Kubism: Disassembling and Reassembling K-Means Clustering for Mobile Heterogeneous Platforms

Seondeok Kim*
Korea University
Seoul, Republic of Korea
seondeok0312@korea.ac.kr

Sangun Choi*
Korea University
Seoul, Republic of Korea
sangun_choi@korea.ac.kr

Jaebeom Jeon
Korea University
Seoul, Republic of Korea
414dragon@korea.ac.kr

Junsu Kim
Korea University
Seoul, Republic of Korea
j0807s@korea.ac.kr

Minseong Gil
Korea University
Seoul, Republic of Korea
ms7859@korea.ac.kr

Jaehyeok Ryu
Korea University
Seoul, Republic of Korea
ryoujh03@korea.ac.kr

Yunho Oh
Korea University
Seoul, Republic of Korea
yunho_oh@korea.ac.kr

Abstract

K-means clustering is widely used in applications such as classification, recommendation, and image processing for its simplicity and efficiency. While often deployed on servers, it is also used on mobile platforms for tasks like sensor data analysis. However, mobile devices face tight hardware and energy constraints, making efficient execution challenging. Prior parallel K-means approaches still suffer from GPU underutilization due to warp divergence and leave CPUs idle. This paper proposes Kubism, a novel software technique that disassembles and reassembles a K-means clustering algorithm to maximize CPU and GPU resource utilization on mobile platforms. Kubism incorporates several key strategies, including reordering operations to minimize unnecessary work, ensuring balanced workloads across processing units to avoid idle time, dynamically adjusting task execution based on real-time performance metrics, and distributing computation efficiently between the CPU and GPU. These methods synergistically improve performance by reducing idle periods and optimizing the use of hardware resources. In our evaluation on the NVIDIA Jetson Orin AGX platform, Kubism achieves up to a 2.65× speedup in individual clustering iterations and an average 1.23× improvement in overall end-to-end execution time compared to prior work.

*Seondeok Kim and Sangun Choi contributed equally to this research.

CCS Concepts: • **Computer systems organization → Embedded software**.

Keywords: K-means Clustering, heterogeneous computing, mobile platforms

ACM Reference Format:
Seondeok Kim, Sangun Choi, Jaebeom Jeon, Junsu Kim, Minseong Gil, Jaehyeok Ryu, and Yunho Oh. 2025. Kubism: Disassembling and Reassembling K-Means Clustering for Mobile Heterogeneous Platforms. In *Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '25), June 16–17, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3735452.3735537

1 Introduction

K-means clustering is widely used across applications, such as data classification, recommendation systems, and image processing [5, 12, 34]. The goal of K-means clustering is to minimize the sum of the squared distance between each data point and its assigned centroid, thus achieving high intra-cluster similarity and low inter-cluster similarity [4, 7, 29]. As the number of data points and clusters increases, the computational and memory requirements for K-means clustering grow linearly [24].

While K-means clustering is frequently deployed in server environments, mobile platforms also rely on it for tasks like sensor data processing. However, mobile platforms face additional hardware and energy constraints compared to servers. Moreover, the linear growth of the computational and memory requirements depending on the data point count results in significant execution time and energy consumption, making K-means clustering execution more challenging. Prior work has proposed techniques that reduce redundant computations by skipping unnecessary distance calculations and accelerate a K-means clustering algorithm by leveraging Graphics Processing Unit (GPU) parallel computing capabilities [11, 33]. However, those software techniques still underutilize hardware resources on mobile platforms equipped with CPUs and GPUs. In case of a GPU-accelerated Yinyang K-means clustering algorithm [33], during skipping process, iterations that are skipped cause the corresponding GPU threads to become idle. Such behavior occurs warp divergence as no adjustments are made to mitigate this issue in this technique. Furthermore, as the implementation does not utilize the CPU, the multicore CPU remains unused.

In this paper, we propose Kubism, a novel software technique that disassembles and reassembles a K-means clustering algorithm to efficiently use the hardware resources in mobile platforms. With the key insights that we mentioned, Kubism addresses these challenges by balancing workloads between the CPU and GPU, reducing idle time, and further improving the parallel processing capabilities of GPUs compared to the prior work.

Inspired by a GPU-accelerated Yinyang K-means algorithm [33], we focus on innovating the local filter algorithm, which accounts for over 90% of the total execution time. Kubism dynamically adapts the local filter algorithm to the skip ratio and eliminates unnecessary computations through reordering, warp balancing, and intelligent task partitioning across mobile GPUs and CPUs, called *heterogeneous task distribution (HETD)*. The *reordering technique* performs early boundary checks, allowing for quick identification of skippable computations. This approach reduces unnecessary work by avoiding redundant thread executions on the GPU, helping to prevent warp divergence. The *warp balancing* technique creates a new GPU kernel for the local filter algorithm, grouping threads that run active distance calculations into adjacent positions within the kernel, which improves hardware resource utilization of GPUs. The *dynamic skip ratio decision* mechanism optimizes warp balancing based on the skip ratio of each iteration. The *heterogeneous platform-aware task distribution (HETD)* technique balances the computational load between the CPU and GPU, further improving performance compared to GPU-only implementations on mobile platforms. We implement HETD without introducing heavy data movement overhead between the CPU and GPU by exploiting page-locked memory allocations, enabling efficient data sharing.

In our evaluation on the NVIDIA Jetson Orin AGX platform, Kubism achieves up to a 2.65× speedup in individual clustering iterations compared to the prior GPU-accelerated Yinyang K-means clustering implementation [33]. Also, Kubism achieves an average 1.23× improvement in overall end-to-end execution time for complete clustering.

The contributions of this paper are as follows:

- We analyze the behavioral characteristics of a state-of-the-art K-means clustering algorithm and demonstrate that its performance can be improved by disassembling and reassembling the entire workflow.
- We propose Kubism that effectively utilizes both CPU and GPU resources and increases GPU thread efficiency.
- Our evaluation shows that Kubism achieves up to a 2.7× speedup in iterations and a 1.2× speedup in overall execution time compared to the prior GPU-accelerated K-means clustering implementation.

The rest of this paper is organized as follows. Section 2 explains the challenges that we define. Section 3 presents Kubism, our proposed solution. Section 4 explains the evaluation methodology and results. Section 5 discusses related work. Section 6 concludes this paper.

2 Why Kubism?

2.1 K-Means Clustering Algorithms

Clustering algorithms are essential for many data-intensive applications such as data classification, recommendation systems, and image processing. K-means clustering is particularly popular due to its efficiency and simplicity in partitioning data into clusters [3, 11, 15, 16, 33, 35]. The objective of K-means clustering is to minimize the sum of the squared distance between each data point and its corresponding cluster centroid, so data points within a cluster exhibit high similarity to each other while maintaining dissimilarity from points in other clusters.

As the number of data points and clusters increases, the computational and memory requirements for K-means clustering grow linearly [24]. This linear growth in computational and memory needs, depending on the number of data points, results in significant execution time and energy consumption. To address this challenge, prior work has proposed novel K-means clustering algorithms utilizing the skipping mechanism to eliminate redundant computations inherent in the original K-means clustering [11–14, 24].

In the original K-means clustering, although most data points retain their cluster assignments during iterations, the algorithm still performs unnecessary distance calculations for all data points. To address this challenge, prior work has proposed the Yinyang K-means clustering algorithm [11]. This algorithm includes a mechanism that skips distance calculation based on a boundary check so that it reduces the execution time compared to the original algorithm. The skipping distance calculation process identifies such data points using three key values: the upper bound, the lower bound, and drift for each data point. The upper bound represents the distance from a data point to its assigned cluster centroid, while the lower bound is the distance to the nearest centroid among the other cluster centroids. Drift denotes the maximum shift of centroids due to updates in each iteration. If the difference between the lower bound and the drift is greater than the upper bound, the nearest centroid cannot

Algorithm 1 Yinyang K-Means Clustering Algorithm

```
Input: Dataset $X = \{x_1, x_2, \ldots, x_n\}$, number of clusters $k$, initial centroids $C = \{c_1, c_2, \ldots, c_k\}$
Output: Final cluster centroids $C$, cluster assignments for each data point
function YINYANG_KMEANS($X$, $k$, $C$)
    // Step 1: Group centroids into groups
    Group centroids into $i$ groups for use in the group filtering step
    $G = \{g_1, g_2, \ldots, g_i\}$

    while not converged do
        // Step 2: Initialization
        Compute initial upper bound $ub_n$ for each data point $x_n$
        Compute initial lower bound $lb_i$ using distance to centroids in group $g_i$

        // Step 3: Calculate centroid drift
        Calculates drift for each $c_k$ and stores maximum drift per $g_i$

        // Step 4: Group filter
        for $dp = 0$ to $n$ do // For all data points
            for $cent = 0$ to $i$ do // For all groups
                if group $lb_{cent}$ is $\geq ub_{dp}$ then
                    Skip distance calculation for $x_{dp}$ and entire group $g_{cent}$
                end if
            end for
        end for

        // Step 5: Local filter
        for $dp = 0$ to $n'$ do // n': # of data points passed through group filter
            for $cent = 0$ to $k$ do // For all centroids
                if $lb_{cent}$ is $\geq ub_{dp}$ then
                    Skip distance calculation for $x_{dp}$ and $c_{cent}$
                end if
            end for
        end for

        //Step 6: Centroid adjust
        Adjust position of each $c_k$

    end while
end function
```

become the new centroid for the data point, as the distance to the currently assigned centroid remains shorter despite the centroid shifts.

Algorithm 1 describes the Yinyang K-means clustering algorithm. Yinyang K-means clustering reduces computational overhead by implementing a two-step filtering process called the group filter and the local filter [10]. The group filter first skips unnecessary distance calculations at the granularity of centroid groups, and then the local filter eliminates redundant distance calculations within each group. Initially, centroids are organized into groups, and in the group filtering stage bounds are applied to filter out centroid groups that do not require additional distance calculations. This approach efficiently reduces a large number of computations with only a small number of distance comparisons. After the group filtering, the algorithm determines the best centroid within the groups that pass through the filter. Within these groups, Yinyang K-means clustering applies a local filter further to skip unnecessary computations.

Both the original K-means clustering and Yinyang K-means clustering algorithms offer opportunities for parallel computing, which can be leveraged to further enhance their performance. Prior work has proposed algorithms that accelerate both methods using GPUs [18, 33]. By taking advantage of the massive parallelism offered by GPUs, computations such as distance calculations and centroid updates are distributed across many processing cores. Such a behavior results in substantial improvements in execution time and efficiency compared to CPU-based implementations. In the case of Yinyang K-means clustering, prior work has shown that its GPU-based implementation outperforms the GPU-based implementation of the original K-means clustering [33].

2.2 K-Means Clustering in Mobile Platforms

Not only servers but also embedded systems often employ K-means clustering for online tasks such as processing images generated by sensors. These tasks are important for features such as image recognition, augmented reality, and environmental sensing [1, 2, 17, 32]. For example, autonomous driving systems commonly use the K-means clustering algorithm to analyze sensor data for tasks like lane and object detection [8, 21]. On mobile platforms, K-means clustering helps with image detection by automatically determining the sizes and number of anchor boxes [6, 19, 20]. The algorithm groups sensor images and configures anchor boxes, removing the need for manual parameter settings. These applications are also important for robotics, which rely on image detection and pattern recognition to process sensor data [22]. Recent improvements in deep neural networks (DNNs) have made clustering algorithms even more useful on mobile devices. Combining K-means clustering with DNNs improves the performance of data analysis tasks [6, 20].

These applications show that K-means clustering plays a key role in the wide range of data analytics tasks on embedded systems. K-means clustering improves the processing capabilities of embedded hardware platforms through efficient data segmentation and feature extraction while maintaining real-time performance requirements. As the demand for data analytics tasks on mobile and embedded systems grows, efficient implementation of K-means clustering is essential for such resource-constrained hardware platforms.

2.3 Key Challenges of K-Means on Mobile Platforms

Many mobile platforms have been equipped with low-power GPUs, enabling more efficient execution of K-means clustering by leveraging parallel processing capabilities compared to traditional mobile platforms equipped only with CPUs. In the case of NVIDIA Jetson, embedded Systems-on-a-Chip (SoC) consists of both a CPU and a GPU, which physically share the same SoC DRAM memory to minimize the cost of host-to-device data transfers [27]. Implementing K-means clustering on mobile platforms presents unique challenges due to limited computational resources and energy constraints. K-means clustering algorithms exhibit computational complexity and memory access operations that scale linearly with both the number of input data points and the number of clusters, which leads to substantial increases in execution time and energy consumption [9, 30, 31].

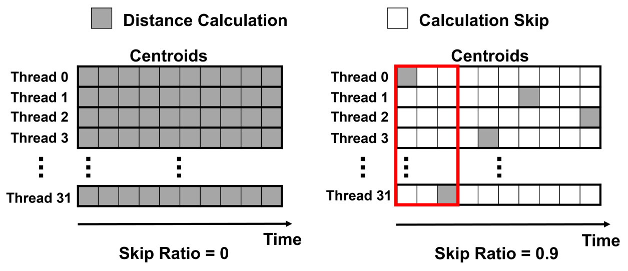

Figure 1. Visualized example for warp divergence in a warp. The left side shows the case where the skip ratio is 0. All threads perform distance calculations, so warp divergence does not occur. The right side demonstrates the case where the skip ratio is 0.9. A few threads within a warp perform computations while the others wait until the calculations are completed.

Table 1. Experiment Configuration

Platform	NVIDIA Jetson Orin AGX
GPU Architecture	Ampere architecture
GPU Cores	8 SMs, 1792 CUDA cores, 56 Tensor cores
Max. # of Threads per SM/Block	1536 / 1024
CPU	12-core Arm Cortex-A78AE v8.2
Memory	32 GB LPDDR5, Bandwidth: 204.8 GB/s

The GPU-accelerated Yinyang K-means clustering implementation proposed by prior work does not take full advantage of the hardware characteristics of embedded GPUs and CPUs [33]. On the GPU side, some threads perform distance calculations while others remain idle until the calculations are complete. Such a behavior is known as warp divergence [18]. Warp divergence incurs an imbalance in the workload across threads in a warp. Such an imbalance causes a large portion of GPU hardware resources to remain idle until the active threads finish their calculations. Figure 1 represents a visualized example of warp divergence in Yinyang K-means clustering. The left side in the example represents the case where the skip ratio is 0. All threads perform distance calculations, so warp divergence does not occur. The right side shows the case where the skip ratio is 0.9. A few threads within a warp perform distance calculations while the other threads stall until the calculations are completed.

We conduct experiments to analyze the behavioral impact of warp divergence in Yinyang K-means clustering on a mobile platform, using the NVIDIA Jetson Orin AGX platform [27]. We use the source code of the GPU-accelerated Yinyang K-means clustering algorithm proposed by the prior work [33]. Detailed system configurations of the NVIDIA Jetson Orin AGX platform are provided in Table 1. The dataset used in our experiments contains one million data points with 128 feature dimensions.

We first profile the *average active thread per warp* metric during execution using Nsight Compute [25]. This metric represents the average number of active threads in a warp for each executed instruction. Nsight Compute calculates it by dividing the total number of executed instructions across all threads by the number of instructions executed by the entire warp. If some threads in a warp remain idle, the metric shows a low value (e.g., 3). If all threads in the warp are active, the metric approaches the maximum value of 32.

Table 2. Example of the number of active threads sampled in five iterations.

Iteration	Skip Ratio	Number of Active Threads per Warp
1	0.91	2.84
2	0.60	12.76
3	0.24	24.16
4	0.06	29.96
5	0.01	31.58

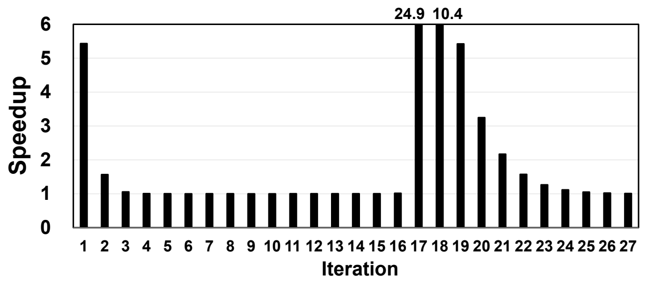

Figure 2. Speedup of Oracle implementation compared to the GPU-accelerated Yinyang K-means proposed in prior work. Oracle represents an implementation with minimized warp divergence. Iterations 1, 17, 18, and 19 exhibit significant speedup due to higher skip ratios, with Oracle achieving at least 5× speedup.

Table 2 shows the profiling results for the first to fifth iterations. Due to the warp divergence, the number of active threads significantly decreases. We observe that the number of active threads is highly related to the (1 - skip ratio), which means a ratio of distance calculation for each iteration. Since the first two iterations exhibit high skip ratios, the number of active threads per warp is much smaller than 32. As the first iteration has the highest skip ratio among five iterations, it exhibits the lowest active threads. In our initial study, 30% of threads are inactive during the end-to-end execution of the GPU-accelerated Yinyang K-means clustering.

Figure 2 presents the experimental results of the speedup depending on the skip ratios for each iteration of the Yinyang K-means clustering algorithm. For comparison, we implement an oracular implementation of Yinyang K-means clustering called Oracle, which heuristically eliminates warp divergence. In this graph, The X-axis shows the corresponding iterations, and the Y-axis shows the speedup per iteration. Overall, this workload runs 30 iterations. Among all the iterations, the workload runs the original K-means clustering algorithm from the first to the third iteration, so we exclude them from our analysis and show 27 iterations. We measure the per-iteration speedup of Oracle over the baseline, the GPU implementation of Yinyang K-means clustering proposed by the prior work [33]. In iterations 1, 17, 18, and 19, the skip ratio exceeds 0.8, resulting in the Oracle implementation achieving a speedup of at least 5× compared to the prior work. Specifically, in iterations 17 and 18, where the skip ratio surpasses 0.95, the Oracle implementation achieves a speedup greater than 10×.

On the CPU side, its underutilization is another significant issue while running K-means clustering on embedded GPUs. The CPU is only utilized for a limited number of tasks, such as launching kernels, while the GPU handles the main computational workload. We profile the utilization of the CPUs in the NVIDIA Jetson Orin AGX platform during the execution of Yinyang K-means clustering. We use Nsight Systems for profiling [26]. Profiling results show that the CPU in Jetson platform remains idle for 98.6% of the total execution time, highlighting the inefficiency in utilizing available CPUs.

The main challenges identified in implementing K-means clustering on mobile platforms involve both GPU and CPU underutilization. On the GPU side, warp divergence caused by the filtering mechanism incurs significant idle time, as some threads skip computations while others remain active, resulting in performance slowdowns. This inefficiency becomes critical if the skip ratio is high, causing substantial execution time increases. Also, the CPU remains idle during execution, as it is only used for tasks such as kernel launches. These issues highlight the need for better utilization of both GPU and CPU to improve performance in mobile platforms.

We make the following observations to resolve this challenge. K-means clustering works in multiple passes, making it well-suited for dynamic and adaptive innovations. Its structure allows for adjustments, like reordering and skipping distance calculations, without impacting result accuracy. This iterative and flexible nature of K-means clustering makes it possible to break down and rebuild the algorithm to better fit the hardware features of mobile platforms.

The execution time and behavior of K-means clustering vary depending on the number of data points and the length of feature vectors. Longer feature vectors also add to the workload by requiring more resources for each calculation, which uses up more memory and increases access time. Processing large, high-dimensional datasets can strain mobile or embedded systems and add extra overhead, especially if tasks are split between the CPU and GPU. Techniques like page-locked memory allocation can help lower data transfer costs, but an adaptive approach could make things even more efficient. By adjusting to the demands of the task based on data size and feature size, adaptive solutions can handle overhead better.

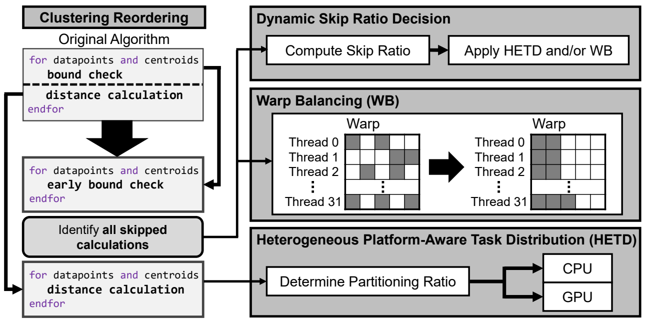
Clustering Reordering
Original Algorithm
for datapoints and centroids
bound check
distance calculation
endfor
for datapoints and centroids
early bound check
endfor
Identify all skipped calculations
for datapoints and centroids
distance calculation
endfor
Dynamic Skip Ratio Decision
Compute Skip Ratio
Apply HETD and/or WB
Warp Balancing (WB)
Warp
Thread 0
Thread 1
Thread 2
Thread 31
Heterogeneous Platform-Aware Task Distribution (HETD)
Determine Partitioning Ratio
CPU
GPU

Figure 3. Kubism overview.

3 Kubism

3.1 Key Idea of Kubism

To address the challenges discussed in the previous section, we propose Kubism, a new software approach designed to improve computational efficiency on mobile heterogeneous platforms. Unlike traditional methods that do not fully utilize either the CPUs or GPUs, Kubism takes full advantage of the parallel processing capabilities of both CPU and GPU.

Inspired by the Yinyang K-means clustering algorithm, Kubism takes advantage of two key features while running over multiple iterations. First, we focus on redesigning the local filter function that accounts for over 90% of the total execution time. Second, within each iteration, it is possible to change the order of skipping steps and calculations without affecting the accuracy of the results. By considering these factors, we design Kubism by disassembling and reassembling the K-means clustering algorithm to improve the threads and hardware resource utilization of mobile GPUs and CPUs.

Figure 3 describes an overview of Kubism. Kubism is based on the following four key ideas: Clustering reordering, warp balancing, dynamic skip ratio decision, and heterogeneous platform-aware task distribution (HETD).

The clustering reordering technique makes the clustering process more efficient by reorganizing the original Yinyang K-Means clustering algorithm. Instead of repeatedly checking whether each data point can be skipped before doing distance calculations, Kubism checks the bound values for all data points early on, before any calculations start. This early check helps Kubism find threads that do not need to do distance calculations, cutting down on unnecessary work.

Warp balancing mitigates warp divergence by organizing threads within a warp to perform only necessary distance calculations. Kubism addresses this issue by creating a new GPU kernel for the local filter algorithm that groups threads performing active distance calculations into adjacent positions within the kernel. With this technique, the threads in the new kernel proceed through the loop together, minimizing idle periods caused by skipped centroids. By completing distance calculations with all centroids in fewer loops and

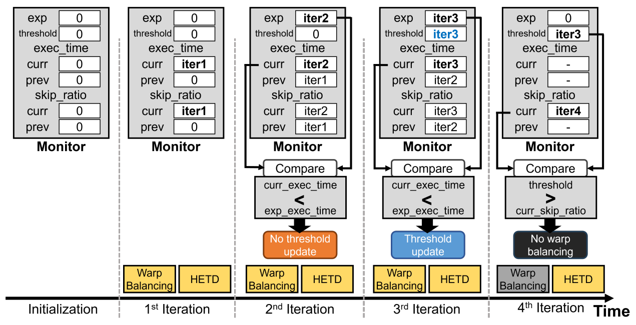

Figure 4. Kubism workflow for five phases, including initialization and four clustering iterations.

maintaining active participation of all threads, Kubism significantly improves the hardware resource utilization.

The dynamic skip ratio decision in Kubism further improves the use of warp balancing by monitoring the effectiveness of the process. The dynamic skip ratio decision performs two key functions. First, the technique determines the skip ratio threshold required for applying optimizations of Kubism during each iteration. Second, the technique decides whether to apply warp balancing based on the skip ratio threshold and the monitoring results of the current iteration. While warp balancing reduces warp divergence, it can add extra execution time if an iteration has a low skip ratio (e.g., 5%). In such a case, the overhead from warp balancing outweighs its benefits, as warp divergence has less impact. To address this issue, Kubism predicts the execution time of the local filter based on the execution time of the previous iteration and the skip ratio, as well as the skip ratio of the current iteration. If the actual execution time exceeds the predicted time, Kubism sets a skip ratio threshold. Kubism applies the warp balancing only to iterations with a skip ratio higher than this threshold, using the technique efficiently.

The HETD technique in Kubism employs heterogeneous computing by distributing distance calculations between the CPU and GPU. Kubism adaptively determines the best-performing partitioning ratio of distance calculations between the two processors, so the CPU and GPU have similar execution times. This balanced distribution maximizes hardware resource usage and enhances overall performance.

3.2 Kubism Workflow

Figure 4 shows the Kubism workflow with a series of phases.

The main goal of the performance monitoring in Kubism is to dynamically adjust the application of techniques based on the runtime behavior of each iteration. As shown in Section 2, the skip ratio decreases as iterations progress, indicating that a larger number of data points need to be processed in later stages. Kubism continuously monitors this behavior, applying warp balancing until it identifies a skip ratio threshold where further warp balancing negatively affects performance.

Initialization Phase: Kubism first initializes key parameters that drive its monitoring mechanism. The performance monitor is initialized with six fields: skip ratio threshold (`skip_threshold`), the current and previous iteration execution times (`curr_exec_time` and `prev_exec_time`), skip ratios of the current and previous iterations (`curr_skip_ratio` and `prev_skip_ratio`), and the expected execution time for the current iteration (`exp_exec_time`). During the initialization kernel, these values are set to zero, and upper and lower bounds are established, similar to the Yinyang K-Means clustering algorithm. This initialization step occurs not only in the first iteration but also if the number of data points filtered in the group filter exceeds a certain threshold.

First iteration: During the first iteration, Kubism applies warp balancing along with reordering and HETD since no skip ratio threshold has been set yet. The monitor records the skip ratio for this iteration in `curr_skip_ratio` before starting the distance calculation. After the calculations, the monitor stores the execution time in `curr_exec_time`. As this is the initial iteration, there is no previous data to compare with, and Kubism proceeds with full optimization.

Second iteration: In the second iteration, the monitor updates the `prev_skip_ratio` and `prev_exec_time` fields using the Values from the first iteration. Before the local filter, the monitor calculates the expected execution time (`exp_exec_time`) for this iteration using the values in the previous iteration and the current skip ratio. Similar to the first iteration, Kubism applies warp balancing as the threshold is still zero. After the local filter, the monitor compares `curr_exec_time` with `exp_exec_time`. In this case, the execution time is shorter than expected, so the threshold remains unchanged, warp balancing is applied in future iterations.

Third iteration and threshold setting: In the third iteration, Kubism repeats the process of updating the previous execution values and recalculating the expected execution time. During this iteration, the monitor detects that `curr_exec_time` is longer than `exp_exec_time`, indicating that applying warp balancing is no longer beneficial. Then, Kubism sets the skip ratio threshold to `curr_skip_ratio`, preventing unnecessary warp balancing in future iterations unless the skip ratio exceeds this threshold.

Fourth iteration and HETD: In the fourth iteration, Kubism uses the established skip ratio threshold to determine whether to apply warp balancing. The monitor calculates `curr_skip_ratio` and compares it to the threshold. As the skip ratio is lower than the threshold, Kubism avoids warp balancing and only performs HETD to maximize efficiency. By monitoring performance in real-time and selectively applying its optimization techniques, Kubism effectively minimizes execution time while maximizing resource utilization on mobile heterogeneous platforms.

Algorithm 2 Kubism Algorithm

```
Input: Dataset $X = \{x_1, x_2, \ldots, x_n\}$, number of clusters $k$, initial centroids $C = \{c_1, c_2, \ldots, c_k\}$
Parameters: CPU-GPU partition ratio $\alpha$, skip ratio threshold $\theta$
Output: Final cluster centroids $C$, cluster assignments for each data point
 1: function KUBISM FOR LOCAL FILTER($X$, $k$, $C$, $\alpha$, $\theta$)
 2:     // Step 1: Initialization
 3:     Compute initial upper bounds $ub[i]$ for each data point $x_i$
 4:     Compute lower bounds $lb[i][j]$ for each data point $x_i$ to each cluster $c_j$
 5:     Assign data points to nearest centroid
 6:     while not converged do
 7:         // Step 2: Update centroids
 8:         for each centroid $c_j$ do
 9:             Update centroid $c_j$ by averaging the assigned points
10:         end for
11:         // Step 3: Reordering and Skip Ratio Calculation
12:         for each data point $x_i$ do
13:             Update upper bound $ub[i]$ using the previous centroid
14:             Update lower bounds $lb[i][j]$ for each centroid group
15:             Determine if data point $x_i$ can skip distance calculation
16:             if lower bound $lb[i][j] > ub[i]$ then
17:                 Skip distance calculation for $x_i$ and centroid $c_j$
18:             else
19:                 Calculate distance between $x_i$ and centroids in the relevant group
20:                 Update cluster assignment for $x_i$ to the nearest centroid
21:             end if
22:         end for
23:         // Step 4: Warp Balancing
24:         if current skip ratio > $\theta$ then
25:             Apply warp balancing to minimize warp divergence on GPU
26:         end if
27:         // Step 5: Heterogeneous Platform-Aware Task Distribution (HETD)
28:         Dynamically distribute tasks between CPU and GPU to minimize idle time
29:         // Step 6: Adaptive Load Balancing
30:         Adjust partition ratio $\alpha$ based on current workload and skip ratio
31:     end while
32:     return Final centroids $C$ and cluster assignments
33: end function
```

3.3 Kubism Implementation Details

Kubism Algorithm: Algorithm 2 describes the key algorithms of Kubism. As shown in Algorithm 2, each thread processes one data point. The first step is to fetch the centroid number currently assigned to each data point. Then, an operation is performed to determine whether the distance calculation can be skipped by comparing the upper and lower bound values. The efficiency of this process is key to reducing overall execution time, if the filtering mechanism allows for skipping a substantial number of distance calculations.

Kubism computes the skip ratio to assess how many distance calculations are skipped due to the filtering mechanism applied during the reordering stage. As outlined in Algorithm 2, the skip ratio is computed by the code running on the GPU, which handles the summation and division of computation counts stored for each data point during the reordering phase. The GPU threads calculate the skip ratio (SR) by dividing the total sum of the computation counts by the number of operations that all data points filtered in the group filter would perform with all centroids, as represented by the following equation:

$$SR = 1 - \frac{\text{\# of Distance Calculations}}{\text{\# of Centroids} \times \text{\# of Data Points}} \tag{1}$$

This calculation helps quantify the proportion of distance calculations skipped during the execution, providing insight into the effectiveness of the filtering mechanism. A higher skip ratio indicates that the algorithm is more successful in reducing unnecessary computations.

The warp balancing technique effectively addresses the warp divergence problem in GPU computation. By scheduling computations across distance calculation loops, this method reduces idle loops while calculating distances to all centroids. We implement a balancing buffer that stores information indicating which centroid each data point should process during each loop. The balancing buffer is implemented in device memory, so the GPU can manage and access the data. Instead of directly storing centroid indices, Kubism utilizes a bitmask representation within the balancing buffer. Each bit in the bitmask indicates whether the distance to a specific centroid should be calculated or skipped. Using this information, Kubism retrieves centroid indices through the bit positions and precomputes operations from future loops that would result in idle loops. This precomputation eliminates idle phases, thereby improving overall computational efficiency by keeping all threads active during calculations.

Kubism distributes data points between the CPU and GPU based on a partitioning ratio. The data points for both the CPU and GPU use page-locked memory, which is essential for heterogeneous computing environments. On embedded platforms like NVIDIA Jetson, pinned memory maps directly to the address space of the devices, allowing access without needing to copy the data. This approach removes the overhead caused by copying. For data structures that the CPU only reads, the system copies them during the first iteration to avoid transferring them between the CPU and GPU in every iteration, improving overall efficiency.

Kubism transfers the data structures used for the CPU local filter from the GPU. Also, during the local filter stage, Kubism updates the data structures containing the bound values, as well as centroid assignment information. As this stage runs in parallel on both the CPU and GPU, Kubism allocates these data structures in page-locked memory to minimize the time needed to transfer the updated information back to the GPU once the CPU local filter operation finishes.

Kubism uses heterogeneous computing by running the local filter in parallel on both the CPU and GPU for the threads distributed by HETD. Kubism initially sets the partitioning ratio based on the CPU and GPU throughput of a given system configuration, and the skip ratio of the current iteration, which indicates the amount of computation. Then, Kubism determines the number of data points assigned to the CPU using the ratio. Kubism runs the CUDA kernel on the GPU using the data points remaining after the CPU assignment. During the clustering iterations, Kubism dynamically adjusts the partitioning ratio by comparing the execution time of CPU and GPU to balance them.

Details of Monitoring Mechanism: In each iteration, Kubism monitors performance to optimize the execution process. During the first iteration, Kubism initializes the system by running the reordering, warp balancing, and heterogeneous computing stages with the skip threshold set to zero. Kubism records the current execution time and skip ratio. From the second iteration onward, it calculates the predicted execution time using the equation:

$$T_{predict} = T_{prev} \times \frac{(1 - SR_{prev})}{(1 - SR_{current})} \tag{2}$$

where $T_{predict}, T_{prev}, SR_{prev}, SR_{current}$ represent the predicted execution time, previous execution time, previous skip ratio, and current skip ratio, respectively.

After completing reordering, warp balancing, and heterogeneous computing in the second iteration, Kubism compares the predicted execution time with the actual local filter execution time. If the predicted execution time is less than the local filter execution time, Kubism maintains the current skip threshold. Otherwise, Kubism adjusts the skip threshold for subsequent iterations. In the case where the current skip ratio is lower than the skip threshold, Kubism skips warp balancing and proceeds with reordering and heterogeneous computing. Otherwise, Kubism performs all stages as usual. This monitoring and adjustment process repeats across iterations, so Kubism dynamically adapts to optimize performance based on the relationship between execution time and skip ratio, iterating through reordering, warp balancing, and heterogeneous computing as needed.

Overhead Analysis: Kubism introduces several stages that contribute to software overhead, but the overall impact remains negligible. During the reordering phase, the primary modification involves adjusting the sequence of skip decisions and distance calculations. To minimize the overhead, Kubism first estimates the skip ratio by sampling a subset of data points instead of computing it for every point. This approach precisely estimates the skip ratio while incurring less overhead, less than 1% of the total execution time. The overhead from warp balancing varies based on the number of data points and the skip ratio. In our initial analysis, this overhead ranges from 7% to 30% of the local filter kernel execution time. Kubism avoids such scenarios using an intelligent monitoring scheme. Kubism also requires data structures to store centroid indices and the total number of computations between data points and centroids. By storing only one bit per centroid index, Kubism minimizes memory usage and the number of memory accesses needed to retrieve centroid indices. The size of these data structures accounts for only 3.1% of the dataset size. For data point partitioning, Kubism maintains additional data structures that store information about which CPU and GPU process data points. Since only the indices of data points are stored, the size of these structures remains relatively small, typically within tens of megabytes.

Table 3. Dataset

Dataset (N,D)	Number of Datapoints (N)	Feature Dimension (D)	Distribution
(0.5 M, 64)	0.5 Million	64	Uniform random distribution, values are ranged from 0 to 100.
(0.5 M, 128)	0.5 Million	128	
(0.5 M, 256)	0.5 Million	256	
(1 M, 64)	1 Million	64	
(1 M, 128)	1 Million	128	
(1 M, 256)	1 Million	256	
(2 M, 64)	2 Million	64	
(2 M, 128)	2 Million	128	
(2 M, 256)	2 Million	256	

Table 4. Evaluation Configurations

Configuration	Key Characteristics
Baseline	Prior GPU-based Yinyang K-means clustering [33]. This configuration suffers from warp divergence and hardware underutilization.
WB+HETD	This configuration consistently applies all three techniques introduced in this paper (clustering reordering, warp balancing, HETD). However, warp balancing may cause significant overhead in some iterations.
Kubism	Kubism selectively applies three techniques to take advantage of each technique and avoid the overhead.

4 Evaluation

4.1 Methodology

We conduct all experiments on the NVIDIA Jetson Orin AGX platform with the identical system specifications in Table 1. To examine the effects of the structure of the dataset, we configure datasets while varying the number of data points and the feature dimension based on the real-world datasets used in prior work [11, 33]. We select three numbers of data points (500K, 1M, and 2M) and feature dimensions (64, 128, and 256) and combine them to build nine datasets, as listed in table 3. We use a uniform random distribution with values ranging from 0 to 100 to generate all datasets. We set the number of clusters (k) to 1024 for all experiments.

Table 4 summarizes the key characteristics of each evaluated method, helping to clarify the differences in how they leverage CPU and GPU resources. To evaluate the performance of Kubism, we compare its execution time to two techniques. The first technique, referred to as *Baseline*, is the GPU-based implementation of the Yinyang K-means clustering algorithm, as described in prior work [33]. The second technique, *WB+HETD*, consistently applies reordering, HETD, and warp balancing techniques for all iterations, aiming for more efficient thread scheduling. Kubism is the combination of reordering, warp balancing, and task distribution techniques. In addition to execution time, we measure power consumption and hardware utilization using Tegrastats [28] and the NVIDIA Nsight Systems profiler [26].

4.2 Performance

Figure 5 shows the end-to-end speedup across the following 9 datasets in the table 3. We normalize all results to the baseline for each dataset. Kubism achieves a speedup of 1.23×

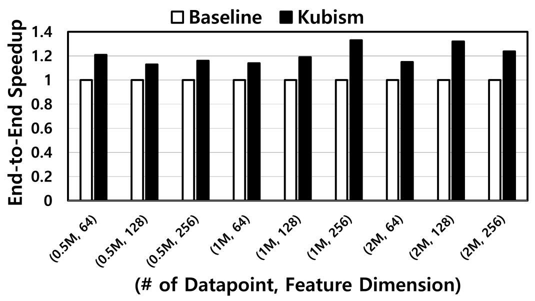

Figure 5. End-to-end speedup for all datasets. The average speedup (geometric mean) of Kubism over the baseline is 1.23×.

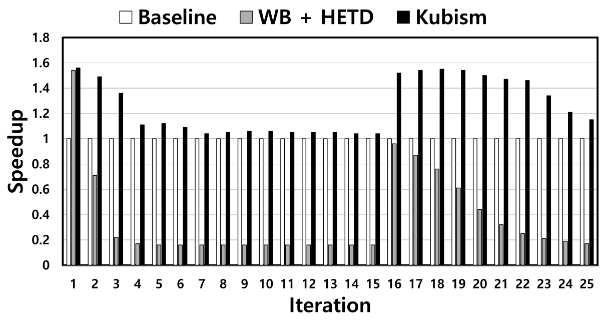

Figure 6. Speedup for all iterations in (0.5M, 64) dataset.

compared to the baseline across all datasets in geometric mean. In the (1M, 256) dataset, Kubism achieves the highest speedup, as the dataset exhibits the highest average skip ratio. While the baseline suffers from warp divergence in (1M, 256) dataset, Kubism significantly improves the performance by employing the warp balancing technique. We also observe that Kubism achieves high performance improvements for large datasets, such as (1M, 256), (2M, 128), and (2M, 256). As we mentioned in Section 3, the software overhead of warp balancing is proportional to the number of data points, not the feature dimension. Hence, processing large datasets exhibits a long execution time with relatively small overhead. In contrast, in small datasets such as (0.5M, 64), the overhead introduced by warp balancing is heavy. Kubism does not apply warp balancing to avoid such overheads.

To further analyze the effects of dataset size, we demonstrate the results for all iterations of (0.5M, 64) and (2M, 256) datasets. Figure 6 shows the speedup of WB+HETD and Kubism over the baseline across all iterations on the (0.5M, 64) dataset. Kubism achieves up to 1.56× speedup in high skip ratio iterations and 1.09× in low skip ratio ones. With the low feature dimensions, WB+HETD shows relatively poor performance due to the overhead from warp balancing. In iterations 4 to 15, WB+HETD shows over 80% performance degradation, despite applying HETD. Kubism improves the performance in those iterations by dynamically employing heterogeneous computing only.

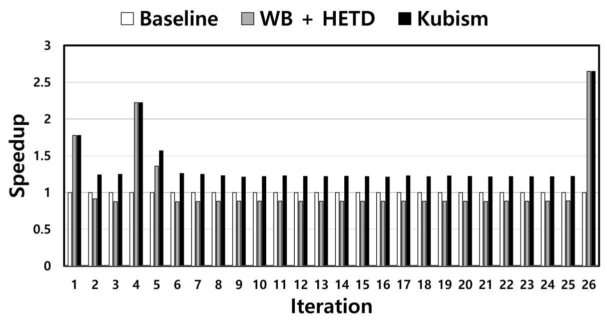

Figure 7. Speedup for all iterations in (2M, 256) dataset.

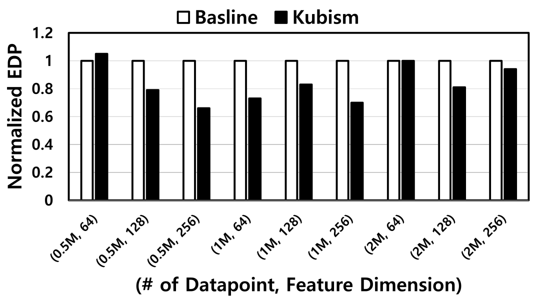

Figure 8. End-to-end normalized EDP for all datasets.

Figure 7 shows the speedup of WB+HETD and Kubism over the baseline across all iterations on the (2M, 256) dataset. On a large dataset, Kubism achieves a higher speedup than the baseline. Kubism achieves up to 2.65× speedup in high skip ratio iterations and 1.2× in low skip ratio ones. As expected, the overhead of warp balancing is much smaller than that shown in Figure 6. Also, the performance improvement of WB+HETD is larger than that in (0.5M, 64). Despite the effectiveness of warp balancing, Kubism applies warp balancing only for four iterations because the dataset exhibits a low average skip ratio. The advantage of Kubism can be increased for datasets with higher skip ratios. These results show that Kubism delivers consistent improvements across various datasets, making it an effective solution for accelerating K-means clustering on heterogeneous platforms.

4.3 Energy Efficiency

We measure the energy efficiency of Kubism compared to the baseline by using the energy-delay-product (EDP). Figure 8 depicts the normalized EDP results. Kubism achieves EDP reductions in 7 out of 9 datasets, showing a 17% EDP reduction on average. Kubism reduces the execution time by 17%, while the total energy consumption of Kubism remains similar to the baseline. CPU power consumption increases by 2.78× due to HETD, which accounts for 5-20% of the total

Table 5. Number of active threads for the baseline and Kubism. We use (1 M, 128) dataset and sample five iterations to maintain consistency with the results in Table 2.

Iteration	Skip Ratio	Number of Active Threads per Warp	
		Baseline	Kubism
1	0.91	2.84	9.06
2	0.60	12.76	16.02
3	0.24	24.16	24.04
4	0.06	29.96	29.90
5	0.01	31.58	31.56

power usage in the baseline. However, GPU power consumption stays almost the same as the baseline since threads in a warp still consume power even if some are idle in the baseline. Overall, Kubism consumes about 18% more power than the baseline, but the faster execution time offsets this increase, resulting in EDP reduction. By dynamically applying warp balancing and HETD, Kubism reduces execution time without excessive energy consumption.

4.4 GPU Behavior

We perform a detailed analysis to see how effectively Kubism reduces warp divergence through warp balancing. Using Nsight Compute [25], we profile the average active thread per warp metric during execution.

Table 5 shows the experimental results of the active threads per warp across iterations from 1 to 5 on the (1M, 128) dataset. For the first iteration, which shows a high skip ratio of 0.91, Kubism achieves a 3.2× improvement in the number of active threads per warp compared to the baseline. With the highest skip ratio among all iterations, the gap between Kubism and the baseline is the largest. On iteration 2, Kubism achieves a 1.3× improvement in the active threads per warp compared to the baseline. We observe that atomic instructions are executed not only in the local filter kernel but also in other kernels. Also, some threads in the local filter kernel exhibit an uneven distribution of computations, even if the proposed warp balancing is applied. So, there remains a gap between the ideal number of active threads (i.e., 32) and the actual experimental results. Considering this aspect, our experimental results show that Kubism effectively mitigates warp divergence. For iterations 3, 4, and 5, the skip ratios are low, which means all threads in the warp remain busy for computation. Hence, there is no difference between Kubism and the baseline since these iterations do not cause warp divergence due to low skip ratios.

5 Related Work

K-means clustering algorithms: Prior work has proposed diverse K-means clustering algorithms based on the Lloyd K-means clustering algorithm [23]. While Lloyd K-means clustering algorithm is simple and effective, the required number of operations scales linearly with the dataset size and the number of clusters. To reduce the number of computations, prior work has proposed K-means clustering algorithms using bound values for skipping redundant distance calculations [11–14, 24]. The algorithm is similar to the local filter explained in Section 2. Drake et al. and Hamerly et al. have proposed algorithms that keep fewer bound values for each datapoint than Elkan et al. to reduce the overhead of bound checking and maintaining the values [12, 14]. Ding et al. have proposed the Yinyang K-means clustering algorithm. Yinyang K-means clustering uses group filters to skip a large number of computations before applying the local filter [11].

Hardware acceleration of K-means clustering: To further acceleration for K-means clustering, architectural studies have been proposed [18, 33, 35]. Zhou et al. accelerated the Yinyang K-means clustering algorithm on ARM-based CPUs through vectorization, along with memory and data layout optimization [35]. However, since the K-means clustering algorithm exhibits significant parallelization opportunities, relying solely on CPUs is not optimal.

Prior work has proposed GPU-based K-means clustering acceleration techniques. Krulis et al. have introduced a comprehensive analysis of CUDA K-means clustering algorithms, focusing on memory behavior and load balancing for threads [18]. However, they have not explored how the filtering process affects the warp divergence. Taylor et al. have proposed accelerating the Yinyang K-means clustering algorithm on GPUs by leveraging the single instruction multiple threads (SIMT) architecture [33]. Despite the parallelization efforts, avoiding distance calculations can lead to warp divergence and result in performance degradation. Unlike prior work, we propose a heterogeneous platform-based Yinyang K-means clustering acceleration technique.

6 Conclusion

In this paper, we propose Kubism, a novel software approach to optimizing the Yinyang K-means clustering algorithm on mobile heterogeneous platforms. Prior work on GPU-accelerated suffers from inefficient GPU hardware resource underutilization due to warp divergence and does not utilize CPUs. Kubism addresses these challenges by reordering operations, applying warp balancing, using dynamic skip ratio decisions, and distributing tasks between the CPU and GPU. Our evaluation shows the potential of Kubism to enhance K-means clustering efficiency on mobile platforms.

Acknowledgments

We thank anonymous reviewers for their invaluable feedback. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2022R1C1C1011021). Yunho Oh is the corresponding author.

References

[1] Tarek S Abdelrahman. 2020. Cooperative software-hardware acceleration of K-means on a tightly coupled CPU-FPGA system. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 3 (2020), 1–24. https://doi.org/10.1145/3406114

[2] Hafsa Kara Achira, Camélia Slimani, and Jalil Boukhobza. 2023. Training K-means on Embedded Devices: a Deadline-aware and Energy Efficient Design. In *2023 31st International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 1–8. https://doi.org/10.1109/MASCOTS59514.2023.10387589

[3] Mohiuddin Ahmed, Raihan Seraj, and Syed Mohammed Shamsul Islam. 2020. The k-means algorithm: A comprehensive survey and performance evaluation. *Electronics* 9, 8 (2020), 1295. https://doi.org/10.3390/electronics9081295

[4] Bilel Ben Ali and Youssef Massmoudi. 2013. K-means clustering based on gower similarity coefficient: A comparative study. In *2013 5th International conference on modeling, simulation and applied optimization (ICMSAO)*. IEEE, 1–5. https://doi.org/10.1109/ICMSAO.2013.6552669

[5] Janki Bhimani, Miriam Leeser, and Ningfang Mi. 2015. Accelerating K-Means clustering with parallel implementations and GPU computing. In *2015 IEEE high performance extreme computing conference (HPEC)*. IEEE, 1–6. https://doi.org/10.1109/HPEC.2015.7322467

[6] Yingfeng Cai, Tianyu Luan, Hongbo Gao, Hai Wang, Long Chen, Yicheng Li, Miguel Angel Sotelo, and Zhixiong Li. 2021. YOLOv4-5D: An effective and efficient object detector for autonomous driving. *IEEE Transactions on Instrumentation and Measurement* 70 (2021), 1–13. https://doi.org/10.1109/TIM.2021.3065438

[7] Ming-Syan Chen, Jiawei Han, and Philip S. Yu. 1996. Data mining: an overview from a database perspective. *IEEE Transactions on Knowledge and data Engineering* 8, 6 (1996), 866–883. https://doi.org/10.1109/69.553155

[8] Xu Chen and Changwei Luo. 2021. Real-time lane detection based on a light-weight model in the wild. In *2021 IEEE 4th International Conference on Computer and Communication Engineering Technology (CCET)*. IEEE, 36–40. https://doi.org/10.1109/CCET52649.2021.9544226

[9] Radha Chitta, Rong Jin, Timothy C Havens, and Anil K Jain. 2011. Approximate kernel k-means: Solution to large scale kernel clustering. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 895–903. https://doi.org/10.1145/2020408.2020558

[10] M Deepa, S Soundarya, S Reshma, B Aakash, T Srivarsa, and R Vidhyapriya. 2023. FPGA Implementation of Yinyang K-Means Clustering. In *2023 IEEE 20th India Council International Conference (INDICON)*. IEEE, 219–224. https://doi.org/10.1109/INDICON59947.2023.10440752

[11] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup. In *International conference on machine learning*. PMLR, 579–587.

[12] Jonathan Drake and Greg Hamerly. 2012. Accelerated k-means with adaptive distance bounds. In *5th NIPS workshop on optimization for machine learning*, Vol. 8. 1–4.

[13] Charles Elkan. 2003. Using the triangle inequality to accelerate k-means. In *Proceedings of the 20th international conference on Machine Learning (ICML-03)*. 147–153.

[14] Greg Hamerly. 2010. Making k-means even faster. In *Proceedings of the 2010 SIAM international conference on data mining*. SIAM, 130–140. https://doi.org/10.1137/1.9781611972801.12

[15] Md Tayeb Himel, Mohammed Nazim Uddin, Mohammad Arif Hossain, and Yeong Min Jang. 2017. Weight based movie recommendation system using K-means algorithm. In *2017 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 1302–1306. https://doi.org/10.1109/ICTC.2017.8190928

[16] Tapas Kanungo, David M Mount, Nathan S Netanyahu, Christine D Piatko, Ruth Silverman, and Angela Y Wu. 2002. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE transactions on pattern analysis and machine intelligence* 24, 7 (2002), 881–892. https://doi.org/10.1109/TPAMI.2002.1017616

[17] Yuto Kitagawa, Tasuku Ishigoka, and Takuya Azumi. 2017. Anomaly prediction based on k-means clustering for memory-constrained embedded devices. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 26–33. https://doi.org/10.1109/ICMLA.2017.0-182

[18] Martin Kruliš and Miroslav Kratochvíl. 2020. Detailed analysis and optimization of CUDA k-means algorithm. In *Proceedings of the 49th International Conference on Parallel Processing*. 1–11. https://doi.org/10.1145/3404397.3404426

[19] Haobin Li and Yi Zhang. 2021. Vehicle Flow Detection Based on Improved Deep Structure and Deep Sort. In *Proceedings of the 2021 6th International Conference on Multimedia and Image Processing*. 72–77. https://doi.org/10.1145/3449388.3449394

[20] Yujie Li, Shuo Yang, Yuchao Zheng, and Huimin Lu. 2021. Improved point-voxel region convolutional neural network: 3D object detectors for autonomous driving. *IEEE Transactions on Intelligent Transportation Systems* 23, 7 (2021), 9311–9317. https://doi.org/10.1109/TITS.2021.3071790

[21] Dongfang Liu, Yaqin Wang, Tian Chen, and Eric T Matson. 2019. Application of color filter adjustment and k-means clustering method in lane detection for self-driving cars. In *2019 Third IEEE international conference on robotic computing (IRC)*. IEEE, 153–158. https://doi.org/10.1109/IRC.2019.00030

[22] Rui Liu, Xuanzhen Xu, Yuwei Shen, Armando Zhu, Chang Yu, Tianjian Chen, and Ye Zhang. 2024. Enhanced Detection Classification via Clustering SVM for Various Robot Collaboration Task. In *2024 6th International Conference on Communications, Information System and Computer Engineering (CISCE)*. 1121–1125. https://doi.org/10.1109/CISCE62493.2024.10653146

[23] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137. https://doi.org/10.1109/TIT.1982.1056489

[24] James Newling and François Fleuret. 2016. Fast k-means with accurate bounds. In *International Conference on Machine Learning*. PMLR, 936–944.

[25] Nvidia. 2024. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute

[26] Nvidia. 2024. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems Accessed: Oct. 3, 2024.

[27] Nvidia Corporation. 2022. NVIDIA Jetson AGX Orin Series: A Giant Leap Forward for Robotics and Edge AI Applications.

[28] Nvidia Corporation. 2024. Tegrastats Utility.

[29] Hermawan Prasetyo and Ayu Purwarianti. 2014. Comparison of distance and dissimilarity measures for clustering data with mix attribute types. In *2014 The 1st International Conference on Information Technology, Computer, and Electrical Engineering*. IEEE, 276–280. https://doi.org/10.1109/ICITACEE.2014.7065756

[30] David Sculley. 2010. Web-scale k-means clustering. In *Proceedings of the 19th international conference on World wide web*. 1177–1178. https://doi.org/10.1145/1772690.1772862

[31] Michael Shindler, Alex Wong, and Adam Meyerson. 2011. Fast and accurate k-means for large datasets. *Advances in neural information processing systems* 24 (2011).

[32] Camélia Slimani, Stéphane Rubini, and Jalil Boukhobza. 2019. K-MLIO: enabling k-means for large data-sets and memory constrained embedded systems. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 262–268. https://doi.org/10.1109/MASCOTS.2019.00037

[33] Colin Taylor and Michael Gowanlock. 2021. Accelerating the yinyang k-means algorithm using the GPU. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1835–1840. https://doi.org/10.1109/ICDE51399.2021.00163

[34] Teng Yu, Wenlai Zhao, Pan Liu, Vladimir Janjic, Xiaohan Yan, Shicai Wang, Haohuan Fu, Guangwen Yang, and John Thomson. 2019. Large-scale automatic k-means clustering for heterogeneous many-core supercomputer. *IEEE Transactions on Parallel and Distributed Systems* 31, 5 (2019), 997–1008. https://doi.org/10.1109/TPDS.2019.2955467

[35] Tianyang Zhou, Qinglin Wang, Shangfei Yin, Ruochen Hao, and Jie Liu. 2022. Optimizing Yinyang K-Means Algorithm on ARMv8 Many-Core CPUs. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 676–690. https://doi.org/10.1007/978-3-031-22677-9_36

Received 2025-03-18; accepted 2025-04-21