Clustering algorithms on GPU for video processing

Jonathan Fin, Mat: 256178, `jonathan.fin@studenti.unitn.it`*, GitRepo:*
`https://github.com/Nathanoj02/GPU-Computing-2025-256178`

***Abstract*—This work presents a GPU-accelerated implementation of the k-means clustering algorithm, tailored for high-dimensional video data. The study evaluates different initialization strategies, distance metrics, and optimization techniques, with particular focus on OpenACC for parallelization. Experimental results show that the squared Euclidean distance achieves the best trade-off between clustering accuracy and computational efficiency, while GPU implementations consistently outperform the CPU baseline across multiple resolutions. The optimized GPU algorithm demonstrates near real-time performance at high resolutions, and a calibration strategy further reduces iteration counts for static video sequences. Additionally, comparisons of alternative approaches highlight the advantages of calibration over k-means++ for video data. The findings confirm the effectiveness of GPU acceleration for clustering tasks and suggest promising directions for further optimization, including CUDA-based low-level implementations and multi-GPU scaling for larger datasets.**

***Index Terms*—K-means, Clustering, GPU, OpenACC, CUDA, Parallelization, Video Processing**

I. Introduction

The aim of this project is to develop an efficient algorithm for performing clustering on large datasets using a GPU, leveraging OpenACC to parallelize the most computationally expensive steps.

Clustering is a fundamental operation in unsupervised learning, widely used for tasks such as data analysis, pattern recognition, and dimensionality reduction. Efficient clustering algorithms are crucial for extracting meaningful patterns from large datasets while minimizing computational time.

In this report, the focus is on the k-means algorithm, specifically applied to clustering video data. Video datasets are particularly challenging due to their high dimensionality and temporal complexity, making GPU acceleration essential for practical performance. The project investigates strategies for parallelizing k-means on the GPU, evaluates their effectiveness in terms of speedup and scalability, and explores computational optimizations as well as heuristics for choosing the initial centroids in video data.

II. Problem Statement

Clustering is a fundamental task in unsupervised machine learning, aiming to group data points into **clusters** such that points within the same cluster are more similar to each other than to points in other clusters. It is widely used in fields such as data analysis, image and video processing, pattern recognition, and recommendation systems.

Among clustering algorithms, **k-means** is one of the most popular due to its simplicity and efficiency. The algorithm partitions a dataset into k clusters by iteratively assigning each data point to the nearest **centroid** and then updating the centroids as the mean of the points in each cluster. Despite its simplicity, k-means can be computationally intensive for large or high-dimensional datasets, especially when applied to video data where each frame contributes multiple dimensions.

K-means clustering is widely used in unsupervised learning and has several standard optimizations:

- **Centroid Initialization:** k-means++ is a widely adopted method to improve convergence by carefully selecting initial centroids based on data distribution;
- **Distance Computation:** efficient calculation of distances between data points and centroids is critical. GPU implementations often parallelize this computation for large datasets;
- **Cluster Evaluation:** metrics for selecting the optimal number of clusters (k) are essential for practical applications, especially in complex datasets such as videos.

A. Parallelization

The focus of this project is on **parallelization** strategies for k-means on the GPU. Initial CPU implementations serve as a baseline for performance evaluation. Early GPU implementations revealed performance bottlenecks, which motivated profiling and further optimization. Key improvements include optimized memory allocation (e.g., using `malloc` efficiently), exploring centroid selection heuristics, and calibrating the algorithm for video datasets.

Given the foundational nature of this work, the primary objective is to experiment with different parallelization strategies and computational optimizations, benchmark their performance, and explore heuristics for cluster selection in video data, rather than to propose novel clustering algorithms.

III. State of the Art

The state-of-the-art solutions for clustering large-scale datasets leverage GPU acceleration to reduce computational time, particularly for high-dimensional data. Libraries such as cuML [1] provide GPU-accelerated implementations of clustering algorithms, including k-means, and are optimized for dense datasets. These libraries typically support multiple precision operations and include routines for distance computation, centroid initialization, and iterative refinement. Notably, cuML's Python API mirrors that of scikit-learn, enabling seamless integration into existing Python-based machine learning workflows.

In addition to cuML, there are other notable implementations of GPU-accelerated k-means clustering. For instance, the repository [2] presents a highly-optimized CUDA implementation of the k-means algorithm. This approach, documented in a conference paper [3], focuses on optimizing the kernel execution and memory access patterns to enhance performance on NVIDIA GPUs.

These existing solutions provide a foundation for GPU-accelerated k-means clustering. However, they often require adaptation to specific use cases, such as clustering video data. This project takes inspiration from these foundations to implement a GPU-accelerated k-means algorithm tailored for video data, incorporating optimizations for memory allocation, centroid initialization, and distance computation, and evaluating its performance against CPU-based implementations.

IV. Methodology and Contributions

The work in this project can be divided into several stages, starting from the CPU baseline implementation and progressively moving towards optimized GPU-based solutions tailored for video clustering. Each stage was designed to address specific computational challenges, improve efficiency, and investigate algorithmic trade-offs.

A. CPU Baseline

A CPU implementation of the k-means algorithm was first developed as a reference baseline. Several **distance metrics** were explored to measure similarity between data points and centroids, allowing for a fair comparison of their computational cost and impact on clustering quality. This stage served as the ground truth for validating GPU-based implementations.

Algorithm 1 report a simple k-means clustering algorithm, where the *nearest* centroid is calculated using the various distance functions.

Algorithm 1 K-means Clustering

Input: Dataset $X = \{x_1, x_2, \ldots, x_n\}$, number of clusters k
Output: Cluster assignments and centroids $C = \{c_1, c_2, \ldots, c_k\}$
Initialize centroids C (e.g., random or with K-means++)
repeat
  foreach $x_i \in X$ **do**
    Assign x_i to nearest centroid c_j
  foreach *cluster* j **do**
    Update c_j as mean of all points assigned to it
until *convergence*;

B. Cluster Evaluation Metrics

To identify the optimal number of clusters k, two widely used evaluation methods were employed:

- **Elbow Method:** evaluates the total within-cluster sum of squares (WCSS) across different values of k, selecting the point where the marginal gain in performance decreases;
- **Silhouette Method:** measures cluster cohesion and separation by comparing intra-cluster distance to inter-cluster distance, with higher silhouette scores indicating better clustering quality.

These methods ensured that the chosen value of k reflected meaningful structures in the dataset, rather than being arbitrarily selected.

C. GPU Implementations

The first GPU implementation of k-means revealed that performance was significantly slower than the CPU baseline. Using *NVIDIA Nsight Systems* (`nsys`), profiling identified bottlenecks primarily in memory transfers. To address these, multiple GPU kernels and algorithmic variations were explored:

- **K-means with Random Initialization:** centroids chosen randomly from the dataset;
- **K-means++ Initialization:** centroids initialized with the k-means++ heuristic to improve convergence;
- **Centroids from Image Pixels:** Initial prototypes selected directly from random pixels of the input video frames;
- **Tiling:** introduced memory tiling strategies to improve memory coalescing and reduce global memory latency.
- **Convergence Skipping:** convergence checks skipped for a fixed number of iterations to reduce synchronization overhead.

Through benchmarking, k-means++ initialization and tiling were found to achieve the best perfomances in terms of speed and clustering accuracy. As a result, a hybrid implementation, *k-means++ with tiling*, was adopted as the primary GPU solution.

Algorithm 2 shows the k-means++ initialization used in the project, while algorithm 3 shows a more detailed GPU implementation of the k-means algorithm.

Algorithm 2 K-means++ Initialization (Distance-Max Variant)

Input: Dataset $X = \{x_1, x_2, \ldots, x_n\}$, number of clusters k, dimensionality d
Output: Initial centroids $C = \{c_1, c_2, \ldots, c_k\}$
Choose first centroid c_1 randomly
for $j = 2$ **to** k **do**
  For each $x \in X$, compute its distance to the nearest centroid in C
  Let c_j be the point $x \in X$ with the maximum such distance
  Add c_j to C

Algorithm 3 GPU-Accelerated K-means Clustering

Input: Dataset X of n pixels with d dimensions, number of clusters k, stability threshold ϵ, maximum iterations T
Output: Cluster assignments

```
Initialize centroids C using K-means++
for t = 1 to T do
    Save current centroids C^old
    // Assignment step (parallel over pixels)
    for each pixel x_i ∈ X do in parallel
        Assign x_i to the nearest centroid in C
    // Update step (parallel accumulation)
    for each cluster j and dimension d do in parallel
        Compute the mean of all pixels assigned to j  Update centroid c_j with this mean
    // Convergence check
    Compute maximum centroid shift Δ = max_j ||c_j − c_j^old||^2
    if Δ ≤ ε then
        break
for each pixel x_i ∈ X do in parallel
    Replace x_i with its assigned centroid
```

D. GPU Calibration for Video Clustering

To further adapt the algorithm to video data, a **calibration** strategy was introduced. A subset of frames (e.g., 10 frames) was clustered first to identify representative prototypes. These calibrated centroids were then reused as initial prototypes throughout the entire video. This approach reduced the variance introduced by random initialization, stabilized clustering results across frames, and improved overall performance in video analysis tasks.

V. System Description and Experimental Set-up

The experiments were conducted on a local workstation equipped with an NVIDIA GeForce GTX 1650 Ti Mobile GPU. The code compilation was performed using the NVIDIA HPC compiler with the flags `-O2`, `-acc`, and `-gpu=cc75`. These flags enabled aggressive optimizations and GPU-specific targeting for compute capability 7.5. The use of the `-acc` flag enabled OpenACC directives, which allow GPU parallelization to be expressed at a higher level. By doing so, the focus of this project remains on investigating algorithmic challenges and optimization strategies, rather than dealing with low-level implementation details of GPU programming.

To ensure reliability and minimize the impact of scheduling noise, each experiment was repeated between 10 and 20 times with at least 3 warm-up cycles, with the results averaged across runs. This methodology reduces variability caused by fluctuations in operating system priorities and ensures consistent benchmarking conditions.

A. System Description

All the systems specifications are described in Table I and Table II.

System Component	Specification
CPU	Intel Core i7-10750H
Cores / Frequency	6 cores @ 2.6 GHz
GPU Accelerator	NVIDIA GTX 1650 Ti Mobile
Compute Capability	7.5

TABLE I
Hardware specifications

Software Component	Version
Operating System	Ubuntu 20.04.6 LTS
CUDA Toolkit	12.9
C++	9.4.0
NVIDIA HPC Compiler (NVC)	25.7-0

TABLE II
Software environment specifications

B. Dataset description

The dataset used for experimentation consists of frames extracted from a single high-definition video source at $1080p$ resolution. To analyze the scalability and robustness of the k-means clustering algorithm across different input sizes, the same video was also downsampled to $720p$, $480p$, and $240p$ resolutions. This multi-resolution approach enables the evaluation of performance trade-offs as a function of input dimensionality, allowing for a thorough comparison of execution time and clustering quality across varying dataset sizes. By using the same video content across all resolutions, the experiments maintain consistency in data distribution while highlighting the computational effects of scaling.

Video	Width (pixels)	Height (pixels)	Total pixels per frame
walking_1080.mp4	1080	1920	$\sim 2.07M$
walking_720.mp4	720	1280	$\sim 922k$
walking_480.mp4	480	854	$\sim 410k$
walking_240.mp4	240	426	$\sim 102k$

TABLE III
Dataset

VI. Experimental Results

A. Evaluation metrics

Elbow method and **average silhouette method** were run on the first frame of the test video to determine the best k to adopt in following calculations. The most suitable k value is usually low, so the tests were run on k ranging from 1 to 10 included.

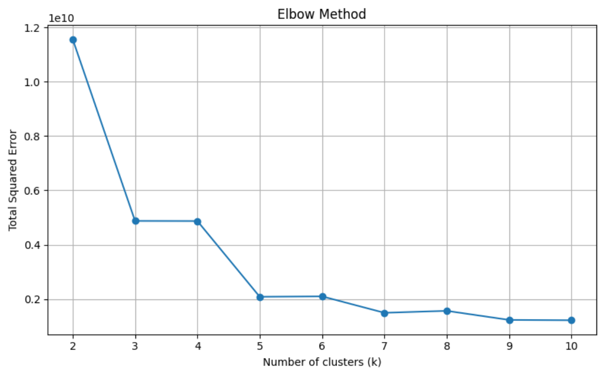

Fig. 1. Elbow method

Elbow method: From Figure 1 $k = 3$ might be a good candidate but it is not sure because the solution is just visual. The average silhouette method is fundamental in this case.

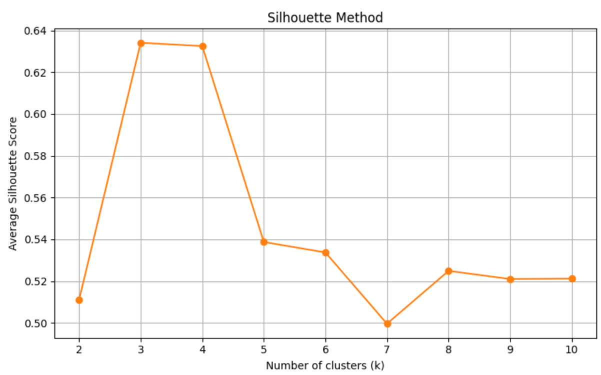

Fig. 2. Average silhouette method

Average silhouette method: After running the average silhouette method, $k = 3$ is clearly the most suitable k for further calculations.

From here $k = 3$ is used in all tests.

B. Distance Metrics Evaluation

To determine the most efficient and effective distance metric for k-means clustering on video frames, several distance measures were tested on CPU. The total execution times were recorded for each metric and are summarized in Figure 3.

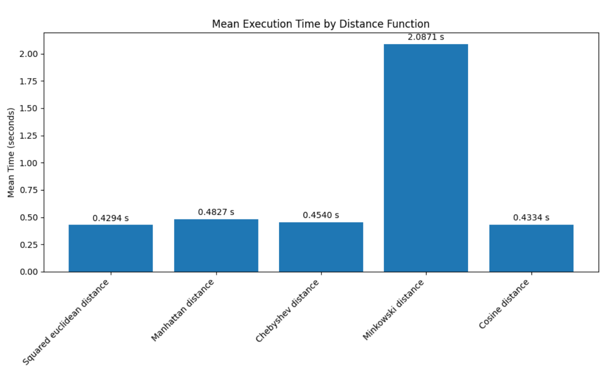

Fig. 3. Execution times of different distance metrics on CPU

From the chart, *squared Euclidean distance* emerges as the best choice, balancing both computational efficiency and clustering quality. The squared form is preferred over the standard Euclidean distance because it avoids costly square root operations while preserving the relative distances necessary for clustering. This distance metric is also the one producing the most visually accurate clusters in the sample video frames.

C. GPU Implementation Performance

The CPU baseline was compared with multiple GPU implementations to assess speedup and performance improvements. Figure 4 reports execution times for each GPU variant, including initial and optimized kernels. A dashed line indicates the CPU baseline.

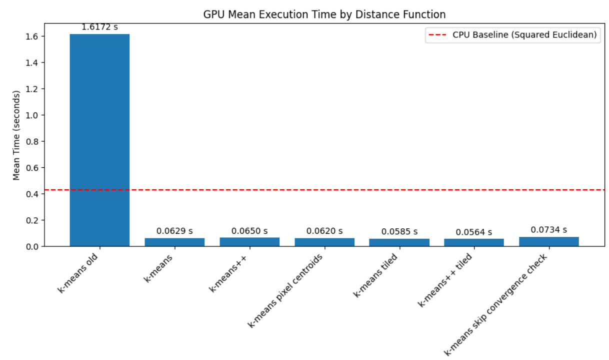

Fig. 4. Execution times of GPU implementations

The charts highlight significant **speedup** in GPU implementations compared to CPU execution. The Initial GPU kernel was slower than expected due to memory access patterns, while optimized kernels with improved memory transfers and centroid heuristics achieved a speedup of approximately $7.6\times$ over CPU. The corresponding processing rate reaches approximately 17.7 FPS for standard 1080p video frames, compared to the CPU's ~ 2.3 FPS.

D. Multi-Resolution Video Performance

To evaluate the scalability of the GPU implementations, the algorithm was tested across multiple video resolutions: 1080p, 720p, 480p, and 240p. Figure 5 compares CPU execution, GPU without calibration, and GPU with calibration.

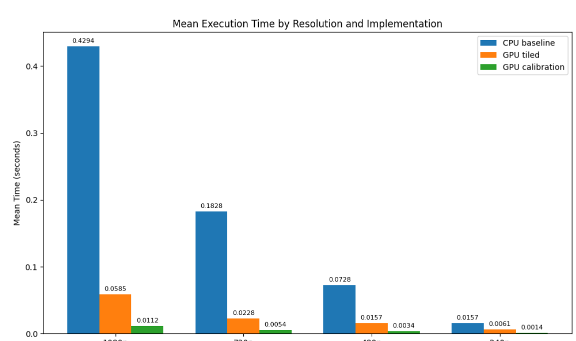

Fig. 5. Execution time comparison across video resolutions for CPU, GPU, and GPU with calibration

The optimized GPU algorithm maintains consistent performance across resolutions, demonstrating near real-time processing at higher resolutions. The calibrated GPU version further improves execution times, reducing the number of iterations to achieve centroid convergence. The FPS achieved for the *full HD* resolution is approximately 89.3, with a speedup of $38.3\times$ with respect to the CPU baseline. At lower resolutions the GPU implementations still outperform the CPU baseline, but the relative speedup is reduced compared to full HD, as the smaller frame sizes limit the amount of parallel work available to fully utilize the GPU.

VII. Other work

Several additional optimization strategies were explored during the development of the GPU k-means implementation:

- **Shared Memory (Cache) Optimization:** attempts to exploit shared memory for caching pixel data were investigated. However, since the OpenACC implementation already manages memory efficiently, these optimizations did not yield significant performance improvements;
- **Pre-allocation of Memory for Video Frames:** for video sequences, allocating all necessary memory on the first frame and reusing it across subsequent frames was tested. This approach did not result in measurable improvements, as memory allocation overhead is negligible relative to the computational workload;
- **Video-specific Strategies:** two approaches were analyzed for video clustering:
  - *Temporal Centroid Reuse:* initializing the centroids of the current frame with the final centroids of the previous frame reduces convergence time and overhead. This method is particularly effective for longer videos where the content changes gradually;
  - *Calibration:* the calibration procedure adopted in this work, which relies on multiple random initializations and selecting the best prototypes, proved to be more accurate than k-means++ initialization. It is especially suited for static or low-motion videos, where it achieves lower clustering error, while temporal reuse is better for highly dynamic sequences.

VIII. Conclusions

This work presented the design and evaluation of a GPU-accelerated k-means clustering algorithm using OpenACC. The results show that the GPU implementations achieve significant speedups compared to the CPU baseline, while maintaining clustering accuracy. Among the distance metrics evaluated, the squared Euclidean distance provided the best balance between computational efficiency and clustering quality. The optimized GPU algorithm demonstrated consistent performance across video resolutions and achieved near real-time processing for high-resolution frames. The calibration strategy further improved convergence, reducing iteration counts and enhancing stability for static video content.

Future Work

Future extensions of this work could focus on:

- **CUDA Implementation:** rewriting the algorithm in CUDA would allow for fine-grained control over memory management and thread scheduling, enabling further optimization beyond OpenACC's abstraction;
- **Multi-GPU Scaling:** leveraging multiple GPUs could extend the method to much larger datasets and higher-resolution video streams, distributing the workload for even greater speedups.

References

[1] V. Singh, "cuml - rapids machine learning library," 2025. [Online]. Available: https://github.com/VishankSingh/cuml

[2] M. Kruliš, "cuda-kmeans: A novel, highly-optimized cuda implementation of the k-means clustering algorithm," 2020. [Online]. Available: https://github.com/krulis-martin/cuda-kmeans

[3] M. Kruliš and M. Kratochvíl, "Detailed analysis and optimization of cuda k-means algorithm," 2020. [Online]. Available: https://jnamaral.github.io/icpp20/slides/Krulis$_{D}etailed.pdf$