctly using a toy dataset, as it produces four distinct clusters as expected. The algorithm converges rather quickly after 3 iterations; this is probably due to the efficient and effective implementation of the K-Means++ centroid initialization. Upon examining the centroids initialization, we see that the initial centroids are well distanced from each other, making the proceeding process more efficient.

In Figure 3, we have three bar charts showing the execution time between the optimized and the unoptimized algorithms. The first bar chart, "Time vs. M," shows a dramatic increase in execution time as M increases, particularly for the non-optimized version. The average time for M equals 100,000 is shorter than when M equals 10,000 because we did not compute for higher levels of data dimensions. This is due to the wall time limit of the Oscar VM. The second bar chart, "Time vs. N," presents a less pronounced but still noticeable trend. The third bar chart, "Time vs. k," illustrates the effect of changing the parameter k on the execution time. For both k = 2 and k = 10, we see that the optimized algorithm performs better than the serial version.

Analyzing the results, we see that the optimizations have a large impact depending on the scale and nature of the parameters M, N, and k. The optimization provides a substantial performance benefit at higher values of M, indicating that the algorithm scales better with size when optimized. This is expected as we use batch parallel processing with CUDA to calculate the distance for multiple data points at a time. For

N, the optimization appears to reduce the variability in execution time, leading to more predictable performance. With respect to k, the benefits of optimization become more pronounced as the value of k increases. These observations suggest that the optimized algorithm has improved computational efficiency, particularly for larger datasets or more complex calculations, as indicated by higher M or k values.

In Figure 4, we see that for all three components, the data points lie below both the compute bounds and the memory bounds, as expected. We also notice that the performance does not increase with the arithmetic intensity. This may be an indication that our algorithm is limited by memory. As we also observe that the data points lie below the memory bandwidth, we hypothesize that the issue lies within our memory transfer and fetch mechanisms. I believe that we could adopt more efficient memory access patterns, such as using shared memory.

Figure 5 provides a snapshot of the execution timeline of our accelerated K-Means algorithm. The distribution and frequency of the blue blocks, denoting the "calculateDistancesBatched" kernel, indicate consistent and high utilization of our GPU.

5. Conclusion

In conclusion, using CUDA and OpenMP to implement K-Means clustering can effectively handle large quantities of data when the number of clusters is large. The optimization of centroid initialization and assignment by OpenMP and parallel distance computations using CUDA significantly improve computational performance and scalability in higher dimension input data with many clusters. By decreasing the possibility of suboptimal clustering and guaranteeing better-quality outcomes, the incorporation of K-Means++, an effective method known for its deliberate selection of starting centroid, further enhances the approach. K-Means++, OpenMP, and CUDA work together to improve the process and increase accuracy, speed, and efficiency. In future works, we should look into opportunities to optimize memory access patterns to further improve the performance of our K-Means algorithm.