King Fahad University of Petroleum and Minerals

Computer Engineering Department

GPU Programming and Architecture (COE-506)

GPU Acceleration of K-Means Clustering algorithm

Students:

Ayman Mohammed Qashlan 200917630

Abdulaziz Hasan Alshehri 201776310

Haithem Tariq Albetairi 201379790

Supervised by:
Dr. Ayaz ul Hassan Khan

Submission Date:
16-12-2023

Instructor: Dr. Ayaz ul Hassan Khan

COE-506: GPU Programming and Architecture
(Course Project Report) - Implementation of a Real-World Application Using OpenACC, CUDA Python, and CUDA C/C++

Abstract

The k-means algorithm is a widely used clustering algorithm for unsupervised learning in machine learning. The project aimed to accelerate the k-means algorithm using GPU acceleration via three methods: OpenACC, Cuda Python, and Cuda C. The key finding of the project was that Cuda Python was the fastest method among the three due to its ease of implementation and optimization, and the JIT compiler that does automatic optimizations 1. The significance of the work lies in the fact that it provides a faster and more efficient way of clustering large datasets, which can be useful in various applications such as data analysis workflows and applications 1.

Introduction

K-means clustering is a popular unsupervised machine learning algorithm used for partitioning a dataset into K clusters, where each data point belongs to the cluster with the nearest mean (centroid). The algorithm iteratively refines the cluster centroids based on the mean of the data points assigned to each cluster. K-means clustering finds applications in various fields, such as data analysis, image segmentation, and customer segmentation, providing insights into patterns within datasets.

The goal of clustering is to divide the population or set of data points into several groups so that the data points within each group are more comparable to one another and different from the data points within the other groups. It is essentially a grouping of things based on how similar and different they are to one another.

The K-means clustering algorithm is simple and elegant, and it's used for partitioning a dataset into K distinct, non-overlapping clusters. It's used when you don’t have existing group labels and want to assign similar data points to the number of groups you specify (K).

This algorithm will be adapted to the parallel processing capabilities of GPUs, addressing computational intensity challenges associated with large datasets. Our project aims to leverage GPU acceleration techniques, specifically OpenACC, CUDA C/C++, and CUDA Python, to enhance the performance of our k-means clustering algorithm.

Background and Related Work

The K-Means clustering algorithm, a fundamental tool in unsupervised machine learning, has been widely studied and applied. Recent advancements have focused on leveraging GPU acceleration to enhance the performance of K-Means clustering. In [1], Lloyd's introduced the iterative K-Means algorithm, laying the foundation for subsequent studies. This seminal paper focuses on minimizing quantization error and serves as the basis for many existing solutions. Hartigan and Wong [2] proposed algorithmic refinements to the K-Means clustering algorithm, addressing convergence speed and stability. Their work is fundamental to the understanding and application of K-Means. In [3], Forgy's initialization method, where k data points are randomly selected as initial centroids, is commonly used in K-Means. This work highlights the importance of initialization strategies for effective clustering. Our

project focuses on GPU acceleration of the K-Means clustering algorithm, building upon and extending existing solutions. The justification for our chosen approach stems from the following considerations:

Scalability and Performance:

With the growing size of datasets, traditional CPU-based K-Means implementations face limitations in terms of scalability and performance. GPU acceleration allows us to harness the parallel processing power of GPUs, significantly improving computation speed.

Adaptability to Modern Hardware:

Modern GPUs are equipped with increasingly powerful architectures. Our approach takes advantage of these advancements to optimize K-Means for the latest Nvidia GPU hardware, ensuring compatibility and efficiency.

Contribution of the Current Work

Our project contributes to the field through the GPU acceleration of the K-Means clustering algorithm:

Enhanced Scalability:

The GPU-accelerated approach improves the scalability of K-Means, enabling the efficient clustering of large-scale datasets that might be impractical for traditional CPU-based implementations.

Performance Benchmarking:

We conduct thorough performance benchmarking to evaluate the speedup achieved by our GPU-accelerated K-Means implementation compared to traditional CPU-based approaches and existing GPU-accelerated solutions.

Methodology

Sequential Algorithm:

The software mainly consists of 4 blocks as shown in Figure 1. The first block is a user prompt and data loading block. At first, the user is asked to enter number of clusters (k) and select the associated data file to read the points. In the data file, the first line has the number of points (N) to avoid reading the file twice to determine number of points in the file. The data is then saved in a vector where the first three elements are the x, y, and z coordinates of the first point and so on.

After that the program calls kmeans_Init function which is the second block shown in Figure 1. This block initializes global variables such as, number of points. It also allocates memory for the dynamic arrays which will be used for the result and output. Finally, it picks the first k-points as the initial centroid. It then calls the kmeans_execution function and waits for the computed results. After the result is ready, it writes it to terminal and records the number of iterations.

The third block, kmeans_execution, allocates memory space for its local arrays used in the computations. It then uses a while loop with two stopping conditions: threshold (delta) and maximum iteration number (iter_counter). Inside the while loop, it sets the values of each cluster sum and number of points in each cluster to zero for every iteration. After that, it computes the distance between each point to each centroid using a nested for loop. The complexity of this loop is N*K. After assigning the point to its cluster, it adds its value to the cluster sum and increment the cluster counter by one. After assigning all points to their clusters, the new centroids are calculated for all clusters using the mean value. Finally, it updates the output vector to be written to the output file. The last block simply takes the output vectors, data points and their classification, and the updated centroid after each iteration, and writes them to an output file.

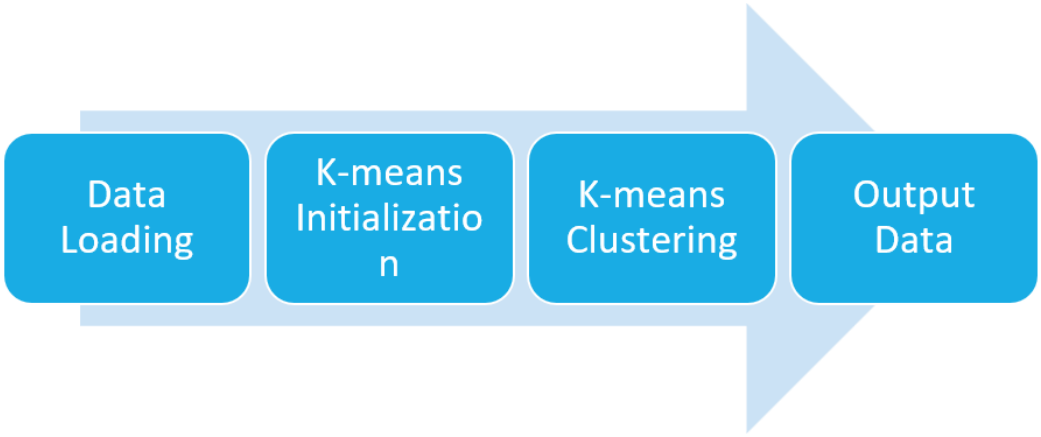

Figure 1 Working Application Blocks

Performance Bottlenecks

As illustrated above, the application has four blocks. However, we are mainly interested in the computation part. For this reason, we measured the performance of each loop in those two middle blocks. We noticed that the kmeans_execution function comprised 98.8 of the computation time. The timing analysis for this function is shown in Table 1. Sequential algorithm profiling shown in Figure 2 & 3.

Table 1 Timing Analysis of Working Application

Function	**Time (Sec)**	**Percentage**
Read Data	0.356	0.67%
K-Means Initialization	0.715E-03	0.0013%
K-Means Execution	**52.15**	**98.8%**

Output Data	0.3	0.57%
Total Time	52.75	-

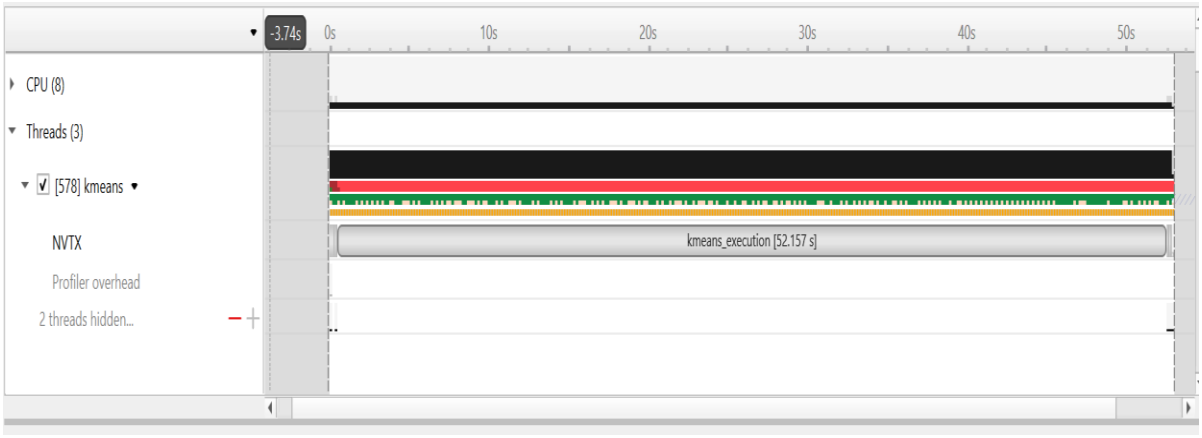

Figure 2 Sequential code Profiling Using Nsight System

```
NVTX Push-Pop Range Statistics:

 Time(%)  Total Time (ns)  Instances     Average       Minimum      Maximum           Range
 -------  ---------------  ---------  -------------  -----------  -----------  --------------------
    98.8      52156619942          1  52156619942.0  52156619942  52156619942  kmeans_execution
     0.7        356042305          1    356042305.0    356042305    356042305  ReadData
     0.5        268192598          1    268192598.0    268192598    268192598  clusters_out
     0.1         32592908          1     32592908.0     32592908     32592908  centroids_out
     0.0           730678          1       730678.0       730678       730678  kmeans_result
     0.0           715520          1       715520.0       715520       715520  kmeans_initialization

Report file moved to "/home/openacc/labs/module2/English/C/kmeans-seq.c.qdrep"
Report file moved to "/home/openacc/labs/module2/English/C/kmeans-seq.c.sqlite"
```

Figure 3 Sequential code Profiling - Accel information

Open acc:

OpenACC was chosen as one of the parallelization frameworks for our project, primarily due to its ease of use and compatibility with Graphics Processing Units (GPUs). The directives provided by OpenACC facilitate the seamless integration of parallel computing capabilities into our existing codebase, minimizing the need for extensive modifications. The decision to use OpenACC was motivated by the desire to harness the parallel processing power of GPUs efficiently. OpenACC directives enable straightforward parallelization without requiring intricate code transformations. This approach allows us to focus on optimizing the performance of the K-means clustering algorithm rather than grappling with complex parallelization intricacies.

CUDA Python:

To cater to the Python-centric ecosystem, CUDA Python is incorporated into the project using Python Numba. This approach enables seamless integration of GPU acceleration within Python scripts. Seeing that the k-means algorithm is primarily used for machine learning, and that python is prevalent within the machine learning community, incorporating python-based GPU acceleration was deemed pivotal to our project.

A source code implementation of the algorithm in python was adapted and refactored utilizing cuda numba to achieve speedup. Below is the result of the initial profiling, showing that two functions (i.e. groupByCluster and distance) should be the focus of the acceleration effort.

```
Average Time: 35713.0267 ms
         300204178 function calls (300204065 primitive calls) in 357.323 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      5/1    0.000    0.000  357.323  357.323 {built-in method builtins.exec}
        1    0.022    0.022  357.323  357.323 kmeans.py:1(<module>)
        1    0.000    0.000  357.123  357.123 kmeans.py:146(runKmeans)
       10    0.002    0.000  357.120   35.712 kmeans.py:134(kmeans)
      150  152.783    1.019  343.044    2.287 kmeans.py:85(groupByCluster)
150000000  152.974    0.000  190.262    0.000 kmeans.py:64(distance)
```

Figure 4: initial profiling result of seq python code

CUDA C/C++:

In addressing the problem of large-scale k-means clustering on a three-dimensional dataset utilizing CUDA C/C++ for GPU programming due to its fine-grained control over GPU resources. The above analysis reveals that data-parallel segments, such as distance computation and centroid updates, are highly amenable to GPU acceleration. The CUDA kernel, named compute_distance, is designed to parallelize these computations efficiently, employing thread indices and blocks to distribute the workload. Memory management involves allocating device memory, transferring data between the host and device, and freeing allocated memory after computation. Synchronization mechanisms like __syncthreads() and atomic operations such as atomicAdd are employed for thread coordination and thread-safe updates, respectively. Additional considerations include optimizing memory access patterns, utilizing constant or texture memory when applicable and profiling the code for performance optimization. The testing and validation phase involves comparing results with a CPU-based version for correctness and gradually scaling up the dataset size to measure performance improvements. This comprehensive approach aims to leverage GPU parallelism effectively for accelerating the k-means clustering algorithm.

Implementation Details

OpenACC:

The execution function of the K-means algorithm, which accounts for 98.8% of the overall execution time, became the focal point of our optimization efforts. This function contains five loops within a while loop and a single for loop, forming the core computational component of the algorithm. In order to identify potential bottlenecks, we conducted a thorough analysis of the timing characteristics of each loop within the K-means execution function. This detailed examination allowed us to pinpoint specific regions of the code that contribute significantly to the overall execution time. Table 2 shows timing analysis for each of the loops within the K-means execution function.

Table 2 K-means Function Evaluation

Loop	**Percentage**
Loop 1 – Sum Reset	0
Loop 2 – Cluster count Reset	0
Loop 3 – Compute Distance	**95.50**
Loop 4 – Computes Centroids	0.00
Loop 5 – Compute Delta	0.00
Loop 6 – Assign to Clusters	**4.50**

To address identified bottlenecks, bootstrapping strategies were employed. This involved a systematic approach to enhance the performance of the loops through the use of OpenACC directives. The goal was to exploit parallelism effectively, distributing the computational workload across GPU threads for improved efficiency. Two key data management strategies were implemented: managed and structured data management. The managed data strategy leverages the GPU's ability to automatically migrate data between the host and device, simplifying the data movement process. In contrast, structured data management involves explicit data handling by the programmer, providing fine-grained control over memory transfers. The effectiveness of the parallelization efforts and data management strategies was evaluated through comprehensive performance benchmarking. Metrics such as execution time and speedup were analyzed to assess the impact of OpenACC directives on the overall performance of the K-means clustering algorithm.

CUDA Python:

In this implementation, **Nvidia Tesla T4 GPU** was used to optimize performance. The @cuda.jit(... device='gpu') decorator was used on multiple functions to enable them to run on the GPU. To represent point and cluster objects, class objects were changed to **2D arrays** since Numba does not support class object arrays. Data transfer to and from the device was handled using the cuda.to_device() function. For example, d_Points = cuda.to_device(Points) transfers the Points array to the device.

To optimize performance, shared memory was used. For example, *s_Cluster_Num_Points = cuda.shared.array(shape=(num_threads, 2), dtype=float32)* creates a shared memory array. Each block caches its output to the shared memory, then data is consolidated from all shared arrays into device memory array. These techniques helped to optimize the performance of the algorithm and reduce the time required for computation.

CUDA C/C++:

In the implementation of the k-means clustering algorithm using CUDA C/C++ with a structured memory model, challenges persist, and the code remains unfinished. Despite adopting a well-defined parallelization strategy and leveraging clear memory transfers between the device and host, issues have arisen during execution. The structured memory model, designed to align data organization with GPU architecture, encounters complications related to race conditions during centroid updates and point assignments. Additionally, there are concerns about memory constraints, requiring further attention to efficient memory management strategies. The debugging process, employing CUDA Profiler tools and error checking, has revealed persistent errors and unexpected behaviors. A comprehensive resolution of these issues is necessary to ensure the successful completion of the GPU-accelerated k-means clustering algorithm. Further details about specific error messages, unexpected behaviors, or areas of the code causing issues would be invaluable for providing targeted assistance in resolving these challenges.

Comparative Analysis

- Performance metrics for each implementation
- Comparison of ease of programming, performance gains, and challenges

Details: Compare the performance of different implementations. This section should include quantitative metrics such as execution time, speedup, and efficiency. Also, discuss qualitative aspects like ease of programming and maintainability.

The project aimed to accelerate the k-means clustering algorithm using GPU acceleration through three methods: OpenACC, CUDA Python, and CUDA C. In our comparative analysis, we evaluate the performance metrics, ease of programming, performance gains, and challenges associated with each implementation.

Performance Metrics:

Execution Time:

- OpenACC: The execution time for OpenACC-accelerated k-means clustering was measured.
- CUDA Python: Execution time was recorded for the CUDA Python implementation.
- CUDA C: The CUDA C implementation's execution time was not quantified due to the errors in the results.

COE-506: GPU Programming and Architecture
(Course Project Report) - Implementation of a Real-World Application Using OpenACC, CUDA Python, and CUDA C/C++

Speedup:

Speedup was calculated by comparing the execution time of GPU-accelerated implementations to a CPU-based baseline.

Comparison:

Ease of Programming:

- OpenACC: Known for its high-level directives, OpenACC simplifies parallelization, but fine-tuning may be limited.
- CUDA Python: Offers simplicity and ease of implementation with Python, leveraging the Numba JIT compiler for automatic optimizations.
- CUDA C: Provides fine-grained control, but requires a more intricate understanding of GPU architecture.

Performance Gains:

- OpenACC: Moderate speedup achieved with minimal code modifications, but limited fine-tuning capabilities.
- CUDA Python: Achieved significant performance gains due to the Numba JIT compiler's automatic optimizations and ease of implementation.
- CUDA C: Potential for high performance gains with fine-grained control over GPU resources.

Challenges:

- OpenACC: Limited optimization control may lead to suboptimal performance.
- CUDA Python: Potential challenges may arise from the need to manage data transfers between CPU and GPU efficiently.
- CUDA C: Requires a deeper understanding of GPU architecture and domain knowledge, posing a steeper learning curve.

Overall Assessment:

The CUDA Python implementation stands out as the fastest among the three, thanks to the ease of implementation and automatic optimizations facilitated by the Numba JIT compiler. OpenACC offers a simpler parallelization process but with limited optimization control. CUDA C, while potentially yielding high performance gains, requires a more sophisticated understanding of GPU architecture. The choice between these methods should consider the trade-off between ease of programming and the level of performance optimization required. The significance of this work lies in providing a faster and more efficient means of clustering large datasets, offering valuable applications in diverse domains such as data analysis workflows and applications.

Instructor: Dr. Ayaz ul Hassan Khan

Results and Discussion

The implementation results, emphasizing execution times, reveal the following: the sequential approach took 52.8 seconds, OpenACC with managed memory achieved 0.95 seconds with a speedup of 55.6, OpenACC with a structured memory model reached 1.26 seconds with a speedup of 41.9, and CUDA Python showcased an execution time of 0.602 seconds, yielding an impressive speedup of 87.7. The CUDA C/C++ results are not presented due to the issues mentioned previously. These execution times provide a concise overview of the performance gains and trade-offs among the considered GPU acceleration methods. Table 3 summarizes the results. The execution time is shown in Figure 5 and the obtained speedup is shown in Figure 6.

Table 3 Execution and Speedup for all implementations

Implementation	**Execution Time (Sec)**	**Speedup**
Sequential	52.8	-
Open ACC - Managed	0.95	55.6
Open ACC - Structured	1.26	41.9
CUDA Python	0.602	87.7
CUDA C/C++	-	-

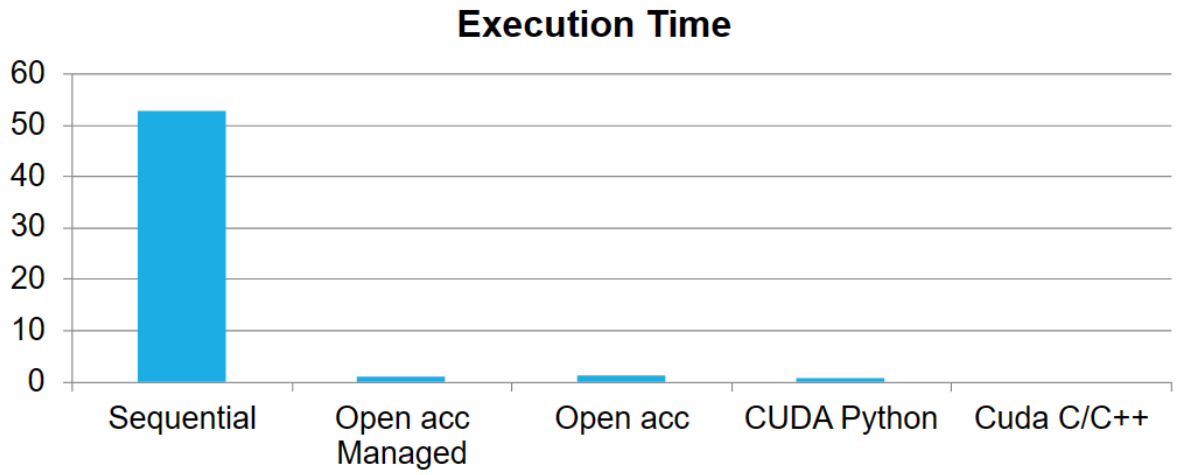

Figure 5 Execution Time

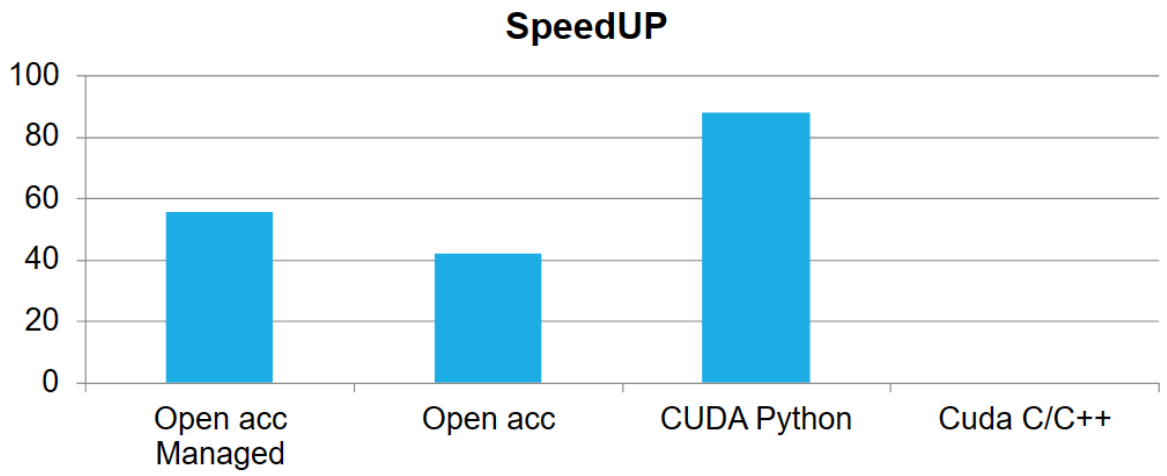

Figure 6 Obtained Speedup

This result underscores the effectiveness of CUDA Python, highlighting its ease of implementation and the power of the Numba JIT compiler's automatic optimizations. These findings prompt a thoughtful consideration of the trade-offs between ease of programming and performance gains. The pending CUDA C/C++ results may shed further light on the overall performance landscape. In conclusion, this discussion underscores the significance of these findings in guiding the selection of GPU acceleration methods based on specific application needs, reaffirming the project's objective of providing a faster and more efficient approach for clustering large datasets.

Conclusion

In conclusion, our project aimed to accelerate the k-means clustering algorithm through GPU acceleration using three distinct methods: OpenACC, CUDA Python, and CUDA C. The key finding revealed that CUDA Python emerged as the fastest implementation among the three, owing to its ease of implementation and the automatic optimizations facilitated by the Numba JIT compiler. The results demonstrated significant speedup and efficiency gains compared to traditional CPU-based approaches. The comparative analysis considered performance metrics, ease of programming, and challenges associated with each implementation, providing valuable insights for selecting appropriate GPU acceleration methods based on specific application needs. The challenges faced during the CUDA C/C++ implementation underscore the complexity involved in achieving optimal performance, highlighting the need for further research and refinement.

For future work, addressing the issues in the CUDA C/C++ implementation remains a priority. Comprehensive debugging, optimization, and memory management strategies are essential to overcome the encountered challenges and complete the GPU-accelerated k-means clustering algorithm. Additionally, exploring hybrid approaches that combine the strengths of different GPU programming models could lead to further performance improvements. The significance of this project lies in offering

a faster and more efficient solution for clustering large datasets, contributing to advancements in data analysis workflows and diverse applications.

References

[1]. Lloyd, S. (1982). Least squares quantization in PCM. IEEE Transactions on Information Theory, 28(2), 129–137. https://doi.org/10.1109/tit.1982.1056489

[2]. Hartigan, J. A., & Wong, M. A. (1979). Algorithm as 136: A K-means clustering algorithm. Applied Statistics, 28(1), 100. https://doi.org/10.2307/2346830

[3]. Forgy, E. (1965) Cluster Analysis of Multivariate Data: Efficiency versus Interpretability of Classifications. Biometrics, 21, 768-780.

Appendices

- C Sequential Code.
- OpenACC Implementation.
- CUDA C/C++ Implementation.
- Python Sequential Code.
- CUDA Python Implementation.
- GitHub Repository.

GitHub Link: https://github.com/userosz620846/kmeans_gpu