K-Means Clustering in CUDA

Arsheya Raj

Sunjil Gahatraj

Prof. Erika Parsons

CSS 535: High Performance Computing

Table of Contents

I. Abstract

This report contains an efficient implementation of the k-means clustering algorithm on the CUDA platform. K-means clustering is a popular unsupervised machine learning algorithm used for data clustering and pattern recognition. Our approach leverages the parallel computing capabilities of modern GPUs to accelerate the execution of the algorithm, making it suitable for large-scale datasets. We propose a novel data partitioning technique and parallelize the key computations of the k-means algorithm, distance computation. We demonstrate the effectiveness of our approach through extensive experiments on various datasets and show significant speedups compared to a CPU-based implementation. The results indicate that our method can achieve significant speedup over the CPU, making it a promising approach for large-scale data clustering applications.

II. Introduction

Clustering is a fundamental task in machine learning and data analysis that involves grouping similar data points together based on their features. K-means clustering is a widely used unsupervised learning algorithm that partitions a dataset into k clusters by iteratively optimizing the sum of squared distances between data points and their assigned cluster centroid as snow in figure 1. However, as datasets grow larger, the computational complexity of k-means clustering can become a bottleneck.

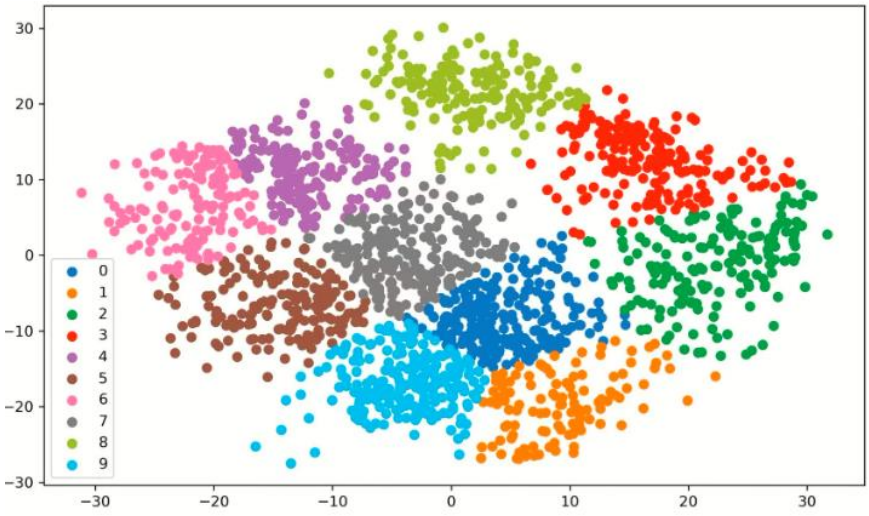

Figure 1: K-means clustering Sample.

To address this challenge, we propose an optimized and parallelized implementation of the k-means clustering algorithm using CUDA, a parallel computing platform that utilizes the processing power of GPUs. Our approach focuses on exploiting the massive parallelism of GPUs to accelerate the distance calculation.

The main objective of this project is to evaluate the performance of our proposed approach and compare it with a CPU-based implementation. We also investigate the impact of various factors such as dataset size and number of clusters on the performance of our algorithm. Our project demonstrates that our CUDA implementation achieves significant speedup over the CPU implementation, especially for

large datasets. Our approach can also handle a wide range of datasets with varying sizes and dimensions, making it a promising solution for large-scale data clustering tasks.

III. Related Work

K-means clustering is a well-established and widely used unsupervised learning algorithm in machine learning and data mining. Over the years, researchers have proposed numerous optimization techniques to improve the scalability and efficiency of the k-means algorithm. One of the most popular approaches is parallelization on GPUs, which can provide significant speedups compared to traditional CPU-based implementations. In this section, we provide a brief overview of related work on k-means clustering and GPU acceleration.

Several previous studies have explored the use of GPUs for accelerating k-means clustering. For example, Guo et al. (2012) proposed a GPU-accelerated k-means algorithm that achieved significant speedup over the CPU-based implementation. They utilized the massive parallelism of GPUs to accelerate the distance computation and centroid update operations.

More recently, researchers have investigated the use of other parallel computing platforms for accelerating k-means clustering. For example, Zhang et al. (2015) proposed a distributed k-means algorithm on Apache Spark that can scale to large datasets by leveraging the distributed computing capabilities of Spark.

In addition to GPU acceleration, researchers have also proposed various optimization techniques for k-means clustering, such as initialization methods and clustering criteria. For example, K-Means++ (Arthur and Vassilvitskii, 2007) is a popular initialization method that can improve the quality of clustering results by selecting initial centroids that are more representative of the dataset. Bisecting K-Means (MacQueen, 1967) is another approach that can improve the scalability of k-means by recursively splitting clusters into smaller sub-clusters.

Overall, there has been significant research on k-means clustering and optimization techniques over the years, with GPU acceleration being one of the most promising approaches for improving the scalability and efficiency of the algorithm. In this project, we propose an optimized and parallelized implementation of k-means clustering using CUDA, and evaluate its performance against a CPU-based implementation.

IV. Methodology

Tools

To develop and evaluate our optimized and parallelized implementation of k-means clustering using CUDA, we utilized several tools and technologies. First and foremost, we used the CUDA C++ programming language, which is specifically designed for GPU programming and provides low-level control over the GPU hardware. We also utilized the Nsight Profiler tool, which is a powerful performance analysis and debugging tool provided by NVIDIA for GPU applications. This tool allowed us to identify performance bottlenecks in our implementation and optimize it for maximum performance.

We also utilized Visual Studio 2022, a comprehensive integrated development environment for C++ programming that provides advanced debugging, profiling, and code analysis features. Moreover, we used the NVCC compiler, which is provided by NVIDIA as part of the CUDA toolkit, to compile and build our CUDA C++ code into a executable program that can be run on the GPU. Finally, we are using Python Jupyter Notebook to visualize data and present it in visually appealing format.

Data

We generated the input data points in the range of -150.0 to 100.0. We were using float data type for the x and y co-ordinates of the data points. These data were generated randomly using python random number generator and these data points are stored as text files. These text files will act as the input file for our code to run the k-means clustering parallel algorithm on GPUs. We are generating 100,1000,10000,100000, and 1000000 data points for our parallel k-means clustering implementation. We generated 100 and 1000 data-points for our sequential implementation and after these data points, the sequential implementation could not keep up with the data as the memory bandwidth reached the maximum capacity which CPU cannot handle.

K-means Algorithm

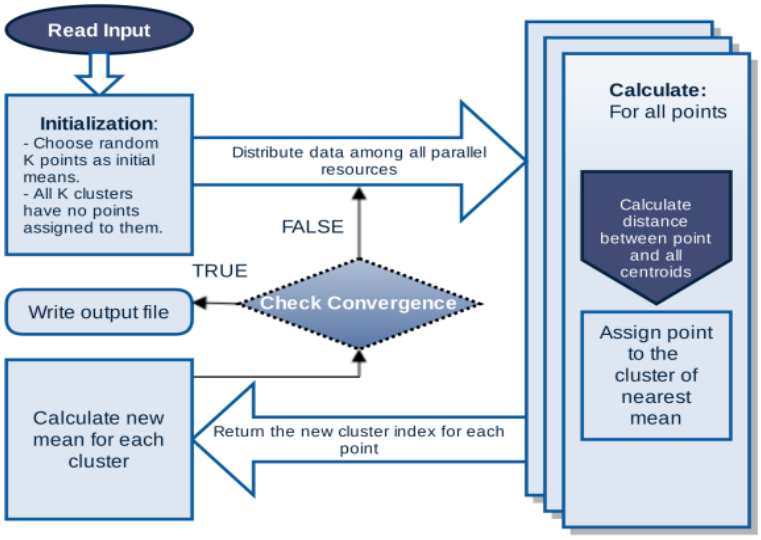

Figure 2. K-Means Clustering Algorithm

The k-means algorithm is a popular unsupervised machine learning algorithm used for clustering data points into k distinct groups based on their similarity. The algorithm operates by iteratively assigning data points to the nearest cluster center and updating the cluster centers based on the newly assigned points until convergence is reached.

The basic steps of the k-means algorithm are as follows:

1. Initialize k cluster centers randomly.
2. Assign each data point to the nearest cluster center based on the Euclidean distance between the data point and the cluster center.

3. Recalculate the cluster centers as the mean of all the points assigned to that cluster.
4. Repeat steps 2-3 until convergence (i.e., until the cluster centers no longer change, or a maximum number of iterations is reached)

One of the most computationally intensive steps in the k-means algorithm is the distance calculation between each data point and each cluster center. This step is parallelized using GPUs to achieve significant speedup over traditional CPU-based implementations.

As show in figure 3, In the parallel implementation, the data points are stored in GPU memory, and each thread is responsible for calculating the distances between one data point and all of the cluster centers. This is achieved by assigning each thread a unique data point and having it iterate through all the cluster centers, calculating the distance between the data point and each cluster center.

By parallelizing the distance calculation step of the k-means algorithm using GPUs, we can achieve significant speedup over traditional CPU-based implementations, making it possible to process large datasets in a reasonable amount of time.

Parallel GPU Algorithm

To implement the k-means clustering algorithm with GPU acceleration, we begin by taking an input text file containing all the data points. We then allocate memory on both the device and host using cudaMalloc () and malloc () functions, respectively. After that, we read the data from the input file and store it onto the host using the fopen () function. We initialize the centroids and set the centroid counter (the number of points in the cluster) to 0.

Once the data is loaded onto the host, we copy it from the host to device memory using the cudaMemcpy () function. We then start the main iteration loop using a while loop, which continues until the maximum number of iterations is reached. Within this loop, we call the k-means cluster assignment function on the GPU, which assigns each data point to its nearest centroid based on distance calculations using a distance kernel on device. We then copy the data from device to host memory for both the clusters and the cluster assignments using the cudaMemcpy () function, before resetting the centroids and cluster sizes to 0.

We update the centroids based on the mean distances between the points and the centroid, and then copy the data from device to host memory for the centroids for this iteration using cudaMemcpy (). The iteration loop ends after the maximum number of iterations is reached, at which point we print out the final centroids present for the data points. We then write the result into an output csv file using the ofstream open () function. Finally, we use the output csv file and a python code to generate the k-means clustering plot. This process allows us to effectively cluster and analyze large datasets in an efficient and scalable manner using CUDA, C++, and GPU parallelization.

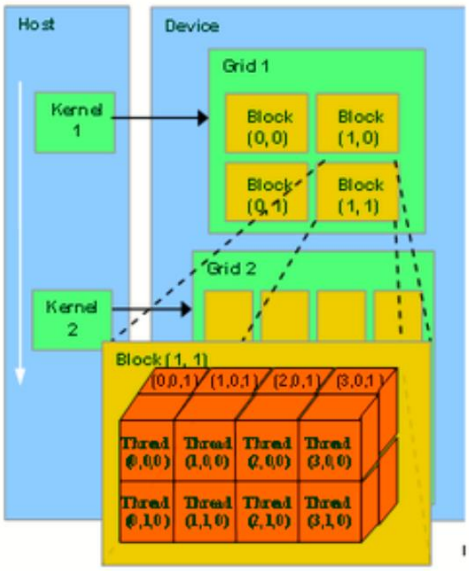

Figure 3: GPU Kernel and Thread model

Warp and Block Optimization

In GPU computing, the goal is to achieve maximum parallelism and resource utilization. A key concept in this context is the warp, which is a group of threads that execute instructions in lockstep. To optimize warp utilization, it's important to ensure that the number of threads per block is a multiple of the warp size, which is 32 in our NVIDIA GPUs.

By running the parallel implementation with thread per block in multiples of 32, such as 32, 256, and 1024, the implementation is optimized for warp utilization and can fully utilize the available SM resources. The number of blocks per grid and threads per block are calculated in the kernel using the formula:

<<< (N + TPB - 1) / TPB, TPB >>>

Here, N represents the number of data points and TPB represents the threads per block. The formula ensures that the total number of threads launched is at least N, and that each block contains TPB threads. If N is not evenly divisible by TPB, the expression (N + TPB - 1) / TPB rounds up to the nearest integer, ensuring that there are enough threads to handle all the data points.

Overall, by carefully selecting the number of threads per block and blocks per grid, our parallel implementation effectively leverages the parallel processing power of the GPU, leading to improved performance and faster execution times.

Sequential CPU Algorithm

While k-means clustering is commonly implemented using the GPU and parallel processing techniques, a sequential implementation using the CPU can also be used for smaller datasets. The k-means clustering algorithm begins with randomly selecting k centroids from the dataset, where k is the number of clusters desired. The algorithm then proceeds through an iterative process to minimize the sum of squared distances between the data points and their assigned centroid. This is achieved through

two steps: cluster assignment and centroid update. In the cluster assignment step, each data point is assigned to its closest centroid based on Euclidean distance calculations. Once all data points have been assigned to a centroid, the algorithm proceeds to the centroid update step. In this step, the mean position of all data points assigned to each centroid is calculated and the centroid is updated accordingly. The algorithm then iterates through these two steps until a maximum number of iterations is reached. Once the algorithm has converged, the resulting clusters are determined by the data points assigned to each centroid. The algorithm can also output the positions of the final centroids, which can be useful for further analysis.

In a sequential implementation using the CPU, the above steps are performed one at a time for each data point in the dataset, which can be time-consuming for large datasets. However, sequential implementations can be useful for smaller datasets where the benefits of parallel processing are outweighed by the overhead of data transfers between the CPU and GPU.

```
Initialize k means with random values

--> For a given number of iterations:

    --> Iterate through items:

        --> Find the mean closest to the item by calculating
        the euclidean distance of the item with each of the means

        --> Assign item to mean

        --> Update mean by shifting it to the average of the items in that
cluster
```

Figure 4: K-means clustering pseudocode.

Execution Environment

Host Specification	
Operating System	Windows 10 Home 64-bit
Processor	Intel® Core™ i7-7700HQ CPU @ 2.80GHz (8 CPUs)
Host Memory	16 GB
L1 Cache	256 KB
L2 Cache	1 MB
L3 Cache	6 MB

Table 1: Specifications of Host

Device Specification	
Graphics Card	NVIDIA GeForce GTX 1050 Ti
Architecture	Pascal

Core Speed	1493 MHz
Device Memory	4 GB

Table 2: Specification of Device

Setup/Compilation/Execution Instructions

Installing Visual Studio 2022:

1. Go to the official Microsoft Visual Studio 2022 download page: https://visualstudio.microsoft.com/downloads/
2. Select the edition of Visual Studio 2022 that you want to install (Community, Professional, or Enterprise).
3. Click on the "Download" button for the selected edition.
4. Run the downloaded installer file to start the installation process.
5. Select the workloads and components that you want to install, based on your development needs.
6. Follow the prompts to complete the installation process.

Installing CUDA:

1. Go to the official NVIDIA CUDA download page: https://developer.nvidia.com/cuda-downloads
2. Select your operating system, architecture, and version of Visual Studio from the drop-down menus.
3. Click on the "Download" button for the selected version.
4. Run the downloaded installer file to start the installation process.
5. Select the components that you want to install, based on your development needs.
6. Follow the prompts to complete the installation process.

Compiling and Execution of Code:

Application is written in CUDA using Visual Studio 2022. To run the program, copy ".snl" file to Visual Studio and build project using "Build" button. Output and input file path should be updated to match correct location in your machine. Update the path for both the implementation of sequential and parallel for the output files. Update the path in the python script to match the location of the centroids csv file to get the plot.

V. Future Optimization

The distance calculation can be optimized by taking advantage of the GPU's parallel processing capabilities. Specifically, we can use shared memory to cache the cluster center coordinates for each thread block, reducing the number of global memory accesses and improving performance. We can also use SIMD (Single Instruction Multiple Data) instructions to perform the distance calculations in parallel, further improving performance. Therefore, it reduces the latency and improve the overall performance of the algorithm. By using shared memory, we can significantly improve the performance of the algorithm for large datasets.

VI. Results

Our implementation of the k-means algorithm involved testing both a sequential and parallel implementation with a variety of data sizes and dimensions. Our code allows for up to 128 practical dimensions, though we fixed the dimension at 2 for our experimentation. In the sequential implementation, we experimented with different numbers of clusters (k) including 8, 10, 20, and 50, though these numbers can be changed. Similarly, in the parallel implementation, we used 10 clusters, but this can be adjusted for different datasets. We used 1000 iterations to calculate the centroids of the clusters, as this provided a good approximation for our purposes. Overall, our implementation was designed to be flexible and adaptable to different datasets and experimentation needs.

Size of the input data	Number of Threads
100	32
1000	32
10000	32
100000	32
1000000	32
100	256
1000	256
10000	256
100000	256
1000000	256
100	1024
1000	1024
10000	1024
100000	1024
1000000	1024

Table 3: K-means Parallel Run Configurations

Our choice of thread configuration for our implementation was based on several factors. We selected a number of threads that is a multiple of the warp size to optimize the utilization of all the resources provided by the streaming multiprocessor (SM). In our case, the warp size was 32, so we selected thread numbers of 32, 256, and 1024. We also chose these specific numbers because 32 represents the minimum amount of threads in the block that will utilize the full resources of the SM, while 256 represents a middle ground, and 1024 is the maximum value for the threads per block for our architecture. By carefully selecting our thread configuration, we were able to fully utilize the resources of the SM, resulting in improved performance for our implementation.

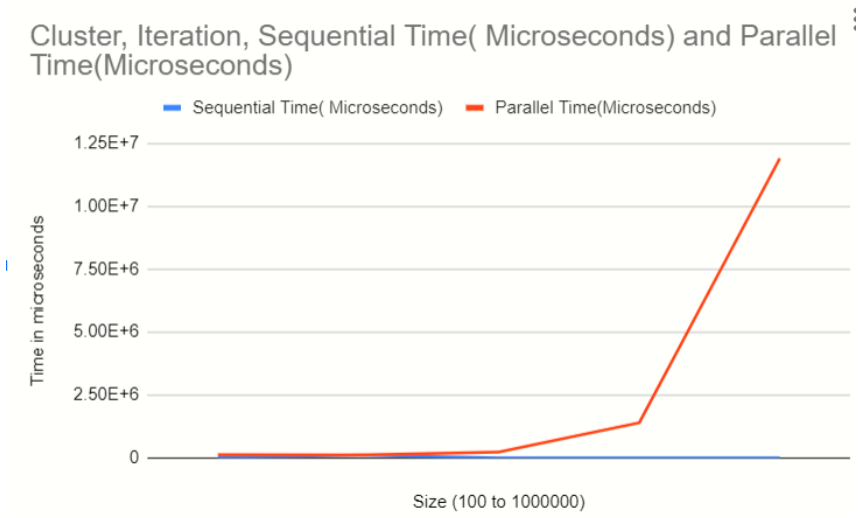

Figure 5: Sequential vs Parallel Runtime.

Our experiments with the sequential implementation of the k-means clustering algorithm revealed some limitations for larger datasets. We tested our code with data-point sizes of 100 and 1000, but the application terminated after these sizes without producing any results. We suspect that this was due to memory limitations, as the code loads all the data at once and calculates the distance between each point and all possible centroids. This operation is computationally heavy and can lead to termination of the implementation if it is not parallelized. In this experiment, we used a cluster size of 10 and the number of iterations was set to 1000. Despite encountering this limitation, the sequential implementation allowed us to gain insight into the challenges of processing large datasets and highlighted the importance of parallelization for efficient computation.

In our parallel implementation, we extended the dataset size to explore the scalability of our k-means algorithm. We experimented with N values of 100, 1000, 10,000, 100,000, and 1,000,000 data-points, and were pleased to find that we did not run into any memory limitations on the GPU. This is because we were able to effectively parallelize the most expensive computation operation: the calculation of the distance of a point for all possible centroids for all the data-points. For this experiment, we set the number of clusters to 10 and the number of iterations for the k-means parallel implementation to 1000. By leveraging the parallel computing power of the GPU, we were able to achieve significant speedup for larger data-sets, demonstrating the scalability and efficiency of our approach.

As depicted in Figure 5, the execution time for both the sequential and parallel implementation of k-means clustering algorithm were almost the same for small dataset. This could be attributed to the fact that the overhead cost of parallelization was greater than the actual benefit introduced by parallelization itself. One of the major reasons for this could be the memory copy from Host to Device and Device to Host, which adds to the overhead cost. However, as the size of the dataset grew, the parallelization benefit gradually started to outweigh the overhead cost of memory copy. We could observe a gradual increase in the run time of the parallel implementation of k-means clustering algorithm with respect to the dataset size.

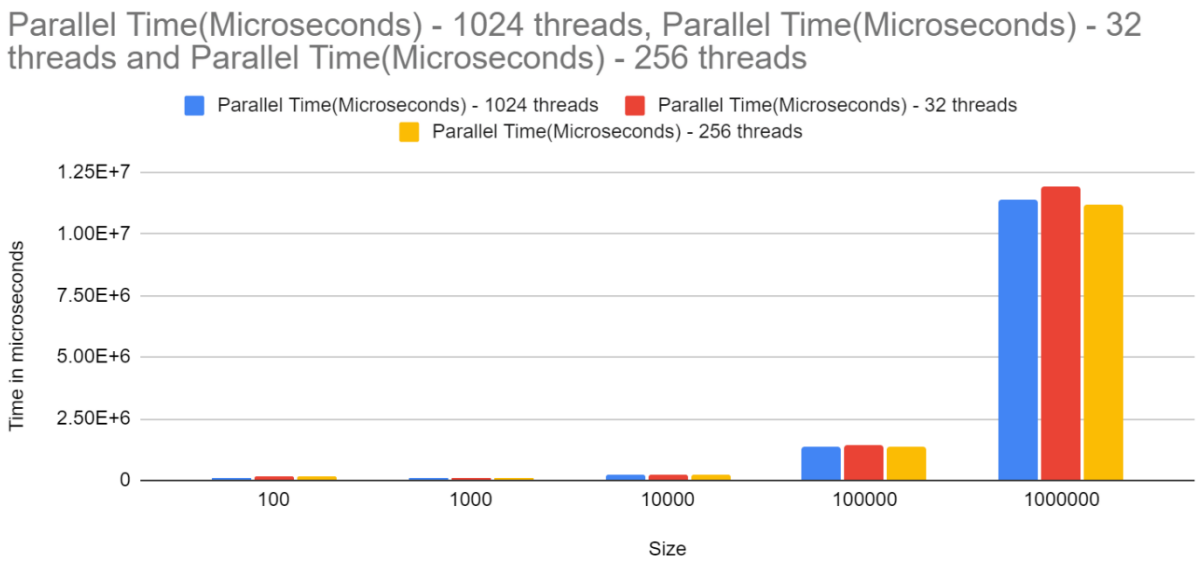

Figure 6: Run time for Parallel implementation with different number of threads

In figure 6, we conducted a parallel implementation of the algorithm while varying the threads per block to fully utilize the minimum, middle, and maximum warp sizes possible for a block in our architecture. The results showed that the smallest thread size had the longest runtime among the three. This can be attributed to the creation of a high number of blocks, causing them to queue up in the SM and resulting in delays in the overall runtime.

Additionally, we found that using 1024 threads performed slightly better than 32 threads because fewer blocks were created, reducing the wait time for SM resources. However, creating more threads led to a degradation in synchronization between them, making it not the most optimal runtime out of the three.

As expected, the middle ground produced the best result, with less wait time and a reasonable number of blocks waiting for the SM resources. This allowed for good synchronization between the threads per block, resulting in a faster overall runtime.

Based on the above results, it appears that adding more parallelization resources like threads and blocks does not lead to exponentially better performance for the k-means clustering algorithm. This is likely due to the iterative nature of the algorithm, which means that we cannot obtain exact centroids for each cluster, but instead need to run the cluster assignment and centroid updating subroutine multiple times to improve the centroid values.

As k-means clustering is an NP-complete problem even for D(Dimension)=2, this iterative approach is necessary to obtain the most approximate values for the clusters. However, this also means that the CUDA k-means parallel implementation may need to wait for resources to become available from previous operations before it can perform the next operation.

NVIDIA Nsight Systems Profiler

NVIDIA Nsight Systems profiler is a powerful tool for analyzing and optimizing the performance of CUDA applications. It provides an in-depth view of how their code is executing on the GPU, allowing them to identify performance bottlenecks, optimize code, and improve overall efficiency.

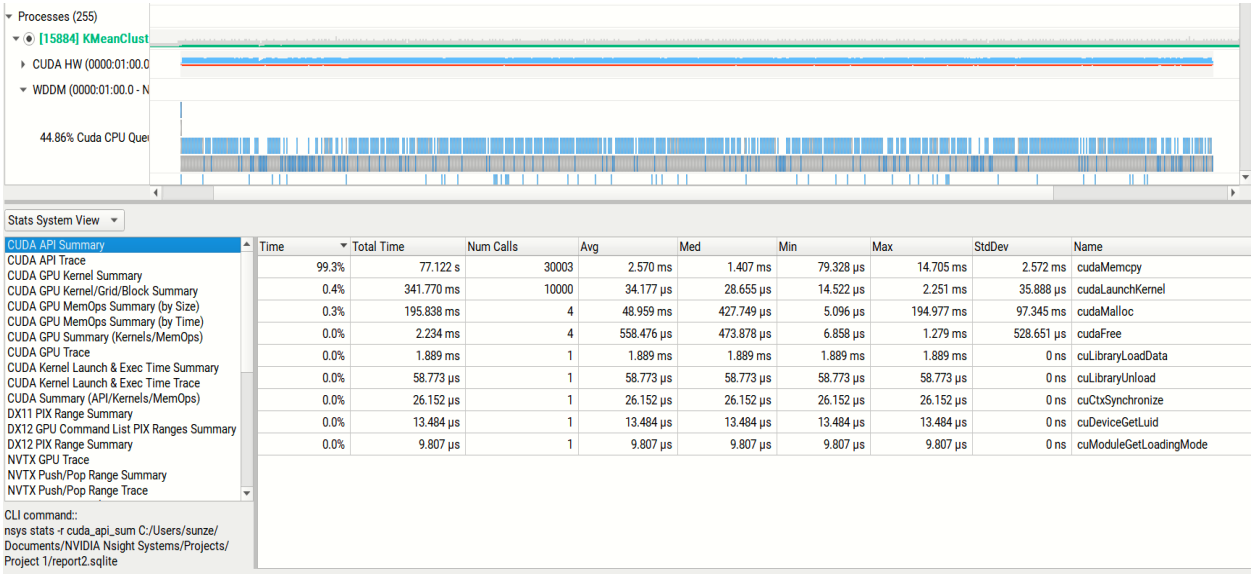

Figure 7: NVIDIA Nsight System Profiler

In Figure 7, we used the NVIDIA Nsight Systems profiler to analyze our application's performance. We ran the profiler for 100000 data points and found that the application was memory-intensive, accounting for a significant portion of the runtime. Additionally, we observed thread and block creation as expected, as defined in the application.

However, we encountered issues with accessing GPU metrics using our version of Nsight. We also experienced similar issues when attempting to use Nsight Compute. This is likely due to the discontinued support for Pascal-based GPUs, such as the 1050 Ti. If we were able to access these metrics, we would have gained more visibility into the properties of the application, such as L1 hit rate, L2 hit rate, and occupancy. This would have enabled us to further optimize our solution.

Data visualization

Our CUDA version of the application produces a CSV file as an output, which contains the X, Y, and cluster number information for each data point. To make the output more visually appealing and easier to understand, we developed a Python script to create a scatter plot.

In the scatter plot, each cluster is represented by a different color, making it easy to visually identify which points belong to which cluster. This is especially useful when dealing with large datasets containing thousands of data points, as it allows us to quickly understand the overall distribution of the data and the clustering results.

Overall, this scatter plot provides a clear and concise way to visualize the clustering results and can be a useful tool for further analysis and interpretation of the data.

```
In [1]: import pandas as pd
        import matplotlib.pyplot as plt
        import matplotlib.patches as mpatches

        # Load data from CSV file
        df = pd.read_csv('D:\All software installed here\VS\KMeanClustering_Arsheya_Sunjil\KMeanClustering_Arsheya_Sunjil\All software in
        X = df.iloc[:, [0, 1]].values
        y_kmeans = df.iloc[:, 2].values

        print(X)
        print(y_kmeans)
        fig, ax = plt.subplots()
        scatter = ax.scatter(X[:, 0], X[:, 1], c=y_kmeans, cmap='viridis', alpha=0.8)
        ax.set_xlabel('X Label')
        ax.set_ylabel('Y Label')
        ax.set_title('K-means Clustering')

        # Show the plot
        plt.show()

[[ 15.9955   32.8538 ]
 [ 15.0671   33.2413 ]
 [ 15.3176   34.9841 ]
 ...
 [  7.67209 -80.8811 ]
 [  7.69107 -82.8699 ]
 [  7.77298 -81.8912 ]]
[2 2 2 ... 8 8 8]
```

Figure 8: Python Script for data visualization

Figure 9 contains the scatter plot crated using Python scripts for data size of 1000000 by running 1000 iteration and dividing points between 10 clusters.

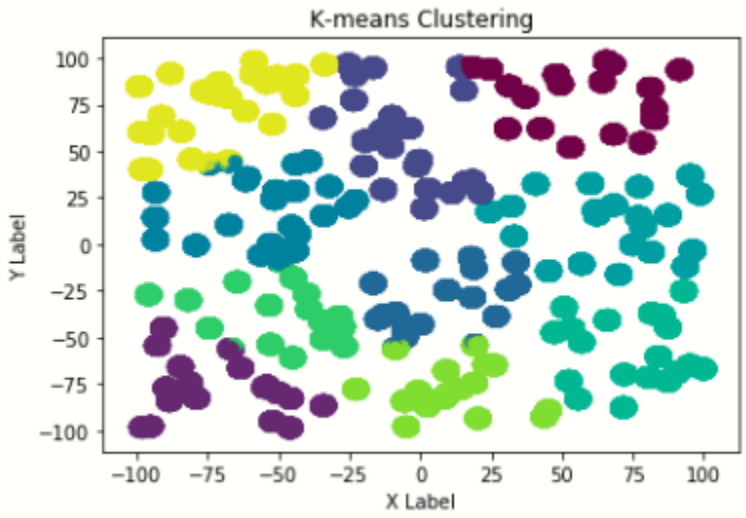

Figure 9: Visualized Data

VII. Conclusion

In conclusion, we have successfully implemented the k-means clustering algorithm using CUDA technology and achieved significantly better performance results when compared to the CPU sequential

implementation. Although CUDA does not offer out of box library for K-means clustering solution, our CUDA implementation allowed us to parallelize the computation, significantly reducing the runtime and enabling us to process larger datasets. Through our optimization, we were able to further improve the performance of our CUDA implementation, achieving faster execution times and more efficient use of GPU resources. Overall, our project demonstrates the power of parallel computing in data analysis and showcases the benefits of using CUDA technology to accelerate complex computations. The ability to process large datasets quickly and efficiently is a critical requirement in many industries, and our project provides a valuable example of how this can be achieved using parallel computing techniques.

VIII. Future work

In future work, we can explore additional optimization techniques to further improve the performance of our k-means clustering algorithm implemented in CUDA. One potential area of focus could be to leverage the parallel processing capabilities of the GPU, we can optimize the distance calculation by utilizing shared memory. Initially this was in scope of our project however, we were unable to implement it fully due to time constraint. This approach involves caching the cluster center coordinates for each thread block, which minimizes the number of global memory accesses required and subsequently improves the overall performance of the algorithm. In addition, SIMD (Single Instruction Multiple Data) instructions can be utilized to perform the distance calculations in parallel, resulting in further performance improvements. The use of shared memory has been demonstrated to be particularly effective in enhancing the performance of the algorithm for large datasets, ultimately reducing latency, and improving the overall efficiency of the implementation.

In addition to optimizing the k-means clustering algorithm in CUDA with shared memory, SIMD instructions, and Krylov solvers, we can also explore its application to modularity clustering problems. Modularity clustering is a graph-based clustering technique that aims to identify densely connected groups of nodes within a larger graph. One approach to solving this problem involves formulating it as an optimization problem, where the objective is to maximize the modularity score, a measure of the quality of the clustering. However, solving this optimization problem can be computationally challenging, especially for large graphs with millions of nodes and edges.

To address this challenge, we can leverage the parallel processing capabilities of the GPU to accelerate the modularity clustering algorithm. Specifically, we can explore the use of CUDA-based implementations of popular modularity clustering algorithms, such as the Louvain method or the Infomap algorithm. By utilizing techniques such as shared memory and SIMD instructions, we can significantly reduce the computation time required to identify the optimal clustering solution. Additionally, by integrating Krylov solvers, we can improve the scalability of the algorithm to handle even larger graphs. Overall, the optimization of modularity clustering algorithms in CUDA with the integration of Krylov solvers represents a promising area of future research for leveraging the power of GPUs to solve complex clustering problems.

Overall, there are many exciting opportunities for research in this field, and this project provides valuable insights into the potential contributions of parallelization of K-means clustering in science and technology.

IX. Iteration 2- Progress Report

Our initial project was based on OpenCV and CUDA implementation for lane detection. During the iteration 2 timeline, we were working on the principle component of lane detection algorithm and based on our finding we were working on following details below:

Canny edge detection is a popular edge detection algorithm that was developed by John F. Canny in 1986. It is a multi-stage algorithm that aims to detect the edges in an image while minimizing noise and false edges. The algorithm has the following steps:

1. Smoothing: The image is first smoothed using a Gaussian filter to remove noise and make the edges more pronounced.
2. Gradient Calculation: The gradient of the image is then calculated using a Sobel operator in both the horizontal and vertical directions. This calculates the strength and direction of the edges.
3. Non-maximum suppression: The gradient magnitude is then thinned out by setting all non-maximum pixels to zero. This is done to preserve only the most significant edges.
4. Double thresholding: A double threshold is then applied to the gradient magnitude image to classify the remaining pixels as strong, weak, or non-edges.
5. Edge tracking: Finally, the strong edges are connected by tracking along the weak edges that are adjacent to them.

Canny edge detection is widely used in computer vision applications such as object detection, lane detection, image segmentation, and motion detection. It is effective at detecting edges in images with low contrast and noisy backgrounds. However, it may not work well on images with complex textures or multiple overlapping edges.

The Hough Transform is a computer vision technique used for detecting simple shapes, such as lines or circles, in an image. The algorithm works by transforming the image from the spatial domain to the Hough space, where the parameters of the shape being detected can be represented as points.

In the case of detecting lines, the Hough Transform works as follows:

1. Edge detection: The first step is to detect edges in the image using an edge detection algorithm, such as Canny edge detection.
2. Hough space representation: Each edge pixel is then transformed into a curve in the Hough space, where the curve represents all possible lines that could pass through the edge pixel in the original image. The curves are represented as points in the Hough space, where each point corresponds to a line in the original image.
3. Accumulator voting: The curves in the Hough space are then counted to determine which lines are present in the original image. This is done by creating an accumulator array in the Hough space and incrementing the count for each point that lies on a curve.
4. Thresholding: The lines are then selected based on a threshold, which determines the minimum number of votes required for a line to be considered valid.
5. Backprojection: Finally, the lines are back-projected onto the original image to display the detected lines.

The Hough Transform is a powerful technique for detecting lines in an image, and it can also be extended to detect other shapes, such as circles or ellipses. However, it can be computationally expensive and may not work well in images with complex structures or overlapping shapes.

For our Lane Detection algorithm implementation, we attempted to use OpenCV and CUDA. We were also planning to incorporate the thrust library into our code, which will allow us to make use of advanced data structures such as device_vector. To smooth the image, we were planning to use the GaussianBlur () library from OpenCV and implement it in our own Canny Edge Detection algorithm. Then use the Canny () library from OpenCV to validate the correctness of our implementation. Next, we crop the image to focus on the region of interest containing the lane. To calculate the rho, we attempted to implement a kernel using the equation rho = x cos(θ) + y sin(θ). To further improve our implementation by finding the local maxima within the accumulator matrix using atomicAdd () function to update the votes in it. We dedicated a lot of time and effort to write the inner workings of this algorithm and test it by printing expected values.

However, we encountered some challenges when we tried to build the actual solution using OpenCV library. We discovered that there was a version mismatch between CUDA and OpenCV, and they were not compatible with each other. Despite our efforts to update and reinstall CUDA, NVCC compiler, and OpenCV, we were unable to build and run our code. Upon receiving feedback from our classmates, we learned that currently, OpenCV with CUDA is not supported, and NVIDIA is working on integration. Although there might be a workaround using CMAKE and compiling both CUDA and OpenCV separately, we could not figure it out in time due to our limited knowledge on OpenCV and CMAKE.

Our issue with OpenCV and CUDA compatibility continued until the last week of class and based on the progress and roadblock we started looking into K-mean clustering project as another great possible project that cover all the knowledge we learned in this class. This is also the topic we presented earlier in the quarter and have good grasp of knowledge. Based on these attributes, we choose K-mean clustering project as our final project.

X. Reference

1. https://www.askpython.com/python/examples/k-means-clustering-from-scratch
2. Arthur, D., & Vassilvitskii, S. (2007). K-means++: The advantages of careful seeding. Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, 1027-1035.
3. Guo, Y., Wang, X., & Wang, X. (2012). Accelerating K-Means algorithm with GPU. Procedia Engineering, 29, 3070-3075.
4. MacQueen, J. B. (1967). Some methods for classification and analysis of multivariate observations. Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, 1(14), 281-297.
5. J. Zhou, Y. Zhang, Y. Jiang, C. L. P. Chen and L. Chen, "A distributed K-means clustering algorithm in wireless sensor networks," 2015 International Conference on Informative and Cybernetics for Computational Social Systems (ICCSS), Chengdu, China, 2015, pp. 26-30, doi: 10.1109/ICCSS.2015.7281143.
6. Janki, Manish, and Kaushik Roy. "Energy-efficient K-means clustering on GPUs." 2015 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 2015.
7. https://kb.iu.edu/d/bcgu
8. https://www.sciencedirect.com/science/article/pii/S1877050917307913?via%3Dihub=
9. https://docs.nvidia.com/nsight-systems/UserGuide/index.html

XI. Appendix

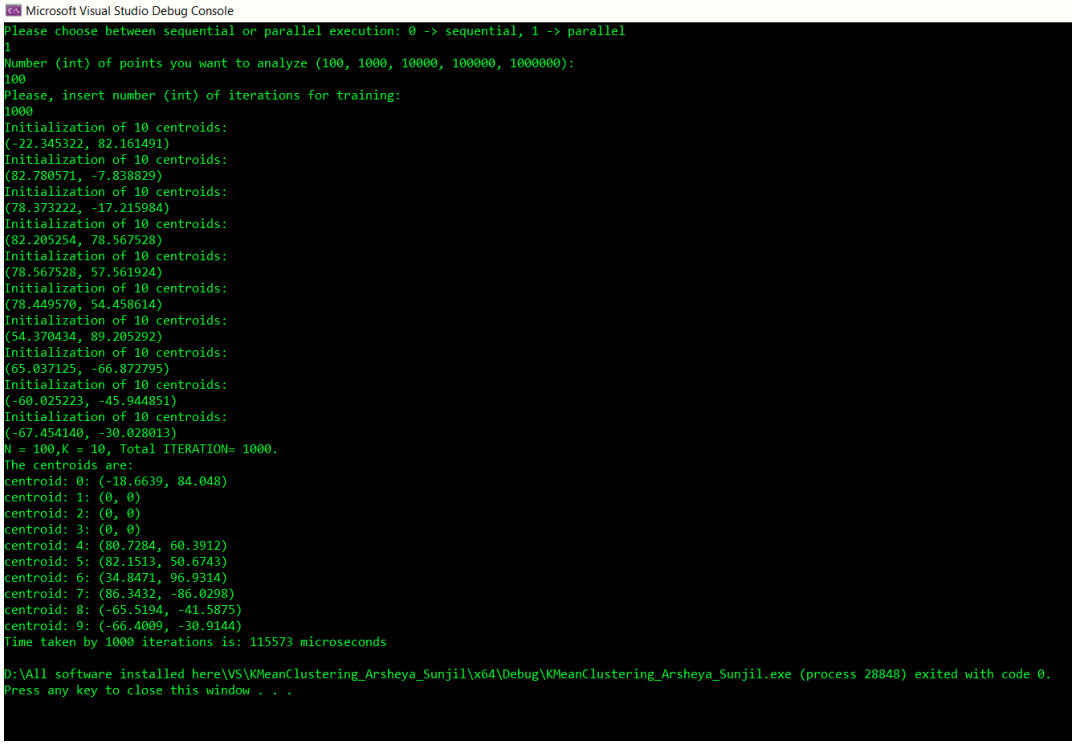

Figure 10: 100 data points, 10 clusters, 1000 iterations for parallel implementation

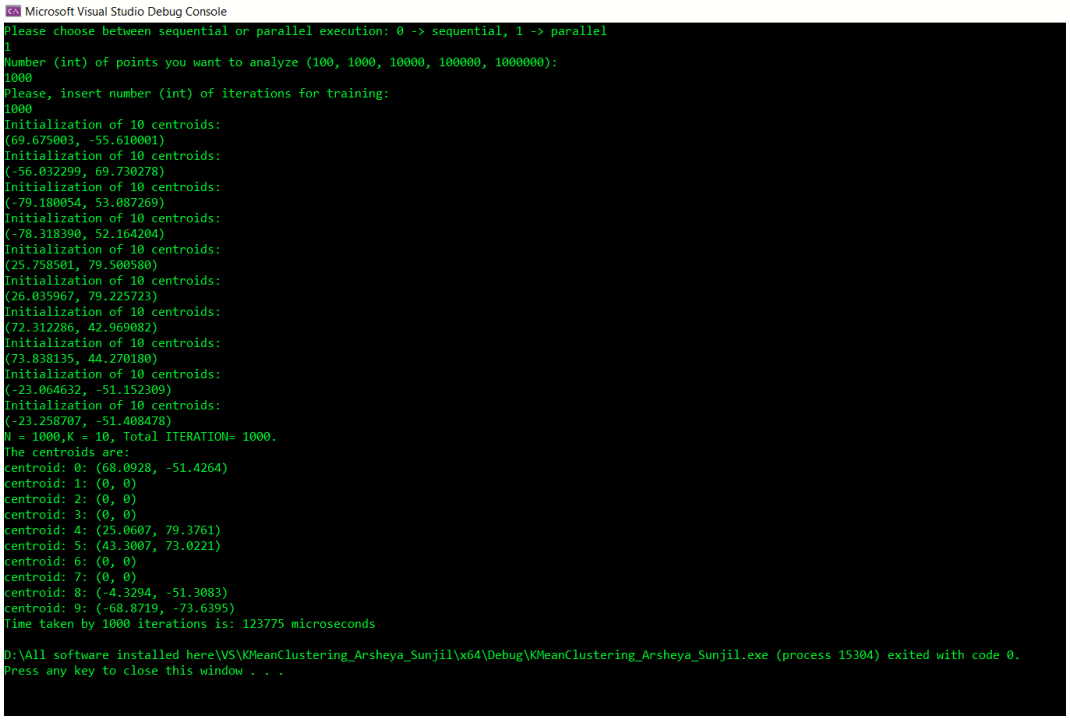

Figure 11: 1000 data points, 10 clusters, 1000 iterations for parallel implementation

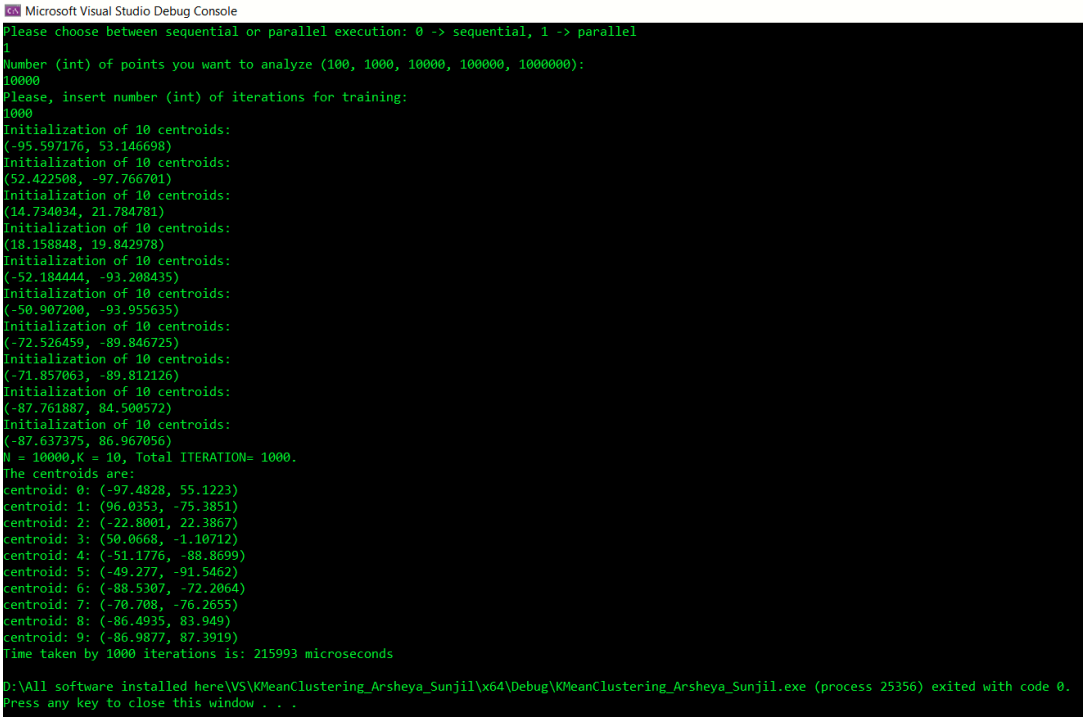

Figure 12: 10000 data points, 10 clusters, 1000 iterations for parallel implementation

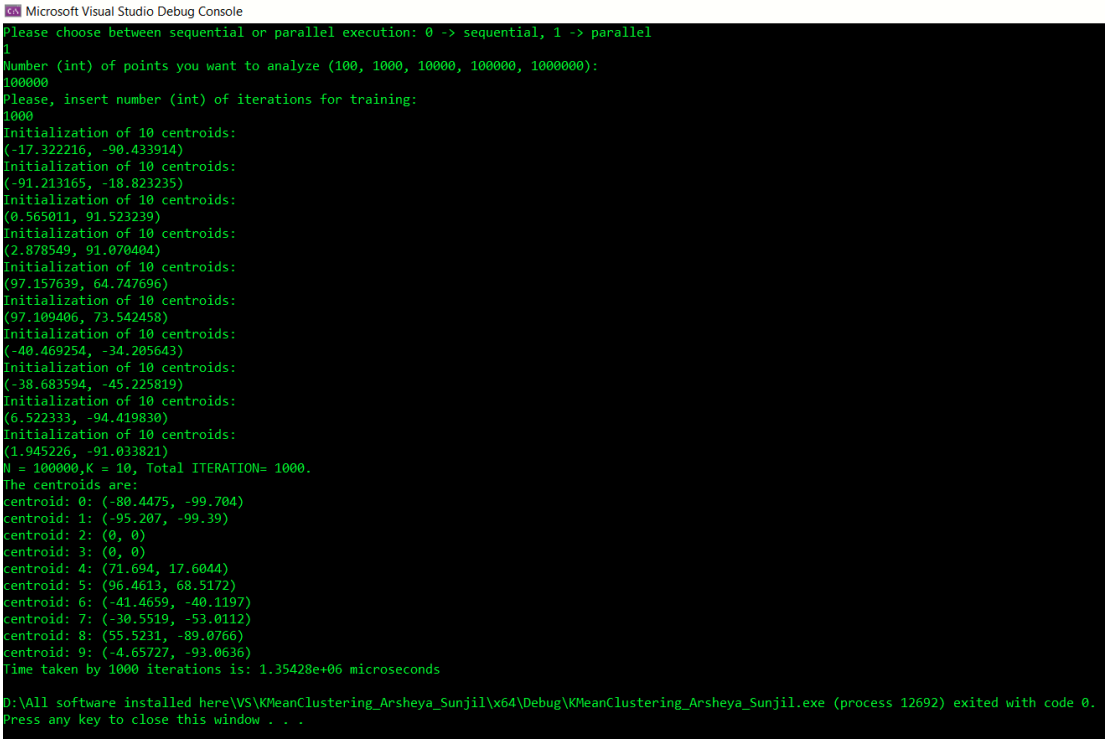

Figure 13: 100000 data points, 10 clusters, 1000 iterations for parallel implementation

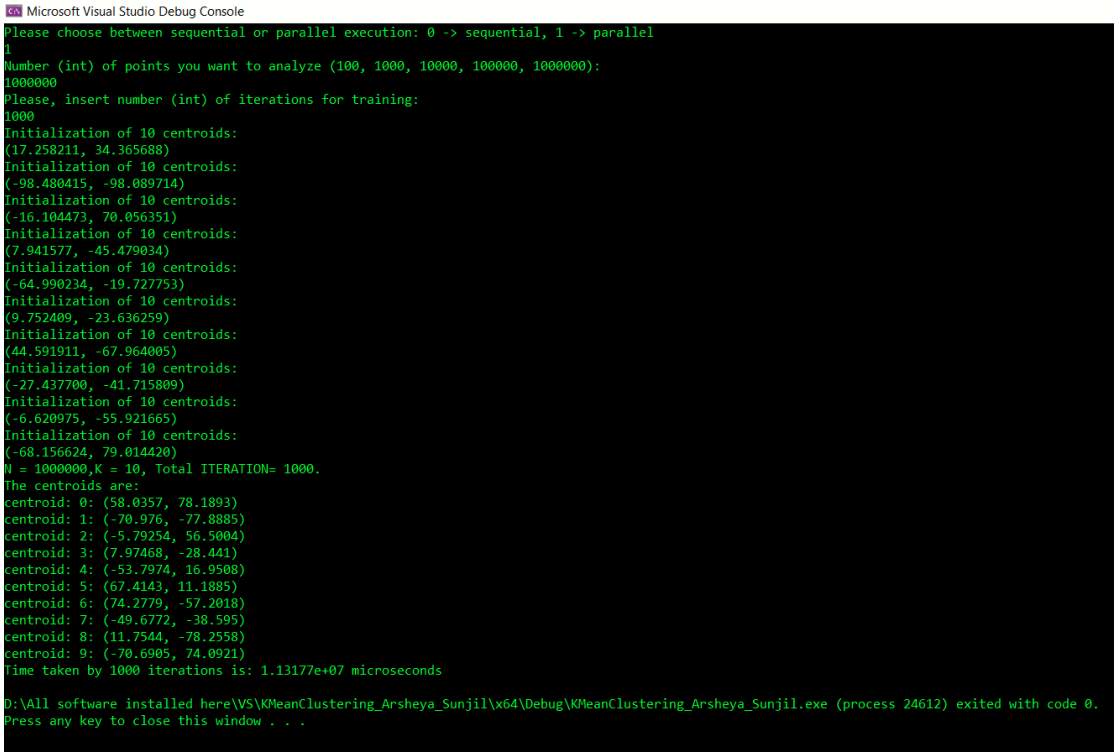

Figure 14: 1000000 data points, 10 clusters, 1000 iterations for parallel implementation

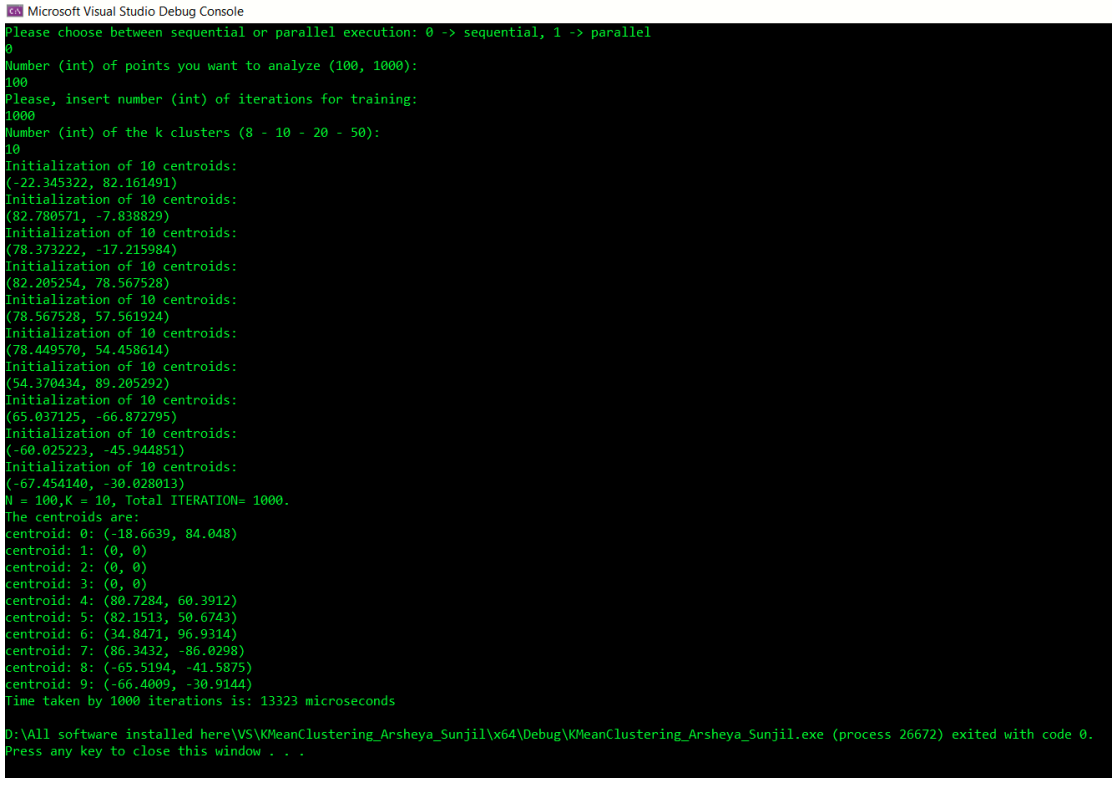

Figure 15: 100 data points, 10 clusters, 1000 iteration for sequential implementation

```
Microsoft Visual Studio Debug Console
Please choose between sequential or parallel execution: 0 -> sequential, 1 -> parallel
0
Number (int) of points you want to analyze (100, 1000):
1000
Please, insert number (int) of iterations for training:
1000
Number (int) of the k clusters (8 - 10 - 20 - 50):
10
Initialization of 10 centroids:
(69.675003, -55.610001)
Initialization of 10 centroids:
(-56.032299, 69.730278)
Initialization of 10 centroids:
(-79.180054, 53.087269)
Initialization of 10 centroids:
(-78.318390, 52.164204)
Initialization of 10 centroids:
(25.758501, 79.500580)
Initialization of 10 centroids:
(26.035967, 79.225723)
Initialization of 10 centroids:
(72.312286, 42.969082)
Initialization of 10 centroids:
(73.838135, 44.270180)
Initialization of 10 centroids:
(-23.064632, -51.152309)
Initialization of 10 centroids:
(-23.258707, -51.408478)
N = 1000,K = 10, Total ITERATION= 1000.
The centroids are:
centroid: 0: (68.0928, -51.4264)
centroid: 1: (0, 0)
centroid: 2: (0, 0)
centroid: 3: (0, 0)
centroid: 4: (25.0607, 79.3761)
centroid: 5: (43.3007, 73.0221)
centroid: 6: (0, 0)
centroid: 7: (0, 0)
centroid: 8: (-4.3294, -51.3083)
centroid: 9: (-68.8719, -73.6395)
Time taken by 1000 iterations is: 116056 microseconds

D:\All software installed here\VS\KMeanClustering_Arsheya_Sunjil\x64\Debug\KMeanClustering_Arsheya_Sunjil.exe (process 15356) exited with code 0.
Press any key to close this window . . .
```

Figure 15: 1000 data points, 10 clusters, 1000 iteration for sequential implementation

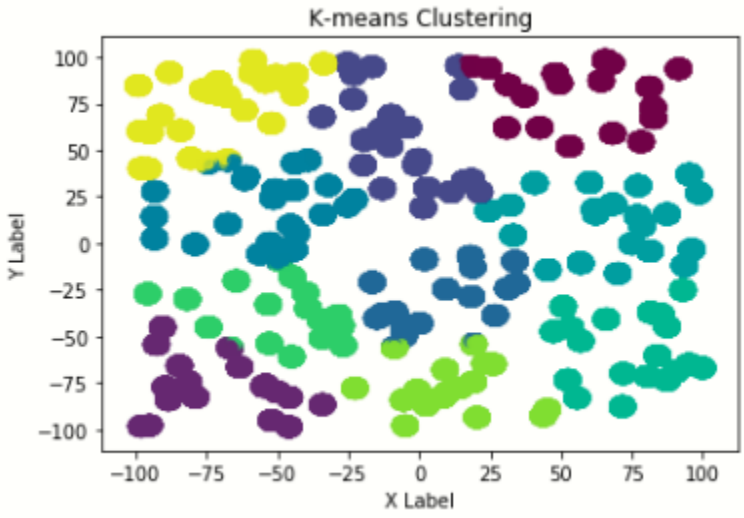

Figure 16: Final scatter plot showing the output clusters.