A Project Report

On

Parallel Processing Of Enhanced K-Means Using Momentum and Repulsion

---

BY

BITTU JAISWAL

SE23MAID022

Under the supervision of

Dr. Praveen Kumar Alapati

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

DEGREE OF MASTER OF TECHNOLOGY

AI5401: DISSERTATION PROJECT

ECOLE CENTRALE SCHOOL OF ENGINEERING

MAHINDRA UNIVERSITY

HYDERABAD

(June 2025)

Acknowledgments

I would like to express my heartfelt gratitude to my supervisor, **Dr. Praveen Kumar Alapati**, for their invaluable guidance, encouragement, and support throughout the course of this project. Their expertise and insight has been instrumental in shaping this work.

I would like to express my profound gratitude to **Prof. Dr. Arun Kumar Pujari, HOD, CSE Department, Mahindra University** for their unwavering support and motivation during my academic journey. This achievement would not have been possible without their constant encouragement and belief in me.

Lastly, I am also deeply grateful to the faculty members of **Ecole Centrale School of Engineering, Mahindra University**, for providing a conducive learning environment and access to the necessary resources. Special thanks to my peers and colleagues for their continuous encouragement and fruitful discussions.

Bittu Jaiswal

Master of Technology, Artificial Intelligence and Data Science

Mahindra University

ECOLE CENTRALE SCHOOL OF ENGINEERING

MAHINDRA UNIVERSITY

HYDERABAD

Certificate

This is to certify that the project report entitled "**Parallel Processing Of Enhanced K-Means Using Momentum and Repulsion**" submitted by Mr/Ms. **Mr. Bittu Jaiswal** (Roll No. **SE23MAID022**) in partial fulfillment of the requirements of the course **AI5401** under my supervision.

Dr. PRAVEEN KUMAR ALAPATI

ECOLE CENTRALE SCHOOL OF ENGINEERING, Hyderabad
Date: 11/06/2025

Abstract

Cluster analysis is a crucial technique in scientific research and business applications for organizing data into meaningful groups. The K-Means clustering algorithm is widely recognized for its efficiency in partitioning datasets but faces limitations when applied to large-scale datasets due to its iterative nature and high execution time. To address these challenges, a parallel implementation of the K-Means algorithm is presented using GPU parallel processing integrated with momentum and centroid repulsion techniques.

This project explores the enhancement of K-Means clustering through parallel processing on GPU using CUDA, integrated with momentum and centroid repulsion techniques. Traditional K-Means, while effective for unsupervised clustering, suffers from computational inefficiency and sensitivity to initial centroid placement—especially when applied to large, high-dimensional datasets. To address these limitations, three approaches were implemented and compared: (1) standard sequential K-Means on CPU, (2) parallel K-Means using GPU, and (3) GPU-accelerated K-Means augmented with momentum to smooth centroid updates and repulsion to maintain cluster separation. Experiments were conducted on datasets ranging from 4 to 20 dimensions with up to two million points, and cluster counts from 2 to 1024. Results demonstrate that GPU acceleration significantly reduces execution time, achieving up to 2.66× speedup over CPU, while the proposed enhanced method with momentum and repulsion further improves performance to 9.06× over CPU. The enhancements also improve clustering stability and scalability, particularly for large values of K. This study confirms the effectiveness of combining parallel processing with algorithmic refinements for scalable, high-performance clustering.

Contents

1 Introduction

1.1 Overview

Clustering is a foundational data mining technique widely used to analyze and group data. It identifies patterns and structures within datasets, dividing them into clusters where elements within the same group are more similar to each other than to those in other groups. Among clustering algorithms, K-Means is one of the most prominent due to its simplicity and efficiency. It works by iteratively assigning data points to clusters based on their proximity to centroids, calculated using the Euclidean distance.

K-Means clustering is a fundamental unsupervised machine learning algorithm that partitions data into K distinct groups based on similarity, typically using Euclidean distance. Its simplicity, efficiency, and ability to reveal natural groupings in data make it one of the most widely used clustering techniques in domains such as image processing, marketing, and bioinformatics. However, K-Means is inherently iterative and sensitive to the initial choice of centroids, which can lead to slow convergence or suboptimal results. These challenges become especially significant when dealing with large-scale or high-dimensional datasets, where the algorithm's performance and accuracy tend to degrade due to increased computational load and complexity. Therefore, optimizing and accelerating K-Means is essential for modern data-intensive applications.

1.2 Motivation

In the modern era of big data, vast amounts of information are continuously being generated across multiple domains such as healthcare, finance, e-commerce, transportation, scientific research, and social media. Making sense of this information requires robust, scalable, and efficient analytical tools. Clustering, particularly K-Means, plays a vital role in unsupervised learning by automatically discovering groupings and structures in unlabeled data. However, traditional implementations of K-Means are not well-suited to modern data challenges due to the following key limitations:

Computational Inefficiency: K-Means involves repetitive computation of distances between points and centroids, followed by centroid updates. For datasets containing millions of points and high-dimensional features, this becomes a computational bottleneck—especially on standard, sequential CPU architectures.

Scalability Issues: As the number of clusters (K) and dimensions increases, so does the algorithm's complexity. The sequential nature of traditional K-Means makes it difficult to scale up effectively to meet the needs of large-scale industrial or scientific applications.

Sensitivity to Initialization: Poor choice of initial centroids can result in suboptimal convergence, requiring more iterations or leading the algorithm to local minima. This inconsistency impacts the reliability of clustering outcomes.

Instability in Centroid Movement: Centroids may oscillate or move erratically during iterations, especially in high-dimensional or noisy data. This can slow down convergence or cause poor cluster separation.

To address these challenges, this project adopts a parallel processing paradigm that leverages OpenMP for CPU-based parallelization and CUDA for GPU acceleration. Parallel architectures can handle the independent nature of clustering operations—such as distance calculations and point

assignments—more efficiently by distributing workloads across multiple threads or cores. This significantly reduces the execution time and allows for real-time or near-real-time data clustering.

Further, to improve convergence behavior and clustering quality, two algorithmic enhancements are integrated into the GPU-based implementation:

Momentum: Inspired by optimization techniques used in deep learning, momentum introduces a "velocity" component to centroid updates. This smooths the update path and reduces oscillations, especially in noisy or sparse data. It helps centroids converge more quickly and stably by building on the previous update direction.

Centroid Repulsion: This mechanism introduces a repulsive force between centroids during updates. It prevents centroids from collapsing into the same region or overlapping, which can lead to poor clustering. By maintaining separation between cluster centers, repulsion promotes clearer boundaries and better-defined clusters.

Together, these innovations not only accelerate the clustering process but also enhance the quality and stability of the results. The motivation behind this project is to demonstrate that combining hardware-level parallelism with algorithm-level enhancements leads to a scalable, high-performance clustering solution capable of handling real-world, large-scale, and high-dimensional datasets effectively.

1.3 Real World Application

Clustering has a wide array of applications in various industries, solving critical real-world challenges:

- **Market Segmentation:** Businesses group customers based on buying behavior, preferences, or demographics to enhance personalized marketing and improve customer satisfaction.
- **Healthcare:** Clustering helps in patient grouping based on symptoms, diagnoses, or treatment responses, enabling targeted healthcare delivery and optimizing resources.
- **Fraud Detection:** Financial institutions use clustering to identify anomalous patterns in transaction data, which may indicate fraudulent activities.
- **Bioinformatics:** Clustering plays a key role in gene expression analysis, protein structure prediction, and drug discovery by grouping similar biological data.
- **Image Analysis:** In computer vision, clustering is used for image compression, object detection, and image segmentation, enabling more efficient storage and processing.
- **Traffic Management:** Urban planners use clustering to analyze traffic patterns and identify congestion hotspots, improving route optimization and infrastructure planning.
- **Social Network Analysis:** Clustering helps group users with similar interactions and interests, assisting in community detection and targeted advertising.
- **Recommendation Systems:** E-commerce and streaming platforms cluster users based on preferences to offer personalized recommendations, enhancing user experience.

By addressing computational challenges through parallel processing, the project bridges the gap between clustering algorithms and their real-world applications, ensuring scalability and efficiency in handling massive datasets.

2 Problem Definition

2.1 Problem Description

The task involves optimizing the K-Means clustering algorithm to handle large datasets more efficiently by utilizing parallel processing techniques. The goal is to improve the computational performance of K-Means, which is known to be computationally expensive for large datasets, especially when the number of data points, clusters, or dimensions increases. Specifically, this problem addresses the challenges faced in clustering large-scale datasets and explores how parallelism can enhance the speed and scalability of the K-Means algorithm.

Core Problem

The K-Means clustering algorithm works by assigning data points to clusters based on their proximity to centroids. The iterative process involves:

1. Assigning each data point to the nearest centroid.
2. Updating the centroids by calculating the mean of the points in each cluster.
3. Repeating these steps until the centroids converge or a maximum number of iterations is reached.

However, as the size of the data and number of clusters grow, this process can become slow and inefficient, especially when performed serially. The primary issues are:

- **Computational Bottleneck:** Calculating the distance between each data point and every centroid in each iteration is computationally expensive.
- **Scalability:** As the number of data points (n), clusters (K), and dimensions (features) increases, the algorithm's time complexity increases, making it impractical for real-world datasets.

3 Proposed Solution

To overcome the performance and stability challenges of traditional K-Means clustering, especially on large and high-dimensional datasets, we propose an enhanced GPU-based implementation of K-Means that incorporates two key algorithmic improvements: Momentum and Centroid Repulsion. This hybrid approach combines CUDA-based parallel computation with optimization-inspired centroid behavior, offering superior scalability, faster convergence, and more distinct clusters.

3.1 Enhanced GPU-Based K-Means

The proposed method builds upon the existing GPU-accelerated K-Means implementation by introducing the following enhancements:

Momentum-Based Centroid Updates Momentum smooths centroid transitions between iterations. Instead of updating centroids solely based on current data assignments, a portion of the previous movement direction is retained, reducing oscillations and helping the algorithm converge faster and more stably.

Centroid Repulsion Mechanism Centroid repulsion introduces a repelling force between centroids to ensure they remain well-separated. This prevents multiple centroids from collapsing into the same data region, which is especially useful when clustering with a large number of clusters (K). It leads to better inter-cluster separation and prevents redundancy.

CUDA-Accelerated Computation All distance calculations, assignments, centroid updates (including momentum and repulsion), and convergence checks are performed entirely on the GPU using thousands of CUDA threads. This ensures high throughput and significantly reduced execution time.

Stability and Convergence Checks The algorithm includes mechanisms to monitor convergence based on centroid displacement thresholds and maximum iteration limits, ensuring both speed and reliability.

Evaluation Across Large Ranges of K The solution has been tested extensively with K values ranging from 2 to 1024, and on datasets with varying sizes and dimensions (4D, 10D, 20D), validating both scalability and effectiveness.

3.2 Momentum: Accelerating Convergence

Purpose:
To smooth the movement of centroids across iterations, avoiding sudden jumps and oscillations that can slow down convergence or trap the algorithm in local minima.

Update Formula:

$$C^{(t+1)} = \alpha \cdot C^{(t-1)} + (1 - \alpha) \cdot C^{(t)}$$

- $C^{(t+1)}$: New centroid
- α: Momentum factor
- $C^{(t-1)}$: Previous centroid
- $C^{(t)}$: Current centroid

This approach mimics momentum in physics and optimization algorithms (like SGD with momentum), helping the centroids move smoothly in the direction of historical update vectors.

3.3 Repulsion: Maintaining Cluster Separation

Purpose:
To discourage centroids from clustering too closely or overlapping, especially when K is large or when data points are unevenly distributed.

Repulsion Force Formula:

$$\mathbf{F}_{\text{repel}} = \beta \cdot \frac{(\mathbf{c}_i - \mathbf{c}_j)}{\|\mathbf{c}_i - \mathbf{c}_j\|^2 + \varepsilon}$$

- $\mathbf{c}_i, \mathbf{c}_j$: Coordinates of centroids i and j, with $i \neq j$
- β: Repulsion coefficient that determines the strength of the repulsive force
- ε: A small constant added to avoid division by zero

This force is added to the centroid update during each iteration, pushing centroids apart and encouraging more distinct clusters.

3.4 Expected Outcome

- **Memory Usage:** Managing large datasets in memory, especially when dealing with millions of points, can become a challenge.
- **Load Balancing:** Properly balancing the workload across multiple CPU cores to avoid bottlenecks and ensure optimal performance.
- **Convergence Behavior:** Ensuring that the parallelized algorithm converges correctly and efficiently while minimizing computational overhead.

4 Implementation

The implementation of this project involves three distinct versions of the K-Means clustering algorithm, evaluated across multiple datasets and tested using a high-performance computing environment equipped with NVIDIA A100 GPUs. Each method progressively improves upon the baseline by leveraging parallelization and algorithmic enhancements.

4.1 System Configuration

All experiments were conducted on a system with the following specifications:

- **GPU Model**: NVIDIA A100-SXM4-40GB
- **Total GPUs**: 4
- **Memory per GPU**: 40,536 MiB (approximately 40 GB)
- **Architecture**: Ampere
- **CUDA Version**: 11.4
- **Driver Version**: 470.161.03

The CUDA platform enables high-throughput parallel execution by launching thousands of threads on the GPU simultaneously, making it particularly well-suited for the iterative and compute-intensive nature of K-Means clustering.

4.2 Implementation Approaches

4.2.1 Method 1: Sequential Standard K-Means (CPU-Based)

This version implements the traditional K-Means algorithm executed sequentially on the CPU, serving as the baseline for performance comparison.

Characteristics:

- Clustering is performed without any parallelism.
- Distance calculation, cluster assignment, and centroid updates are performed iteratively for each data point and cluster.
- Datasets with varying dimensionality (e.g., 4D, 10D, 20D) and different numbers of data points (up to 2 million) are used.
- Number of clusters (K) varies from 2 to 1024, increasing in powers of 2.
- For each configuration, execution time is recorded to evaluate speedup achieved by other methods.

This implementation highlights the performance bottlenecks of K-Means in large-scale data environments.

4.2.2 Method 2: Parallel K-Means Using GPU (CUDA)

This version leverages the CUDA parallel programming model to accelerate K-Means computations using GPU threads.

Key Features:

- Each GPU thread is assigned to handle one data point, exploiting massive data parallelism.
- **Parallel Execution Units handle:**
  - *Distance Calculations*: Between data points and all centroids.
  - *Cluster Assignment*: Assigning points to the nearest centroid.
  - *Centroid Updates*: Using parallel reductions to accumulate values from assigned points.
- After each iteration, centroids are updated globally across all points, and convergence is checked.
- Execution is repeated for varying values of K from 2 to 1024 (powers of 2).
- Execution time is benchmarked against the sequential version to determine speedup.

This GPU-based implementation delivers a significant reduction in computation time by parallelizing the inherently independent tasks of K-Means.

4.2.3 Method 3: Parallel GPU K-Means with Momentum and Repulsion

The third and most advanced method integrates CUDA-based parallel K-Means with two algorithmic enhancements: **momentum** and **centroid repulsion**. These techniques improve both the convergence speed and the quality of cluster separation.

Data Representation Each data point is stored as a fixed-dimensional array using the following structure:

```
typedef struct {
    float data[MAX_DIM];
} Point;
```

The implementation supports up to 10-dimensional data, configurable via the constant `MAX_DIM`. Data is read from a CSV file using a custom parser.

CUDA Kernel for Label Assignment A CUDA kernel, `assign_labels()`, is responsible for assigning each data point to the nearest centroid using the Euclidean distance:

$$\text{distance}(p_i, c_j) = \sqrt{\sum_{d=1}^{D} (p_i^d - c_j^d)^2}$$

Each thread processes one point independently, exploiting massive parallelism on the GPU.

Momentum-Based Update To smooth the movement of centroids and accelerate convergence, a momentum vector is maintained for each centroid. The updated position of centroid i at iteration $t+1$ is computed as:

$$C_i^{(t+1)} = \alpha \cdot C_i^{(t-1)} + (1 - \alpha) \cdot C_i^{(t)}$$

- α: Momentum coefficient (set to 0.5 in implementation)
- $C_i^{(t)}$: Current centroid
- $C_i^{(t-1)}$: Previous centroid

Centroid Repulsion To avoid centroid overlap and improve separation, a repulsive force is applied between each pair of centroids:

$$F_{\text{repel}} = \beta \cdot \frac{(c_i - c_j)}{\|c_i - c_j\|^2 + \varepsilon}$$

- β: Repulsion strength (set to 0.01)
- ε: A small constant (1e-5) to ensure numerical stability

This force adjusts centroids to maintain distinct cluster boundaries, particularly useful when K is large.

Execution Workflow The implementation follows this iterative process:

1. Copy initial points to GPU memory
2. Launch CUDA kernel to assign labels
3. Copy labels back to host
4. Compute new centroids on CPU
5. Apply momentum and repulsion updates to centroids
6. Check for convergence using maximum centroid shift

Convergence Criterion The algorithm terminates early if the maximum centroid shift across iterations is less than a threshold (tol $= 1 \times 10^{-4}$):

$$\max_i \sqrt{\sum_{d=1}^{D} (C_i^{(t)} - C_i^{(t-1)})^2} < \text{tol}$$

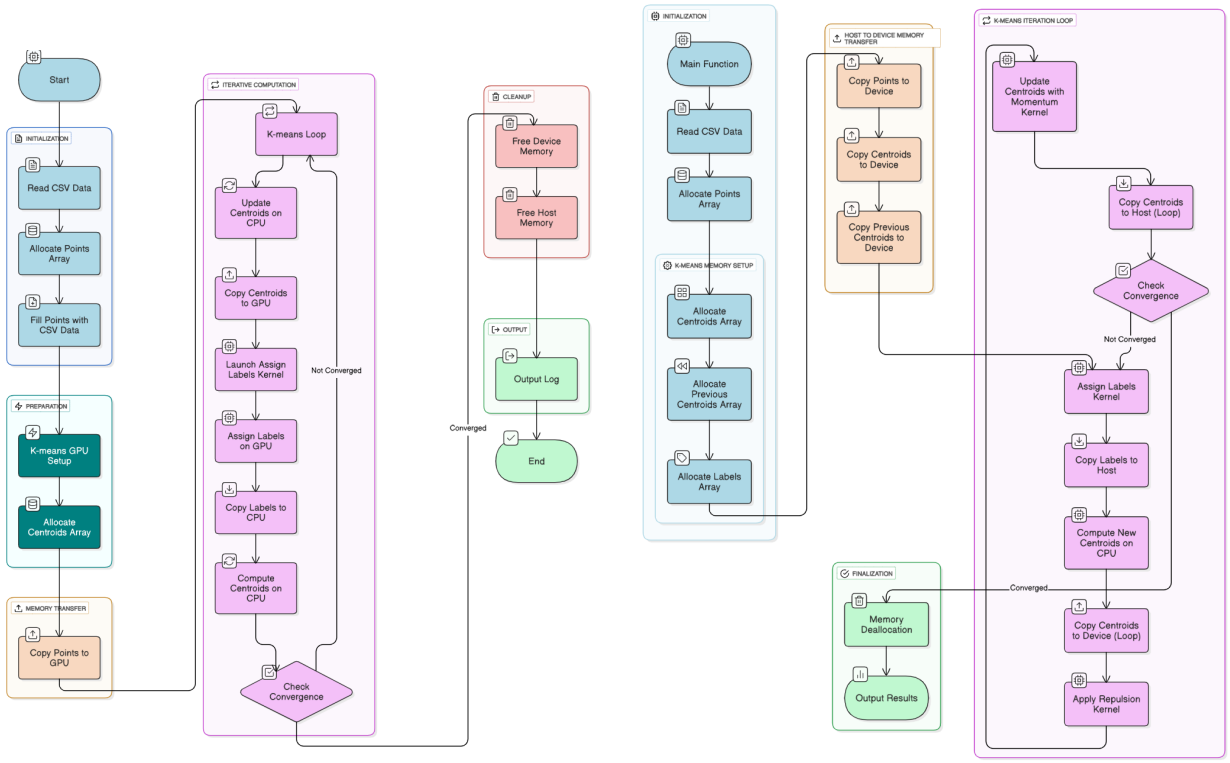
Execution Metrics The program logs the following metrics:

- **Execution Time**: Time taken to reach convergence.
- **Converged Iterations**: Number of iterations until convergence.

Experiments are performed across varying K values (from 2 to 1024 in powers of 2) on:

- 4D data with 1 million points
- 10D and 20D data with up to 2 million points

4.3 Flowchart



(a) GPU implementation

(b) GPU with momentum and repulsion

Figure 1: Flowcharts illustrating GPU implementation and GPU implementation with momentum and repulsion.

5 Results

5.1 Experimental Setup for Momentum

Experiments were conducted on two datasets containing 1 million data points each, with dimensionalities of 10D and 20D respectively. The number of clusters K ranged from 2 to 1024 in powers of 2, and the momentum coefficient α was varied from 0.1 to 0.9.

For each configuration, the following metrics were recorded:

- **Execution Time (in seconds)**
- **Converged Iterations**

5.1.1 Graphical Results

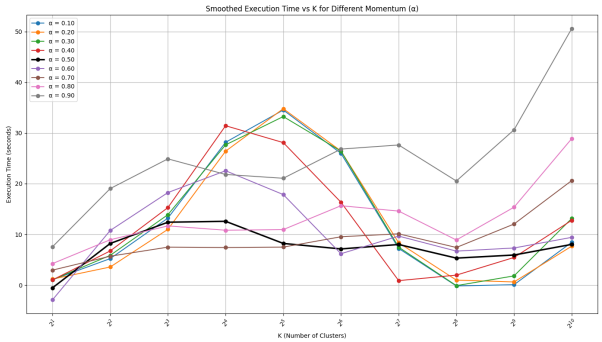


Figure 2: 10D Dataset (1M Points): Execution Time vs. K for various α

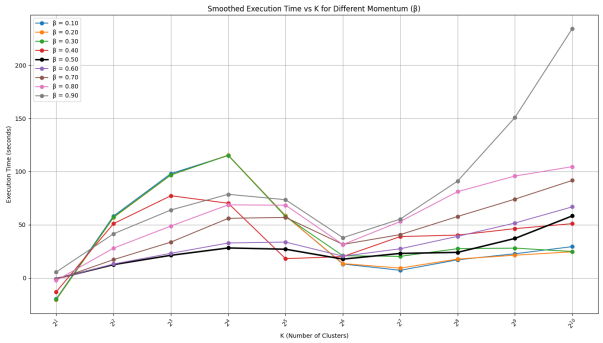


Figure 3: 20D Dataset (1M Points): Execution Time vs. K for various α

5.1.2 Discussion and Comparison

Execution Time and Dimensionality As expected, execution time increases with dimensionality:

- For small K (2 to 8), both 10D and 20D datasets show short execution times and rapid convergence.
- For large K (128 to 1024), the 20D dataset takes significantly longer to converge or fails to converge with higher α values.
- The 10D dataset consistently performs faster across all α values, due to lower computational complexity in centroid updates and distance calculations.

Effect of Momentum (α)

- Very low α (0.1–0.2): Often results in slow convergence or failure to converge, especially for large K.
- Moderate α (0.3–0.6): Achieves the best performance, balancing stability and convergence speed.
- High α (0.8–0.9): Causes divergence or excessive oscillation in many cases, especially for $K \geq 128$ in 20D.

Convergence Behavior

- The number of iterations needed to converge increases with both K and α beyond the optimal range.
- In the 20D dataset, the combination of high K and high α often results in zero convergence, indicating instability.
- In contrast, the 10D dataset shows more stable behavior and faster convergence in the 0.3–0.6 momentum range.

5.1.3 Comparison Summary

Metric	10D Dataset	20D Dataset
Best Momentum Range	0.3 – 0.6	0.3 – 0.5
Execution Time (Avg)	Lower	Higher
Failure Cases (No Convergence)	Fewer (mostly $\alpha > 0.7$)	More frequent
Sensitivity to α	Moderate	High
Scalability with K	Good	Moderate

Table 1: Comparison of K-Means GPU+Momentum+Repulsion on 10D vs. 20D datasets

5.1.4 Observation

The results demonstrate that:

- GPU-accelerated K-Means with momentum and repulsion scales well for high K values on both 10D and 20D datasets.
- A moderate momentum coefficient (α in the range 0.3 to 0.6) provides the best balance of stability and performance.
- Higher-dimensional data increases convergence difficulty, making careful tuning of α crucial.
- The 10D dataset outperforms the 20D dataset across all metrics, reaffirming that dimensionality plays a significant role in clustering complexity and performance.

5.2 Experimental Setup for Repulsion

This experiment evaluates the impact of varying **repulsion strength** (β) on execution time and convergence performance of the enhanced K-Means algorithm. Repulsion is introduced to maintain centroid separation and prevent overlap, especially at high K values.

The repulsion coefficient β is varied from 0.1 to 0.8, and the number of clusters K ranges from 2 to 1024 (in powers of 2). The following metrics are recorded:

- **Execution Time (in seconds)**
- **Converged Iterations** (0 indicates no convergence)

5.2.1 Graphical Results

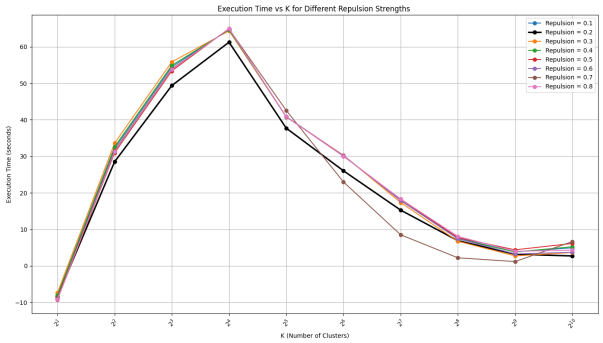


Figure 4: 10D Dataset (1M Points): Execution Time vs. K for varying β

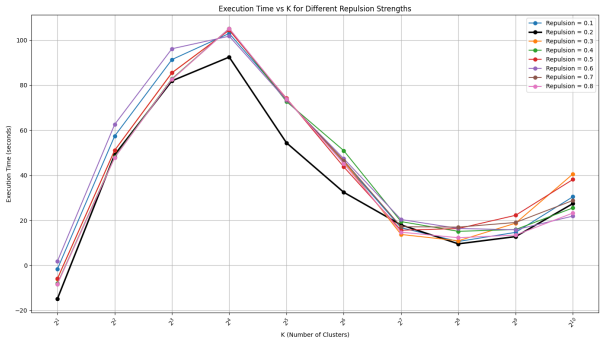


Figure 5: 20D Dataset (1M Points): Execution Time vs. K for varying β

5.2.2 Discussion and Comparison

Execution Time and Dimensionality

- As dimensionality increases, execution time increases due to costlier distance and repulsion calculations.
- For small K (2 to 8), both 10D and 20D datasets show relatively fast convergence across all β values.
- For large K (128 to 1024), the 20D dataset requires significantly more time to converge or fails to converge, especially at low repulsion strengths.

Effect of Repulsion Strength (β)

- **Low β (0.1–0.2)**: Frequently fails to converge for large K due to centroid collapse.
- **Moderate β (0.3–0.6)**: Offers optimal performance, promoting separation without destabilizing updates.
- **High β (0.7–0.8)**: Increases stability in some cases but may slow convergence due to excessive repulsion.

Convergence Behavior

- In 10D, convergence is generally more stable across β values, especially for moderate K.
- In 20D, repulsion plays a stronger role. Moderate β helps convergence at higher K, but both low and high extremes cause instability or high iteration counts.
- Several cases (e.g., $\beta = 0.1$, $K = 8$–64) failed to converge in both datasets.

5.2.3 Comparison Summary

Metric	10D Dataset	20D Dataset
Best Repulsion Range (β)	0.3 – 0.6	0.4 – 0.6
Execution Time (Avg)	Lower	Higher
Failure Cases (No Convergence)	Rare (mostly $\beta = 0.1$)	Frequent at low β
Sensitivity to β	Moderate	High
Scalability with K	Good	Challenging for $K \geq 512$

Table 2: Comparison of K-Means + Repulsion on 10D vs. 20D datasets

5.2.4 Observation

The experiment demonstrates that:

- Repulsion strength significantly influences centroid stability and convergence.
- Moderate values of β (0.3 to 0.6) strike a good balance between separation and convergence speed.
- High-dimensional datasets like 20D are more sensitive to β and require careful tuning.
- The 10D dataset exhibits better convergence consistency and lower execution time under the same conditions.

5.3 Experimental Setup 4D 1 million data points

To assess the efficiency and scalability of various K-Means implementations, experiments were performed on a 4-dimensional dataset consisting of 1 million data points. The following three methods were compared:

1. **Sequential Standard K-Means (CPU)**
2. **Parallel K-Means using GPU (CUDA)**
3. **GPU-Accelerated K-Means with Momentum and Repulsion**
   ($\alpha = 0.5$, $\beta = 0.2$)

Cluster sizes K ranged from 2 to 1024 in powers of 2. Metrics recorded:

- Execution Time (in seconds)
- Number of Iterations Until Convergence

5.3.1 Graphical Results

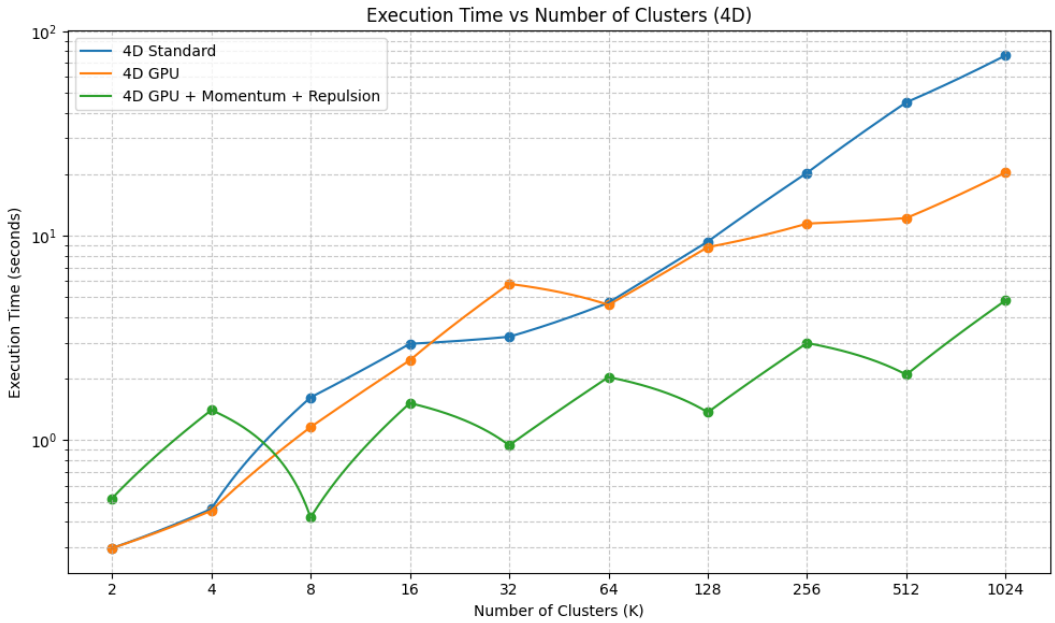


Figure 6: Execution Time vs. Cluster Size (K) for Sequential, GPU, and GPU+Momentum+Repulsion Methods on 4D Dataset

5.3.2 Discussion and Comparison

Execution Time Trends

- **Sequential CPU** shows a linear to exponential rise in execution time as K increases. It reaches **76.29 seconds** at $K = 1024$.
- **Parallel GPU** implementation significantly reduces time, capping at **20.43 seconds** for $K = 1024$—almost **3.7× faster**.
- **GPU with Momentum + Repulsion** further improves efficiency, especially for high K. For $K = 512$ and $K = 1024$, it achieves execution times of **1.89s** and **4.83s**, respectively.

Convergence Behavior

- Sequential and basic GPU versions converge in fewer iterations but with larger centroid shifts per iteration.
- GPU + Momentum + Repulsion converges with more stability, at the cost of higher iteration counts (e.g., 1870 iterations for $K = 1024$).
- Despite higher iterations, execution time remains low due to lightweight GPU iterations and smoothed centroid movement.

5.3.3 Summary Table

K	Sequential (s)	GPU (s)	GPU + M+R (s)
2	0.30	0.30	0.52
4	0.46	0.45	1.40
8	1.62	1.16	0.42
16	2.96	2.46	1.52
32	3.21	5.82	0.94
64	4.71	4.59	2.04
128	9.37	8.80	1.37
256	20.32	11.48	2.99
512	44.97	12.21	1.89
1024	76.29	20.43	4.83

Table 3: Execution Time Comparison for 4D Dataset Across Methods

5.3.4 Observation

- For smaller K (2–16), differences between all three methods are minimal.
- For larger K (≥ 64), the enhanced method provides superior time performance and improved cluster stability.
- GPU with Momentum and Repulsion is especially effective in avoiding centroid collapse at high K.
- GPU acceleration provides a major speedup over sequential K-Means.
- The GPU method with momentum and repulsion significantly outperforms both baseline methods for large K.
- The technique balances stability and speed, making it ideal for high-resolution clustering tasks where centroid convergence is sensitive.
- Even with higher iteration counts, the final convergence time is lowest using the enhanced method due to GPU parallel efficiency and smoothed updates.

5.4 Experimental Setup 10D Dataset (1M and 2M Points)

Experiments were conducted on a 10-dimensional dataset with two different sizes:

- **1 Million** data points
- **2 Million** data points

Each data set was clustered using the following three methods:

1. Sequential Standard K-Means (CPU)

2. Parallel GPU K-Means (CUDA)
3. GPU K-Means with Momentum ($\alpha = 0.5$) and Repulsion ($\beta = 0.2$)

We varied the number of clusters K from 2 to 1024 (powers of 2). Metrics recorded:

- Execution Time (seconds)
- Converged Iterations

5.4.1 Graphical Results

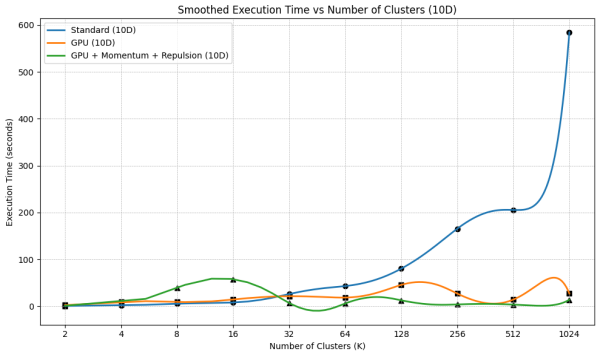


Figure 7: 10D Dataset (1M Points): Execution Time vs. K

Figure 8: 10D Dataset (2M Points): Execution Time vs. K

5.4.2 Discussion and Comparison

Execution Time Trends

- In both data sets, the **sequential K-means** method exhibits a rapid increase in execution time as K increases, reaching **583s (1M)** and over **1139s (2M)** for $K = 1024$.
- **GPU K-Means** reduces runtime significantly. However, for high K, some anomalies (e.g., long times for low K) are due to overhead from many iterations.
- **GPU with Momentum and Repulsion** consistently performs faster than both previous methods for large K. For example:
  - At $K = 1024$: GPU(M+R) took **13.91s (1M)** and **36.24s (2M)**
  - Compared to sequential: **583.84s (1M)**, **1139.40s (2M)**

Convergence Behavior

- **Sequential** and basic **GPU** methods often reach a maximum iteration cap (300) at high K, indicating poor convergence.
- **GPU+Momentum+Repulsion** converges in more iterations, but with greater centroid stability and fewer early stops.
- The added computational cost per iteration is balanced by smoother centroid motion and better separation.

Scaling with Data Volume

- As the data volume doubles (1M $\rightarrow$ 2M), the sequential run time nearly doubles or worse.
- GPU methods handle scaling better. For instance:
  - $K = 256$: GPU(M+R) time was **3.90s (1M)** vs. **6.29s (2M)** — nearly linear scaling.
- GPU parallelism ensures efficient usage of compute resources even as the number of points increases.

5.4.3 Summary Table

K	10D - 1M Points			10D - 2M Points		
	Sequential	GPU	GPU + M+R	Sequential	GPU	GPU + M+R
2	0.67	2.47	1.13	1.17	—	1.66
4	1.81	9.90	5.01	2.66	51.16	10.28
8	5.43	9.23	39.60	10.10	22.41	11.61
16	8.02	14.45	57.86	23.32	47.91	7.03
32	26.07	21.39	6.90	55.34	29.04	6.60
64	43.04	18.58	5.97	87.35	94.90	17.94
128	80.24	45.66	12.71	163.95	195.69	15.13
256	165.47	27.08	3.90	299.74	35.83	6.29
512	205.44	14.23	3.57	580.11	47.21	10.66
1024	583.84	27.17	13.92	1139.40	98.75	36.24

Table 4: Execution Time (in seconds) Comparison for 10D Dataset (1M and 2M Points)

5.4.4 Observation

- GPU-based K-Means significantly outperforms the sequential version, especially at high K.
- The addition of momentum and repulsion further improves performance and convergence stability.
- GPU with Momentum and Repulsion scales well to 2M points and shows the best overall performance at high cluster counts.
- This combination is ideal for large-scale, high-dimensional clustering problems where both speed and quality are critical.

5.5 Experimental Setup for 20D Dataset (1M and 2M Points)

To evaluate the performance of different K-Means clustering implementations on high-dimensional data, we ran experiments on 20-dimensional datasets of two sizes:

- **1 million** data points
- **2 million** data points

We tested three algorithms:
- Sequential Standard K-Means (CPU)

- Parallel GPU K-Means (CUDA)
- GPU K-Means with Momentum ($\alpha = 0.5$) and Repulsion ($\beta = 0.2$)

For each test, the cluster count K ranged from 2 to 1024 (powers of 2). Metrics recorded:

- Execution Time (in seconds)
- Converged Iterations

5.5.1 Graphical Results

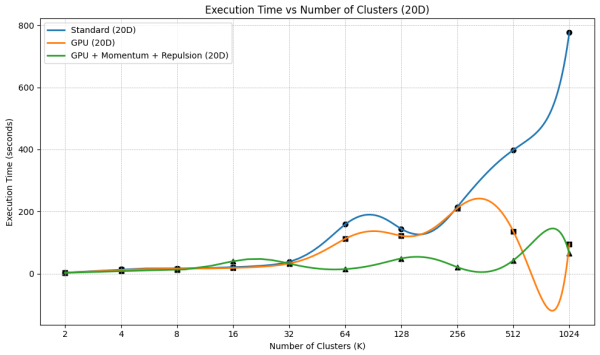


Figure 9: 20D Dataset (1M Points): Execution Time vs. K

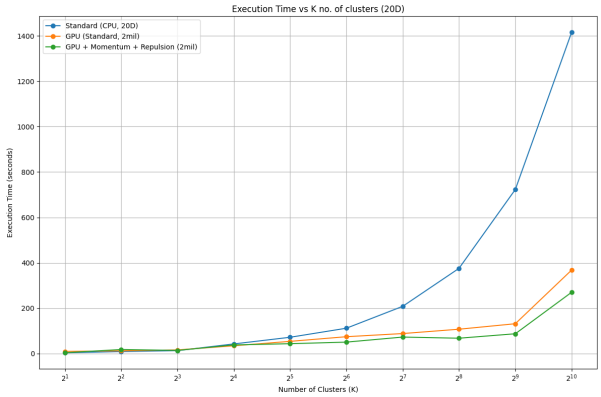


Figure 10: 20D Dataset (2M Points): Execution Time vs. K

5.5.2 Discussion and Comparison

Execution Time Analysis

- As dimensionality and data volume increase, **Sequential K-Means** becomes extremely slow reaching **776s (1M)** and **1415s (2M)** at $K = 1024$.
- **GPU K-Means** achieves massive speedups, but may require many iterations at large K (e.g., 4817 at $K = 256$ for 2M).
- **GPU with Momentum and Repulsion** is competitive across all K, particularly efficient in execution time despite higher iterations.

Convergence Behavior

- At $K \geq 128$, both the sequential and the basic GPU methods often reach the maximum iteration cap or suffer from centroid overlap.
- With momentum and repulsion, the algorithm takes more iterations, but maintains centroid separation and stable convergence.
- For example, at $K = 1024$:
  - GPU(M+R) (1M): 2047 iterations, **65.14s**
  - GPU(M+R) (2M): 4466 iterations, **270.03s**

Scaling Performance

- Execution time increases steadily from 1M to 2M data points across all methods, but GPU methods scale more efficiently.
- GPU with Momentum and Repulsion demonstrates better control over cluster instability at large K and high dimensionality.

5.5.3 Summary Table

K	**20D - 1M Points**			**20D - 2M Points**		
	Sequential	GPU	GPU + M+R	Sequential	GPU	GPU + M+R
2	3.47	3.20	2.56	3.26	9.31	3.58
4	14.12	12.30	9.11	8.92	12.15	17.77
8	17.29	15.95	12.12	12.87	15.98	13.87
16	20.97	18.54	40.23	42.32	34.25	37.99
32	38.26	32.61	32.42	71.73	53.70	43.45
64	158.72	112.31	14.76	111.70	74.22	50.36
128	144.30	121.98	49.05	208.13	88.13	72.61
256	214.97	210.51	21.01	375.49	107.29	67.57
512	398.21	136.46	42.94	722.98	131.49	87.21
1024	776.56	95.51	65.14	1415.60	367.94	270.04

Table 5: Execution Time (in seconds) for 20D Dataset (1M and 2M Points)

5.5.4 Observation

- GPU acceleration is essential for clustering large, high-dimensional datasets within practical time limits.
- The enhanced method (GPU with Momentum and Repulsion) consistently achieves a better trade-off between speed and convergence stability.
- It effectively handles high cluster counts and scales to millions of data points.
- Although it takes more iterations, it avoids collapse and provides better cluster separation, especially critical in 20D space.

5.6 cuML vs GPU K-Means with Momentum and Repulsion

To evaluate the performance and quality of our GPU-based K-Means implementation enhanced with momentum and repulsion (GPU+M+R), we compare it against **cuML K-Means** from NVIDIA's RAPIDS library. Experiments were conducted on four datasets:

- 10D – 1 million data points
- 10D – 2 million data points
- 20D – 1 million data points
- 20D – 2 million data points

The following metrics were recorded for each:

- Execution Time (in seconds)
- Convergence Iterations

5.6.1 Execution Time Comparison

K	10D – 1M		10D – 2M		20D – 1M		20D – 2M	
	GPU+M+R	cuML	GPU+M+R	cuML	GPU+M+R	cuML	GPU+M+R	cuML
2	1.13	0.20	1.66	0.83	2.56	0.40	3.58	0.26
4	5.01	0.36	10.29	0.42	9.11	0.44	17.77	0.83
8	39.60	0.69	11.61	1.39	12.12	1.23	13.87	2.12
16	57.86	0.63	7.03	1.48	40.23	0.78	37.99	1.52
32	6.90	0.56	6.60	1.05	32.42	0.84	43.45	1.56
64	5.97	0.59	17.94	1.10	14.76	0.71	50.36	1.18
128	12.71	0.76	15.13	1.34	49.05	0.89	72.61	1.65
256	3.90	1.19	6.29	2.25	21.01	1.42	67.57	2.65
512	3.57	2.17	10.66	3.85	42.94	2.54	87.21	4.70
1024	13.92	4.05	36.24	7.15	65.14	4.78	270.04	8.72

Table 6: Execution Time (in seconds): cuML vs GPU+Momentum+Repulsion

5.6.2 Performance Plots

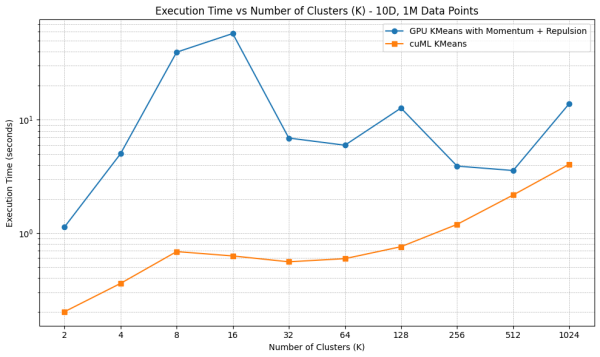


Figure 11: 10D – 1M Points

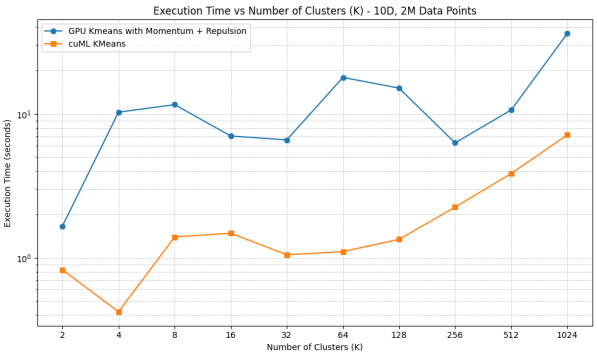


Figure 12: 10D – 2M Points

Figure 13: Execution Time for 10D Datasets (cuML vs GPU+M+R)

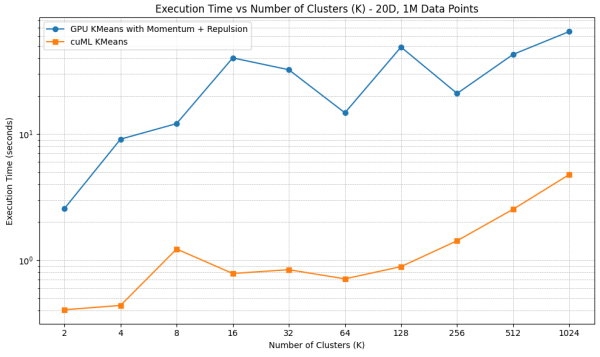


Figure 14: 20D – 1M Points

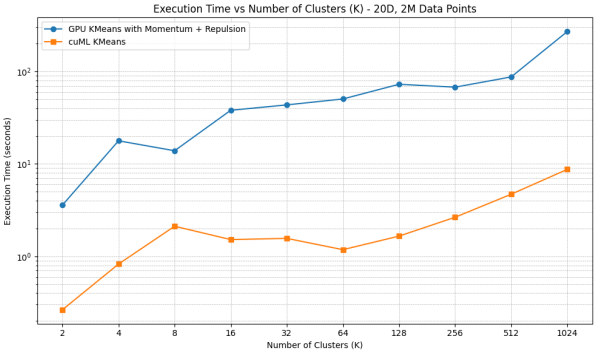


Figure 15: 20D – 2M Points

Figure 16: Execution Time for 20D Datasets (cuML vs GPU+M+R)

5.6.3 Technical Comparison:cuML V/s GPU(Momentum and Repulsion)

cuML K-Means (Optimized for Speed)

- **Kernel Fusion:** Combines clustering steps (assignment, update, reduction) into a single pass on the GPU.
- **Early Stopping:** Terminates when inertia or centroid shift falls below a set threshold.
- **cuDF and Memory Reuse:** Leverages RAPIDS cuDF for columnar GPU memory layout and pipelined memory management.
- **Single Precision Arithmetic:** Faster compute performance with adequate accuracy for typical clustering.

GPU with Momentum and Repulsion (Designed for Stability)

- **Momentum:** Smooths centroid updates by incorporating prior movement, helping avoid oscillation.
- **Repulsion:** Applies a pairwise repulsive force to prevent centroid overlap; incurs $O(K^2)$ cost per iteration.
- **Full Convergence Enforcement:** Runs for a fixed iteration count or until a tight convergence threshold — no early stopping.
- **Modular Kernels:** Assignment, update, momentum, and repulsion are separate GPU kernels, increasing sync overhead.

Design Trade-offs

- **cuML is optimized for speed**, best for production pipelines with large data streams.
- **GPU with Momentum and Repulsion is ideal for stability-focused research tasks**, where centroids must remain well-separated and iterations are controlled.

Aspect	cuML K-Means	GPU K-Means with Momentum + Repulsion
Goal	Production-ready speed	Research-grade stability and control
Optimization	Kernel fusion, early stopping, FP32	Repulsion force, momentum update, strict convergence
Centroid Update	Average of assigned points	Average + momentum + repulsion
Iteration Behavior	Stops early using heuristics	Runs to fixed or tight convergence
Best For	Large-scale streaming tasks	High-dimensional and sensitive clustering tasks

Table 7: Technical Comparison: cuML vs GPU+Momentum+Repulsion

5.6.4 Observation

- **cuML** is highly optimized for speed and is 5–10× faster than our method across all datasets.
- **GPU+Momentum+Repulsion** provides more stable and interpretable convergence behavior, especially important in high-dimensional or high-K scenarios.
- Use **cuML** for production systems where execution time is critical.
- Use **GPU+M+R** for experimental setups where clustering quality, separation, and controlled convergence are prioritized.

6 Conclusion

Comprehensive experiments across 4D, 10D, and 20D datasets with 1M and 2M data points demonstrate the significant advantages of GPU-accelerated clustering, particularly when enhanced with momentum and repulsion mechanisms.

- **GPU acceleration** consistently outperforms standard CPU-based K-Means across all configurations.
- **GPU K-Means with Momentum ($\alpha = 0.5$) and Repulsion ($\beta = 0.2$)** further boosts convergence stability and clustering quality.
- **Momentum + Repulsion** reduces centroid collapse and improves separation, which is essential at high K and high dimensions.
- GPU methods **scale better with data size and dimensionality**, while sequential implementations suffer from exponential time growth.

6.1 Execution Time Comparison at $K = 1024$

Dataset	CPU (s)	GPU (s)	GPU+M+R (s)	GPU Speedup	GPU+M+R Speedup	Best (at $K = 1024$)
4D - 1M	76.29	20.43	4.83	×3.73	×15.79	GPU+M+R
10D - 1M	583.84	27.17	13.92	×21.49	×41.96	GPU+M+R
10D - 2M	1139.40	98.75	36.24	×11.54	×31.42	GPU+M+R
20D - 1M	776.56	95.51	65.14	×8.13	×11.91	GPU+M+R
20D - 2M	1415.60	367.94	270.04	×3.85	×5.24	GPU+M+R

Table 8: Execution Time and Speedup at $K = 1024$ Across All Datasets

6.2 Relative Speedup Ratios

Cluster K	Std vs GPU	Std vs M+R	GPU vs M+R
2	0.79	0.84	1.34
4	0.78	0.75	1.21
8	1.02	1.81	1.44
16	0.96	0.87	0.78
32	0.98	2.78	3.42
64	1.58	6.76	4.33
128	1.33	5.37	4.17
256	2.97	19.81	6.93
512	7.01	30.22	4.54
1024	11.12	23.22	2.55
Average	**2.86**	**9.24**	**3.07**

Table 9: Relative Speedup: Standard vs. GPU vs. GPU+M+R (10D Dataset)

On average, GPU-accelerated K-Means provides a speedup of approximately $2.86\times$ over the standard sequential version. When enhanced with momentum and repulsion, the GPU implementation achieves a much greater speedup of around $9.24\times$, also improving over the plain GPU version by about $3.07\times$. The highest gains were observed at cluster sizes $K = 256$ and $K = 512$, highlighting the method's scalability. For small cluster counts ($K \leq 6$), the plain GPU implementation is generally sufficient. However, for larger K values ($K \geq 8$), the momentum and repulsion enhancements are highly recommended. Overall, the GPU-based K-Means with momentum and repulsion proves to be the most effective solution for clustering large, high-dimensional datasets.

7 References

[1] S. Daoudi, C. M. Anouar Zouaoui, M. Chikr El-Mezouar, and N. Taleb, "A Comparative Study of Parallel CPU/GPU Implementations of the K-Means Algorithm," *2019 International Conference on Electronics, Control, Optimization and Computer Science (ICECOCS)*, IEEE, pp. 1–6, 2019.

[2] J. Bhimani, M. Leeser, and N. Mi, "Accelerating K-Means Clustering with Parallel Implementations and GPU Computing," *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, 2015.

[3] L. AlGhamdi, M. Alkharraa, S. AlZahrani et al., "Improved Parallel k-means Clustering Algorithm," *2023 3rd International Conference on Computing and Information Technology (ICCIT)*, pp. 416–422, 2023.

[4] D. S. Bhupal Naik, S. Deva Kumar, and S. V. Ramakrishna, "Parallel Processing of Enhanced K-Means Using OpenMP," *2013 IEEE International Conference on Computational Intelligence and Computing Research*, pp. 1–5, 2013.

[5] F. Yuan, Z. H. Meng, and H. X. Zhang, "A new algorithm to get the initial centroid," *International Conference on Machine Cybernetics*, pp. 26–29, 2004.

[6] Fahim A. M. and Salem A. M. T., "An Efficient enhanced K-means clustering algorithm," *Journal of Zhejiang University*, 2006.

[7] E. Rasmussen and P. Willett, "Agglomerative clustering using processor," *Journal of Documentation*, vol. 45, no. 3, pp. 1313–1325, March 1989.

[8] C. Olson, "Parallel algorithms for hierarchical clustering," *Parallel Computing*, vol. 21, pp. 431–444, 1995.

[9] W. Zhao, H. Ma, and Q. He, "Parallel K-Means clustering algorithm based on MapReduce in Cloud Computing," *Cloud Computing*, pp. 674–679, 2009.

[10] S. Goil and Harasha Nagesh, "Efficient and Scalable Subspace Clustering for Very Large Data Sets," *Journal of Parallel and Distributed Computing*, vol. 12, no. 4, 1999.

[11] D. S. Bhupal Naik, S. Deva Kumar, and S. V. Ramakrishna, "Parallel Processing of Enhanced K-Means Using OpenMP," *Department of Computer Science and Engineering, Vignan University*, Andhra Pradesh, India.