Flash-KMeans: Fast and Memory-Efficient Exact K-Means

Shuo Yang* **Haocheng Xi*** **Yilong Zhao** **Muyang Li** **Xiaoze Fan** **Jintao Zhang**
Han Cai **Yujun Lin** **Xiuyu Li** **Kurt Keutzer** **Song Han** **Chenfeng Xu** **Ion Stoica**
University of California, Berkeley MIT NVIDIA The University of Texas at Austin
andy_yang@berkeley.edu, xuchenfeng@utexas.edu
Code: https://github.com/svg-project/flash-kmeans

Abstract

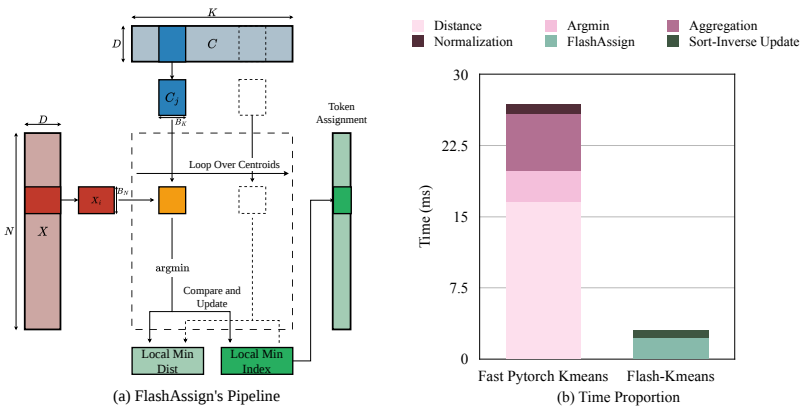
k-means has historically been positioned primarily as an offline processing primitive, typically used for dataset organization or embedding preprocessing rather than as a first-class component in online systems. In this work, we revisit this classical algorithm under the lens of modern AI system design and enable *k*-means as an online primitive. We point out that existing GPU implementations of *k*-means remain fundamentally bottlenecked by low-level system constraints rather than theoretical algorithmic complexity. Specifically, the assignment stage suffers from a severe IO bottleneck due to the massive explicit materialization of the $N \times K$ distance matrix in High Bandwidth Memory (HBM). Simultaneously, the centroid update stage is heavily penalized by hardware-level atomic write contention caused by irregular, scatter-style token aggregations. To bridge this performance gap, we propose flash-kmeans, an IO-aware and contention-free *k*-means implementation for modern GPU workloads. Flash-kmeans introduces two core kernel-level innovations: (1) *FlashAssign*, which fuses distance computation with an online argmin to completely bypass intermediate memory materialization; (2) *sort-inverse update*, which explicitly constructs an inverse mapping to transform high-contention atomic scatters into high-bandwidth, segment-level localized reductions. Furthermore, we integrate algorithm-system co-designs, including chunked stream overlap and cache-aware compile heuristic, to ensure practical deployability. Extensive evaluations on NVIDIA H200 GPUs demonstrate that flash-kmeans achieves up to **17.9**× end-to-end speedup over best baselines. At the kernel level, FlashAssign and Sort-Inverse Update deliver up to **21.2**× and **6.3**× speedups. Inside real pipelines, including IVF index construction and video diffusion, flash-kmeans accelerates the *k*-means stage by 5.3–10.7× over NVIDIA cuVS. By systematically restructuring execution around underlying hardware constraints, flash-kmeans delivers mathematically exact, scalable, and highly deployable acceleration across diverse AI workloads.

1 Introduction

k-means [1, 2] is one of the most classical and widely used clustering algorithms, and has long been treated as a mature, general-purpose component. In traditional settings, *k*-means typically appears as part of offline data processing pipelines [3], and most optimizations focus on the algorithmic level, such as faster convergence, fewer distance evaluations, or better approximations [4, 5, 6]. Although these algorithmic improvements successfully reduce theoretical FLOPs, many of them fail to translate into real end-to-end speedups on modern GPUs. This discrepancy arises because traditional optimizations are often incompatible with modern hardware design principles: dense compute

*Equal contribution.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).



(a) FlashAssign's Pipeline (b) Time Proportion

Figure 1: Overview of flash-kmeans and Performance Breakdown. (a) Inspired by IO-aware attention mechanisms, FlashAssign streams data blocks from HBM to SRAM, fusing distance computation with an online argmin operator to completely bypass the materialization of the massive $N \times K$ distance matrix. (b) Compared to standard *k*-means implementations, flash-kmeans drastically compresses both the assignment IO bottleneck and the update synchronization bottleneck.

is exceedingly cheap [7], while frequent, fragmented memory operations incur massive overheads. Consequently, lowering FLOPs does not necessarily correlate with wall-clock time reduction, and heavily overlooks the dominant constraints of memory bandwidth and data movement [8, 9].

This performance gap caused by implementation-level factors is significantly amplified in modern AI workloads. *k*-means usage is undergoing a clear paradigm shift, evolving from an offline analytics tool into a high-frequency, online operator within training and inference pipelines (e.g., vector quantization and sparse routing) [10, 11, 12]. This shift manifests in three dimensions: (1) execution moves from CPUs to GPUs, creating clear potential to exploit hardware accelerators; (2) usage moves from offline to online invocation, making low latency per invocation strictly essential; and (3) computation moves to longer sequences and larger batches, making throughput and scalability crucial requirements.

To understand why standard implementations fail to meet these new requirements, we must first define the *k*-means workload. A standard Lloyd's iteration consists of two distinct stages: an *assignment stage*, which computes the distance between every data point and all centroids to find the nearest cluster, and a *centroid update stage*, which aggregates the features of all points assigned to each cluster to form the new centroids.

When profiling this standard pipeline on modern GPUs [13, 14, 15], we find that the end-to-end performance is severely bottlenecked by kernel-level dataflow and write-path serialization. Concretely, this performance gap is driven by three primary challenges:

1. **IO-bound assignment in assignment stage:** Consider a workload consisting of N data points in d-dimensional space, clustered into K centroids. In each iteration, the assignment step compares every point against all K centroids, resulting in a computational cost of $O(NKd)$. Standard implementations execute this by first computing distances and then applying $\arg\min$, which explicitly materializes a massive distance matrix $D \in \mathbb{R}^{N \times K}$. Writing D to High Bandwidth Memory (HBM) and immediately reading it back incurs immense memory traffic.
2. **Atomic write contention in update stage:** The centroid update stage requires aggregating data grouped by cluster indices. Standard implementations typically perform scatter-style updates, where each thread atomically accumulates its data into a shared sum and count buffer based on its cluster id. Consequently, many threads concurrently attempt to update the same centroid, causing severe atomic contention and hardware-level serialization.
3. **System-level constraints:** Beyond kernel-level bottlenecks, real-world deployments face severe system constraints. When the input batch cannot reside entirely in VRAM, chunk-wise execution introduces heavy host-to-device communication overheads. Furthermore,

modern AI workloads often exhibit dynamic shapes, which amplify compilation and tuning costs, drastically increasing the time-to-first-run and hurting practical usability.

To bridge this performance gap, we propose flash-kmeans, an efficient implementation of *k*-means designed for modern GPU workloads. Flash-kmeans does not alter the mathematical formulation of the standard Lloyd *k*-means, nor does it introduce approximations. Instead, it restructures the assignment stage, the centroid update stage, and the system execution path around the three hardware bottlenecks discussed above.

Concretely, we introduce three key techniques. First, to eliminate distance materialization, we propose *FlashAssign* (Figure 1a), which fuses streaming distance computation with an online argmin procedure. This avoids constructing the $N \times K$ intermediate matrix entirely. Second, to resolve atomic contention, we introduce *sort-inverse update* (Figure 2b), which explicitly sorts the assignment vector by cluster id and performs aggregations in this sorted logical order. This rewrites high-contention, per-token atomic scatters into highly regular, segment-level localized merges. Finally, we introduce a set of *algorithm-system co-design* optimizations, including chunked stream overlap to hide PCIe communication overhead for large-scale execution, and a cache-aware compile heuristic to reliably match near-optimal configurations without expensive tuning, solving the dynamic deployment challenge.

Online argmin and sorted segmented reduction are established primitives [8, 16, 17]; our contribution is the *k*-means-specific dataflow around them. Fusion that leaves the running minimum in global memory saves capacity but not time (§ 5.3), so flash-kmeans keeps both reductions on chip and passes only a compact label from FlashAssign to Sort-Inverse Update.

We prototype flash-kmeans with customized GPU kernels and systematically evaluate it on an NVIDIA H200 GPU. We first conduct a kernel-level breakdown, revealing that FlashAssign speeds up the assignment kernel by up to **21.2**×, and Sort-Inverse Update accelerates the centroid update kernel by up to **6.3**×. Scaling to end-to-end iterations, we sweep across various numbers of points N, clusters K, feature dimensions d, and batch sizes B. flash-kmeans consistently achieves state-of-the-art performance across all representative workload regimes. As shown in Figure 1b, it outperforms the strongest baseline by up to **17.9**×. It also delivers massive speedups of up to **33**× and over **200**× against widely adopted industry-standard libraries like NVIDIA `cuML` [13] and `FAISS` [14], respectively. Together, these results demonstrate that flash-kmeans delivers not only theoretical kernel efficiency, but also stable, highly deployable end-to-end acceleration for modern AI workloads.

2 Related Work

The evolution of K-Means workloads. Traditionally, *k*-means was deployed in offline data mining and classical computer vision [1, 2]. However, modern AI pipelines have transformed it into a high-frequency, online primitive across diverse domains. In large-scale data processing and retrieval, *k*-means is heavily utilized for web-scale semantic deduplication [18] and embedding quantization for late-interaction search [19, 20, 21]. In Large Language Models (LLMs), online clustering is critical for context scaling, enabling dynamic token routing for sparse attention [10, 22, 11] and semantic state merging for KV cache compression [12, 23]. More recently, this paradigm has extended to state-of-the-art generative video models, where batched *k*-means is invoked repeatedly during forward passes to perform semantic-aware token permutation in Diffusion Transformers [24, 25] and extreme low-bit KV-cache quantization for auto-regressive generation [26]. Consequently, the primary evaluation metric for *k*-means has gradually shifted from offline analytical throughput to strict online invocation latency.

Algorithmic optimizations for K-Means. To accelerate the clustering process, prior research has focused on reducing the theoretical computational complexity. Methods leveraging the triangle inequality safely skip redundant distance calculations [4, 5]. Other approaches shrink the effective dataset size using summarization or sampling to estimate centroid updates [6, 3]. Recent theoretical literature also continues to explore dual-distance metrics to improve mathematical convergence [27]. While these algorithmic innovations successfully decrease arithmetic operations, standard GPU implementations of *k*-means remain fundamentally IO-bound, making implementation-level optimization a critical prerequisite to fully translating algorithmic FLOP reductions into end-to-end wall-clock speedups.

GPU K-Means libraries. Exact GPU Lloyd implementations include `cuML` [13] (built on RAFT and cuVS [28, 17]), `FAISS` [14], `fast_pytorch_kmeans` [29], and `fastkmeans` [15]. cuVS avoids distance materialization with a fused kernel (`fusedL2NN`) but keeps each point's running minimum in global memory under locks and updates centroids with atomic scatters; `FAISS` reuses generic nearest-neighbor search and updates centroids on the CPU.

Hardware-aware and IO-optimized ML primitives. Recent systems research achieves massive performance breakthroughs by optimizing operator data paths and memory hierarchies rather than altering the underlying mathematical formulas. The most prominent example is FlashAttention [8, 9, 30], which fuses attention computation to entirely bypass the explicit materialization of the massive $N \times N$ attention matrix in HBM. This IO-aware philosophy has rapidly expanded to yield heavily optimized primitives for dynamic LLM serving [31] and memory management [32]. Furthermore, in handling highly dynamic and skewed data distributions, systems frequently transform irregular scatter writes into sorting followed by regular segmented operations to eliminate severe atomic contention [16, 33]. These works collectively demonstrate that restructuring an operator's memory dataflow to respect hardware synchronization and bandwidth limits is the most effective path to accelerating modern AI primitives.

3 Preliminary and Motivation

3.1 Lloyd's Algorithm and Standard GPU Implementation

Given data points $X \in \mathbb{R}^{N \times d}$ and centroids $C \in \mathbb{R}^{K \times d}$, Euclidean *k*-means minimizes the objective:

$$\min_{a \in \{1,\dots,K\}^N,\, C} \sum_{i=1}^{N} \|x_i - c_{a_i}\|_2^2 \tag{1}$$

Lloyd's algorithm solves this by alternating between two stages. The first is the **assignment stage**, which computes a distance matrix $D \in \mathbb{R}^{N \times K}$ and assigns each point to the nearest centroid:

$$D_{ik} = \|x_i - c_k\|_2^2, \qquad a_i = \arg\min_k D_{ik} \tag{2}$$

The second is the **centroid update stage**, which aggregates the points assigned to each cluster to compute the new centroids:

$$n_k = \sum_{i=1}^{N} \mathbb{I}[a_i = k], \quad s_k = \sum_{i=1}^{N} \mathbb{I}[a_i = k]\, x_i, \quad c_k \leftarrow \frac{s_k}{n_k} \tag{3}$$

In practice, the squared distance in the assignment stage is expanded as $\|x_i - c_k\|_2^2 = \|x_i\|_2^2 + \|c_k\|_2^2 - 2x_i^\top c_k$, which enables the reuse of norms and maps the computation to high-throughput matrix multiplications.

Many existing GPU libraries follow a direct translation of these equations, as outlined in Algorithm 1. It first computes and writes the massive distance matrix D to High Bandwidth Memory (HBM), then reads it back for row-wise reduction. Finally, it performs scatter-style atomic additions at the token granularity to accumulate cluster statistics.

3.2 Kernel-Level Bottlenecks in Standard Implementation

While the standard implementation is mathematically straightforward, our profiling reveals that its end-to-end performance on modern GPUs is severely limited by two low-level bottlenecks: the memory wall in the assignment stage and atomic serialization in the update stage.

Distance materialization bottlenecks the assignment stage. The key inefficiency in the assignment stage is the explicit materialization of the distance matrix $D \in \mathbb{R}^{N \times K}$. The matrix D is a short-lived intermediate that is written to HBM by the distance kernel and immediately read back by the assignment kernel. This dataflow introduces at least $\Theta(NK)$ writes plus $\Theta(NK)$ reads per iteration, which heavily dominates the total memory traffic when N and K are large.

Algorithm 1: Standard k-means implementation (one iteration)

Input: Data $X \in \mathbb{R}^{N \times d}$, Centroids $C \in \mathbb{R}^{K \times d}$ resident in HBM
Output: Assignments $a \in \{1, \ldots, K\}^N$, Updated Centroids C_{new}

`// Stage 1: Assignment`
Kernel 1 (Distance): Load blocks of X and C from HBM;
Compute $D_{ik} = \|x_i\|_2^2 + \|c_k\|_2^2 - 2x_i^\top c_k$;
Write full matrix $D \in \mathbb{R}^{N \times K}$ to HBM ; `// Massive intermediate materialization`

Kernel 2 (Argmin): Load row D_{i*} from HBM, Compute $a_i = \arg\min_k D_{ik}$ for each point i;
Write assignments $a \in \{1, \ldots, K\}^N$ to HBM;

`// Stage 2: Centroid Update`
Kernel 3 (Aggregation): Initialize sum $s \leftarrow 0$, count $n \leftarrow 0$ in HBM;
foreach *point index* $i \in \{1, \ldots, N\}$ *in parallel* **do**
  Read feature x_i and assignment a_i from HBM;
  `atomic_add(` s_{a_i}, x_i `) ;` `// Severe atomic contention on hot clusters`
  `atomic_add(` $n_{a_i}, 1$ `);`

Kernel 4 (Normalization): **Read** s, n from HBM;
Compute $c_k = s_k / n_k$ for each k, and Write C_{new} to HBM;

In contrast, reading the inputs X and C only costs $\Theta(Nd + Kd)$, and writing the outputs costs $\Theta(N + Kd)$. Consequently, even if the distance computation utilizes high-throughput primitives, the end-to-end assignment latency is fundamentally bandwidth-limited by the $2 \cdot \Theta(NK)$ HBM round trips required to materialize D.

Scatter-style updates cause severe atomic write contention. In the centroid update stage, the standard implementation relies on a token-to-cluster aggregation view. Each Cooperative Thread Array (CTA) processes a contiguous block of tokens in their original order, reading the feature x_i and its assigned cluster id a_i, and then issues scatter-style atomic adds to update the global sum and count buffers.

While this implicitly achieves the necessary gather operation, the write path is highly inefficient. Local token blocks often contain repeated and interleaved cluster ids. As a result, multiple warps frequently attempt to update the same centroid simultaneously, leading to irregular and highly concentrated write destinations. This causes severe atomic contention, serialization, and cache-line thrashing. Since atomics are issued at token granularity, the number of atomic operations scales as $O(Nd)$. In our profiling on an NVIDIA H200 GPU, this write-path contention restricts the standard update implementation to only 50 GB/s effective bandwidth, drastically falling short of the hardware's theoretical reduction bandwidth limits.

3.3 System-Level Constraints in Real Deployments

Beyond the core kernel inefficiencies, *k*-means faces additional system-level constraints when deployed in modern AI workloads. First, these workloads are frequently invoked with extremely large batch sizes, causing the input data to exceed the GPU's VRAM capacity. Processing the data in chunks can reduce peak memory usage, but it introduces frequent Host-to-Device (CPU-to-GPU) communication and synchronization, shifting the bottleneck to the PCIe bandwidth. Second, real-world AI pipelines exhibit highly dynamic shapes—the number of points, clusters, and feature dimensions can change rapidly between invocations. Standard implementations that rely heavily on shape specialization require exhaustive auto-tuning to achieve optimal performance. Under dynamic workloads, this configuration search triggers frequent recompilations, significantly increasing the time-to-first-run and hindering practical usability in online scenarios.

4 Methodology

In this section, we introduce flash-kmeans, an efficient *k*-means implementation for modern GPU workloads. We first present FlashAssign in § 4.1, which fuses distance computation and reduction

to eliminate IO overheads. We then present sort-inverse update in § 4.2, which explicitly constructs an inverse mapping to rewrite high-contention atomic scatter into a regular segment-level aggregation. Finally, we discuss several algorithm-system co-design optimizations in § 4.3 to improve deployability in real systems.

4.1 FlashAssign: Materialization-Free Assignment via Online Argmin

To eliminate the severe HBM traffic caused by materializing the distance matrix $D \in \mathbb{R}^{N \times K}$ (§ 3.2), we propose *FlashAssign*. FlashAssign fuses the distance computation and row-wise reduction into a single streaming procedure, ensuring that the full $N \times K$ distance matrix is never explicitly constructed in memory.

Online argmin. FlashAssign relies on an online argmin update. For each point x_i, we maintain two running states in registers: the current minimum distance m_i and the corresponding centroid index a_i. We initialize $m_i = +\infty$ and $a_i = -1$, and then scan the centroids in tiles. For each centroid tile, the kernel computes local distances on chip, identifies the local minimum within that tile, and compares it with the running (m_i, a_i) to keep the smaller one. Repeating this update over all centroid tiles yields the exact global minimum.

Tiling and asynchronous prefetch. FlashAssign uses two-dimensional tiling over both the points and the centroids. For each tile, the kernel maintains the running (m, a) states on chip and scans centroid tiles sequentially. To hide memory latency, we implement double buffer and asynchronous prefetch, ensuring that loading the next centroid tile from HBM overlaps seamlessly with the distance computation of the current tile.

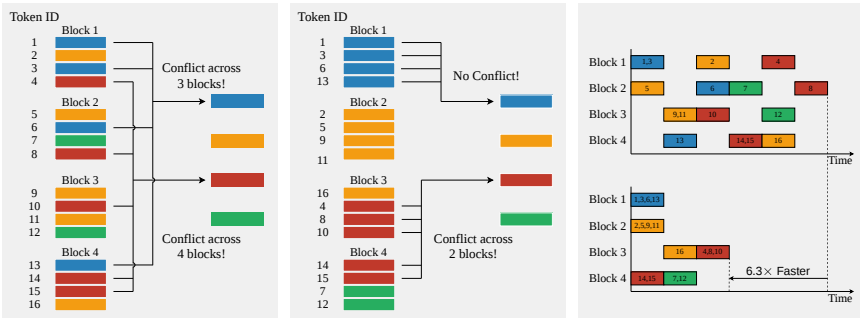
Algorithm 2: FlashAssign (materialization-free assignment)

Input: Data $X \in \mathbb{R}^{N \times d}$, centroids $C \in \mathbb{R}^{K \times d}$, point tile size B_N, centroid tile size B_K
Output: Assignments $a \in \{1, \ldots, K\}^N$
Precompute norms $\|x_i\|_2^2$;
foreach *point tile* X_{tile} *of size* B_N *in parallel* **do**
  Initialize on-chip running states: $m \leftarrow +\infty$, $a \leftarrow -1$;
  Prefetch the first centroid tile $C_{\text{tile}}^{(0)}$ from HBM into on-chip buffer;
  for $t \leftarrow 0$ **to** $\lceil K/B_K \rceil - 1$ **do**
    if $t + 1 < \lceil K/B_K \rceil$ **then**
      Prefetch $C_{\text{tile}}^{(t+1)}$ into the alternate buffer;
    Compute local distances between X_{tile} and $C_{\text{tile}}^{(t)}$ on chip;
    Compute tile-local minima and indices $(\tilde{m}, \tilde{a})$ for each point in X_{tile};
    Update running states using online argmin;
    $m \leftarrow \min(m, \tilde{m})$, and update a with the corresponding index;
    Swap buffers;
  Write final assignments a for X_{tile} to HBM;

IO complexity. By fusing distance computation and reduction, FlashAssign fundamentally changes the dataflow. Under ideal streaming execution, each assignment pass only needs to read the point matrix X and centroid matrix C once, and write the assignment vector a once. This reduces the dominant IO complexity from $O(NK)$ to $O(Nd + Kd)$, entirely removing the $2 \cdot \Theta(NK)$ HBM traffic penalty of standard implementations. Unlike GEMM-epilogue fusion, which keeps (m_i, a_i) in HBM and updates it under locks once per centroid tile, FlashAssign keeps this state in registers for the entire centroid sweep and writes only the final label.

4.2 Sort-Inverse Update: Low-Contention Centroid Aggregation

To resolve the severe write-side serialization in the centroid update stage (§ 3.2), we propose *sort-inverse update* as illustrated in Figure 2. Instead of performing scattered atomic add at the token granularity, we explicitly restructure the execution to enable localized, segment-level reductions.



(a) Scatter-style update (b) Sort-inverse update (c) Execution timeline

Figure 2: Illustration of the centroid update stage and atomic contention. **(a) Standard scatter-style update:** Tokens within the same cluster are scattered in multiple blocks. The highly irregular mapping leads to severe write-side atomic contention when multiple threads update the same centroid simultaneously. **(b) Sort-inverse update:** flash-kmeans sorts the tokens by their cluster IDs and constructs an inverse mapping. This transforms the unstructured scatter into regularized and localized reductions. **(c) Execution timeline comparison:** Standard approach stalls frequently due to atomic lock contention on HBM, whereas Sort-Inverse Update issues contention-free memory writes, significantly hiding latency and accelerating the reduction phase.

Explicit inverse mapping via argsort. The core idea is to transform the token-to-cluster update into a cluster-to-token gather. We first apply an `argsort` operation to the assignment vector a to obtain the permutation index `sorted_idx`, and then form the sorted cluster-id sequence a^{sorted}. In this sorted logical order, identical cluster ids naturally group together to form contiguous *cluster-id segments*. Note that this sorting is performed only on the 1D assignment vector a; we do *not* physically permute the heavy point matrix X in memory.

Segment-level localized aggregation. Once the inverse mapping is built, each CTA processes a contiguous chunk of the sorted sequence a^{sorted}. The CTA identifies the cluster-id segments boundaries inside its assigned chunk and uses `sorted_idx` to gather the corresponding token features from the original X matrix. Crucially, the CTA accumulates partial sums and counts entirely in fast on-chip memory (registers or shared memory) for each segment. It only issues global `atomic_add` operations to HBM at segment boundaries. By keeping the read-side gather from HBM but moving the reduction logic on chip, we effectively bypass the write-path bottleneck. Figure 2(b) provides an overview of this reorganization.

Algorithm 3: Sort-Inverse Update (low-atomic centroid update)

Input: Points $X \in \mathbb{R}^{N \times d}$ (original order), assignments $a \in \{1, \dots, K\}^N$, chunk size B_N
Output: Centroid sums $s \in \mathbb{R}^{K \times d}$, counts $n \in \mathbb{R}^K$, updated centroids C
Sort by cluster id: compute `sorted_idx`$\leftarrow$ `argsort`(a);
Construct sorted cluster ids $a^{\text{sorted}}[j] \leftarrow a[\texttt{sorted_idx}[j]]$;
Initialize $s \leftarrow 0$, $n \leftarrow 0$;
for $l \leftarrow 0$ **to** $N-1$ *step* B_N **do**
  $r \leftarrow \min(l + B_N, N)$;
  Load $a^{\text{sorted}}[l:r]$ and `sorted_idx`$[l:r]$;
  Identify contiguous segments of identical cluster ids in $a^{\text{sorted}}[l:r]$;
  for *each segment* (u, v, k) *in* $a^{\text{sorted}}[l:r]$ **do**
    Gather token features from original order using indices `sorted_idx`$[u:v]$;
    Accumulate local partial sum Δs_k and local partial count Δn_k on chip;
    Atomic merge (once per segment): `atomic_add`$(s_k, \Delta s_k)$ and `atomic_add`$(n_k, \Delta n_k)$;
for $k \leftarrow 1$ **to** K **do**
  $c_k \leftarrow s_k / n_k$;

Atomic add count analysis. Sort-inverse update drastically reduces the total number of atomic operations. In a standard scatter update, atomics are issued per token, scaling as $O(Nd)$. In our design, atomics are issued per contiguous cluster-id segment. After sorting, the sequence contains exactly K contiguous segments. Since the workload is partitioned into chunks of size B_N across CTAs, chunk boundaries may split a segment, adding at most $\lceil N/B_N \rceil$ extra boundaries. Therefore, the worst-case number of atomic merges drops to $O((K + \lceil N/B_N \rceil)d)$. This theoretical reduction directly translates to the elimination of write contention.

4.3 Efficient Algorithm-System Co-design

Large-scale data processing via chunked stream overlap. When the input cannot reside entirely in GPU memory and must be staged from CPU memory, we use a chunked stream overlap design. We partition the data into chunks and use CUDA streams to coordinate asynchronous host-to-device transfers and *k*-means computation. Each chunk is copied to the GPU with non-blocking transfers, then processed by assignment and centroid update kernels, while the next chunk is transferred in parallel, following a double-buffer streaming pattern.

Cache-aware compile heuristic for fast time-to-first-run. Configuration tuning in flash-kmeans compilation often introduces a large time-to-first-run overhead, especially under dynamic shapes and cross-hardware deployment. To address this issue, we design a cache-aware compile heuristic that selects high-quality kernel configurations directly from hardware characteristics (in particular, L1 and L2 cache sizes) and problem shape, instead of relying on expensive exhaustive tuning. The goal is not to perfectly tune every shape, but to reliably match near-best configurations at very low compilation cost.

5 Experiments

5.1 Experimental Setup

We evaluate flash-kmeans on an NVIDIA H200 GPU with CUDA 12.8. We benchmark Euclidean *k*-means over a comprehensive sweep of data sizes N, cluster counts K, feature dimensions D, and batch sizes B. We compare flash-kmeans against four heavily optimized baselines: `fast_pytorch_kmeans`, `fastkmeans` [15], NVIDIA `cuML` [13], and `FAISS` [14]. We report end-to-end latency, individual kernel latencies (assignment and centroid update), and time-to-first-run for dynamic shape deployments. We use `fast_pytorch_kmeans` [29] and `cuML` 26.02, add NVIDIA cuVS [17] in § 5.3, and label H100 and B200 results. All methods use FP32 and the same fixed number of iterations; end-to-end runs include random initialization and per-call setup, while per-iteration and kernel runs use identical pre-set centroids. We report the mean of 50 runs after 10 warm-ups (Appendix B.1).

5.2 Efficiency Evaluation

End-to-end speedup benchmark. Figure 3 compares the end-to-end latency per iteration across implementations under different workloads. For clarity, we group the results into three representative regimes: large N with large K, large N with small K, and small N with small K. In the memory-intensive large-N, large-K regime (e.g., $N = 1\text{M}, K = 64\text{K}, D = 512$), standard PyTorch implementations run out of memory due to the massive distance matrix, whereas flash-kmeans achieves the largest absolute speedup, outperforming the best baseline (`fastkmeans`) by over **5.4**×.

In the compute-intensive large-N, small-K regime (e.g., $N = 8\text{M}, K = 1024$), flash-kmeans remains the fastest, reducing end-to-end latency by **94.4%** (a **17.9**× speedup over `fast_pytorch_kmeans`). In the small-N, small-K regime, where kernel launch and framework overheads dominate, flash-kmeans still delivers consistent acceleration, scaling up to a **15.3**× speedup in highly batched scenarios (e.g., $B = 32$). Overall, flash-kmeans remains robust across batch sizes and feature dimensions, demonstrating that its IO-aware optimizations generalize across diverse shape configurations. The 33× (`cuML`) and over 200× (`FAISS`) speedups correspond to $N = 8\text{M}$, $K = 1024$, $D = 128$ in Figure 3; Appendix B.6 breaks them down.

Kernel-level efficiency breakdown. To isolate the sources of our end-to-end acceleration, we benchmark the two core *k*-means stages individually against highly optimized standard imple-

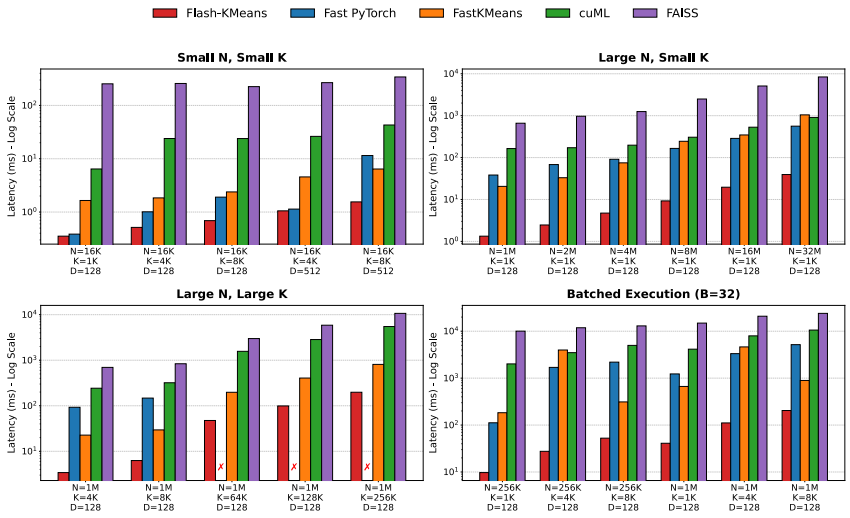


Figure 3: End-to-End Latency of flash-kmeans compared to standard baselines. We group the evaluation into four representative regimes. The y-axis is presented in log scale to accommodate magnitude differences. Red × marks denote Out-Of-Memory failures (e.g., standard PyTorch fails to explicitly materialize the $N \times K$ distance matrix in Large K workloads). flash-kmeans delivers consistent and substantial speedups across diverse shapes and handles extreme limits gracefully.

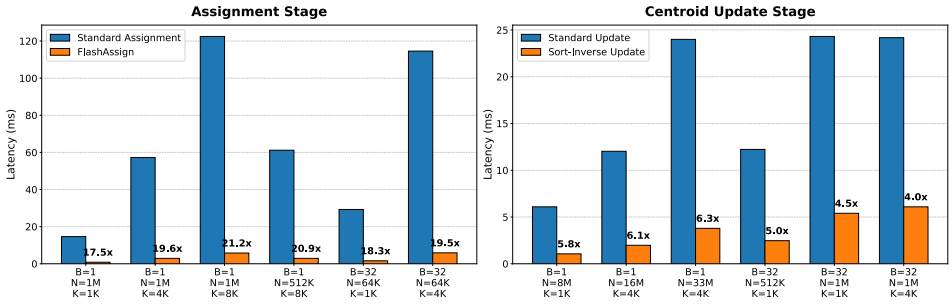


Figure 4: Kernel-level latency breakdown. Latency comparison of our custom kernels versus standard implementations across diverse extreme workloads ($D = 128$). **Left:** FlashAssign completely removes HBM distance materialization, scaling up to a 21.2× speedup. **Right:** Sort-Inverse Update replaces per-token scatter atomic adds with sorted localized merges, eliminating atomic contention and achieving up to a 6.3× speedup. Standard (H200): eager PyTorch materializing the FP32 $N \times K$ matrix before `argmin`; token-wise FP32 atomic scatter.

mentations. Figure 4 presents the absolute latency comparisons and relative speedups across six representative boundary configurations for each stage. On a highly demanding configuration ($N = 1\text{M}, K = 8192$), FlashAssign drastically reduces execution time from 122.5ms down to just 5.8ms, achieving up to a **21.2**× speedup over the standard assignment implementation. Concurrently, sort-inverse update systematically accelerates the reduction step, reaching up to a **6.3**× speedup on massive-scale workloads (e.g., $B = 1, N = 33\text{M}, K = 4096$). These microbenchmarks definitively confirm that flash-kmeans effectively neutralizes both the dominant IO limits and synchronization bottlenecks, aligning directly with our hardware analysis in § 3.2.

Factorial ablation. Table 1 isolates each technique. FlashAssign alone yields 9.89×. Sort-Inverse Update alone yields only 1.04× because the materializing assignment dominates, but once assignment is fast, it cuts the remaining latency from 12.44 ms to 7.34 ms, for **16.77**× in total. The two techniques are thus complementary.

Table 1: Factorial ablation (H200, $N = 8\text{M}$, $K = 1024$).

Assign	Update	ms	Speedup
Standard	Atomic	123.08	1.00×
FlashAssign	Atomic	12.44	9.89×
Standard	Sort-Inv.	117.97	1.04×
FlashAssign	**Sort-Inv.**	**7.34**	**16.77**×

Table 2: Speedup over NVIDIA cuVS (FP32, $D = 128$); pipeline rows swap only the *k*-means engine (Appendix B.5).

Level	Workload	GPU	cuVS (ms)	Ours (ms)	Speedup
Assignment	cuML/cuVS fused path, $N = 4\text{M}$, $K = 1000$	H200	29.64	5.47	5.4×
	`fusedL2NN` kernel, $N = 8\text{M}$, $K = 1024$	H100	51.8	10.6	4.9×
Iteration	KMeans `fit`, $N = 8\text{M}$, $K = 1024$	H100	66.1	12.4	5.3×
Pipeline	IVF-Flat index build, SIFT1M, nlist $= 4096$	H100	577	108	5.3×
	IVF-PQ index build, SIFT1M, $m = 16$	H100	684	122	5.6×
	PQ codebooks: one batched call vs. 16 fits	H100	98.2	9.2	10.7×
	Video DiT clustering, $B = 32$, $N = 75.6\text{K}$, $K = 1000$	H100	279.6	43.1	6.5×

Algorithm-system co-design. We defer the large-scale out-of-core and dynamic-shape co-design evaluation to Appendix A, where flash-kmeans scales to one billion points and reduces time-to-first-run by up to **175**× with negligible runtime loss. The heuristic is used in every experiment; the out-of-core pipeline is evaluated separately (6.3× over `fastkmeans` at $N = 10^9$).

5.3 Fused Baselines, Real Workloads, and Convergence

Comparison with fused baselines. cuVS [17, 28] fuses L2 distance with a 1-NN reduction (`fusedL2NN`) in a GEMM epilogue but keeps each running minimum in global memory under locks; at $N = 8\text{M}$, $K = 1024$ on H100, it is slower than an unfused GEMM plus `argmin` (51.8 versus 28.5 ms). FlashAssign keeps the state in registers, running 4.7–5.0× faster than `fusedL2NN` (Appendix B.4); per iteration, flash-kmeans is 4.4–5.3× faster than cuVS KMeans for $N = 8\text{M}$–33M, $K = 1\text{K}$–64K.

Real datasets and AI pipelines. Swapping the *k*-means engine inside real consumers (Table 2) speeds up IVF-Flat and IVF-PQ index construction on SIFT1M [34] by 5.3× and 5.6× with unchanged inertia, and cuts the per-step clustering of a 50-step video Diffusion Transformer generation [25] from about 14 s to 2 s. The SIFT1M gap matches that on same-shape synthetic data (5.6× versus 5.7×).

Robustness and convergence. Across N/K from 256 to 16,384 and D up to 1024, FlashAssign and Sort-Inverse Update are 3.95–19.65× and 2.45–6.28× faster (Appendix B.2); including `argsort`, Sort-Inverse Update is 4.23× faster at $N = 33\text{M}$ (Appendix B.3); on B200, flash-kmeans is 13.1–51.4× faster than `fast_pytorch_kmeans` (Appendix B.8). Over 100 trials, flash-kmeans converges in 24.79 ± 4.58 iterations versus 24.19 ± 4.04 for PyTorch Lloyd (WCSS gap 1.91×10^{-4}; Appendix B.7).

6 Conclusion & Limitation

In this paper, we introduced flash-kmeans, a highly optimized, IO-aware *k*-means implementation designed to overcome the severe memory and synchronization bottlenecks of modern GPU workloads. Rather than altering the underlying mathematics, we systematically restructured the execution dataflow through FlashAssign to eliminate massive distance matrix materialization, and Sort-Inverse Update to resolve write-side atomic contention. Coupled with an asynchronous out-of-core data pipeline and shape-aware compile heuristics, flash-kmeans delivers up to a **17.9**× end-to-end speedup over best baselines, and massively outperforms industry standards like `cuML` and `FAISS` by **33**× and over **200**×, respectively. By gracefully scaling to extreme workloads of one billion points and slashing compilation overheads by **175**× with near-zero performance degradation, flash-kmeans provides a robust, mathematically exact, and highly deployable clustering primitive for next-generation generative AI infrastructure. One limitation of our current study is that, although the proposed algorithm has the potential to be utilized by specialized hardware accelerators such as TPUs or AWS Trainium, we have not yet implemented this adaptation. We leave this hardware-specific optimization as an avenue for future work. Moreover, the FlashAssign advantage narrows at large D (3.95× at $D = 1024$), and `argsort` takes 24.5% of the update time at $N = 33\text{M}$.

Acknowledgments and Disclosure of Funding

Computing resources for this work were provided by the Berkeley Sky Computing Lab.

References

[1] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[2] J. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability - Vol. 1*, pages 281–297. University of California Press, Berkeley, CA, USA, 1967.

[3] D. Sculley. Web-scale k-means clustering. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 1177–1178, New York, NY, USA, 2010. Association for Computing Machinery.

[4] Charles Elkan. Using the triangle inequality to accelerate k-means. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, page 147–153. AAAI Press, 2003.

[5] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 579–587, Lille, France, 07–09 Jul 2015. PMLR.

[6] Olivier Bachem, Mario Lucic, and Andreas Krause. Scalable k-means clustering via lightweight coresets, 2018.

[7] NVIDIA. Nvidia a100 tensor core gpu. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf`, 2021.

[8] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[9] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023.

[10] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers, 2020.

[11] Kan Zhu, Tian Tang, Qinyu Xu, Yile Gu, Zhichen Zeng, Rohan Kadekodi, Liangyu Zhao, Ang Li, Arvind Krishnamurthy, and Baris Kasikci. Tactic: Adaptive sparse attention with clustering and distribution fitting for long-context llms, 2025.

[12] Guangda Liu, Chengwei Li, Jieru Zhao, Chenqi Zhang, and Minyi Guo. Clusterkv: Manipulating llm kv cache in semantic space for recallable compression, 2025.

[13] Sebastian Raschka, Joshua Patterson, and Corey Nolet. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *arXiv preprint arXiv:2002.04803*, 2020.

[14] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[15] Benjamin Clavié and Benjamin Warner. fastkmeans: Accelerated kmeans clustering in pytorch and triton. `https://github.com/AnswerDotAI/fastkmeans/`, 2025.

[16] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-decoding for long-context inference. `https://crfm.stanford.edu/2023/10/12/flashdecoding.html`, Oct 2023. Stanford CRFM Blog.

[17] RAPIDS Development Team. cuVS: Vector search and clustering on the GPU. `https://github.com/rapidsai/cuvs`, 2024.

[18] Amro Abbas, Kushal Tirumala, Dániel Simig, Surya Ganguli, and Ari S. Morcos. Semdedup: Data-efficient learning at web-scale through semantic deduplication, 2023.

[19] Omar Khattab and Matei Zaharia. Colbert: Efficient and effective passage search via contextualized late interaction over bert, 2020.

[20] Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. Colbertv2: Effective and efficient retrieval via lightweight late interaction, 2022.

[21] Keshav Santhanam, Omar Khattab, Christopher Potts, and Matei Zaharia. Plaid: An efficient engine for late interaction retrieval, 2022.

[22] Shuohang Wang, Luowei Zhou, Zhe Gan, Yen-Chun Chen, Yuwei Fang, Siqi Sun, Yu Cheng, and Jingjing Liu. Cluster-former: Clustering-based sparse transformer for long-range dependency encoding, 2021.

[23] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization, 2025.

[24] Haocheng Xi, Shuo Yang, Yilong Zhao, Chenfeng Xu, Muyang Li, Xiuyu Li, Yujun Lin, Han Cai, Jintao Zhang, Dacheng Li, Jianfei Chen, Ion Stoica, Kurt Keutzer, and Song Han. Sparse videogen: Accelerating video diffusion transformers with spatial-temporal sparsity, 2025.

[25] Shuo Yang, Haocheng Xi, Yilong Zhao, Muyang Li, Jintao Zhang, Han Cai, Yujun Lin, Xiuyu Li, Chenfeng Xu, Kelly Peng, Jianfei Chen, Song Han, Kurt Keutzer, and Ion Stoica. Sparse videogen2: Accelerate video generation with sparse attention via semantic-aware permutation, 2025.

[26] Haocheng Xi, Shuo Yang, Yilong Zhao, Muyang Li, Han Cai, Xingyang Li, Yujun Lin, Zhuoyang Zhang, Jintao Zhang, Xiuyu Li, Zhiying Xu, Jun Wu, Chenfeng Xu, Ion Stoica, Song Han, and Kurt Keutzer. Quant videogen: Auto-regressive long video generation via 2-bit kv-cache quantization, 2026.

[27] Naitik Gada. A novel k-means clustering approach using two distance measures for gaussian data, 2025.

[28] RAPIDS Development Team. RAFT: Reusable accelerated functions and tools for vector search and more. `https://github.com/rapidsai/raft`, 2022.

[29] DeMoriarty. fast_pytorch_kmeans: Fast pytorch implementation of k-means clustering. `https://github.com/DeMoriarty/fast_pytorch_kmeans`, 2020.

[30] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024.

[31] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving, 2025.

[32] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023.

[33] Wentao Guo, Mayank Mishra, Xinle Cheng, Ion Stoica, and Tri Dao. Sonicmoe: Accelerating moe with io and tile-aware optimizations, 2025.

[34] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[35] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1027–1035, 2007.

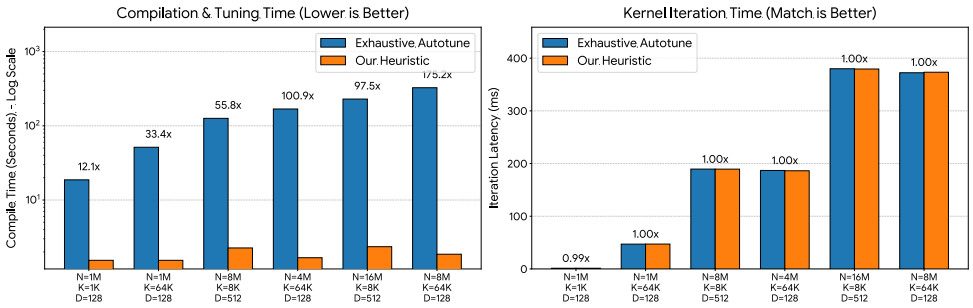


Figure 5: Effectiveness of the cache-aware compile heuristic. **Left:** Compilation and configuration search time (log scale). The heuristic slashes tuning overhead by up to 175×, entirely bypassing the severe compilation bottleneck of exhaustive auto-tuning. **Right:** Kernel iteration latency. The heuristic seamlessly matches the optimal performance found by exhaustive search across various shapes, ensuring near-zero runtime degradation.

A Algorithm-System Co-design Evaluation

Large-scale out-of-core data processing. To evaluate flash-kmeans in memory-constrained, massive-scale scenarios, we benchmark workloads where the dataset severely exceeds GPU VRAM capacity, scaling up to one billion points. Under these extreme conditions, standard PyTorch implementations immediately fail due to Out-Of-Memory errors. We therefore compare flash-kmeans against `fastkmeans`, the most robust out-of-core baseline available. On an extreme workload of one billion points ($N = 10^9, K = 32768, D = 128$), flash-kmeans completes an iteration in just **41.4 seconds**, whereas the baseline requires 261.8 seconds, yielding a **6.3**× speedup. Furthermore, on a configuration of $N = 400\text{M}$ and $K = 16384$, flash-kmeans achieves an exceptional **10.5**× end-to-end speedup (8.4s vs. 88.4s). These results demonstrate that our pipelined execution successfully bounds the peak GPU memory footprint while delivering order-of-magnitude acceleration for massive data processing.

Fast time-to-first-run for dynamic shapes. To evaluate our cache-aware compile heuristic, Figure 5 compares its compilation time and runtime performance against exhaustive tuning across six representative scale-up workloads. As shown in the left chart, exhaustive tuning time explodes as the problem size scales, requiring over **325 seconds** to sweep configurations for massive shapes (e.g., $N = 8\text{M}, K = 65536$). In contrast, our heuristic analytically derives the optimal configuration in less than **2.5 seconds** across all tested shapes, achieving up to a **175**× reduction in time-to-first-run. Crucially, as shown in the right chart, this massive reduction in configuration overhead does not compromise execution efficiency. The kernel iteration latency driven by the heuristic consistently matches the exhaustively tuned oracle, with a negligible performance difference of less than **0.3%**. This confirms that flash-kmeans provides highly deployable, out-of-the-box acceleration for dynamic AI pipelines without requiring offline warm-ups.

B Additional Experimental Results

This appendix provides the measurement protocol and the detailed results summarized in § 5. As in the main text, experiments run on a single NVIDIA H200 GPU unless labeled otherwise.

B.1 Measurement Protocol and Variance

Table 4 reports the mean and standard deviation of each phase of a Lloyd iteration over 50 runs at $B = 1, N = 8\text{M}, K = 1024, D = 128$. This is a separate measurement session from Table 1, so the totals differ by less than the run-to-run standard deviation. The end-to-end coefficient of variation is 0.12% for the standard implementation and 1.33% for flash-kmeans.

Table 3: Measurement protocol for all latency comparisons.

Setting	Protocol
Precision	FP32 for all methods, unless otherwise noted
Lloyd iterations	Same fixed number for all methods in latency comparisons
Initialization	End-to-end: random initialization for all methods; per-iteration and kernel: identical pre-set centroids
Convergence	Same stopping criterion for all methods in the convergence study
One-time setup	Included end to end; excluded from per-iteration and kernel measurements
Timing	Mean of 50 timed runs after 10 warm-up runs; rotated method order; no outlier removal
Configuration	Cache-aware compile heuristic for flash-kmeans in all experiments; no per-shape tuning
Software	CUDA 12.8, `cuML` 26.02

Table 4: Per-phase latency (ms, mean $\pm$ standard deviation over 50 runs; H200, $B = 1$, $N = 8\text{M}$, $K = 1024$, $D = 128$, FP32).

Implementation	Assign	Update	Other	Total
Standard + atomic	117.016 ± 0.145	6.088 ± 0.010	0.0151 ± 0.0005	123.119 ± 0.152
FlashAssign + Sort-Inverse	6.287 ± 0.097	1.0416 ± 0.0023	0.0151 ± 0.0005	7.343 ± 0.097
Phase speedup	18.61×	5.84×	1.00×	16.77×

B.2 One-Factor-at-a-Time Sweep over N, K, and D

Table 5 varies one of N, K, and D at a time around the reference shape $B = 1$, $N = 1\text{M}$, $K = 1\text{K}$, $D = 128$, and reports the speedup of each stage over the standard implementation of Figure 4. FlashAssign is 3.95–19.65× faster across all shapes. Its advantage over the materializing baseline narrows as D grows because the baseline's $N \times K$ materialization traffic does not depend on D, whereas the distance computation shared by all implementations grows linearly with D; as the problem becomes compute-bound, every exact implementation approaches the GEMM bound. FlashAssign stays at that bound: against the fused cuVS kernel, which avoids materialization, it remains 5.5–9.5× faster at $N = 1\text{M}$, $K = 1024$, and $D = 128$–2048 (FP16 inputs, H100). Sort-Inverse Update is 2.45–6.28× faster. With K and D fixed, its speedup grows from 3.63× to 5.22× to 6.28× as N/K increases from 1024 to 4096 to 16384. This trend is consistent with the atomic-count analysis in § 4.2: the baseline issues atomics per point, whereas Sort-Inverse Update issues them per segment, so the reduction grows with the number of points per cluster. We emphasize $D = 128$ in the main text because online LLM and video workloads operate on per-head vectors.

Table 5: Stage-wise speedup over the standard implementation when varying one factor at a time (H200, $B = 1$, FP32).

N	K	D	N/K	FlashAssign speedup	Sort-Inverse Update speedup
1M	1K	128	1024	17.63×	3.63×
4M	1K	128	4096	18.29×	5.22×
16M	1K	128	16384	18.32×	6.28×
1M	2K	128	512	18.51×	3.53×
1M	4K	128	256	19.65×	3.31×
1M	1K	256	1024	8.63×	4.35×
1M	1K	512	1024	5.48×	2.45×
1M	1K	1024	1024	3.95×	3.43×

B.3 Full Accounting of the Update Path

Sort-Inverse Update adds an `argsort` and an indexed gather that the atomic baseline does not need. Table 6 times every component of the update path separately at the largest update setting of Figure 4 ($B = 1$, $N = 2^{25} \approx 33\text{M}$, $K = 4096$, $D = 128$, FP32). The `argsort` accounts for 24.5% of

the update time at this large N, and the fully accounted update is still 4.23× faster than the atomic update. The indexed gather and the localized reduction are fused in a single Triton kernel and are therefore reported together.

Table 6: Component-wise latency of the update path (ms, mean ± standard deviation over 50 runs; H200, $N = 2^{25}$, $K = 4096$, $D = 128$, FP32).

Update component	Latency (ms)	Share
`argsort`	1.366 ± 0.002	24.5%
Index preparation	0.0983 ± 0.0004	1.8%
Buffer initialization	0.0072 ± 0.0003	0.1%
Indexed gather + localized reduction	4.075 ± 0.001	73.2%
Normalization	0.0203 ± 0.0003	0.4%
Sort-Inverse Update total	$\mathbf{5.567 \pm 0.003}$	100%
Atomic update total	23.545 ± 0.010	–

B.4 Comparison with cuVS `fusedL2NN`

Table 7 compares FlashAssign with the cuVS `fusedL2NN` kernel at matched shapes on H100 (FP32, $D = 128$). At $N = 8$M, `fusedL2NN` is slower than an unfused GEMM followed by `argmin` (28.5 ms), and profiling shows that it uses only 1.7% of HBM bandwidth and under 10% of matrix-multiply throughput. The kernel avoids materializing the distance matrix but attaches the 1-NN reduction to the GEMM epilogue, so each point's running minimum resides in global memory and requires a lock-guarded read-modify-write per centroid tile (four per point at $K = 1024$). FlashAssign keeps this state in registers within the streaming main loop, needs no locks, overlaps the reduction with asynchronous loads, and writes only the final 4-byte label, which is exactly the input consumed by Sort-Inverse Update. At $N = 4$M and $K = 4096$, a complete flash-kmeans iteration (23.2 ms) is 4.8× faster than the `fusedL2NN` assignment alone.

Table 7: Assignment latency of cuVS `fusedL2NN` versus FlashAssign (H100, $D = 128$, FP32).

N	K	`fusedL2NN` (ms)	FlashAssign (ms)	Speedup
1M	1024	6.55	1.38	4.7×
8M	1024	51.8	10.6	4.9×
4M	4096	111.1	22.2	5.0×

B.5 Real-Pipeline Engine Swaps

The pipeline rows of Table 2 run on H100 in FP32. In each pipeline, only the *k*-means engine changes between flash-kmeans and cuVS; the data, initial centroids (passed to cuVS with `init_method="Array"`), and iteration counts are fixed. The IVF-Flat index is built on SIFT1M [34] with nlist = 4096 and 20 coarse-training iterations. The IVF-PQ index uses $m = 16$ sub-quantizers with 256 codewords each; because cuVS exposes no batched interface, its 16 PQ codebooks are trained as 16 sequential fits, whereas flash-kmeans trains them in a single batched call with $B = 16$. The video workload is the per-step semantic clustering of Sparse VideoGen2 [25] ($B = 32$ attention heads, 75,600 tokens per head, $D = 128$, $K = 1000$, 10 iterations); since this clustering runs at every denoising step, it costs about 2 s with flash-kmeans and about 14 s with cuVS over a 50-step generation. On SIFT1M, the final inertia is 44,441.8 for flash-kmeans and 44,443.2 for cuVS. Matching initial centroids is conservative for cuVS: with its default k-means++ initialization [35], the IVF-Flat build takes 3.26 s, about 30× longer than with flash-kmeans (108 ms).

B.6 Analysis of the `cuML` and `FAISS` Comparisons

The 33× (`cuML`) and over 200× (`FAISS`) speedups in § 5.2 are end-to-end results from Figure 3 at $N = 8$M, $K = 1024$, $D = 128$. They are not a single kernel-level multiplier but the compounding product of algorithmic, structural, and hardware-level factors. cuVS, on which `cuML` builds, avoids materialization in assignment, but its update remains an atomic scatter. Working down from the ∼33× `cuML` result:

(1) Initialization and per-call setup ($\sim$1.7$\times$). These costs are fixed per call, so their share depends on the iteration count: at $N = 8$M, the end-to-end ratio rises from $\sim$5$\times$ at 50 iterations to 7.5$\times$ at a single iteration. Removing them leaves $\sim$19$\times$ at fixed iteration counts.

(2) Native batching (2.7$\times$). cuVS exposes no batched interface for independent per-head problems. For $B = 32$, $N = 75{,}600$, $K = 1000$, and $D = 128$, one batched flash-kmeans call takes 43.1 ms; looping our own kernels over the batch takes 117.7 ms, while looping cuVS takes 279.6 ms.

(3) Per-iteration execution ($\sim$5–7$\times$). Under matched protocols ($N = 8$M, $K = 1024$, $D = 128$, H100), one cuVS KMeans iteration takes 66.1 ms versus 12.4 ms for flash-kmeans (5.3$\times$). This combines the $\sim$5$\times$ gain of FlashAssign over the fused assignment (Table 7) and the $\sim$6$\times$ gain of Sort-Inverse Update over the atomic update, and remains 4.4–5.3$\times$ across $N = 8$M–33M and $K = 1$K–64K.

For `FAISS`, we profile GPU `faiss.Kmeans` at $B = 1$, $N = 8$M, $K = 1024$, and $D = 128$ with subsampling disabled and matched initial centroids: one iteration takes 712 ms versus 12.4 ms for a complete flash-kmeans iteration (57$\times$ at fixed iteration counts). `FAISS` attributes 82% of this time to its generic 1-NN search, which materializes distance tiles, performs top-k selection, and re-streams the host-resident dataset; the remaining 18% is spent on CPU centroid updates and transfers. The bridge from 57$\times$ to the over-200$\times$ end-to-end result is native batching and per-call index setup.

B.7 Convergence Study

Table 8 reports 100 independent trials in which flash-kmeans and a PyTorch implementation of Lloyd's algorithm start from the same initial centroids in each trial and use the same stopping criterion (H200, $B = 1$, $N = 8192$, $K = 32$, $D = 128$, FP32, tolerance 0.1). Both methods converge in every trial. Across paired trials, the mean relative gap in final within-cluster sum of squares (WCSS) is 1.91×10^{-4} and the mean difference in iteration count is 0.60.

Table 8: Convergence over 100 shared-initialization trials (mean $\pm$ standard deviation).

Method	Iterations	Final WCSS	Converged
PyTorch Lloyd	24.19 ± 4.04	$996{,}815.74 \pm 1{,}628.05$	100/100
flash-kmeans	24.79 ± 4.58	$996{,}780.47 \pm 1{,}636.28$	100/100

B.8 Cross-Architecture Results on B200

flash-kmeans supports A100, H100, H200, B200, and GB10 GPUs with identical kernels; per-architecture performance comes from the cache-aware compile heuristic, which re-selects tile sizes for each GPU's memory hierarchy. On Blackwell B200, re-selecting tile sizes with the heuristic alone yields up to 7.5$\times$ over reusing the configurations chosen for the previous generation. On randomized validation shapes, including batched problems and feature dimensions up to 2560, the heuristic lands within 0.8–1.6% (geometric mean; 95th percentile $\leq$5.6%) of an exhaustive per-shape oracle, with exact label agreement on every shape. Table 9 compares flash-kmeans with `fast_pytorch_kmeans` on B200 with FP16 inputs and matched settings. The gaps are wider than on H200 because these shapes lie in the large-K regime, where the baseline's $N \times K$ materialization dominates, and because B200's compute throughput grew faster than its memory bandwidth, which favors fused execution. As on H200, the ratio narrows as D grows toward the GEMM bound.

Table 9: Per-iteration latency on B200 (FP16 inputs).

N	K	D	flash-kmeans (ms)	`fast_pytorch_kmeans` (ms)	Speedup
1M	16,384	128	5.04	248.2	49.2×
1M	16,384	1024	36.6	514.2	14.0×
2M	8,192	128	5.00	256.8	51.4×
2M	8,192	1024	34.3	504.9	14.7×
2M	4,096	128	2.73	125.4	45.9×
2M	4,096	1024	18.5	243.1	13.1×
4M	4,096	128	5.39	238.4	44.2×
4M	4,096	1024	36.9	493.0	13.4×

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Our claims are empirically verified in § 5 and Appendix B.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discuss limitations, including hardware scope, the narrowing advantage at large feature dimensions, and the cost of `argsort`, in § 6.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.

- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: No theoretical result in the paper.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Hardware, software versions, shapes, precision, initialization, and timing protocol are specified in § 5.1 and Appendix B.1, and the code is publicly available.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.

(b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
(c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The code is open-sourced under the Apache 2.0 license at `https://github.com/svg-project/flash-kmeans`, including benchmark scripts. Synthetic data are generated by these scripts, and SIFT1M is publicly available.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Details are included in § 5.1 and Appendix B.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: Latencies are means over 50 timed runs after 10 warm-up runs. Appendix B.1 and Appendix B.3 report one-standard-deviation run-to-run variation (end-to-end coefficient of variation at most 1.33%), and Appendix B.7 reports mean $\pm$ standard deviation over 100 independent trials. Because the variation is small, figures omit error bars.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Each experiment runs on a single NVIDIA H200, H100, or B200 GPU, as specified in § 5.1 and the corresponding table captions; per-run latencies are reported directly.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research conducted in the paper conform with the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.

- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: This paper introduces a method to accelerate k-means algorithm, so it does not have direct societal impact. However, the use of k-means may have social impact, which can be amplified by this technique.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: We will explicitly specify the usage permission of our method with proper licenses.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: We cite all baseline libraries (with versions where applicable) and the SIFT1M dataset, and our usage respects their original licenses.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: We release the flash-kmeans code under the Apache 2.0 license, with installation instructions, API documentation, and benchmark scripts in the repository.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: This paper does not involve crowdsourcing nor research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: The core method development in this paper does not involve LLMs as any important, original or non-standard components.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.