Optimizing Lloyd's K-Means Algorithm for GPU

Jakob Gollreiter, Jiaxin Ge, Amil Dravid
EECS Department
UC Berkeley

Abstract

This paper presents a high-performance CUDA implementation of Lloyd's k-means clustering algorithm that outperforms `scikit-learn's` CPU implementation by up to $12x$ on an NVIDIA A100 GPU. After analyzing performance bottlenecks in the scikit-learn implementation, we developed optimizations including constant memory for centroids, triangle inequality pruning, and a two-stage reduction with shared memory to minimize global memory atomics. Our implementation efficiently addresses the memory bandwidth and cache inefficiency challenges that limit CPU-based approaches when processing large-scale, high-dimensional datasets. All code is released as open-source available at `https://github.com/JakobMichaelGollreiter/CUDA_KMeans`.

1 Introduction

Clustering algorithms form the backbone of modern data analysis, with applications ranging from segmentation and anomaly detection to image compression and representation learning. Among these techniques, Lloyd's k-means algorithm stands out for its simplicity, efficiency, and widespread adoption. The algorithm's popularity stems from its minimal parameter requirements beyond specifying the number of clusters, its natural parallelization across data points, and its versatility as a fundamental building block for more sophisticated techniques such as vector quantization and Bag-of-Features models.

1.1 The K-means Algorithm

For a dataset $X = x_1, x_2, ..., x_n \subset \mathbb{R}^d$ with n samples in d dimensions, Lloyd's algorithm iteratively refines k cluster centroids (user–specified cluster count) through two alternating steps:

(i) an *assignment step* (E-step) that computes all ℓ_2 distances $\|x_i - c_j\|_2$ and assigns each sample to its nearest centroid, and
(ii) an *update step* (M-step) that recomputes every centroid as the arithmetic mean of the samples assigned to it.

The arithmetic complexity per iteration is therefore $\mathcal{O}(nkd)$, and the memory traffic is dominated by reading the nd floating-point coordinates once per iteration. In modern workloads, n often ranges up to 10^8 and d from up to 10^3. A naïve implementation quickly becomes the runtime bottleneck.

1.2 State of the Art CPU Implementation

The reference KMeans algorithm in `scikit-learn` (`sklearn.cluster.KMeans`) showcases a well-optimized, hand-tuned CPU implementation that reflects current best practices. It employs Cython for static typing and elimination of Python interpreter overhead, OpenMP parallelization to distribute computation across available CPU cores, BLAS matrix multiplication for efficient distance

Algorithm 1 Lloyd's k-Means Algorithm

```
Require: Dataset $X = \{x_1, x_2, ..., x_n\} \subset \mathbb{R}^d$, number of clusters $k$
 1: Initialize centroids $C = \{c_1, ..., c_k\}$ (e.g., randomly select $k$ points from $X$)
 2: repeat
 3:     for each point $x_i \in X$ do
 4:         Compute distances $d(x_i, c_j)$ for $j = 1, ..., k$
 5:         Assign $x_i$ to the nearest centroid
 6:     end for
 7:     for each centroid $c_j$ do
 8:         Update $c_j$ as the mean of points assigned to cluster $j$
 9:     end for
10: until convergence or maximum iterations reached
11: return Final centroids $C$ and assignments
```

computation, and Elkan's triangle inequality optimization to skip unnecessary distance calculations. Through these techniques, `scikit-learn` delivers what is widely regarded as the fastest publicly available CPU implementation.

Despite these sophisticated optimizations, our performance analysis (Section 5) reveals a curious non-monotonic scaling pattern as dataset size increases. For small datasets ($n < 10^4$), performance is dominated by fixed overhead costs Python-to-C transitions, thread spawning, and buffer allocation, resulting in relatively poor efficiency. As dataset size grows to the medium range ($10^4 < n < 10^6$), the implementation achieves near-ideal scaling as computation becomes the dominant factor and parallelization delivers linear speedups. However, with large datasets ($n > 10^6$), the per-point processing cost increases as memory bandwidth becomes the limiting factor, and the algorithm becomes memory-bound. This "slow–fast–slow" behavior motivates the exploration of alternative hardware platforms, specifically GPUs, to overcome these limitations.

1.3 Potential of GPU Acceleration

Modern NVIDIA GPUs offer substantial computational resources that align well with k-means' inherent parallelism: up to 80 Streaming Multiprocessors (SMs) with $\mathcal{O}(10^4)$ hardware threads and memory bandwidth exceeding $\sim 1\,\mathrm{TB\,s^{-1}}$ HBM2 bandwidth. However, effectively mapping k-means to the GPU architecture requires addressing several challenges. The closest-centroid search can lead to different execution paths within thread warps, causing control flow divergence that reduces efficiency. Naïve centroid updates would require atomic operations on global memory, leading to severe serialization and performance degradation. Additionally, moving data between host and device memory can nullify computational gains for smaller datasets if not managed carefully. These challenges require careful algorithm redesign, as described in our methods part (see section 3).

1.4 Research Questions

Based on our analysis of existing CPU limitations of K-means clustering and the potential of GPU acceleration, we formulate three research questions:

1. Can a GPU implementation outperform the best Cython–OpenMP–BLAS baseline?
2. Which algorithmic optimizations (triangle-inequality pruning, batch processing, shared-memory reductions) are necessary and sufficient to reach that goal?
3. How do architectural limits (HBM bandwidth, SM occupancy, atomic contention) shape the ultimate speed-up curve?

1.5 Contributions

This paper makes the following contributions:

1. **Micro-architectural dissection of the CPU baseline.** We instrument `scikit-learn's` Cython kernels and expose, for the first time, the exact breakdown between Python overhead, prange chunk loop, BLAS distance kernel, and centroid reduction.

2. **A memory-hierarchy-aware CUDA design.** Our implementation stores centroids in `__constant__` memory, performs distance pruning in registers, and aggregates partial sums in shared memory before a warp-coalesced global write, thereby eliminating global atomics.
3. **Extensive empirical evaluation.** On an NVIDIA A100 (40 GB) and an AMD EPYC 7763 (64c/128t) we test up to 10^7 samples and 16 dimensions. The GPU achieves $10x - 20x$ speed-up over the CPU baseline.
4. **Open-source artefacts.** All kernels, host code, and scaling scripts are released at `https://github.com/JakobMichaelGollreiter/CUDA_KMeans` under an MIT licence.

The rest of this paper is structured as follows. In Section 2, we review previous CPU and GPU implementations. Section 3 introduces our CUDA kernel pipeline, while Section 4 describes the experimental setup. In Section 5, we analyze the scaling behavior of the `scikit-learn` CPU implementation. Section 6 presents a comparison between our GPU implementation and the CPU version from `scikit-learn`. Finally, Section 7 summarizes our findings and outlines possible directions for future work.

2 Related Work

Exact k-means on CPUs. The canonical Lloyd–Forgy algorithm dates back to the 1950s, yet the first provably good seeding strategy, *k-means++*, appeared only in 2007 (Arthur & Vassilvitskii, 2007). Most industrial CPU libraries, including `scikit-learn`, combine this seeding with highly–optimized inner loops: Cython for static typing, BLAS GEMM for dense distance calculation, and OpenMP work-sharing across samples. To reduce the arithmetic constant, Elkan (2003) introduced triangle–inequality upper/lower bounds; a memory–lean single-bound variant was later proposed by Hamerly (2010). Mini-batch stochastic updates push the scalability frontier further, as demonstrated by Sculley (2010) at Google Web-scale.
GPU acceleration. Early CUDA prototypes parallelized only the assignment step and suffered from global–memory atomics in the update phase. Modern GPU codes exploit shared–memory tiling, warp-level primitives, and distance pruning. Li et al. (2020) introduced *ASB k-means*, which streams data in batches too large to fit in HBM while preserving exactness. The open-source RAPIDS `cuML` library (May et al., 2020) implements both Lloyd and mini-batch variants and serves as an off-the-shelf baseline. FAISS extends GPU k-means to billion-vector regimes as part of its similarity-search stack (Johnson et al., 2019).
Other hardware back-ends. FPGA designs exploit custom pipelines for distance evaluation and reduction, trading programmability for energy efficiency (Lavenier, 2000). On distributed memory, bisecting and prediction- guided variants offer better load balance than naïve Lloyd; see Zhao et al. (2007) for an MPI realisation.
Position of this work. Unlike prior GPU studies, we first perform a *micro-architectural* dissection of the `scikit-learn` CPU path, exposing its overhead-, compute-, and bandwidth-dominated regimes (Section 5). Our CUDA kernel then integrates Elkan/Hamerly pruning within a two-stage shared-memory reduction, allowing us to outperform the highly-optimized CPU baseline by up to $19\times$ at 10^7 samples while remaining competitive at mid-scale.

3 Method

We implement k-means on GPUs by mapping the algorithm's key steps (assignment, centroid update, and reduction) onto the hierarchical memory structure of modern CUDA-enabled GPUs. This approach takes full advantage of GPU parallelism and memory bandwidth, while minimizing redundant computations and memory access bottlenecks. The following subsections describe our design choices in detail, including both algorithmic optimizations and GPU-specific memory optimizations.

3.1 Algorithmic Optimization: Triangle Inequality Pruning

The assignment step in Lloyd's k-means involves calculating the distance from each point to all k centroids. This is computationally expensive, particularly as the number of points or dimensions increases. We apply triangle inequality pruning based on Elkan's algorithm Elkan (2003), which allows us to skip unnecessary distance computations. Specifically, for each point, we calculate the

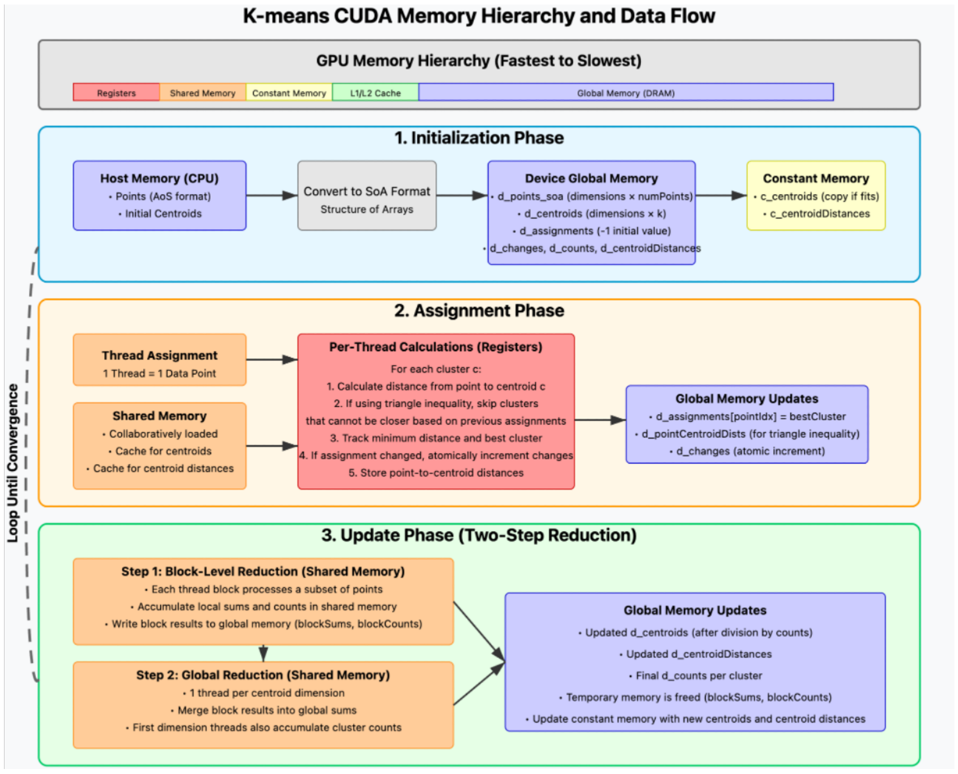

K-means CUDA Memory Hierarchy and Data Flow
GPU Memory Hierarchy (Fastest to Slowest)
Registers
Shared Memory
Constant Memory
L1/L2 Cache
Global Memory (DRAM)
1. Initialization Phase
Host Memory (CPU)
• Points (AoS format)
• Initial Centroids
Convert to SoA Format
Structure of Arrays
Device Global Memory
• d_points_soa (dimensions × numPoints)
• d_centroids (dimensions × k)
• d_assignments (-1 initial value)
• d_changes, d_counts, d_centroidDistances
Constant Memory
• c_centroids (copy if fits)
• c_centroidDistances
Loop Until Convergence
2. Assignment Phase
Thread Assignment
1 Thread = 1 Data Point
Shared Memory
• Collaboratively loaded
• Cache for centroids
• Cache for centroid distances
Per-Thread Calculations (Registers)
For each cluster c:
1. Calculate distance from point to centroid c
2. If using triangle inequality, skip clusters
that cannot be closer based on previous assignments
3. Track minimum distance and best cluster
4. If assignment changed, atomically increment changes
5. Store point-to-centroid distances
Global Memory Updates
• d_assignments[pointIdx] = bestCluster
• d_pointCentroidDists (for triangle inequality)
• d_changes (atomic increment)
3. Update Phase (Two-Step Reduction)
Step 1: Block-Level Reduction (Shared Memory)
• Each thread block processes a subset of points
• Accumulate local sums and counts in shared memory
• Write block results to global memory (blockSums, blockCounts)
Step 2: Global Reduction (Shared Memory)
• 1 thread per centroid dimension
• Merge block results into global sums
• First dimension threads also accumulate cluster counts
Global Memory Updates
• Updated d_centroids (after division by counts)
• Updated d_centroidDistances
• Final d_counts per cluster
• Temporary memory is freed (blockSums, blockCounts)
• Update constant memory with new centroids and centroid distances

Figure 1: The pipeline of our method.

distance to only those centroids for which the triangle inequality does not guarantee that a closer centroid exists.

For a given point x_i, and centroids c_j and c_l, the triangle inequality states that:

$$d(x_i, c_j) \leq d(x_i, c_l) + d(c_l, c_j)$$

If this inequality holds, we can skip computing $d(x_i, c_j)$ since it is guaranteed to be larger than $d(x_i, c_l)$. This pruning step reduces the number of distance calculations, especially when the centroids are well-separated, resulting in significant performance improvements.

3.2 GPU Memory Hierarchy Design

To optimize memory access patterns and minimize latency, we carefully map the different stages of the algorithm to the GPU memory hierarchy, which includes registers, shared memory, constant memory, and global memory. The key idea is to maximize cache locality and minimize global memory accesses by storing frequently accessed data in faster memory spaces.

3.2.1 Initialization and Memory Layout

In the initialization step, we randomly select k centroids from the dataset. These centroids are stored in *constant memory* on the GPU, which is optimized for broadcast access to all threads in a warp. The dataset $X = \{x_1, x_2, ..., x_n\}$ is stored in *global memory*, organized in a structure-of-arrays (SoA) format to ensure coalesced accesses during the distance calculations. This format groups points by dimension rather than by point, leading to efficient memory access patterns during parallel processing.

3.2.2 Assignment Step: Parallel Distance Computation

During the assignment phase, each thread is responsible for calculating the distance between a single data point and all k centroids. To exploit parallelism, we assign each thread to a single point, and each block of threads handles multiple points in parallel. For each point, the thread computes the distance to every centroid, utilizing the triangle inequality pruning to skip unnecessary calculations.

Once the distances are calculated, threads in a block collaboratively assign points to their nearest centroid. The assignment of points is done in parallel, with each thread atomically updating a global array that tracks the nearest centroid for each point.

3.2.3 Centroid Update: Shared and Global Memory Optimization

The centroid update step is crucial for convergence. After the assignment step, the centroids must be updated by computing the mean of the points assigned to each centroid. This requires a global reduction, which is notoriously memory-intensive. To optimize this, we use a two-step reduction strategy.

- In the first step, each block of threads computes partial sums for the points assigned to each centroid. These partial sums are stored in *shared memory*, a high-bandwidth memory space that is private to each block. By keeping partial sums in shared memory, we reduce the need for high-latency global memory accesses.
- In the second step, we perform a block-level reduction to accumulate the partial sums across blocks. This final sum is then written back to global memory, where the centroids are updated. To avoid contention on global memory, we use a coalesced access pattern, where each thread writes to a unique memory location.

This two-step reduction minimizes the use of atomic operations and ensures efficient memory access, enabling the update step to scale with larger datasets.

3.2.4 Memory and Batch Management

For datasets exceeding GPU memory capacity, points are processed in mini-batches (Dynamic Batching). Batch size B is auto-tuned based on available VRAM:

$$B = \left\lfloor \frac{\text{Free VRAM} - \text{Centroid Buffer}}{d \times \text{sizeof(float)}} \right\rfloor,$$

where d is the data dimensionality.

3.2.5 Convergence Check and Iteration

The convergence check is based on the change in centroid positions between iterations. After the centroid update, we compare the new centroids with the old ones to determine whether the algorithm has converged. This check is performed in parallel, with each thread computing the Euclidean distance between the corresponding centroids. If the total movement of the centroids is below a threshold, the algorithm terminates; otherwise, it repeats the assignment and update steps.

3.3 GPU Optimization Summary

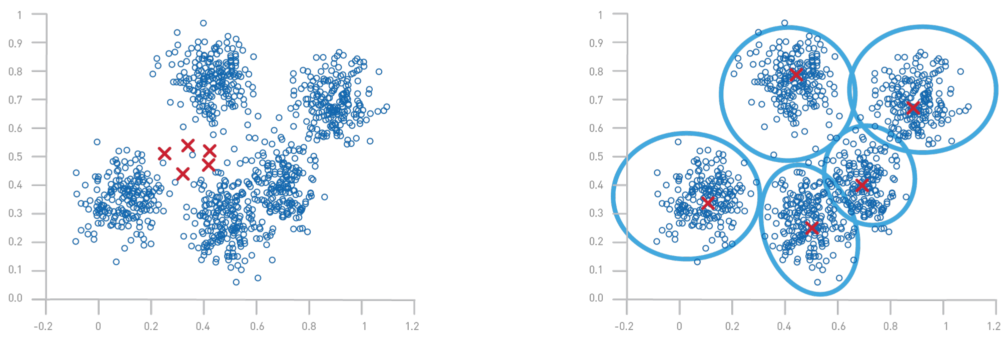
We summarize the GPU optimizations as follows:

- **Triangle Inequality Pruning:** Reduces redundant distance computations by pruning unlikely centroid assignments based on geometric properties.
- **Constant Memory for Centroids:** Uses constant memory to store centroids for fast broadcast access by all threads.
- **Shared Memory for Partial Sums:** Uses shared memory to accumulate per-thread partial sums before a block-level reduction.
- **Two-Step Centroid Update:** A two-stage reduction minimizes global memory access and atomic contention, speeding up centroid updates.

These optimizations allow the GPU implementation to efficiently handle large-scale k-means clustering tasks while maintaining the correctness of the algorithm.

4 Experiment

The objective of this experiment is to evaluate the performance gains achieved by a GPU-accelerated K-means implementation, compared to the highly optimized `scikit-learn` (v 1.6.1) CPU implementation. Our benchmark consists of synthetic datasets with varying dimensionality and amount of data points (see Section 4.1) We measure per-iteration time and total execution time.



(a) Initial data distribution showing randomly generated points in a feature space before clustering. Data points are colored based on their true underlying cluster distributions, representing a synthetic dataset with clear separable structures. The red crosses mark the initial random positions of the K centroids prior to the first iteration of Lloyd's algorithm.NVIDIA

(b) Final clustering results after convergence of the K-means algorithm. The colored regions represent the decision boundaries between clusters, with data points assigned to their nearest centroid.NVIDIA

Figure 2: Visualization of a simplified low-dimensional K-means initialization and clustering.

4.1 Dataset Generation and Initialization

Using `scikit-learn's` *make_blobs* function, we generate synthetic datasets comprising points sampled from multiple Gaussian distributions in high-dimensional feature space. The datasets are parameterized by dimensionality, number of samples, cluster count, and cluster standard deviation. For initialization, we implemented a deterministic approach where initial centroids were selected directly from data points using random sampling without replacement, controlled by a fixed seed. This ensured that all implementations began from identical starting configurations, eliminating initialization as a confounding variable in performance comparisons. An simplified low-dimensional visualization of this is depicted in Figure 2a.

4.2 Hardware Setup

All experiments were conducted on a high-performance computing node equipped with an NVIDIA A100 GPU (40GB memory) and an AMD EPYC™ 7763 CPU processor (64 cores/128 threads). Our GPU K-means implementation executed on the A100, while the baseline `scikit-learn` (v1.6.1) implementation ran on the EPYC CPU utilizing all available cores.

5 Scikit-Learn (CPU) Scaling Analysis

In our experiments, we noticed some interesting scaling effect with the `scikit-learn` (CPU) implementation. Therefore, we conduct an analysis on the optimization techniques of this library code. We start by reading the code and breaking down the time into different components, then, we investigated the effect of different components including CHUNK_SIZE, BLAS parallelization, and OMP Threads parallelization.

5.1 Investigation Method

We began with the Cython code in `sklearn.cluster.KMeans`. The main work is done in `_kmeans_single_lloyd`, which calls a dense or sparse `lloyd_iter_chunked_*` routine each iteration.

- **E-step ("chunk-loop").** An OpenMP `prange` loop partitions the data into fixed-size chunks (`CHUNK_SIZE`, default in the library is 256). Each chunk: 1. builds a local pairwise-distance matrix with one BLAS `dgemm` call; 2. assigns every sample to its nearest center; 3. accumulates partial sums for a later reduction.
- **M-step ("center-update").** After the parallel section, each thread enters a critical region, adds its partial sums into the global centroid buffers and then the master thread normalises them.
- **Extra work.** After convergence a final E-step is executed (labels must match the final centroids) and inertia is recomputed for the score.

To time this, we adapt the Cython code to record the time spent in each section. The elapsed times for the E-step and M-step are tracked across iterations and accumulated:

$$t_{\text{E}},\ t_{\text{M}},\ t_{\text{total}},\ t_{\text{overhead}} = t_{\text{total}} - t_{\text{E}} - t_{\text{M}}.$$

5.2 BLAS Parallelization

Figure 3 shows the wall-time *per sample* when we vary `threadpoolctl` so that the `dgemm` call runs with 1, 2, 4 or 8 BLAS threads, while `OMP_NUM_THREADS` is fixed to 1. We observed that:

1. **Small datasets ($N < 10^4$).** Adding BLAS threads *hurts* performance. The cost of syncing the thread pool and merging results dominates the tiny amount of arithmetic, so the curves with more threads performs worse than single-thread baseline.
2. **Large datasets ($N \geq 10^4$).** The matrix–multiply becomes the compute-bound and the extra cores are used efficiently. Adding more threads helps the performance.

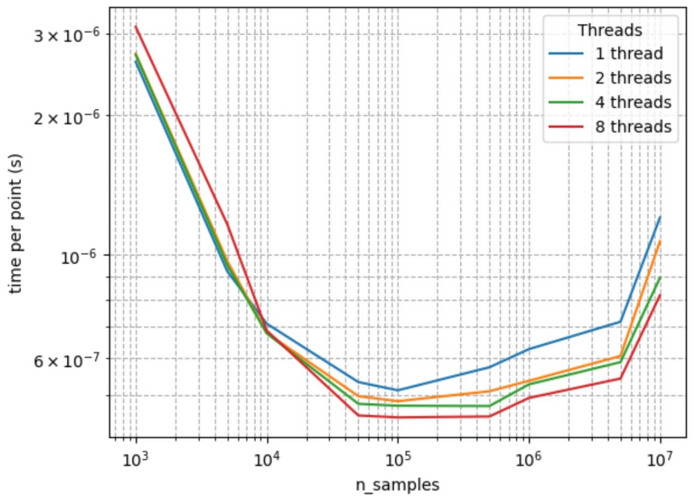
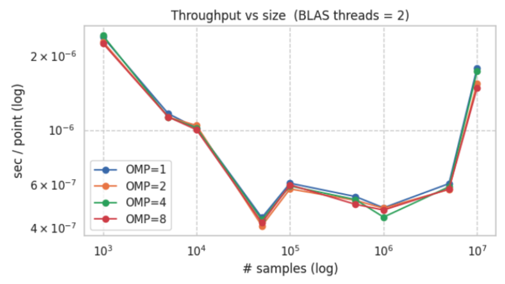


Figure 3: **BLAS threads vs. data size.** Log–log plot of total time per sample for 1, 2, 4 and 8 BLAS threads (`dgemm`); OpenMP is fixed to a single thread. Left section ($N < 10^4$) shows thread-pool overhead, right section shows near-linear speed-up until bandwidth limits dominate.

5.3 OpenMP Parallelization

We then investigate the effect of using different number of OpenMP threads. In the Cython kernel, OpenMP is used only inside the *chunk loop*: each thread processes a block of samples, assigns labels and accumulates partial sums; a short critical section then merges the partial centroids (M-step). The heavy `dgemm` call that builds the distance matrix is *not* parallelised with OpenMP—it is delegated to BLAS. In Figure 4, we made the following observations:

- **Left panel: BLAS threads = 2.** The four OpenMP curves are almost indistinguishable; wall-time is set by the `dgemm` call rather than by the OpenMP chunk loop.
- **Right panel: BLAS threads = 1 or 2.** Doubling the BLAS thread pool shifts the whole curve downward, but again the OpenMP thread count has no visible impact.



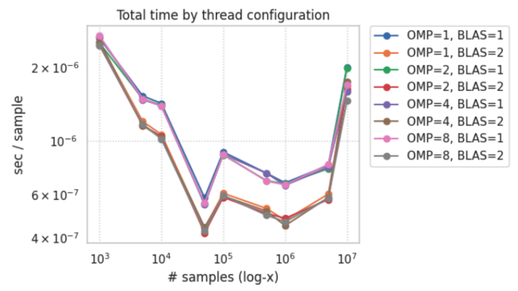


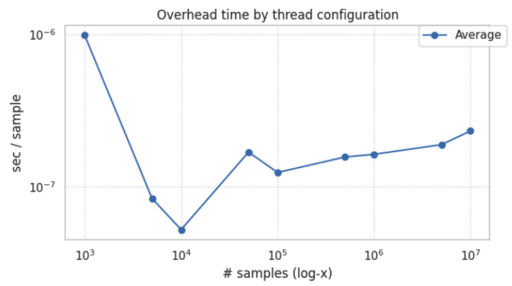
Figure 4: **Effect of OpenMP threads under different BLAS settings.** Left: BLAS threads =2. Right: BLAS threads = 1 or 2. Each curve shows wall-time per sample for `OMP_NUM_THREADS` =1, 2, 4 and 8 (log–log axes). OpenMP doesn't have much effect; but the change from BLAS 1 $\rightarrow$ 2 matters.

5.4 Overhead cost

The term t_{overhead} is the part of the wall-time that is *not* spent in the E-step (*chunk-loop*) or the M-step (*center-update*). Figure 5 (left) plots this value for all data sizes.

- The overhead is highest at the smallest problem ($N = 10^3$), where it is about 40 ms, then falls sharply and stabilises at $\approx$ 8–10 ms for $N \geq 10^5$.
- Because the fixed cost is divided by more samples, the overhead-per-sample becomes negligible for large datasets.

Main sources of overhead. 1. **Thread-pool start-up** (OpenMP and BLAS) on the first iteration. 2. **Extra E-step** executed after convergence so that labels match the final centroids. 3. **Final inertia calculation** used for the KMeans score. 4. **Python-level checks** (input validation, NumPy $\rightarrow$ C pointer conversion) and small memory allocations.



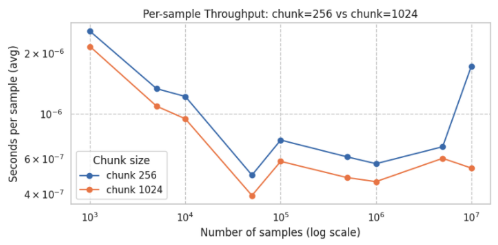


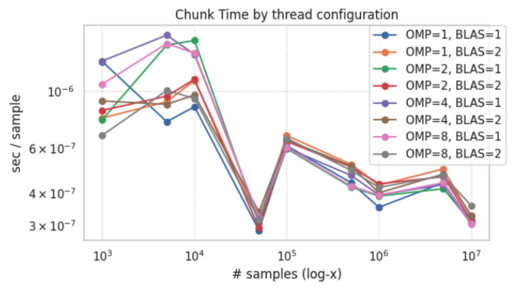
Figure 5: **Effect of the overhead time (Left) and the chunk size (Right).**

5.5 Chunk Size cost

The Cython kernel processes the input in blocks of `CHUNK_SIZE` samples so that each OpenMP thread works on one block. `scikit-learn's` default is `256`. We rebuilt the extension with `CHUNK_SIZE = 1024` and reran the benchmark; the results are in Figure 5 (right). We observed **Lower curve, same slope.** Every data size runs faster but the log–log slope remains similar. A larger block therefore reduces the constant factor in the $O(N)$ term instead of changing the algorithmic order. Fewer blocks mean fewer reductions and lock entries in the M-step. The temporary distance buffer ($1024 \times k$ doubles) still fits in cache, so there is no cache penalty. The relative gain widens slightly at $N = 10^7$ potentially because the reduction lock is used fewer times.

5.6 Phase decomposition

To see where the time goes we split every Lloyd iteration into its two parts: (1) **E-step (chunk time).** (2) **M-step (update time).** Figure 6 plots the per-sample cost of each phase for eight thread combinations (OMP $\in \{1, 2, 4, 8\}$, BLAS $\in \{1, 2\}$). In both cases we noticed a decrease in time over data size. When data size is small the time is large. In Figure 7, we found that the E-step dominates most of the compute time over the M-step.



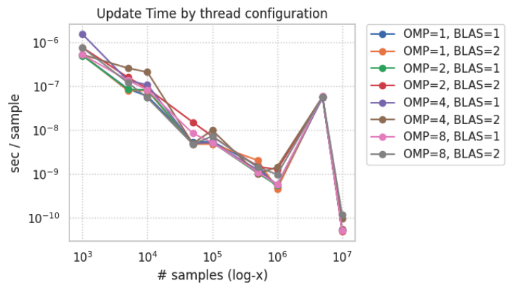


Figure 6: **Time breakdown into the 2 phases.** Left: Chunk Time. Right: Update Step.

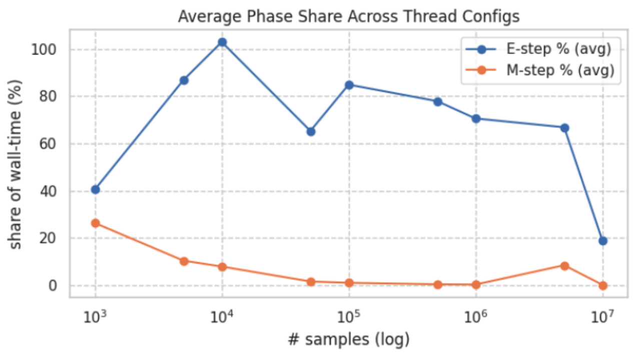


Figure 7: **Percentage between the 2 phases.** The E-step dominates most of the compute

5.7 Summary

These results identify three distinct regimes: *Overhead-dominated (small n)*: fixed costs (GIL transitions, thread spawn) overshadow linear work. *Compute-dominated (mid n)*: Cython prange loops and BLAS/GEMM vectorization fully amortize overhead, yielding ideal $O(n)$ scaling and near–perfect thread parallelism. *Bandwidth-limited (large n)*: per-point compute reaches hardware memory-bandwidth limits, flattening the scaling curve and causing constant M-step cost to regain significance.

6 Results

To evaluate the scalability and efficiency of our GPU-based KMeans clustering implementation, we compare its performance against the CPU-based `scikit-learn` implementation across varying dataset sizes and feature dimensionalities. Our analysis highlights how both data volume and feature dimensionality influence computational performance.

6.1 Scaling with Dataset Size

Figure 8 illustrates how execution time scales with increasing dataset size for both CPU and GPU implementations. Our GPU implementation demonstrates superior performance on larger datasets (>10^6 points), achieving speedups of up to 12× compared to `scikit-learn`. For smaller data sets (<10^6 points), our GPU implementation is even relatively faster than the CPU version. The apparently counterintuitive relationship between the two implementations, where the GPU advantage is pronounced at both the smallest and largest dataset sizes, is explained by the CPU implementation's distinctive scaling characteristics (see Section 5) rather than any anomaly in our GPU implementation. The performance curve of the `scikit-learn` CPU implementation exhibits the characteristic non-monotonic U-shape identified in our analysis, transitioning through overhead-dominated ($10^3 - 10^4$ points), compute-dominated ($10^4 - 10^5$ points), and bandwidth-limited ($> 10^6$ points) regimes. This explains why the CPU implementation's performance initially improves with increasing dataset size before deteriorating again at larger scales. In contrast, our GPU implementation follows a more consistent and predictable scaling pattern throughout the tested range. The GPU's massive parallelism effectively mitigates the overhead issues that plague the CPU implementation at small scales, while its superior memory bandwidth architecture prevents the severe performance degradation observed in the CPU implementation at large scales.

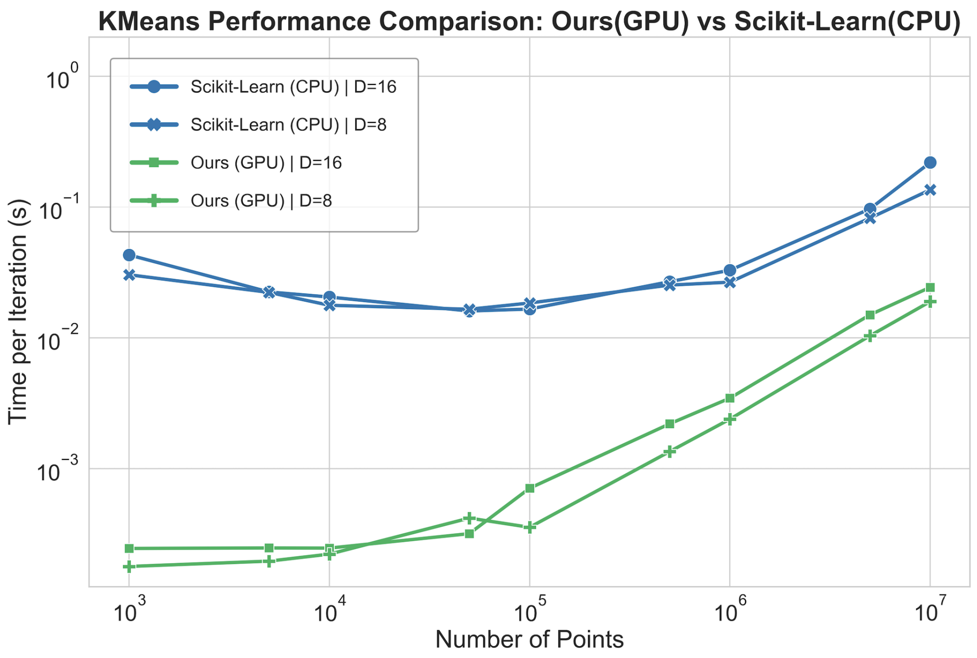


Figure 8: Log-log plot showing time per iteration versus dataset size ($10^3 - 10^7$ points) for KMeans clustering implementations. GPU-based methods (green) outperform CPU `scikit-learn` implementations (blue) by 1-2 orders of magnitude across all dataset sizes. Untypical scaling graph due to a non-monotonic scaling pattern of the `scikit-learn` CPU implementation, that under performs siginificantly in small number of data points. Lower dimensionality (D=8) provides slight performance improvements over higher dimensionality (D=16) for both implementations.

6.2 Impact of Dimensionality

As shown in Figure 8, dimensionality has a measurable but modest effect on performance for both CPU and GPU implementations. In general, lower dimensionality (D=8) results in slightly faster execution times than higher dimensionality (D=16), as expected due to the reduced computational load per point. However, the GPU implementation not only maintains its advantage across dimensions but in fact shows greater relative speedup at higher dimensions specifically in the largest dataset. For instance, at 16 dimensions with 10 million data points, the GPU implementation achieves a 12× speedup over the CPU baseline, compared to a 9× speedup for 8 dimensions.

7 Conclusion

Our research provides both analytical insights into CPU-based K-means performance bottlenecks and a practical GPU implementation that overcomes them. Through detailed profiling of `scikit-learn's` implementation, we identified three distinct scaling regimes: overhead-dominated for small datasets, compute-efficient for medium datasets, and bandwidth-limited for large datasets. Building on these insights, we developed a CUDA implementation with four key optimizations: (1) constant memory storage of centroids for efficient access, (2) triangle-inequality pruning to reduce distance calculations, (3) two-stage shared memory reduction to minimize global memory operations, and (4) batch processing for datasets exceeding GPU memory. Our experimental results demonstrate up to 12× speedup over the `scikit-learn` baseline for large datasets on an NVIDIA A100 GPU. Notably, our GPU implementation outperforms the CPU version across all tested dataset sizes by mitigating overheads in the small-data regime and avoiding the bandwidth saturation that limits CPU performance at scale.

References

David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 1027–1035, 2007.

Charles Elkan. Using the triangle inequality to accelerate k-means. In *Proc. Int. Conf. Machine Learning (ICML)*, pp. 147–153, 2003.

Greg Hamerly. Making k-means even faster. In *Proc. SIAM Int. Conf. Data Mining (SDM)*, pp. 130–140, 2010.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. arXiv:1702.08734, 2019.

Dominique Lavenier. Fpga implementation of the k-means clustering algorithm for image segmentation. In *Proc. Int. Conf. Field Programmable Logic and Applications*, pp. 145–154, 2000.

Yuhui Li, Kai Wang, and Dave Anderson. Asynchronous selective batched k-means on gpus. *Proc. IEEE High Performance Extreme Computing*, pp. 1–7, 2020.

Jacob May, Michael Luong, et al. Combining speed and scale to accelerate k-means in rapids cuml. RAPIDS AI Blog, 2020. `https://medium.com/rapids-ai/combining-speed-scale-to-accelerate-k-means-in-rapids-cuml-8d45e5ce39f5`.

NVIDIA. K-means. `https://www.nvidia.com/en-us/glossary/k-means/`. Accessed: 2025-05-12.

David Sculley. Web-scale k-means clustering. In *Proc. WWW*, pp. 1177–1178, 2010.

Rong Zhao et al. Parallel bisecting k-means with prediction clustering algorithm. *The Journal of Supercomputing*, 39(1):45–63, 2007.