

Master's Thesis

Scalable GPU Clustering with Split-
and-Pool-Driven Weighted K-means

Minseong Kim

The Graduate School

Sungkyunkwan University

Department of Immersive Media Engineering

M a s t e r ' s T h e s i s

Scalable GPU Clustering with Split-and-Pool-
Driven Weighted K-means

2 0 2 2

Minseong Kim

Master's Thesis

Scalable GPU Clustering with Split-
and-Pool-Driven Weighted K-means

Minseong Kim

The Graduate School

Sungkyunkwan University

Department of Immersive Media Engineering

Scalable GPU Clustering with Split- and-Pool-Driven Weighted K-means

Minseong Kim

A Master's Thesis Submitted to the Department of Immersive Media
Engineering and the Graduate School of Sungkyunkwan University in
partial fulfillment of the requirements for the degree of Master of
Engineering

October 2025

Supervised by

Sungkil Lee

Major Advisor

This certifies that the Master's Thesis
of Minseong Kim is approved.

Committee Chair : Jaepil Heo

Committee Member : Jaemin Jo

Major Advisor : Sungkil Lee

The Graduate School
Sungkyunkwan University

December 2025

Table of Contents

Chapter 1. Introduction	1
Chapter 2. Related Work.....	4
2.1 Algorithmic Accelerations of K-Means	4
2.2 GPU-based Accelerations of K-Means	6
Chapter 3. Preliminary: K-means and GPU Acceleration	8
3.1 Standard K-means Algorithm.....	8
3.2 CUDA Acceleration	10
Chapter 4. GPU-Accelerated Scalable K-means Clustering with Split- and-Pool Approach.....	13
4.1 LSH-based Initial Grouping.....	15
4.2 Split-and-pool for Weighted Centroids	17
4.2.1 Group splitting	18
4.2.2 Weighted Centroid Pooling	20
4.3 Weighted K-means Clustering	21
4.4 Merging Subgroup Points to Clusters.....	22

4.5 Global Iteration	2 3
4.6 Analysis of Time Complexities	2 4
Chapter 5. Experimental Results	2 7
5.1 Experimental Configuration.....	2 7
5.2 Datasets	2 7
5.3 Method	2 9
5.4 Results and Discussion	3 0
Chapter 6. Conclusion	3 9
References	4 0

List of Tables

Table 1.	28
Table 2.	31
Table 3.	32
Table 4.	35

List of Equations

Equation 1.	8
Equation 2.	9
Equation 3.	10
Equation 4.	16
Equation 5.	18
Equation 6.	20
Equation 7.	21
Equation 8.	21
Equation 9.	22
Equation 10.	22
Equation 11.	22
Equation 12.	24
Equation 13.	24
Equation 14.	25
Equation 15.	25
Equation 16.	25

List of Figures

Figure 1.	9
Figure 2.	11
Figure 3.	14
Figure 4.	17
Figure 5.	19
Figure 6.	30
Figure 7.	30
Figure 8.	34
Figure 9.	36

Abstract

Scalable GPU Clustering with Split-and-Pool-Driven Weighted K-means

This thesis presents a highly scalable and hardware-efficient K-means clustering algorithm that leverages CUDA to handle large-scale datasets efficiently. Standard K-means methods require iterating over all data points, leading to increasing inefficiency as data size, dimensionality, and memory footprint grow. To address this, our method introduces a split-and-pool approach to accelerate iterations. Given coarse clusters, they are split into smaller subgroups to aggregate individual points, which are then pooled as size-weighted centroids. These centroids are subsequently clustered into coarser groups and fed into the next global iteration. This pooling mechanism enables efficient batch-based distance computation and eliminates costly per-point data access. The procedure is further accelerated using a locality-sensitive hashing-driven initialization. Importantly, the iterative split-and-pool design aligns naturally with GPU architecture, making it highly suitable for parallel processing. Extensive experiments on real-world datasets demonstrate that

our method significantly outperforms standard CUDA-based implementations of K-means, while preserving comparable clustering quality.

Keywords : K-means, Split-and-Pool, GPU, CUDA

Chapter 1. Introduction

The exponential growth of data in modern applications has created an urgent need for scalable and efficient unsupervised learning algorithms [1], [2]. Clustering, as a fundamental tool for discovering structure in unlabeled data, plays a vital role in domains such as natural language processing, computer vision, bioinformatics, and recommender systems. Among the various clustering algorithms, K-means remains one of the most widely adopted methods due to its simplicity, interpretability, and practical performance [3]. However, as datasets grow in size and dimension, its computational cost (driven by repeated distance computations and global centroid updates) becomes a critical bottleneck for large-scale deployment.

To address these limitations, prior work has introduced complementary acceleration strategies. Improved initializers such as k-means++ [4] and scalable seeding schemes [5] enhance convergence speed but do not reduce the per-iteration complexity. Distance-bounding techniques such as Elkan's algorithm [6] can prune redundant computations, but their benefits often diminish in high-dimensional settings. GPU-accelerated variants [7–9]

leverage parallelism for faster distance evaluation but still rely on exhaustive all-point assignments and full centroid recomputation at each iteration. As a result, the efficiency of many existing designs remains limited because the dominant cost of global recomputation persists.

These challenges motivate the development of a redesigned clustering pipeline that moves beyond direct parallelization. The key inefficiency in standard K-means is that every iteration revisits the entire dataset, even though many data points exhibit local homogeneity and could be represented more compactly. This insight underlies a split-and-pool strategy: partition the dataset into locality-aware groups, summarize each group by a centroid and weight, and refine clustering decisions on these weighted summaries rather than on raw points. Such an approach significantly reduces redundancy while preserving clustering fidelity through decision propagation back to individual samples.

In this thesis, we present a GPU-accelerated scalable weighted K-means algorithm that employs a split-and-pool clustering strategy. The method begins with a coarse grouping of input data projected using locality-sensitive hashing, forming the initial input to the split-and-pool pipeline. Each group is then subdivided into finer subgroups, whose centroids are pooled with weights proportional to subgroup sizes. Weighted K-means is applied to these pooled

representatives to iteratively refine the cluster structure. The resulting centroids are fed into the next iteration of the pipeline. By operating on a significantly reduced set of weighted centroids instead of the full dataset, the method substantially lowers the per-iteration cost while maintaining clustering accuracy. The pipeline is implemented entirely in CUDA, with efficient kernels for projection and weighted updates, supporting practical scalability to large-scale datasets.

What follows summarize our key contributions:

- Introduction of weighted K-means that reorganizes the clustering pipeline based on split-and-pool summarization;
- A size-weighted centroid pooling that compacts homogeneous groups into centroids for batch-clustering;
- A CUDA-based efficient parallel implementation, incorporating LSH-based initialization and weighted updates.

The remainder of this paper is organized as follows. **Chapter 2** reviews GPU acceleration for K-means and summarization strategies. **Chapter 3** introduces preliminaries and overview of K-means. **Chapter 4** details our algorithm and framework, including LSH-based coarse grouping and weighted summarization for representative centroids. **Chapter 5** presents experimental results and performance analysis. **Chapter 6** concludes the paper with discussions.

Chapter 2. Related Work

2.1 Algorithmic Accelerations of K-Means

Algorithmic approaches have accelerated K-means by improving initialization and reducing redundant distance evaluations, while still preserving the global clustering objective. Seeding methods help reduce iteration counts and avoid poor local minima. K-means++ selects well-spread centers using distance-aware sampling [4], Scalable K-means++ extends this idea to large datasets by distributing the selection process across multiple rounds [5], and Global K-means incrementally adds one center at a time to optimize placement [10].

Pruning-based techniques address inefficiencies in redundant distance computations while maintaining the semantics of Lloyd's algorithm. Elkan's method reduces unnecessary evaluations by maintaining lower and upper bounds for each data point using triangle inequalities [6]. Yinyang K-means improves upon this by grouping centroids to apply hierarchical pruning—first at the group level, then locally [11]. Hamerly's algorithm simplifies the pruning strategy by tracking a single bound per point, lowering memory usage while achieving significant speedups [12].

Another line of work tackles the challenge of high-dimensional indexing. Hierarchical K-means trees organize data to allow logarithmic-depth search, enabling the construction of large-scale visual vocabularies [13]. FLANN auto-tunes between K-means trees and randomized k -d forests to handle high-dimensional features efficiently [14, 15]. Additionally, p -stable locality-sensitive hashing (LSH) supports sublinear-time approximate nearest neighbor search in Euclidean space [16]. For non-numeric data, LSH-based K-representatives combine hashed candidate sets with distributional summaries to reduce reassignment cost while maintaining quality [17].

To scale K-means clustering in terms of both computation and memory, several frameworks have been introduced. Mini-batch K-means updates centroids using small, random subsets of the data, optionally with weighting, to increase throughput at the cost of some per-iteration precision [18]. Coreset-based methods construct weighted summaries that offer provable approximation guarantees, though they are often computed only once and at coarse resolution [19]. Other memory-efficient approaches improve data locality by reorganizing assignments and updates, which is especially beneficial for datasets that exceed available memory and must be processed in streaming or out-of-core settings [20].

2.2 GPU-based Accelerations of K-Means

GPU acceleration has made K-means practical on modern hardware by aligning computation kernels and memory traffic with Single Instruction Multiple Thread (SIMT) execution and by enabling efficient out-of-core pipeline scheduling. Early CUDA implementations introduced reusable parallelization patterns, including coalesced global memory access, shared-memory optimization for centroids and data points, and optimized host-device coordination. These techniques showed significant performance gains over CPU-based implementations [7–9].

Over time, GPU acceleration has become a key strategy for scaling K-means clustering. Initial studies demonstrated the feasibility of CUDA-based designs [21, 22], which were later extended with practical GPU-parallel frameworks [23] and kernel-level optimizations that enhanced memory locality and thread efficiency [24]. More recent approaches introduced warp-centric execution models to improve SIMT utilization [25] and hybrid execution pipelines to manage clustering on very large datasets [26]. Kernelized variants of K-means have leveraged sparse-dense linear algebra operations, enabling the use of highly optimized GPU libraries [27]. In parallel, pruning-based techniques like Yinyang K-means have been adapted to GPU environments, showing that bound-based acceleration methods can be effectively mapped onto massively parallel architectures [28].

GPU-based frameworks have further advanced the scalability of K-means by removing computational bottlenecks and supporting both single- and multi-GPU execution. For instance, the Facebook AI Similarity Search (FAISS) library fuses distance computation with top-K selection in the assignment phase, eliminating critical bottlenecks and enabling clustering across multiple GPUs [29]. Similarly, the RAPIDS cuML library provides optimized K-means implementations with support for GPU-accelerated primitives across varying hardware setups [30].

While most prior work focuses on either reducing per-iteration cost through GPU parallelism or using static summaries to simplify computation, these methods often still require reprocessing the entire dataset in each round. In contrast, the split-and-pool strategy introduced here builds weighted summaries by grouping points into locality-aware clusters and applying iterative refinement through weighted K-means. This approach significantly reduces redundancy, enhances scalability, and yields more stable clustering results in large-scale GPU environments.

Chapter 3. Preliminary: K-means and GPU Acceleration

3.1 Standard K-means Algorithm

K-means is an iterative partitional clustering algorithm that alternates between assigning data points to clusters and updating cluster representatives. Consider a dataset $X = [x_1, \dots, x_N]^T \in \mathbb{R}^{N \times D}$ where each $x_i \in \mathbb{R}^D$ denotes a data point in a D -dimensional feature space. The objective is to partition X into K clusters, with $K \in \mathbb{N}_+$ and $K < N$ such that each cluster is represented by a centroid that summarizes the data points assigned to it. Euclidean distance is commonly used to measure the similarity between data points and centroids. The algorithm alternates between two phases. In the labeling phase, each point x_i is assigned to the cluster whose centroid is closest in Euclidean space, inducing a partition $\{C_j \mid j \in \{1, \dots, K\}\}$ of the dataset:

$$C_j = \{x_i \mid j = \operatorname{argmin}_k \|x_i - c_k\|_2\} \quad (1)$$

Where c_j denotes the centroid of cluster C_j .

Algorithm 1 K-means Clustering (Lloyd's Algorithm)

- 1: $X = \{x_i\}_{i=1}^N \in \mathbb{R}^{N \times D}$, K , $y_i \in \{1, \dots, K\}$, ϵ , P , $c_j^{(t)}$ ▷ N points of D dims., number of clusters, labels, tolerance, patience, centroid at iter t
 - 2: $\{c_j^{(0)}\}_{j=1}^K \subset X$; $t \leftarrow 0$, $q \leftarrow 0$, $E \leftarrow +\infty$ ▷ initialize centroids, counters, and initial SSE
 - 3: **repeat**
 - 4: $y_i \leftarrow \operatorname{argmin}_j \|x_i - c_j^{(t)}\|_2^2$ ($\forall i$) ▷ assign each point to nearest centroid (labeling phase)
 - 5: $c_j^{(t+1)} \leftarrow \frac{1}{|\{i: y_i=j\}|} \sum_{i: y_i=j} x_i$ ($\forall j$) ▷ update centroids using mean of assigned points
 - 6: $E' \leftarrow \sum_{i=1}^N \|x_i - c_{y_i}^{(t+1)}\|_2^2$; $\Delta \leftarrow \frac{E-E'}{\max(E, 10^{-12})}$ ▷ compute new SSE and relative improvement
 - 7: $q \leftarrow (\Delta \geq 0 \wedge \Delta < \epsilon) ? q+1 : 0$; $E \leftarrow E'$; $t \leftarrow t+1$ ▷ increase patience if improvement is below tolerance
 - 8: **until** $q \geq P$ ▷ stop when convergence is sustained for P iterations
-

Figure 1. The pseudocode for K-means Clustering (Lloyd's Algorithm)

In the reduction phase, each centroid c_j is updated as the arithmetic mean of the data points assigned to its cluster:

$$c_j = \left(\frac{1}{|C_j|} \right) \sum_{x_i \in C_j} x_i \quad (2)$$

Where $|\cdot|$ denotes the cardinality operator.

These two steps are repeated until convergence, which is typically determined by stabilized cluster assignments or minimal change in the clustering objective. From an optimization standpoint, K-means aims to find a set of centroids $\mathcal{C} = \{c_1, \dots, c_K\}$ that minimize the within-cluster sum of squared distances:

$$\phi(X; C) = \sum_{i=1}^N \min_{c_j \in C} \|x_i - c_j\|_2^2 \quad (3)$$

The standard approach to minimizing the Eq. (3) is Lloyd's algorithm in Figure 1, which proceeds as follows:

1. Initialization: Select K initial centroids, typically at random.
2. Labeling: Assign each data point x_i to the cluster whose centroid is closest in Euclidean distance.
3. Reduction: Recompute each centroid c_j as the mean of the points currently assigned to cluster C_j .
4. Convergence: Repeat steps 2 and 3 until cluster assignments no longer change or the objective function converges.

This alternating process monotonically decreases the objective function and is guaranteed to converge to a local minimum in a finite number of iterations.

3.2 CUDA Acceleration

In recent years, GPUs have evolved from fixed-function graphics accelerators into highly programmable parallel computing devices. NVIDIA CUDA extends the C language with abstractions such as kernels, threads, and hierarchical memory spaces, allowing developers to explicitly express fine-grained parallelism. The CUDA execution model treats the GPU as a

coprocessor that launches thousands of lightweight threads organized into blocks and grids, following the SIMT paradigm.

Algorithm 2 CUDA-based Standard K-means

```

1:  $X = \{x_i\}_{i=1}^N \in \mathbb{R}^{N \times D}$ ,  $K$ ,  $T$ ,  $\epsilon$ ,  $P$ ,  $c_j^{(t)}$   $\triangleright n$  samples of  $d$  dims., #clusters, max iters, tolerance,
   patience, centroid at iter  $t$ 
2:  $t \leftarrow 0$ ,  $q \leftarrow 0$ ,  $E \leftarrow +\infty$   $\triangleright$  initialization on host
3: move  $X$  and working buffers to GPU; sample  $\{c_j^{(0)}\}_{j=1}^K$  on GPU
4: repeat
5:    $y_i \leftarrow \operatorname{argmin}_j \|x_i - c_j^{(t)}\|_2^2 (\forall i)$   $\triangleright$  label each point in parallel (GPU)
6:    $E' \leftarrow \sum_{i=1}^N \|x_i - c_{y_i}^{(t)}\|_2^2$ ;  $\Delta \leftarrow \frac{E - E'}{\max(E, 10^{-12})}$   $\triangleright$  compute SSE in parallel (GPU)
7:    $q \leftarrow (\Delta \geq 0 \wedge \Delta < \epsilon) ? q + 1 : 0$ ;  $E \leftarrow E'$ 
8:   if  $q \geq P$  then break  $\triangleright$  early stop if converged
9:    $c_j^{(t+1)} \leftarrow \frac{1}{|\{i: y_i = j\}|} \sum_{i: y_i = j} x_i (\forall j)$   $\triangleright$  centroid update in parallel (GPU)
10:   $t \leftarrow t + 1$ 
11: until  $t \geq T$ 

```

Figure 2. The pseudocode for CUDA-based Standard K-means Algorithm

A CUDA kernel defines a function that is executed concurrently by multiple threads. Upon launch, the kernel is instantiated across a grid of threads, with each thread responsible for a distinct portion of the input data. Threads within the same block can communicate via low-latency shared memory and synchronize through barriers, while all threads can access the larger but slower global memory. This architecture is particularly well-suited to data-parallel workloads such as K-means clustering, where each thread can independently compute point-to-centroid distances or assignment decisions.

The standard K-means algorithm maps naturally to the GPU execution model due to its inherent data-parallel structure. In each iteration, the GPU carries out both the labeling phase, which assigns each data point to its nearest centroid, and the reduction phase, which recomputes centroids based on the current assignments. Executing these phases entirely on the device avoids redundant host-device transfers and stalls, thereby maximizing GPU utilization.

As summarized in Algorithm 2 in Figure 4, a typical CUDA-based implementation of K-means clustering [21] begins by allocating the dataset and the requisite working buffers in GPU memory, followed by the initialization of cluster centers. Each iteration is composed of three stages: labeling, error evaluation, and centroid update. In the labeling stage, all data points are processed in parallel to determine their nearest centroids, and the sum of squared errors (SSE) is accumulated. A labeling kernel performs nearest-centroid assignments, while a reduction kernel aggregates partial sums and population counts per cluster using atomic operations to preserve correctness. A subsequent centroid-update kernel computes new cluster means from the aggregated statistics. These kernels execute iteratively until a convergence condition is met or a maximum iteration budget is reached. This modular kernel decomposition enables CUDA to exploit thread-level parallelism effectively while maintaining proper synchronization and balanced memory access.

Chapter 4. GPU–Accelerated Scalable K–means

Clustering with Split–and–Pool Approach

This page is typeset with the above font sizes, so you can write the text by using the typeset form of this page. The key idea of our method is to reorganize the clustering pipeline using a split–and–pool approach, designed to balance scalability and accuracy for large–scale datasets. A brief overview of the method is shown in Figure 4. Traditional K–means becomes inefficient on large datasets due to high memory usage and redundant distance evaluations, especially from repeatedly processing locally similar points. To address this, our main iteration operates on subgroups instead of individual data points, significantly improving iteration speed. Initially, data points are grouped based on their local proximity using locality–sensitive hashing (LSH). These coarse groups are then split into smaller subgroups and pooled into size–weighted centroids. A weighted K–means step is applied to these pooled centroids to refine the cluster structure. This ensures that dense regions have a stronger influence on updates, while small or noisy groups do not disproportionately affect the outcome. By shifting computation from all data points to a smaller set of representatives, our batch–clustering design reduces redundancy and

Algorithm 3 Scalable GPU Clustering with Split-and-Pool-Driven Weighted K-means

```
1:  $X = \{x_i\}_{i=1}^N \in \mathbb{R}^{N \times D}$ ,  $K$ ,  $R$ ,  $T$ ,  $\epsilon$ ,  $P$   $\triangleright N$  samples of  $D$  dims., #clusters, outer rounds, max iters,
   tolerance, patience
2: partition  $X$  into  $B=\tau_0 K$  groups  $\{G_i\}_{i=1}^B$   $\triangleright$  LSH-based bucketing
3: for  $r = 1$  to  $R$  do
4:    $U \leftarrow \max(2, \lceil \tau K / |\{G_i\}| \rceil)$   $\triangleright$  adaptive split factor
5:    $S \leftarrow \{g_{i,u}\}$  by splitting each  $G_i$  into  $U$  subgroups
6:    $p(g) = \frac{1}{w(g)} \sum_{i \in g} x_i$ ;  $w(g) = |g|$   $\triangleright$  for each  $g \in S$ 
7:    $\{c_j^{(0)}\}_{j=1}^K$   $\triangleright$  random if  $r=1$ , else warm-start from previous round
8:    $t \leftarrow 0$ ,  $q \leftarrow 0$ ,  $E \leftarrow +\infty$ 
9:   repeat
10:     $y(g) \leftarrow \operatorname{argmin}_j \|p(g) - c_j^{(t)}\|_2^2$   $\triangleright$  assignment on representatives
11:     $c_j^{(t+1)} \leftarrow \frac{\sum_{g: y(g)=j} w(g) p(g)}{\sum_{g: y(g)=j} w(g)}$   $\triangleright$  weighted update
12:     $E' \leftarrow \sum_{g \in S} \|p(g) - c_{y(g)}^{(t+1)}\|_2^2$ ;  $\Delta \leftarrow \frac{E - E'}{\max(E, 10^{-12})}$ 
13:     $q \leftarrow (0 \leq \Delta < \epsilon) ? q+1 : 0$ ;  $E \leftarrow E'$ ;  $t \leftarrow t+1$ 
14:   until  $t \geq T$  or  $q \geq P$ 
15:    $G'_j \leftarrow \bigcup_{g: y(g)=j} \{\mathbf{x} \in g\}$  ( $j = 1, \dots, K$ )  $\triangleright$  merge subgroups back to clusters
16:    $G \leftarrow \{G'_j\}_{j=1}^K$ ;  $B \leftarrow K$   $\triangleright$  update coarse groups for next round
```

Figure 3 The pseudocode for Split-and-Pool-Driven Weighted K-means

improves per-iteration efficiency. Moreover, this approach aligns well with GPU parallelism, making it well-suited for scalable deployment. A visual overview of the pipeline is shown in Figure 5, and the detailed steps are summarized in Figure 3 in Algorithm 3.

4.1 LSH-based Initial Grouping

Locality-Sensitive Hashing (LSH) is a probabilistic technique that projects high-dimensional data into a lower-dimensional hash space while approximately preserving the similarity between data points [16]. Its core idea is to construct hash functions in such a way that similar points are much more likely to collide than dissimilar ones, making it effective for approximate nearest neighbor search in high-dimensional settings. This property has been widely leveraged in large-scale clustering and retrieval applications where exact distance computations are too costly to perform at scale.

In our work, we leverage locality-sensitive hashing (LSH) to efficiently perform initial coarse grouping on the GPU. Each input data point $x \in \mathbb{R}^D$ is projected onto a basis vector $r \in \mathbb{R}^D$, which is randomly generated. Unlike conventional multi-probe or multi-level LSH methods that require multiple random vectors and hash tables, our method uses a single, globally shared basis vector that effectively preserves locality for coarse clustering. Each component r_j of the vector r corresponds to the j -th feature dimension and is independently sampled from a standard normal distribution $\mathcal{N}(0,1)$. This corresponds to the 2-stable (Gaussian) case in the general p -stable LSH framework [16], and it preserves the Euclidean (L_2) distance structure in the projected space. After projection, the values are normalized and quantized into discrete buckets to form coarse hash groups. Given B buckets, the hash

function is defined as:

$$h_r(x) = \lfloor \text{normalize}(r \cdot x) \cdot B \rfloor. \quad (4)$$

The points grouped into each bucket are likely to be similar in the original high-dimensional input space. These buckets serve as initial coarse clusters; however, their granularity is much lower than that of more refined clustering. Therefore, to ensure that the number of buckets exceeds the number of final clusters, we scale the number of buckets B using a factor τ_0 , resulting in $B = \tau_0 K$ groups, as shown in Algorithm 3 in Figure 3. In our experiments, setting $\tau_0 = 8$ offers the best balance between computational efficiency and clustering quality. This coarse but effective partitioning enables the split-and-pool pipeline to begin with groups that are already locally homogeneous. The resulting B coarse groups are then used as input to the split-and-pool process, as illustrated in Figure 5.

To further improve performance, we implement the LSH projection and assignment as a CUDA kernel. The random basis vector is generated on the host and transferred to the device, where each thread processes a data point by computing its dot product with the basis vector, normalizing the result, and assigning the point to a corresponding bucket. By exploiting CUDA’s parallel execution model, thousands of data points can be grouped concurrently, resulting in a substantial reduction in runtime compared to CPU-based implementations.

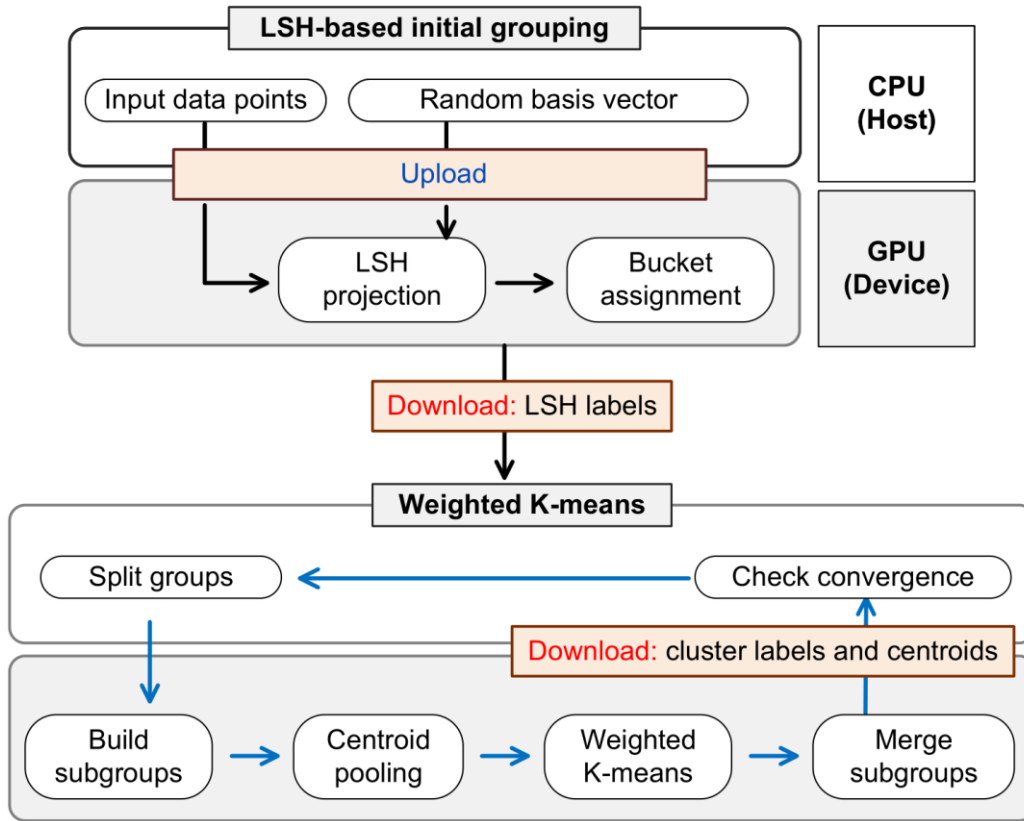


Figure 4 Overview of our GPU-based weighted K-means algorithm with split-and-pool approach.

4.2 Split-and-pool for Weighted Centroids

The split-and-pool procedure begins with a set of groups $\{G_i\}$. In the first iteration, these groups correspond to the B buckets generated from the LSH-based initialization. In subsequent global iterations, the intermediate clustering results from the previous round are used as the new coarse groups. At this stage, the number of groups $|\{G_i\}|$ becomes equal to the number of clusters K .

4.2.1 Group splitting

Our batch-clustering method operates on subgroups rather than on individual data points. The total number of subgroups in the dataset is defined τK , where τ is a density factor that controls the granularity of the batch clustering process. A larger τ creates more subgroups, which results in finer granularity but increases the size of the centroid pool and runtime. In contrast, a smaller τ leads to coarser partitions, improving efficiency but potentially under-representing local structures. In our experiments, setting $\tau = 16$ achieved the best balance between efficiency and clustering quality.

In the group splitting step, each input group G_i is divided into a set of smaller subgroups $\{g_{\{i,u\}}\}_{u=1}^U$. The per-group adaptive split factor U determines the number of subgroups per coarse group. Since the total number of subgroups is τK , the value of U is calculated as:

$$U = \max\left(2, \lceil \frac{(\tau K)}{|G_i|} \rceil\right) \quad (5)$$

Each subgroup is then summarized as a weighted centroid, which is passed into the next clustering step.

To ensure that batch-clustering continues to improve over global iterations, data points should be randomly permuted within each group before splitting.

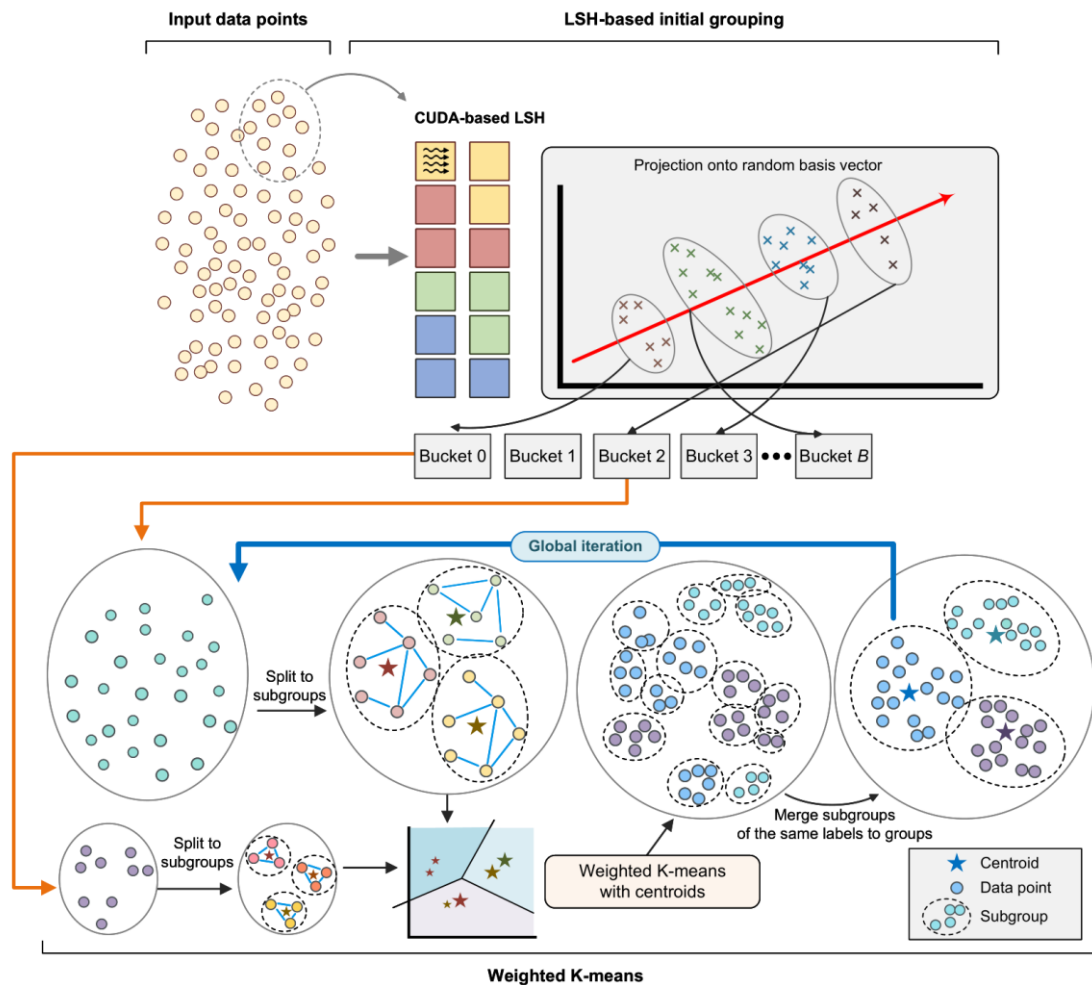


Figure 5 Illustration of the data flow in split-and-pool-based weighted K-means clustering. In the initialization stage, input data points are projected into a low-dimensional hash space using locality-sensitive hashing (LSH) on the GPU, assigning nearby points to the same hash bucket. Each coarse group is then represented by a weighted centroid, summarizing both the spatial position and the number of points it contains. These centroids (shown as stars) are processed through weighted K-means to refine cluster assignments across subgroups. The procedure is globally iterated, progressively improving clustering quality while maintaining efficiency on large-scale, high-dimensional datasets.

Without this randomization, similar data points may remain in the same subgroup, limiting diversity. For this reason, after each global clustering round,

the merge step randomly shuffles the data points. The initial LSH-based groups are not permuted, as they are already formed with high randomness. As a result, splitting is implemented by partitioning the contiguous array representing G_i into equal-sized segments.

4.2.2 Weighted Centroid Pooling

Let the complete set of all split subgroups be defined as $\mathcal{S} = \{g_{i,u} \mid i = 1, \dots, B; u = 1, \dots, U\}$. Given \mathcal{S} , we pool each subgroup $g \in \mathcal{S}$ into its centroid $p(g)$ and weight $w(g)$ as follows:

$$p(g) = \left(\frac{1}{w(g)}\right) \sum_{x_i \in g} x_i \quad (6)$$

Here, $w(g) = |g|$ denotes the number of data points in subgroup g . This aggregation step compresses all data points in g into a single weighted representative $(p(g), w(g))$, forming a compact set $\mathcal{P} = \{(p(g), w(g)) \mid g \in \mathcal{S}\}$. This set \mathcal{P} is then used as the input to the next stage of the weighted K-means algorithm.

Since subgroup sizes can vary greatly, applying $w(g)$ as a weight prevents small subgroups from disproportionately influencing the global optimization. For instance, if $|g_1| = 100$ and $|g_2| = 5$, treating both equally would cause the small

group g_2 to affect the centroid update as much as g_1 , leading to instability. The weight $w(g)$ ensures that each subgroup contributes in proportion to its size during refinement.

4.3 Weighted K-means Clustering

We perform a weighted K-means over the entire representative set P , allowing different subgroups to interact and be reassigned to cluster labels. Starting from the initial K cluster centers $\{c_j^{t=0}\}_{j=1}^K$ the algorithm alternates between assignment and update steps at each iteration t , as follows:

$$y(g) = \operatorname{argmin}_j \|p(g) - c_j^t\|_2^2 \quad (7)$$

$$c_j^{t+1} = \frac{\sum_{g: y(g)=j} w(g) \cdot p(g)}{\sum_{g: y(g)=j} w(g)} \quad (8)$$

Here, $y(g)$ denotes the cluster index (1 to K) assigned to subgroup g , and each cluster center $c_j^{(t)}$ is updated to the weighted mean of its assigned representatives.

This clustering procedure minimizes the sum of squared errors (SSE) at each iteration:

$$E^{(t)} = \sum_{g \in S} \left\| p(g) - c_{y(g)}^{(t)} \right\|_2^2 \quad (9)$$

Convergence is determined by the relative improvement:

$$\Delta = \frac{(E^{(t-1)} - E^{(t)})}{\max(E^{(t-1)}, 10^{-12})} \quad (10)$$

The iteration stops when $\Delta < \epsilon$ for P consecutive steps (patience) or when the maximum number of iterations T is reached.

4.4 Merging Subgroup Points to Clusters

After convergence, subgroups assigned to the same cluster are merged, and their original data points are extracted and combined as follows:

$$G'_j = \bigcup_{g: y(g)=j} \{x \in g\}, \quad j = 1, \dots, K \quad (11)$$

Here, x represents the individual data points that belong to subgroup g , and all subgroups with the same cluster label j are unified into a single group G'_j .

In this way, each data point is batch-assigned to the label of the subgroup it belongs to, allowing efficient consolidation of subgroup contents into coherent

clusters.

The data points within each merged cluster are randomly permuted. Specifically, after clusters are merged, the indices of data points belonging to each cluster are uniformly shuffled using a random permutation before forming mini-batches. This shuffling is applied independently at each global iteration, ensuring that different subsets of points are grouped into batches over time. Such a design prevents fixed batch composition, promotes diverse sample exposure within each batch, and enables the batch-clustering process to progressively approximate full-data K -means behavior across iterations. The resulting merged clusters $\{G'_j\}$ for $j = 1$ to K , serve as the new coarse groups that are used as input to the next global iteration.

4.5 Global Iteration

To progressively improve our batch-clustering method, we apply a global iteration over the full pipeline consisting of split-and-pool, clustering, and merging steps. This global iteration is repeated for a fixed number of times R . Since each run of the pipeline already converges locally, we do not check an additional global stopping condition. In practice, increasing R beyond a certain point does not necessarily improve clustering results. This observation is further discussed in the Results chapter.

4.6 Analysis of Time Complexities

The primary computational bottleneck of conventional K-means lies in the assignment step, where each data point must be compared with all cluster centroids in every iteration. Given N data points of dimension D , K clusters, and T iterations, the time complexity of standard K-means is:

$$O(N \cdot K \cdot D \cdot T) \tag{12}$$

While GPU parallelism can accelerate the actual computation, the asymptotic complexity remains the same because each of the N points still requires comparison with all K centroids during every iteration.

In contrast, our method begins with an LSH-based coarse grouping step that projects each point onto a random vector and assigns it to a bucket. This process has a linear time complexity of:

$$O(N \cdot D) \tag{13}$$

and can be efficiently parallelized on the GPU. Crucially, it avoids computing full pairwise distances between data points and centroids at this stage.

After initial grouping, each coarse group is split into U subgroups. Let M denote the number of resulting subgroup centroids, where typically $M \ll N$.

The weighted K-means algorithm is then applied to this compact set, with a reduced complexity of:

$$O(M \cdot K \cdot D \cdot T) \tag{14}$$

Because M is significantly smaller than N , this step drastically reduces computational cost.

The entire process is globally iterated R times. Therefore, the total time complexity across all global iterations is:

$$\sum_{r=1}^R O(M \cdot K \cdot D \cdot T), \text{ with } M \ll N \tag{15}$$

Since each iteration operates on a much smaller set of centroids, the total computational cost remains far lower than that of traditional K-means.

In summary, our approach reduces dependence on N by replacing point-to-centroid comparisons with coarse grouping followed by centroid-level clustering. Compared to the conventional complexity $O(N \cdot K \cdot D \cdot T)$ our method achieves a theoretical complexity of:

$$O(N \cdot D) + \sum_{r=1}^R O(M \cdot K \cdot D \cdot T), \text{ with } M \ll N \tag{16}$$

which explains its efficiency advantages, particularly on large-scale and high-dimensional datasets.

Chapter 5. Experimental Results

5.1 Experimental Configuration

All experiments were conducted on a high-performance computing server equipped with an Intel Xeon Gold 5220R CPU (2.20 GHz) and 128 GB of DDR4 RAM. The system included an NVIDIA GeForce RTX 3090 GPU featuring 10,496 CUDA cores, 24 GB of GDDR6X memory, and a memory bandwidth of up to 936 GB/s. The GPU supports up to 35.6 TFLOPS of single-precision performance and has a maximum power consumption of 350 W. The software environment was based on Ubuntu 20.04 LTS, with NVIDIA driver version 570.181 and the CUDA Toolkit version 12.8. All GPU-accelerated computations were carried out using the CUDA 12.8 platform.

5.2 Datasets

To evaluate our algorithm on real-world data, we used a diverse set of widely adopted benchmark datasets spanning multiple domains to ensure reliable and generalizable clustering performance (see Table [1] for a summary).

Table 1 Summary of datasets used in our experiments.

Dataset	Size (N)	Dim. (D)	#Clusters (K)
GloVe.twitter.27B.25d	1,193,514	25	32–1024
GloVe.twitter.27B.50d	1,193,514	50	32–1024
GloVe.twitter.27B.100d	1,193,514	100	32–1024
GloVe.twitter.27B.200d	1,193,514	200	32–1024
MNIST	70,000	784	32–1024
SUSY	5,000,000	18	32–1024
Word2Vec (GoogleNews)	3,000,000	300	32–1024

Specifically, we included the Twitter GloVe dataset [31], which provides pre-trained word embeddings in multiple dimensions (25, 50, 100, and 200) and is commonly used in natural language processing tasks. We also used the MNIST dataset [32], consisting of 70,000 grayscale images of handwritten digits, where each image is represented as a 784-dimensional vector serving as a standard benchmark for high-dimensional image clustering. For structured numerical data, we employed the SUSY dataset [33] from the UCI repository, which contains 5 million examples with 18 real-valued features derived from particle physics simulations. Lastly, we evaluated clustering on semantic vector representations using the Word2Vec embedding dataset [34], which captures linguistic similarity from large-scale text corpora.

5.3 Method

In our pipeline, the coarse initial grouping step is executed only once, and the results from each global iteration are used as input for the next. This global iteration is repeated a fixed number of times R . Within each global iteration, the clustering process is allowed to run for up to 1,000 local iterations. Convergence is considered reached when the relative decrease in the sum of squared errors (SSE) falls below 10^{-4} for five consecutive iterations.

To evaluate performance, we measured execution time using four benchmark datasets: Twitter GloVe (with embedding dimensions of 25, 50, 100, and 200), MNIST, SUSY, and Word2Vec. For the scalability analysis, experiments were conducted across varying numbers of clusters, with $K \in \{32, 64, \dots, 1024\}$. The number of global iterations R and the grouping parameters (τ_0, τ) were chosen through preliminary experiments to determine optimal performance settings. For comparative evaluation, we included three representative GPU-based clustering methods: the standard GPU implementation of Lloyd’s algorithm [3], a GPU version of Yinyang K-means [11], and the Simplified Yinyang GPU method (SIMPLEGPU) presented by Taylor and Gowanlock [28]. SIMPLEGPU implements a simplified version of Yinyang K-means, applying two of the original three filtering steps. All methods were executed on the same hardware platform under identical convergence conditions.

Unlike existing methods, our approach uses weighted centroids that operate at a different level of data granularity. To properly assess clustering quality, we report results using normalized SSE (nSSE), a commonly used metric in K-means clustering

to evaluate intra-cluster compactness [35].

5.4 Results and Discussion

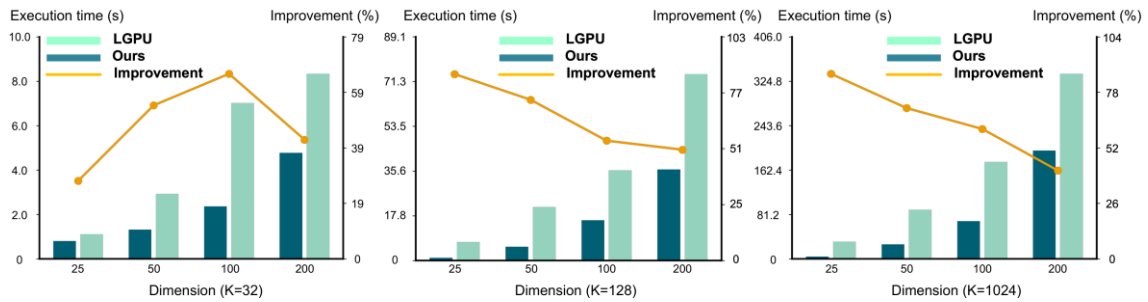


Figure 6 Bar chart results on the Twitter GloVe dataset measured at four embedding dimensions (25, 50, 100, 200) and three cluster sizes ($K=32$, $K=128$, and $K=1024$). The bars show performance of ours and the baseline (LGPU), and the yellow lines are speedups of ours against the baseline.

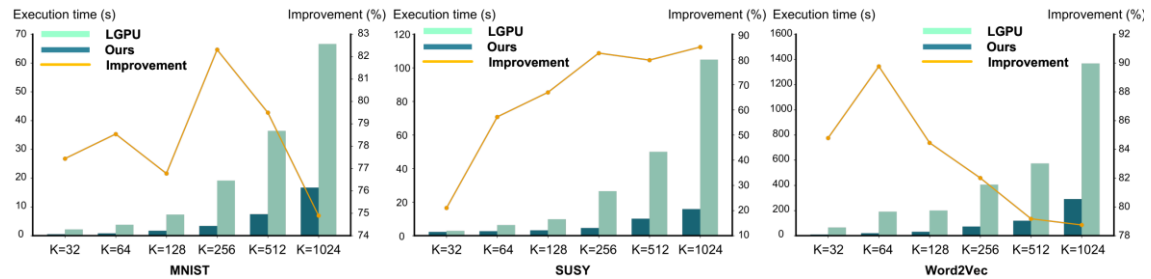


Figure 7 Comparison of execution times between the baseline (LGPU) and FastK across three datasets (MNIST, SUSY, and Word2Vec). The bars show execution times for baseline and our method, and the yellow line represents the speedups of our method against the baseline.

Table 2 Execution time (s) and normalized SSE (nSSE) on the Twitter GloVe dataset. LGPU, YGPU, and SYGPU denote Lloyd GPU, Yinyang GPU, and Simplified Yinyang GPU, respectively.

Dataset	Method	Number of Clusters (K)					
		32	64	128	256	512	1024
GloVe-25D	LGPU (time)	1.114	2.097	3.601	7.349	15.054	31.303
	LGPU (nSSE)	14.255	13.265	12.342	11.475	10.684	9.934
	YGPU (time)	0.748	1.219	2.199	4.322	8.788	19.796
	YGPU (nSSE)	14.255	13.265	12.341	11.475	10.684	9.934
	SYGPU (time)	0.579	1.016	1.844	3.747	7.514	17.414
	SYGPU (nSSE)	14.255	13.265	12.341	11.475	10.684	9.934
	Ours (time)	0.803	0.590	0.665	1.026	1.575	4.087
	Ours (nSSE)	14.271	13.271	12.330	11.494	10.717	9.981
GloVe-50D	LGPU (time)	2.925	5.329	11.797	21.263	46.968	89.808
	LGPU (nSSE)	19.168	18.265	17.427	16.648	15.884	15.163
	YGPU (time)	1.359	2.686	5.152	10.392	21.778	44.582
	YGPU (nSSE)	19.168	18.265	17.427	16.648	15.885	15.162
	SYGPU (time)	1.066	2.111	4.221	8.644	17.875	36.815
	SYGPU (nSSE)	19.168	18.265	17.427	16.648	15.885	15.162
	Ours (time)	1.316	1.545	2.933	5.486	13.724	26.194
	Ours (nSSE)	19.147	18.288	17.458	16.670	15.909	15.200
GloVe-100D	LGPU (time)	7.039	8.023	18.141	35.902	86.765	176.959
	LGPU (nSSE)	26.580	25.768	24.967	24.211	23.482	22.774
	YGPU (time)	2.806	4.431	10.261	22.013	52.748	129.824
	YGPU (nSSE)	26.581	25.768	24.968	24.210	23.481	22.774
	SYGPU (time)	2.251	3.477	7.897	16.418	39.417	89.730
	SYGPU (nSSE)	26.581	25.768	24.968	24.210	23.481	22.774
	Ours (time)	2.369	3.815	7.290	16.045	32.894	68.978
	Ours (nSSE)	26.628	25.775	24.999	24.212	23.488	22.796
GloVe-200D	LGPU (time)	8.359	15.625	35.157	74.248	140.980	338.368
	LGPU (nSSE)	33.805	33.078	32.377	31.709	31.057	30.457
	YGPU (time)	4.349	8.738	20.510	58.460	127.533	296.820
	YGPU (nSSE)	33.807	33.078	32.378	31.708	31.056	30.456
	SYGPU (time)	3.395	6.424	14.721	41.826	83.205	198.030
	SYGPU (nSSE)	33.807	33.078	32.378	31.708	31.056	30.456
	Ours (time)	4.789	8.634	16.945	36.282	85.270	197.999
	Ours (nSSE)	33.818	33.093	32.410	31.714	31.033	30.431

Table 3 Execution time (s) and normalized SSE (nSSE) on the MNIST, SUSY, and Word2Vec datasets.

Dataset	Method	Number of Clusters (K)					
		32	64	128	256	512	1024
MNIST	LGPU (time)	2.190	3.770	7.325	19.124	36.437	66.605
	LGPU (nSSE)	32.770	29.807	27.317	24.964	22.868	21.215
	YGPU (time)	0.804	1.519	3.150	7.703	17.693	34.784
	YGPU (nSSE)	32.772	29.807	27.318	24.963	22.867	21.216
	SYGPU (time)	0.737	1.377	2.707	6.836	14.748	28.631
	SYGPU (nSSE)	32.772	29.807	27.318	24.963	22.867	21.216
	Ours (time)	0.494	0.809	1.701	3.383	7.470	16.720
	Ours (nSSE)	33.092	30.105	27.371	25.130	23.154	21.551
SUSY	LGPU (time)	2.870	6.034	9.750	26.471	49.973	104.870
	LGPU (nSSE)	4.358	3.625	3.033	2.549	2.143	1.826
	YGPU (time)	1.842	3.794	5.815	14.229	29.215	58.200
	YGPU (nSSE)	4.358	3.625	3.033	2.549	2.143	1.826
	SYGPU (time)	1.538	3.389	5.203	13.472	26.385	57.592
	SYGPU (nSSE)	4.358	3.625	3.033	2.549	2.143	1.826
	Ours (time)	2.267	2.699	3.224	4.630	10.121	15.814
	Ours (nSSE)	4.448	3.663	3.061	2.568	2.153	1.813
Word2Vec	LGPU (time)	64.121	189.932	200.023	405.350	573.537	1368.239
	LGPU (nSSE)	4.204	4.051	3.913	3.755	3.602	3.468
	YGPU (time)	18.295	100.587	213.806	547.504	696.294	1479.339
	YGPU (nSSE)	4.204	4.051	3.913	3.754	3.602	3.468
	SYGPU (time)	16.167	83.795	130.721	307.601	419.480	803.283
	SYGPU (nSSE)	4.204	4.051	3.913	3.754	3.602	3.468
	Ours (time)	9.751	19.430	31.090	72.927	119.439	290.833
	Ours (nSSE)	4.182	4.030	3.880	3.747	3.615	3.505

Table 2 and Figure 6 compare the execution time and normalized sum of squared errors (nSSE) of the method with three representative GPU-based baselines: Lloyd GPU (LGPU), Yinyang GPU (YGPU), and Simplified Yinyang GPU (SYGPU). The experiments were conducted using the Twitter GloVe dataset, across various embedding dimensions (25D, 50D, 100D, and 200D) and cluster

counts ranging from 32 to 1024. The results show that our approach consistently achieves faster execution across all configurations while maintaining comparable clustering quality in terms of nSSE. For instance, on the 100-dimensional GloVe dataset with $K = 1024$, the runtime is reduced from 176.9 seconds (LGPU) to 68.9 seconds, achieving a $2.56\times$ speedup. As dimensionality increases, the performance gain becomes more pronounced. Even when compared with the more optimized SYGPU baseline, improvements are evident—e.g., from 89.7s to 68.9s at 100D and $K = 1024$, and from 17.9s to 13.7s at 50D and $K = 512$. In lower-dimensional settings, such as 25D with $K = 256$, the runtime is reduced from 7.3s (LGPU) to just 1.0s. These results suggest that the grouping and weighted refinement strategies used in the method are highly effective in reducing redundant computations without sacrificing clustering accuracy.

Our method consistently outperforms the three GPU-based baselines across the MNIST, Word2Vec, and SUSY datasets, while maintaining comparable clustering quality in terms of normalized SSE (nSSE), as shown in Table 3 and Figure 7. On the MNIST dataset, the runtime was reduced from 3.77 seconds (LGPU), 1.52 seconds (YGPU), and 1.38 seconds (SYGPU) to 0.81 seconds at $K = 64$. At $K = 1024$, execution time was reduced from 66.6s, 34.8s, and 28.6s to 16.7s, achieving speedups of up to $4.0\times$, $2.1\times$, and $1.7\times$, respectively. For the high-dimensional Word2Vec dataset, the runtime at $K = 1024$ was significantly reduced from 1368.2s (LGPU), 1479.3s (YGPU), and 803.3s

(SYGPU) to 290.8s, corresponding to $4.7\times$, $5.1\times$, and $2.8\times$ improvements. On the SUSY dataset, which includes structured numerical data, all methods achieved competitive performance, with our method reaching up to $6.6\times$ faster runtime than LGPU at $K = 1024$. These results demonstrate robust and scalable performance improvements across both low- and high-dimensional datasets, effectively reducing redundant computations without compromising clustering accuracy.

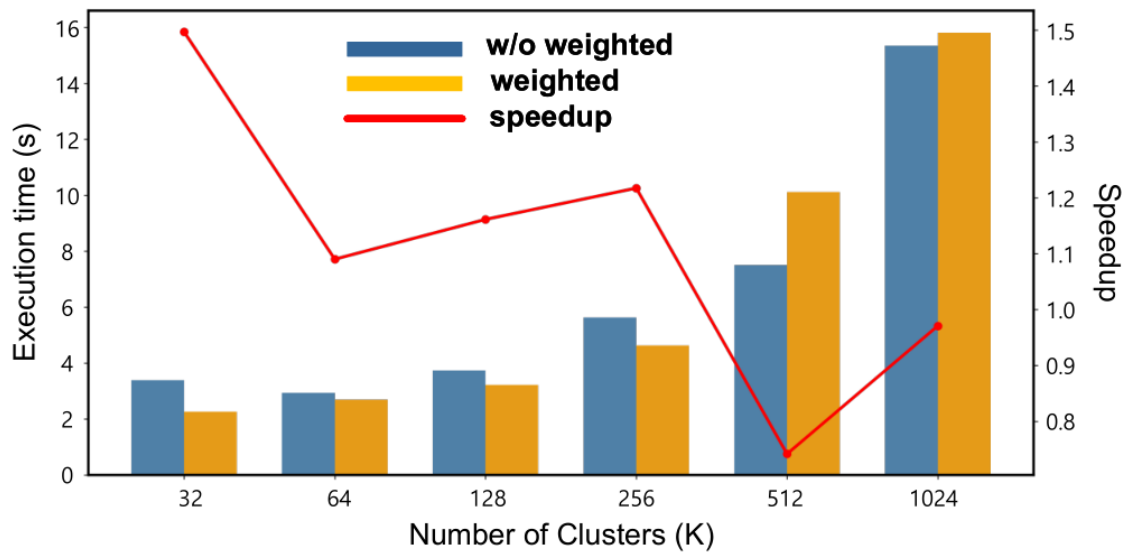


Figure 8 Comparison of execution times between the weighted centroid approach and the mean centroid approach on the SUSY dataset. The weighted centroid consistently achieves lower runtime while maintaining comparable clustering quality across different numbers of clusters ($K = 32-1024$).

Table 4 Comparison of Execution time (s) and normalized SSE (nSSE) using different centroid computation methods (Mean vs. Weighted Sum) on the SUSY datasetl

K	Execution Time (s)		nSSE	
	Mean	Weighted Sum	Mean	Weighted Sum
32	3.394	2.267	4.4208	4.4482
64	2.942	2.699	3.6596	3.6630
128	3.744	3.224	3.0630	3.0606
256	5.636	4.630	2.5654	2.5676
512	7.505	10.121	2.1590	2.1534
1024	15.350	15.814	1.8133	1.8126

Table 4 and Figure 8 compare the performance of centroid computation strategies on the SUSY dataset, specifically, simple mean centroids versus weighted centroids that incorporate group sizes. The weighted centroids show a comparable or better balance between runtime and clustering accuracy. For lower cluster counts ($K \leq 128$), both methods achieve similar nSSE values, with the weighted approach demonstrating faster runtimes. At $K = 256$, runtime decreases from 5.64s to 4.63s while maintaining comparable accuracy. For larger cluster sizes ($K = 512$ and $K = 1024$), the weighted centroid approach provides better clustering results with more stable convergence. These results suggest that accounting for group size when computing centroids contributes to both performance and accuracy, particularly in large-scale settings.

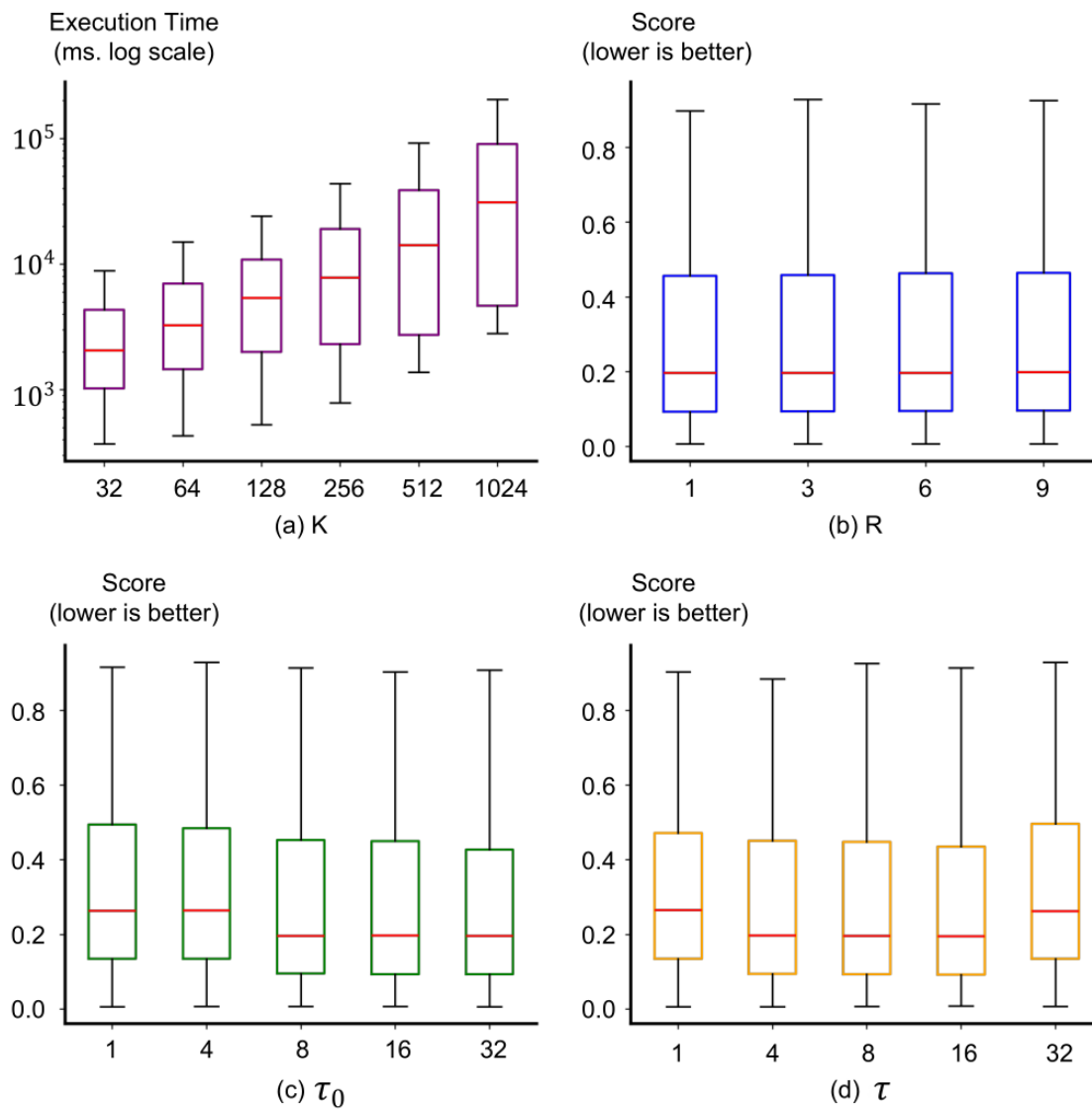


Figure 9 Effect of the parameters on the Twitter GloVe dataset. The boxplots illustrate the distribution of clustering runtimes under varying settings of (a) the number of clusters (K), (b) global iteration (R), (c) the initial grouping factor (τ_0), and (d) the subdivision factor (τ).

Figure 9 presents a sensitivity analysis of key internal parameters. To quantify their effects, we used a combined metric that jointly evaluates execution time and SSE. Each parameter was varied independently: the number

of global iterations $R \in \{1, 3, 6, 9\}$, the initial grouping factor $\tau_0 \in \{1, 4, 8, 16, 32\}$, and the subgroup density $\tau \in \{1, 4, 8, 16, 32\}$. Based on the results, the optimal configuration was selected as $R = 3$, $\tau_0 = 8$, and $\tau = 16$, providing the best trade-off between runtime and clustering quality.

Clustering quality in accelerated K-means methods is strongly affected by data distribution, particularly in non-uniform or multimodal datasets. When standard K-means is applied to a randomly sampled subset of the data (10%), clustering quality often degrades due to insufficient representation of minority structures. For example, on GloVe-25D with $K = 512$, 10% sampling results in a noticeably higher nSSE (10.9) compared to our method (10.717), despite achieving shorter execution time (0.6 s). In contrast, our method preserves representative structure through its split-and-pool process, achieving substantially lower nSSE while maintaining efficient execution time (1.575 s). This indicates that the improved clustering quality is not simply due to data reduction, but to structured aggregation that better reflects the underlying distribution.

To further validate the effectiveness of our design, we conduct an ablation study in which the coarse grouping stage is replaced with standard K-means. Although this variant exhibits similar execution time to the full method (1.6 s), it leads to consistently higher nSSE (10.85 vs. 10.717 on GloVe-25D with $K = 512$), particularly for non-uniform datasets. These results suggest that directly applying K-means for coarse grouping is less effective at capturing global

structure than our LSH-based coarse grouping strategy. Overall, these findings highlight the importance of distribution-aware processing and explain why our method achieves a favorable quality-efficiency trade-off compared to naive subsampling and K-means-based alternatives.

Chapter 6. Conclusion

We introduced Scalable GPU Clustering with Split-and-Pool-Driven Weighted K-means, a GPU-accelerated framework designed to efficiently handle high-dimensional and large-scale datasets. The method utilizes weighted centroids within a split-and-pool clustering strategy, effectively balancing runtime efficiency and clustering quality. The pipeline begins with coarse grouping based on locality-sensitive hashing, enabling proximity-aware parallel initialization. This approach significantly reduces execution time while preserving clustering accuracy, and produces a compact representation of cluster structures. It can be seamlessly integrated into existing K-means based workflows for scalable, high-performance clustering.

References

- [1] A. Fahad *et al.*, "A survey of clustering algorithms for big data: Taxonomy and empirical analysis," *IEEE transactions on emerging topics in computing*, vol. 2, no. 3, pp. 267–279, 2014.
- [2] A. S. Shirkhorshidi, S. Aghabozorgi, T. Y. Wah, and T. Herawan, "Big data clustering: a review," in *International conference on computational science and its applications*, 2014: Springer, pp. 707–720.
- [3] S. Lloyd, "Least squares quantization in PCM," *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.
- [4] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," Stanford, 2006.
- [5] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable k-means++," *arXiv preprint arXiv:1203.6402*, 2012.
- [6] C. Elkan, "Using the triangle inequality to accelerate k-means," in *Proceedings of the 20th international conference on Machine Learning (ICML-03)*, 2003, pp. 147–153.
- [7] B. Hong-Tao, H. Li-Li, O. Dan-Tong, L. Zhan-Shan, and L. He, "K-means on commodity GPUs with CUDA," in *2009 WRI World Congress on Computer Science and Information Engineering*, 2009, vol. 3: IEEE, pp. 651–655.
- [8] R. Wu, B. Zhang, and M. Hsu, "Clustering billions of data points using GPUs," in *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, 2009, pp. 1–6.
- [9] M. Zechner and M. Granitzer, "Accelerating k-means on the graphics processor via cuda," in *2009 First International Conference on Intensive Applications and Services*, 2009: IEEE, pp. 7–15.
- [10] A. Likas, N. Vlassis, and J. J. Verbeek, "The global k-means clustering algorithm," *Pattern recognition*, vol. 36, no. 2, pp. 451–461, 2003.
- [11] Y. Ding, Y. Zhao, X. Shen, M. Musuvathi, and T. Mytkowicz, "Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup," in *International conference on machine learning*, 2015: PMLR, pp. 579–587.
- [12] G. Hamerly, "Making k-means even faster," in *Proceedings of the 2010 SIAM international conference on data mining*, 2010: SIAM, pp. 130–140.

- [13] D. Nister and H. Stewenius, "Scalable recognition with a vocabulary tree," in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, 2006, vol. 2: IEEE, pp. 2161–2168.
- [14] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," *VISAPP (1)*, vol. 2, no. 331-340, p. 2, 2009.
- [15] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE transactions on pattern analysis and machine intelligence*, vol. 36, no. 11, pp. 2227–2240, 2014.
- [16] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*, 2004, pp. 253–262.
- [17] T. N. Mau and V.-N. Huynh, "An LSH-based k-representatives clustering method for large categorical data," *Neurocomputing*, vol. 463, pp. 29–44, 2021.
- [18] D. Sculley, "Web-scale k-means clustering," in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 1177–1178.
- [19] S. Har-Peled and S. Mazumdar, "On coresets for k-means and k-median clustering," in *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, 2004, pp. 291–300.
- [20] M. Shindler, A. Wong, and A. Meyerson, "Fast and accurate k-means for large datasets," *Advances in neural information processing systems*, vol. 24, 2011.
- [21] Y. Li, K. Zhao, X. Chu, and J. Liu, "Speeding up k-means algorithm by gpus," *Journal of Computer and System Sciences*, vol. 79, no. 2, pp. 216–229, 2013.
- [22] J. Wu and B. Hong, "An efficient k-means algorithm on CUDA," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011: IEEE, pp. 1740–1749.
- [23] S. Cuomo, V. De Angelis, G. Farina, L. Marcellino, and G. Toraldo, "A GPU-accelerated parallel K-means algorithm," *Computers & Electrical Engineering*, vol. 75, pp. 262–274, 2019.
- [24] M. Kruliš and M. Kratochvíl, "Detailed analysis and optimization of CUDA k-means algorithm," in *Proceedings of the 49th International Conference on Parallel Processing*, 2020, pp. 1–11.
- [25] M. B. Cordeiro and W. M. N. Zola, "Parallel K-means on GPU using Warp-Centric Strategies," in *2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS)*, 2024: IEEE, pp. 720–727.
- [26] M. Li, E. Frank, and B. Pfahringer, "Large scale K-means clustering using GPUs," *Data Mining and Knowledge Discovery*, vol. 37, no. 1, pp. 67–109, 2023.

- [27] J. Bellavita, T. Pasquali, L. Del Rio Martin, F. Vella, and G. Guidi, "Popcorn: Accelerating Kernel K-means on GPUs through Sparse Linear Algebra," in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2025, pp. 426–440.
- [28] C. Taylor and M. Gowanlock, "Accelerating the yinyang k-means algorithm using the GPU," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021: IEEE, pp. 1835–1840.
- [29] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.
- [30] T. R. RAPIDS, "Collection of libraries for end to end GPU data science," ed: NVIDIA, 2023.
- [31] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.
- [32] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 2002.
- [33] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature communications*, vol. 5, no. 1, p. 4308, 2014.
- [34] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.
- [35] L. Bian *et al.*, "Automatic extraction of fine structural information in angle-resolved photoemission spectroscopy by multi-stage clustering algorithm," *Communications Physics*, vol. 7, no. 1, p. 398, 2024.

논문요약

분할-집약 기반 가중 K-평균을 활용한 확장 가능한 GPU 클러스터링

김민성

실감미디어공학과

성균관대학교

본 논문은 CUDA를 활용하여 대규모 데이터셋을 효율적으로 처리할 수 있는, 확장성과 하드웨어 효율성이 뛰어난 K-means 클러스터링 알고리즘을 제안한다. 기존 K-means 방식은 모든 데이터 포인트를 반복적으로 처리해야 하므로, 데이터의 크기와 차원, 메모리 사용량이 증가할수록 연산 효율이 급격히 저하된다. 이를 해결하기 위해 본 논문에서는 데이터를 나누고 모으는 방식(split-and-pool, 분할-집약 방식)의 반복 구조를 도입하여 연산 속도를 향상시켰다. 초기의 대략적인 클러스터는 더 작은 하위 그룹으로 분할되며, 각 그룹의 포인트들은 하나의 대표 중심점(centroid)으로 요약되고, 해당 그룹의 크기를 가중치로 반영하여 계산된다. 이렇게 생성된 중심점들은 다시 상위 클러스터로 군집화되며, 다음 반복 단계의 입력으로 사용된다. 이와 같은 집약 메커니즘은 개별 포인트 수준이 아닌 묶음 단위로 거리를 계산할 수 있게 하여, 연산량을 크게 줄이고 메모리 접근 비용을 절감한다.

또한, 초기에는 locality-sensitive hashing (LSH, 지역 민감 해싱)을 활용하여 데이터의 근접성을 고려한 병렬 처리를 효과적으로 수행한다. 제안하는 반복 구조는 GPU 아키텍처의 병렬 처리 특성과 잘 맞아 높은 실행 효율을 보이며, 실제 대규모 데이터셋을 대상으로 한 실험에서도 기존 CUDA 기반 K-means보다 실행 시간이 크게 단축되면서도 유사한 클러스터링 품질을 유지함을 확인하였다.

주제어: K-평균 알고리즘, 분할-집약 방식, 그래픽 처리 장치/GPU, 쿠다/CUDA