# EFFICIENT MULTI-GPU *K*-MEANS CLUSTERING

by

Kian O'Ryan, B.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
May 2023

Committee Members:

Martin Burtscher, Chair

Byron Gao

Vangelis Metsis

## DEDICATION

To my parents, brother, and last but not least Happy and Crocket.

## ACKNOWLEDGMENTS

I want to show my utmost gratitude towards my graduate advisor, Dr. Martin Burtscher, for assisting me throughout this long, grueling, but advantageous process.

## TABLE OF CONTENTS

**Page**

## LIST OF TABLES

**Page**

## LIST OF FIGURES

**Page**

## LIST OF ABBREVIATIONS

| Abbreviation | Description |
|---|---|
| FAIIS | Facebook AI Similarity Research |
| PC | Parallel CPU |
| SC | Serial CPU |
| MG | Multi GPU |
| OG | One GPU |

## ABSTRACT

One of the best ways of digesting ever-growing large troves of gathered data is by grouping, also known as clustering, similar data points based on their attributes. A successful clusterization is accomplished by minimizing intra-cluster and maximizing inter-cluster attribute variation. Although theoretically simple, clustering has many real-life uses in fields such as astronomy, data processing, medicine, digital marketing, biology, and computer vision. One of the standard algorithms for accomplishing clusterization is k-means, also known as Lloyd's algorithm.

Lloyd's algorithm for computing clusters has a simple heuristic implementation, but due to the extreme size of the typical input target data, it can be a very time-consuming algorithm and is, therefore, a suitable parallelization candidate.

I report my novel methods for significantly speeding up this algorithm for both GPU and CPU. This was accomplished by minimizing the amount of communication between the device and the host. My implementations utilized CUDA for the GPU code and relied on OpenMP for CPU parallelization. I achieved respectable speedup levels compared to the current state of art implementations by NVIDIA and Meta. My GPU code can execute *k*-means 3000 times faster than Meta's parallel CPU and 55 times faster than NVIDIA's GPU code.

# I. INTRODUCTION

As the rate at which data collection is growing, so does our need to find more efficient methods for processing and analyzing what we collect. One pathway for evaluating data is by grouping the elements based on their likenesses. This method is more commonly known as "clustering."

The mechanisms for clustering fall under two main categories: supervised [1], [2] and unsupervised learning [3] algorithms. Although both categories can cluster data, they each have their unique strengths.

To successfully train a precise and accurate supervised learning model, we would need a large amount of training data. Generating vast amounts of training data can be a time-consuming task and may require direct human involvement. Alternatively, unsupervised learning offers the option to classify given data without requiring prior training and instead uses provided algorithms to classify data as necessary.

This paper focuses on a specific type of unsupervised clustering: centroid clustering. Centroid clustering functions by organizing the data around virtual centroids. This approach has been used in many different fields, including astronomy [4], [5], data processing, medicine [6], [7], digital marketing [8], biology [9], and computer vision.

One method of centroid clustering is applying Lloyd's algorithm, more commonly known as *k*-means. This algorithm was initially introduced by Stuart P. Lloyd in his "Least squares quantization in PCM" paper [10]. Although researchers have built on and developed different variations of *k*-means (some of which are discussed in the related work section), my multi-GPU implementation will focus on the original algorithm. At a high level, this algorithm relies on repeatedly recategorizing data points until they are successfully clustered into k unique clusters.

The following section describes this algorithm in more detail.

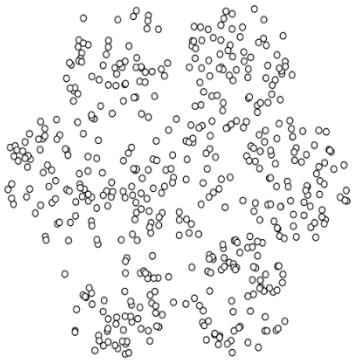
## II. LLOYD'S ALGORITHM

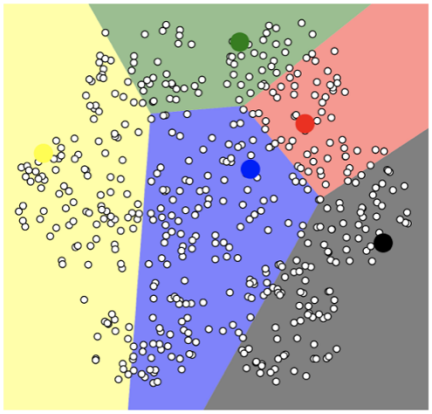
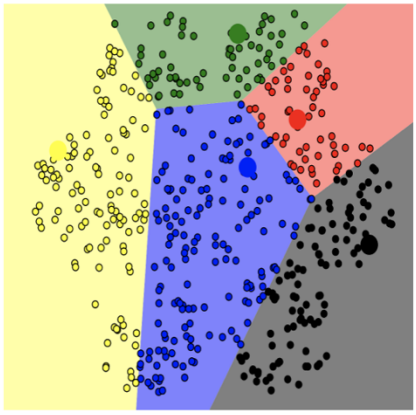
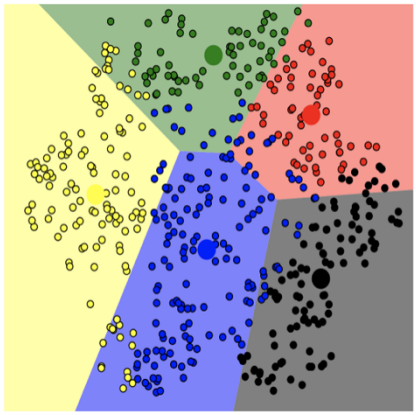
To cluster *p* points into *k* clusters using Lloyd's algorithm, the following steps need to be performed:

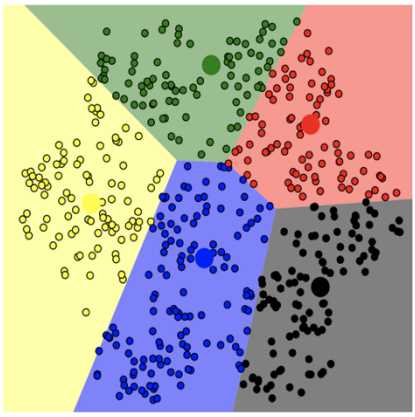
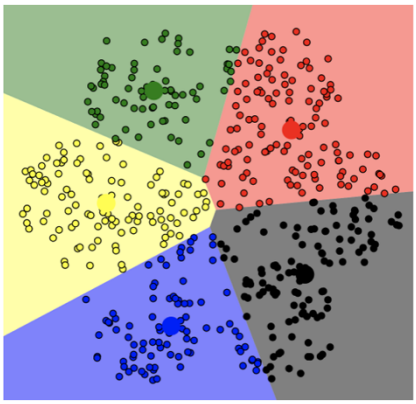
1. Select *k* arbitrary center points called "centroids"
2. Assign each point to its nearest centroid (i.e., form clusters)
3. Calculate new centroid locations by computing the mean of all points assigned to the same cluster
4. Repeat steps 2 and 3 until the centroid values no longer change

Visual representation of *k*-means clustering, as showcased in Table 1 [11] :

***Table 1.*** *Step by Step breakdown for clustering input data*

| | |
|---|---|
|  | Step 1: Original un-assigned points |

| | |
|---|---|
|  | Step 2: Selecting centroids at random (the yellow, green, blue, red, and black dots) |
|  | Step 3: Coloring points the same color as their nearest centroid |
|  | Step 4: Calculating the average distance of all same-colored points and moving the centroids to these new respective locations |

| | |
|---|---|
|  | Step 5: Recoloring points, so they match their nearest centroid (after centroid location update) |
|  | Step 6: Repeat the last two steps until each cluster's point median matches the corresponding centroid location. |

Although implementing the *k*-means algorithm seems straightforward, it can fall victim to over-clustering or under-clustering.

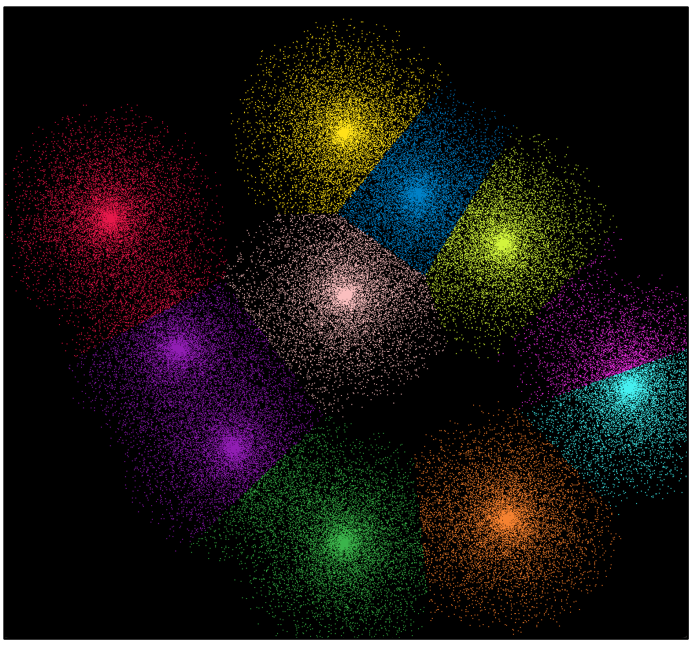


***Figure 1.*** *Example of both under and over-clusterization of points*

Figure 1 shows an example of both under and over-clusterization of the data. The purple points (bottom left) incorrectly include two separated clusters. In contrast, the pink points (center right) belong to their adjacent yellow and cyan neighbors but were mistakenly clustered as the same group. Figure 2 illustrates the correct categorization of the same data.

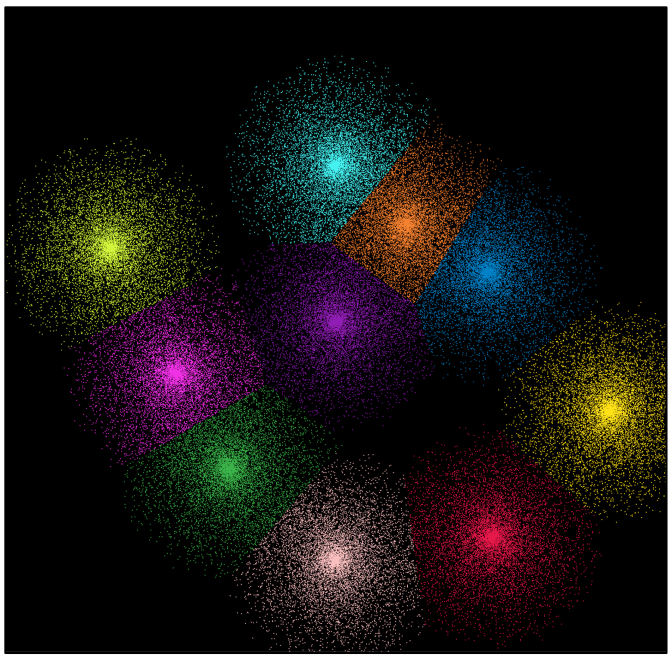


***Figure 2.*** *Correct clusterization of 100,000 points using ten centroids.*

## III. MY IMPLEMENTATION OF LLOYD'S ALGORITHM

I re-cluster the original data multiple times using new centroid starting points to decrease the odds of over-clustered results. Additionally, to find the best clustering attempt, I measure the intra-cluster variance. Under the assumption that the program is using the correct number of clusters, I keep track of the attempt with the lowest variance value and output its point-centroid assignment.

Serial CPU approach:

1. Select *k* arbitrary center points called "centroids"
2. Assign each point to its nearest centroid (i.e., form clusters)
3. Calculate new centroid locations by computing the mean of all points assigned to the same cluster
4. Repeat steps 2 and 3 until the centroid values no longer change

## IV. PARALLEL CPU IMPLEMENTATION

My parallel CPU code strictly follows the general approach but relies on OpenMP to utilize multiple CPU threads. For my implementation, I focused on updating the most computationally heavy section of my program to use threads. Lloyd's *k*-means requires every point to be compared with every centroid. This comparison is the computationally heaviest segment of the code, which is why I chose to parallelize it.

An OpenMP pragma is used to divide the points between the threads. Once assigned a point, a thread is responsible for calculating the point-centroid distance and updating the point's centroid assignment, and the threads continue until they have processed all of their assigned points. As threads find the nearest centroid for each point, they must also update the nearest centroid's values correctly. These values (the count and details of the points assigned to each cluster) are used to calculate the new centroids.

To avoid a possible data race [12], which could be caused by all threads attempting to read and write to the same information associated with each centroid, I elected to provide each thread with its own private copy of each centroid. Since these copies are local to each thread, the threads are able to update values without the possibility of a data race. Once the threads processed all their assigned points and updated the required information, I utilized OpenMP's reduction clause to combine the cluster information, which is used to compute the centroid information.

**A. N-Dimensionality Support for CPU**

The implementation covered by the previous section would only function properly if the input data has two dimensions. Although the previous implementation can adapt to higher input dimensions by using resizing techniques such as Principal Components Analysis [13], which is more commonly known as PCA. Although PCA can assist with dimensionality reduction, they, unfortunately, can introduce unwanted side effects such as noise and also may remove or diminish important attributes and impose additional computational overhead. Due to these factors, I have decided to add native support for higher dimensions to my CPU and GPU implementations.

For the CPU implementation to successfully cluster any input size, regardless of dimension count, I have elected to store the input data for all point and centroid dimensions within two separate arrays. Figure 3 illustrates how different coordinates for the input and centroids are stored:

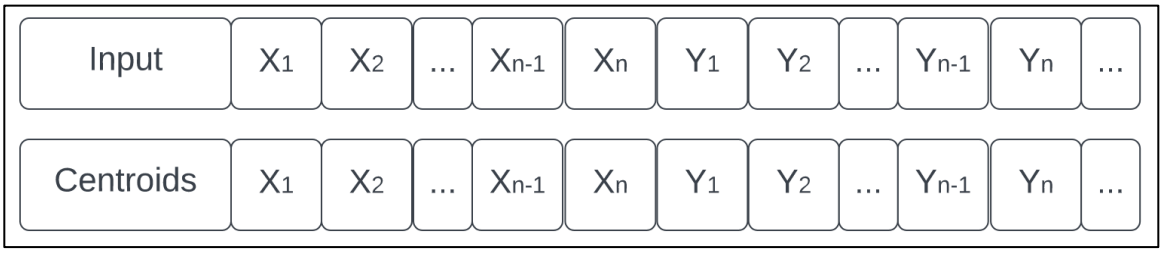



***Figure 3.*** *Visualization of arrays that store input and centroid information.*

Since all dimensions are stored within the same array, the following formula is used to navigate them:

$$Element\ index * Dimension\ count + Dimension$$

In the above formula, “Dimension count” is a constant value set to the total input dimensions

count. Alternatively, the index and dimension depend on the specific element being accessed.

## V. SINGLE GPU IMPLEMENTATION

My single-GPU implementation is very similar to the general approach. Still, since the calculation will be done on two separate pieces of hardware, the host (CPU) and device (GPU), I need to choose when and how these two devices communicate.

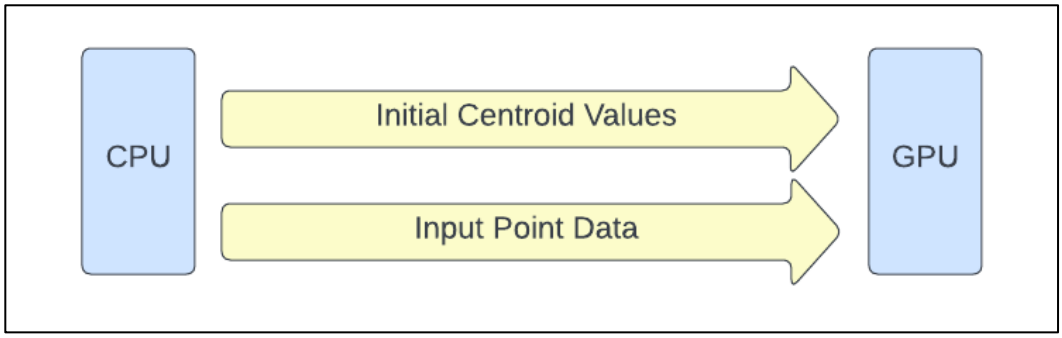



***Figure 4.*** *Initial centroid values and point data are only transferred once throughout the computation.*

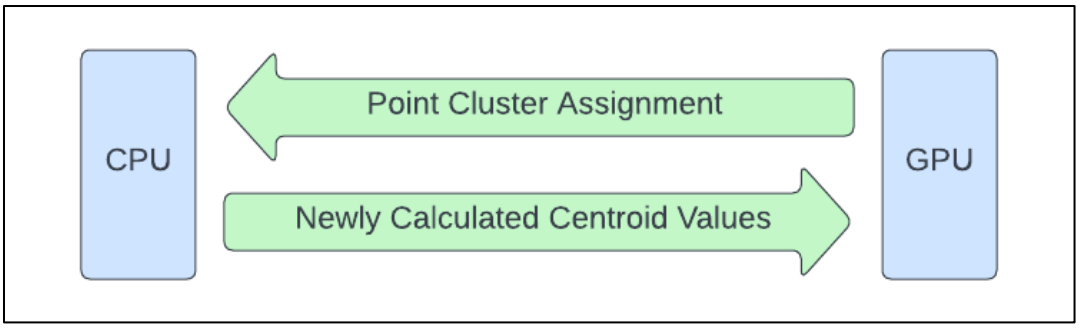



***Figure 5.*** *This step is repeated until convergence.*

As shown in Figure 4, the host must communicate the point and centroid data to the device, and the device sends the host the point cluster association (Figure 5). Data transfer between host and device can be time-consuming. Therefore, I minimize the times the host and device communicate by only transferring information when necessary. Consequently, excluding the point centroid assignment, the point data is only transmitted to the device once. Moreover, the host and device communicate new centroid values and point cluster assignments to each other.

This step is repeated until convergence.

To speed up the process and take advantage of the GPU's computing power, I parallelized this code by dividing the points among different threads. Since GPUs can generate many threads, as soon as the GPU receives point data from the CPU, each point is designated to a single GPU thread. Each thread is responsible for finding the closest centroid to its assigned point so that the point's centroid assignment can be updated accordingly. After GPU threads have finished their task, the new centroid assignment data are sent back to the CPU to be used for new centroid calculations.

## VI. MULTI-GPU IMPLEMENTATION

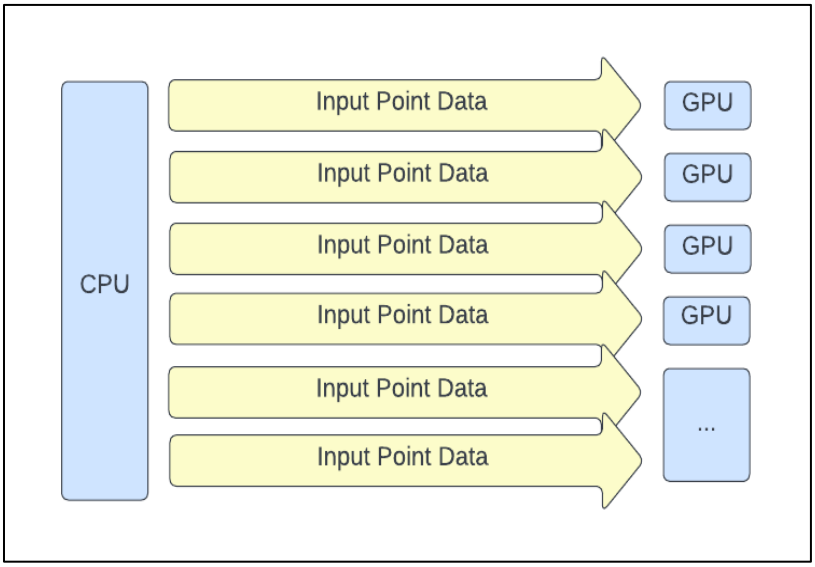



***Figure 6.*** *Each GPU only receives a portion of the Points to work on*

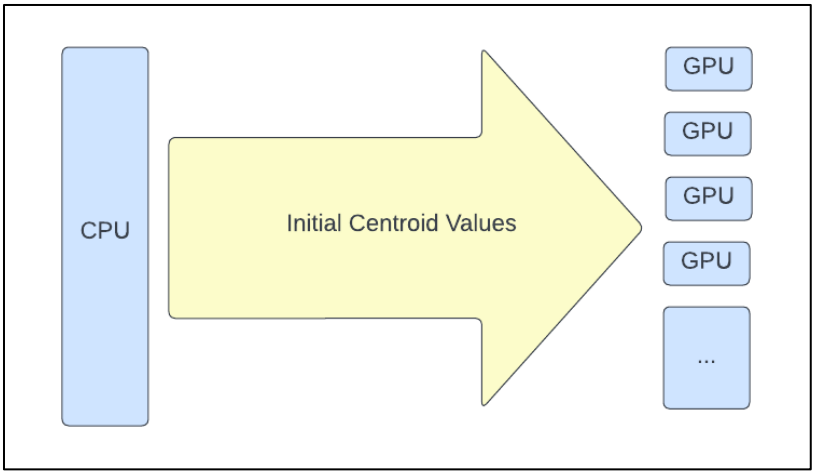



***Figure 7.*** *All GPUs receive data on every centroid value.*

Although my multi-GPU code resembles the single-GPU version, it requires additional steps to guarantee correct cluster outputs. Throughout this implementation, I keep two factors in mind: How much data and how often the CPU and GPUs should communicate. Additionally, since each GPU is responsible for its own portion of the calculation, it is essential to decide if the GPUs need to communicate with each other throughout the process.

If the CPU transfers data about all points to every GPU, then the GPUs would output duplicate responses; therefore, since transferring data between the CPU and the GPU is time-consuming, this would not only provide no speedup and instead cause a slowdown due to the unnecessary communication overhead. To resolve this issue, I need to ensure that every GPU gets access to a unique portion of the points. This step is accomplished by assigning each GPU a roughly equal number of points. Since each GPU receives its own unique portion, this eliminates the possibility of multiple GPUs producing duplicate work.

The information about the centroids needs to be transferred to every GPU (and re-transferred when the centroids are updated). This is necessary because each GPU will need access to all centroids to calculate the correct point assignment; since there are many more points than clusters, transferring all centroids to all GPUs does not take long as it only communicates a small amount of data.

As mentioned previously, each GPU is responsible for producing point assignments on only their own portion of points. Once a GPU has accomplished its task, it calculates the new point centroid assignments for future centroid calculations. Afterward, the CPU requests new point assignments from each GPU and uses this data to compute new centroid values. As previously stated, these steps are repeated for the serial CPU implementation until centroid values are no longer updatable (point assignments have stayed the same throughout two consecutive runs). It is essential to keep in mind that, since the GPUs only need to communicate with the CPU and not each other, the code executed by each GPU is the same as the single-GPU implementation, but each GPU only receives part of the point data (point count divided by GPU count).

**A. N-Dimensionality support for GPU**

Adding more than two-dimensionality support for the GPU is done similarly to the CPU implementation, but the dimensions are stored in a new fashion relative to each other. Unlike the CPU, the GPU model contiguously stores each element's dimensions' data, as shown in Figure 8.

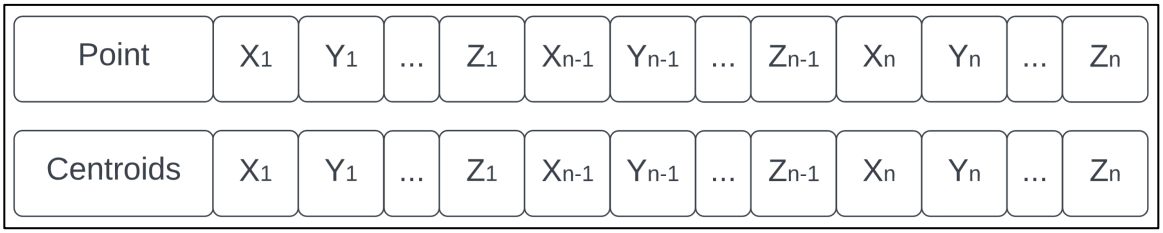



***Figure 8.*** *Visualization of arrays that store input and centroid information for the GPU implementation.*

Storing information in this manner streamlines the necessary communication between the CPU and GPU since the algorithm can allocate points to each GPU by simply dividing the point array by the total number of GPUs.

Under this point design the point and centroid arrays are iterated using the following formula:

$$\text{Element index} + \text{Dimension count} * \text{Dimension}$$

## VII. RELATED WORK

Alguliyev et al. [14] explored creating a more effective *k*-means algorithm specializing in clustering large inputs. They advocate for dividing the dataset into several different batches. To find the optimal batch size, the authors use a method suggested by Parker & and Hall [15]. Once an optimal batch size is calculated, the original dataset is distributed into an equal number of batches. These are independently clustered and consequently output new centroids. Subsequently, the centroids are clustered, and the final centroid set is generated. During the concluding step for this method, the original input dataset is mapped to the last set of centroids, thus clustering the original dataset. This process is outlined in Figure 9.

This algorithm is similar to my multi-GPU implementation. However, I divided my data into batches as I send portions of the input points to each GPU, whereas this paper did so as a response to large input sizes. In addition, their batching method would not offer any advantages when used with smaller input sizes.

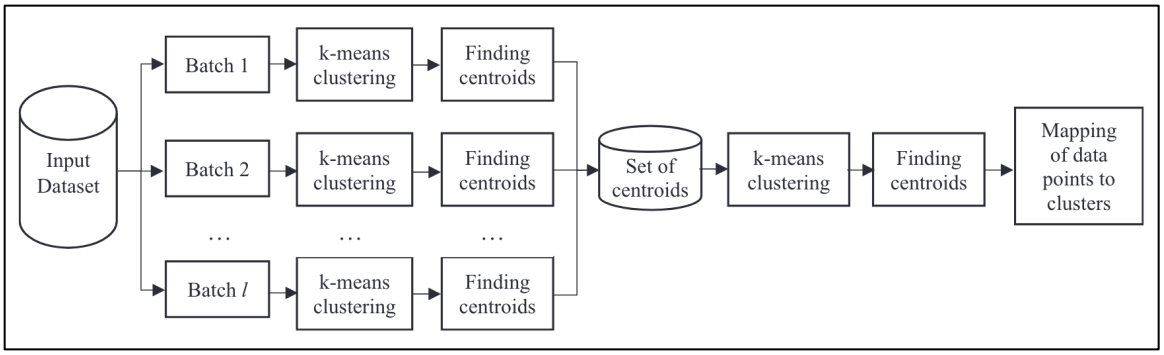



***Figure 9.*** *Flow diagram for method suggested by Alguliyev et al. [12]*

For an accelerated approach, Cuomo et al. [16] investigate a new approach to help speed up the algorithm by addressing two limitations: the space limitation of GPUs and the host-device

communication time. They propose resolving these constraints by limiting the number of times the host and device communicate and focusing on using lighter data structures to reduce transfer size and, as a result, transfer time. Moreover, to increase global memory throughput, following the most optimal access patterns according to the device's capabilities is recommended, relying on data sizes that meet alignment requirements and padding data as necessary.

Similar to my execution, this method minimizes the frequency and size of communication between the host and the GPU. However, unlike my approach of atomically updating point cluster assignments on the GPUs, they depend on a master thread. This can substantially slow down the execution of the code. They also use three separate vectors for data transfer, whereas I break down my input and centroid information into ten different arrays, which provides me with more control over the device host communication patterns. Furthermore, Cuomo et al. were able to gain a speedup by parallelizing the calculations on the CPU, but bigger input sizes made these gains irrelevant.

Kanungo et al. introduce a new implementation for *k*-means [17]. Their filtering algorithm relies on a kd-tree [18], [19] as its only central data structure. To distribute the points correctly into a kd-tree, the paper suggests the following: "Each node of the kd-tree is associated with a closed box, called a cell. The root's cell is the bounding box of the point set. If the cell contains at most one point (or, more generally, fewer than some small constant), then it is declared to be a leaf. Otherwise, it is split into two hyperrectangles by an axis-orthogonal hyperplane. The points of the cell are then partitioned to one side or the other of this hyperplane. (Points lying on the hyperplane can be placed on either side.) The resulting subcells are the children of the original cell, thus leading to a binary tree structure."

This algorithm requires data pre-processing since it needs a k-d tree to function.

Additionally, this approach has only been implemented serially, and it is not easily parallelizable. Due to this, it is at a severe disadvantage since it is not exploiting the computational power provided by the GPU.

## VIII. RELATED ALGORITHMS

This section describes several other variations of *k*-means. Although, as mentioned previously, *k*-means was originally introduced by Lloyd, researchers have offered new algorithms that introduce novel tactics to achieve the same goal. These algorithms include but are not limited to *k*-medians, Fuzzy *C*-Means, and Mini-batch *k*-means.

The first algorithm, *k*-median, is similar to the original *k*-means algorithm, but as the name implies, it relies on calculating the median rather than the mean to cluster points [20]. Additionally, instead of depending on the Euclidian distance, the median value is calculated using the Manhattan-distance [21] formula. Figure 10 [22] illustrates the difference between these distinct distance measurements.

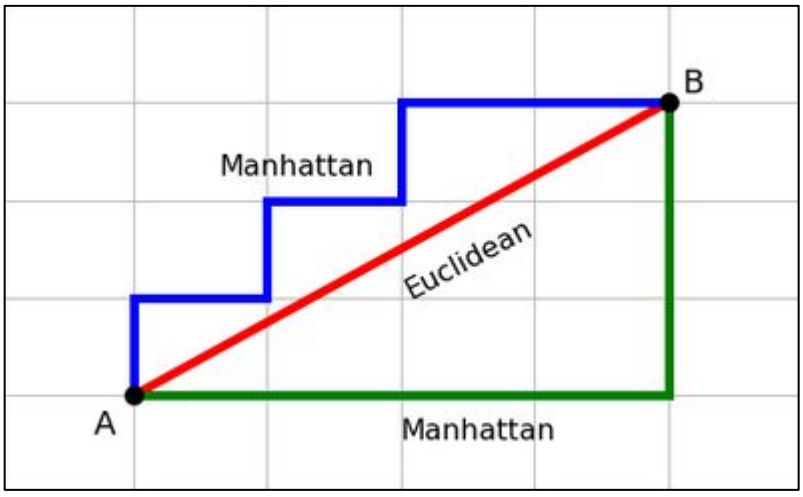



***Figure 10.*** *The difference between Euclidean and Manhattan distances.*

Another method is Fuzzy C-Means, which can be a suitable algorithm for inputs with noise or conditions where a point can belong to multiple clusters [23]. This algorithm minimizes intra-cluster similarities while maximizing inter-cluster differences [24]. This is done by assigning a

cluster membership probability to each point. Consequently, points closer to a cluster's center have a higher rating; inversely, points further from the center have a lower rating.

Third, Mini-batch *k*-means is offered as a solution for real-time clustering [25]. Though this algorithm offers faster clustering for smaller given inputs, it can suffer from increased computational costs with bigger input sizes [25]. Mini-batch *k*-means functions by dividing the data into fixed-sized batches, which are used to update the centroids.

## IX. EXPERIMENTAL METHODOLOGY

The goal of my thesis is to design a high-speed multi-GPU *k*-means implementation. The primary metric for comparing different versions of my code is the running time of the *k*-means algorithm. Since input data is provided in practice, I did not include point generation in my measurements.

To increase flexibility, I generate the points dynamically. This allows us to choose how many points and clusters (and therefore centroids) the test would run with.

The point generation requires the point and cluster counts and a starting seed to be fed to the randomizer. Using the same seed across multiple runs (and keeping cluster and point count the same) guarantees that my tests across CPU and GPU experiments are based on the same input values and start with the same initial centroids. These factors allow me to generate as many points and clusters as I require for each test, and, as a result, I can easily observe algorithm efficiency across both large and small point and cluster counts.

Our experimental timings were measured on Linux servers with the following details:

System:

- Linux 5.17.9-200.fc35.x86_64 x86_64
- Main memory - 128 GB

CPU:

- Two Intel Xeon Gold 6226R CPU @ 2.9 GHz
- Number of Cores - 32 (2x 16-core NUMA)
- Number of Threads – 64
- Cache – 22 MB

GPUs:

- Two NVIDIA GA102 – GeForce RTX 3080 TI
- CUDA compute capability - 8.6
- Base and max frequency - 1.37 GHz | 1.67 GHz
- Memory size -12 GB
- SM count - 80
- L1 & L2 cache: 128 KB (per SM) | 6 MB

Table 4 includes a speedup comparison between serial and parallel CPU/GPU measurements. To calculate these values, I first measured the amount of time each implementation takes to cluster a given input set. Afterward, to calculate the speedup, I divide one measurement by the other, i.e., to calculate the speedup of parallel GPU vs. serial CPU, the following formula would be used:

$$Parallel\ GPU\ vs\ Serial\ CPU = \frac{Serial\ CPU\ time}{Parallel\ GPU\ time}$$

To make the rest of the chapter more comprehensible, Table 2 includes a central list of abbreviations.

***Table 2.*** *Different abbreviations used in this thesis.*

| | |
|---|---|
| Serial CPU | SC |
| Parallel CPU | PC |
| One GPU | OG |

| | |
|---|---|
| Multi GPU (2 GPUS) | MG |

After executing the algorithm for 10 clusters, and two dimensions, I gathered the following measurements showcased in the following tables. Table 3 includes the absolute runtimes and as mentioned previously, Table 4 contains the speedup comparisons.

***Table 3.*** *Time (seconds) measurements for different implementations of k-means.*

| Point Count | SC | PC | OG | MG |
|---|---|---|---|---|
| 500 | 0.0013 | 0.0030 | 0.0004 | 0.0005 |
| 5000 | 0.0073 | 0.0036 | 0.0011 | 0.0011 |
| 50000 | 0.0643 | 0.0050 | 0.0013 | 0.0015 |
| 500000 | 1.1761 | 0.0426 | 0.0066 | 0.0052 |
| 5000000 | 7.8957 | 0.2512 | 0.0297 | 0.0203 |
| 50000000 | 69.7024 | 2.2790 | 0.2477 | 0.1589 |
| 500000000 | - | 34.3035 | 3.9415 | 2.4306 |

***Table 4.*** *Speed up comparison of different implementations.*

| Point Count | PC vs. SC | OG vs SC | OG vs. PC | MG vs. SC | MG vs. PC | MG vs. OG |
|---|---|---|---|---|---|---|
| 500 | 0.42 | 3.23 | 7.69 | 2.71 | 6.45 | 0.84 |
| 5000 | 2.04 | 6.85 | 3.36 | 6.65 | 3.25 | 0.97 |
| 50000 | 12.84 | 48.69 | 3.79 | 43.59 | 3.39 | 0.90 |
| 500000 | 27.63 | 178.08 | 6.44 | 224.27 | 8.12 | 1.26 |
| 5000000 | 31.44 | 265.68 | 8.45 | 389.74 | 12.40 | 1.47 |
| 50000000 | 30.58 | 281.43 | 9.20 | 438.59 | 14.34 | 1.56 |
| 500000000 | - | - | 8.70 | - | 14.11 | 1.62 |

As the number of input points increases, so does the possible speedup by both the CPU and GPU parallel codes. Since my GPU, and GPUs in general, can create thousands of threads, they

are more efficient at bigger tasks. For the GPU implementation, as anticipated, the speedup went up as the input point count increased; in other words, GPUs are not suitable for parallelizing small-scale jobs. My GPU codes, on single and two GPUs, reached an efficiency plateau above 50,000,000 input values, and they both saw a lower speedups compared to the CPU codes. Nevertheless, they were much faster than the CPU codes -- about 439 times faster than the CPU for the largest tested input size.

Compared to the single GPU time measurements, my multi GPU implementation has a maximum speedup of 1.62 out of 2.0 (the maximum possible speedup). The theoretical maximum is 2 since I worked with two GPUs and, therefore, twice as much computational power. The 19% drop, when compared to 2.0, is due to the parallelization overhead, in particular the device-host communication (described in detail in Section 6).

## A. Timing Evaluations

This section covers the timing comparisons for my implementation of *k*-means clustering against Meta's "Facebook AI Similarity Search" (FAISS) parallel CPU and NVIDIA's state-of-the-art cuML GPU algorithms. The timing results were gathered using a different dataset than the one mentioned in the previous section. The new input dataset includes a more extensive selection of point counts and introduces larger than two dimensions. This new pool ranges from 100 points and two dimensions to a maximum of 100,000,000 points and 128 dimensions.

To minimize testing anomalies, each experiment was repeated 10 times for each input, and the median value was selected as the final time measurement.

Table 5 showcases the test results in seconds. Serial, OpenMP, and GPU columns for my implementations. All tests were executed on the same hardware as mentioned in the previous section, Serial, OpenMP, and FAISS were executed on the CPU, whilst GPU and cuML were executed across all available GPUs (two).

***Table 5.*** *Time (seconds) measurements for my implements, cuML and FAISS. OOM fields indicate out-of-memory errors.*

| Points | Dim. | Serial | OpenMP | GPU | cuML | FAISS |
|---|---|---|---|---|---|---|
| 100 | 2 | 0.001 | 0.009 | 0.002 | 0.028 | 0.14 |
| 100 | 3 | 0.001 | 0.009 | 0.002 | 0.028 | 0.19 |
| 100 | 4 | 0.001 | 0.010 | 0.003 | 0.027 | 0.16 |
| 100 | 5 | 0.001 | 0.010 | 0.003 | 0.027 | 0.13 |
| 100 | 9 | 0.001 | 0.009 | 0.003 | 0.028 | 0.15 |
| 100 | 15 | 0.001 | 0.009 | 0.004 | 0.028 | 0.15 |
| 100 | 64 | 0.002 | 0.011 | 0.013 | 0.044 | 0.14 |
| 100 | 128 | 0.003 | 0.013 | 0.024 | 0.064 | 0.19 |
| 10000 | 2 | 0.059 | 0.018 | 0.008 | 0.036 | 0.4 |
| 10000 | 3 | 0.076 | 0.020 | 0.011 | 0.036 | 0.44 |
| 10000 | 4 | 0.096 | 0.021 | 0.016 | 0.036 | 0.46 |
| 10000 | 5 | 0.087 | 0.016 | 0.014 | 0.036 | 0.51 |

| 10000 | 9 | 0.114 | 0.022 | 0.024 | 0.039 | 0.52 |
|---|---|---|---|---|---|---|
| 10000 | 15 | 0.192 | 0.031 | 0.041 | 0.039 | 0.73 |
| 10000 | 64 | 0.389 | 0.385 | 0.382 | 0.075 | 0.62 |
| 10000 | 128 | 0.809 | 0.251 | 0.812 | 0.120 | 0.62 |
| 100000 | 2 | 0.388 | 0.031 | 0.006 | 0.103 | 2.36 |
| 100000 | 3 | 0.428 | 0.034 | 0.007 | 0.104 | 2.52 |
| 100000 | 4 | 0.606 | 0.039 | 0.011 | 0.105 | 2.3 |
| 100000 | 5 | 0.609 | 0.039 | 0.010 | 0.105 | 2.59 |
| 100000 | 9 | 0.679 | 0.043 | 0.015 | 0.140 | 3.11 |
| 100000 | 15 | 1.209 | 0.068 | 0.025 | 0.138 | 3.11 |
| 100000 | 64 | 2.412 | 0.192 | 0.253 | 0.396 | 3.12 |
| 100000 | 128 | 4.286 | 0.188 | 0.400 | 0.677 | 4.52 |
| 1000000 | 2 | 4.405 | 0.141 | 0.015 | 0.598 | 22.51 |
| 1000000 | 3 | 3.790 | 0.129 | 0.014 | 0.620 | 22.76 |
| 1000000 | 4 | 3.511 | 0.113 | 0.018 | 0.602 | 22.13 |
| 1000000 | 5 | 4.279 | 0.137 | 0.021 | 0.609 | 24.15 |
| 1000000 | 9 | 4.949 | 0.165 | 0.031 | 0.877 | 25.27 |
| 1000000 | 15 | 7.369 | 0.259 | 0.113 | 0.912 | 30.09 |
| 1000000 | 64 | 12.239 | 0.592 | 1.296 | 3.286 | 36.02 |
| 1000000 | 128 | 20.139 | 1.040 | 2.443 | 6.115 | 44.24 |
| 10000000 | 2 | 35.947 | 0.964 | 0.086 | 5.371 | 245.8 |
| 10000000 | 3 | 40.952 | 1.157 | 0.105 | 5.586 | 260.54 |
| 10000000 | 4 | 36.062 | 0.941 | 0.123 | 5.576 | 259.23 |
| 10000000 | 5 | 35.613 | 0.958 | 0.112 | 5.665 | 255.77 |
| 10000000 | 9 | 42.647 | 1.286 | 0.185 | 8.604 | 267.63 |
| 10000000 | 15 | 68.354 | 2.249 | 1.266 | 8.673 | 291.3 |
| 10000000 | 64 | 91.734 | 4.724 | 9.998 | 31.283 | 370.19 |
| 100000000 | 2 | 330.802 | 8.884 | 0.817 | 53.184 | 2401.61 |
| 100000000 | 3 | 404.521 | 11.512 | 1.114 | 53.380 | 2587.85 |
| 100000000 | 4 | 326.012 | 8.578 | 1.081 | 55.684 | 2827.8 |
| 100000000 | 5 | 385.092 | 10.453 | 1.218 | 55.090 | 2785.14 |
| 100000000 | 9 | 430.032 | 13.174 | 1.653 | OOM | 2802.48 |
| 100000000 | 15 | 661.875 | 22.430 | 12.398 | OOM | 3146.41 |

Table 6 displays the calculated speedup of my GPU implementation compared with all other measurements; the table is ordered ascendingly by points and dimensions count. I was able to

gain considerable speed-ups over both the cuML and FAISS libraries.

***Table 6.*** *Speed up of my multi-GPU code over my Serial, OpenMP, NVIDIA's cuMl and Meta's FAISS. This showcases considerable speedup over both cuML and FAISS.*

| Points | GPU vs Serial | GPU vs OpenMP | GPU vs cuML | GPU vs FAISS |
|---|---|---|---|---|
| 100 | 0.4 | 4.9 | 14.7 | 73.4 |
| 100 | 0.3 | 3.9 | 11.6 | 78.9 |
| 100 | 0.3 | 3.6 | 9.8 | 58.1 |
| 100 | 0.3 | 3.7 | 10.6 | 50.9 |
| 100 | 0.3 | 2.7 | 8.7 | 46.5 |
| 100 | 0.3 | 2.3 | 7.1 | 38.0 |
| 100 | 0.1 | 0.9 | 3.3 | 10.5 |
| 100 | 0.1 | 0.6 | 2.7 | 8.1 |
| 10000 | 7.7 | 2.4 | 4.6 | 51.9 |
| 10000 | 6.7 | 1.8 | 3.2 | 38.8 |
| 10000 | 5.9 | 1.3 | 2.2 | 28.0 |
| 10000 | 6.0 | 1.1 | 2.5 | 35.4 |
| 10000 | 4.8 | 0.9 | 1.6 | 21.7 |
| 10000 | 4.7 | 0.8 | 0.9 | 17.7 |
| 10000 | 1.0 | 1.0 | 0.2 | 1.6 |
| 10000 | 1.0 | 0.3 | 0.1 | 0.8 |
| 100000 | 64.5 | 5.1 | 17.1 | 392.6 |
| 100000 | 64.3 | 5.1 | 15.6 | 378.5 |
| 100000 | 55.9 | 3.6 | 9.7 | 211.9 |
| 100000 | 58.7 | 3.7 | 10.1 | 249.4 |
| 100000 | 43.9 | 2.8 | 9.0 | 201.0 |
| 100000 | 47.8 | 2.7 | 5.4 | 123.0 |
| 100000 | 9.5 | 0.8 | 1.6 | 12.3 |
| 100000 | 10.7 | 0.5 | 1.7 | 11.3 |
| 1000000 | 290.9 | 9.3 | 39.5 | 1486.4 |
| 1000000 | 278.1 | 9.4 | 45.5 | 1670.0 |
| 1000000 | 192.5 | 6.2 | 33.0 | 1213.2 |
| 1000000 | 208.6 | 6.7 | 29.7 | 1177.6 |
| 1000000 | 160.3 | 5.4 | 28.4 | 818.4 |
| 1000000 | 65.0 | 2.3 | 8.0 | 265.5 |

| | | | | |
|---|---|---|---|---|
| 1000000 | 9.4 | 0.5 | 2.5 | 27.8 |
| 1000000 | 8.2 | 0.4 | 2.5 | 18.1 |
| 10000000 | 417.1 | 11.2 | 62.3 | 2852.3 |
| 10000000 | 391.7 | 11.1 | 53.4 | 2491.8 |
| 10000000 | 293.4 | 7.7 | 45.4 | 2108.8 |
| 10000000 | 318.2 | 8.6 | 50.6 | 2285.5 |
| 10000000 | 230.8 | 7.0 | 46.6 | 1448.5 |
| 10000000 | 54.0 | 1.8 | 6.8 | 230.0 |
| 10000000 | 9.2 | 0.5 | 3.1 | 37.0 |
| 100000000 | 405.0 | 10.9 | 65.1 | 2940.7 |
| 100000000 | 363.0 | 10.3 | 47.9 | 2322.2 |
| 100000000 | 301.6 | 7.9 | 51.5 | 2616.0 |
| 100000000 | 316.1 | 8.6 | 45.2 | 2286.1 |
| 100000000 | 260.2 | 8.0 | OOM | 1695.9 |
| 100000000 | 53.4 | 1.8 | OOM | 253.8 |

Figure 11, Figure 12, and Figure 13 visualize the speedup comparisons of my GPU algorithm versus other implementations. To provide a more detailed overview of various speedups, each consecutive graph removes the slowest algorithm.

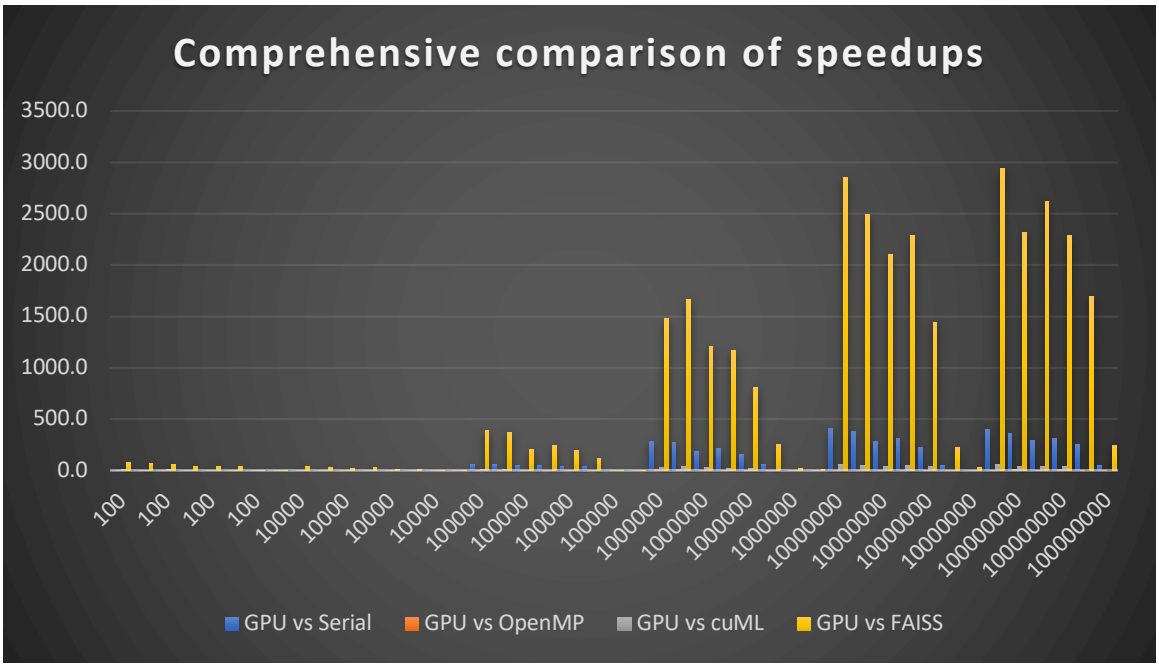



***Figure 11.*** *Speedup comparisons of Serial, OpenMP, cuML, and FAISS. A larger value equals a more significant speedup.*

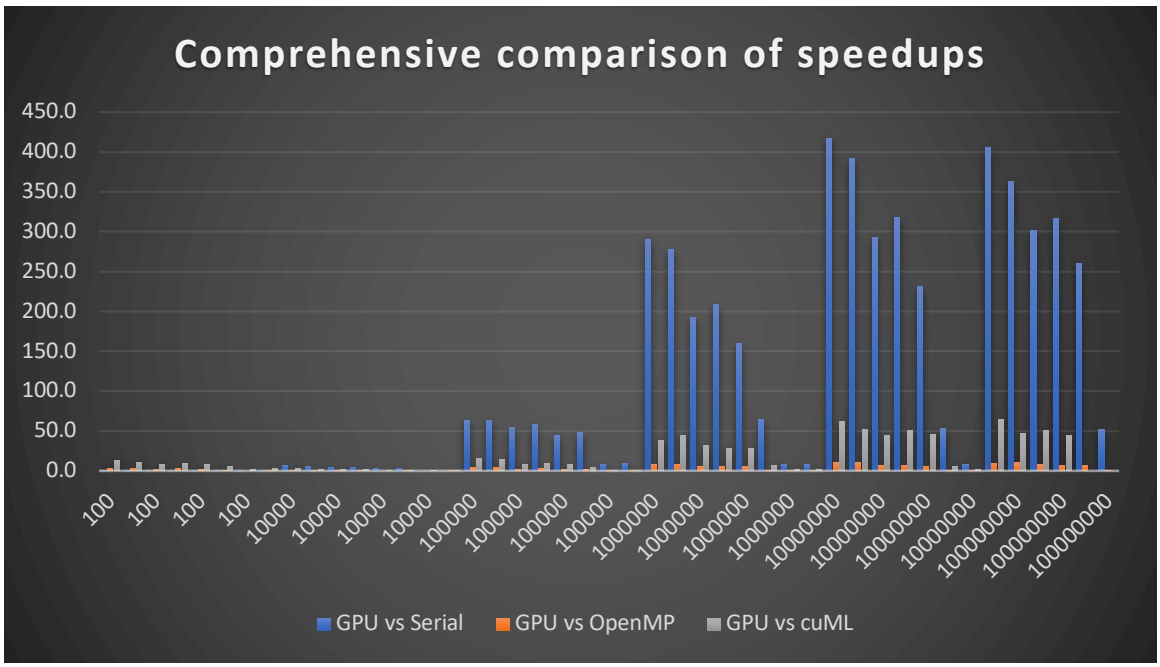



***Figure 12.*** *Speedup comparisons after removing FAISS from the graph. A larger value equals a more significant speedup.*

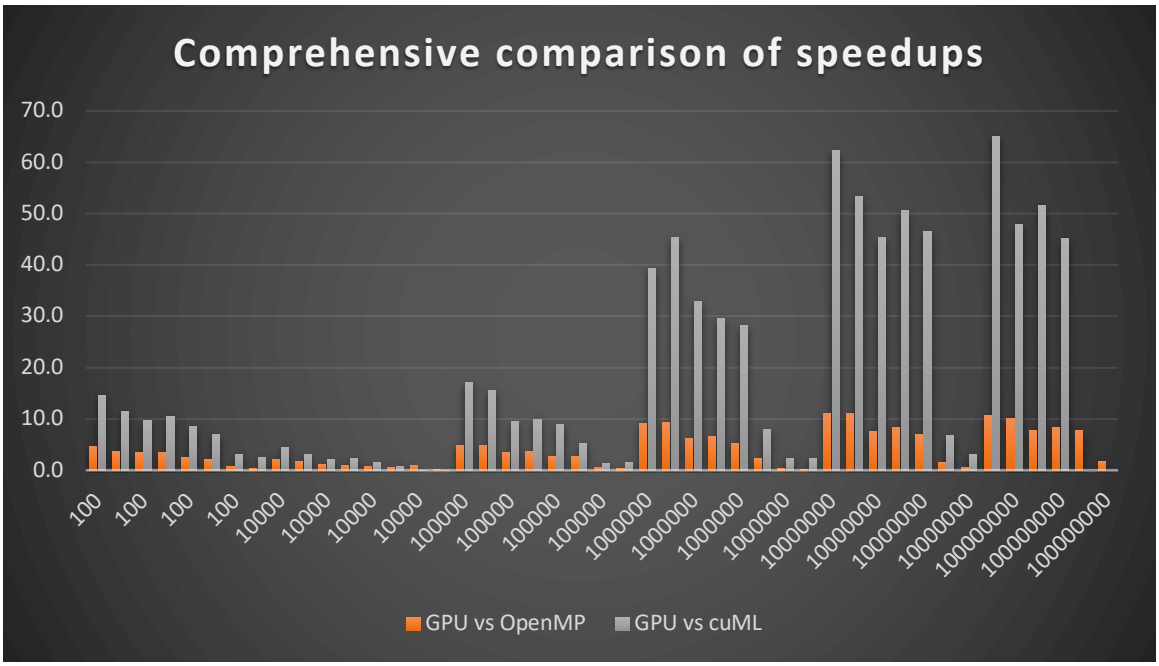



***Figure 13.*** *Speedup comparisons of OpenMP, cuML. A larger value equals a more significant speedup.*

## X. CONCLUSION

Although simple in its implementation, the *k*-means algorithm can be time-consuming. Throughout this paper, I explored previous attempts at efficient *k*-means parallelization and alternatives to *k*-means that provide unique benefits.

I additionally presented my novel methods to speed up this algorithm's performance for CPU's OpenMP and GPU's CUDA. This was achieved by minimizing the data size and amount of required communication between the device and host when necessary to increase this algorithm's efficiency. My CUDA and OpenMP variations achieved considerable speedups versus two of the current state of art implementations by Meta and NVIDIA.

Furthermore, my thesis showcased novel methods for enhancing the speed of the original algorithm; these methods are not limited to the original version of *k*-means and can be applied to other various adaptations of this algorithm. As discussed in the related work and algorithm sections, various implementations of *k*-means may use varying data structures and techniques. My proposed techniques can be applied and integrated into these methods to achieve similar improvements with only minor code modifications.

For example, *k*-median employs a different distance calculation than *k*-means, Manhattan rather than Euclidian distance, and I was able to adapt my algorithm to follow suit by removing the square root operation from one line of code. Similarly, Fuzzy C-Means assigns an assignment probability to each point. Still, since the overall structure of the algorithm remains the same, my algorithm would require basic modifications to support Fuzzy C-Means' soft clustering feature. Moreover, Mini Batch *k*-means subsamples the input data, and the implementation suggested by Alguliyev et al. relies on batching the input data; although both novel approaches, my algorithm can adapt to either algorithm by simply accepting their batched or subsampled input data as its

point input, which again should only require slight adjustments to the code. Lastly, Kanungo et al. rely on a k-d tree as the primary data structure. Although this would require my algorithm to undergo modifications to process the input data into a k-d tree, the clusterization aspects of the code would not need any major modifications.

In summary, my approach follows two simple guidelines: Minimizing data structures and frequency of communication between different devices. These principles may be applied to any *k*-means variant or implementation, and in some cases, applying my enhancements may be as simple as updating a few lines of code.

## REFERENCES


[1] T. Finley and T. Joachims, "Supervised clustering with support vector machines," in *Proceedings of the 22nd international conference on Machine learning - ICML '05*, Bonn, Germany, 2005, pp. 217–224. doi: 10.1145/1102351.1102379.

[2] P. Awasthi and R. Zadeh, "Supervised Clustering," in *Advances in Neural Information Processing Systems*, 2010, vol. 23. Accessed: May 31, 2022. [Online]. Available: https://papers.nips.cc/paper/2010/hash/18997733ec258a9fcaf239cc55d53363-Abstract.html

[3] K. P. Sinaga and M.-S. Yang, "Unsupervised K-Means Clustering Algorithm," *IEEE Access*, vol. 8, pp. 80716–80727, 2020, doi: 10.1109/ACCESS.2020.2988796.

[4] R. Garcia-Dias, C. A. Prieto, J. S. Almeida, and I. Ordovás-Pascual, "Machine learning in APOGEE: Unsupervised spectral classification with *K-means*," *Astron. Astrophys.*, vol. 612, p. A98, Apr. 2018, doi: 10.1051/0004-6361/201732134.

[5] I. Ordovás-Pascual and J. S. Almeida, "A fast version of the k-means classification algorithm for astronomical applications," *Astron. Astrophys.*, vol. 565, p. A53, May 2014, doi: 10.1051/0004-6361/201423806.

[6] S. Ferro, D. Bottigliengo, D. Gregori, A. S. C. Fabricio, M. Gion, and I. Baldi, "Phenomapping of Patients with Primary Breast Cancer Using Machine Learning-Based Unsupervised Cluster Analysis," *J. Pers. Med.*, vol. 11, no. 4, p. 272, Apr. 2021, doi: 10.3390/jpm11040272.

[7] G. Florimbi *et al.*, "Towards Real-Time Computing of Intraoperative Hyperspectral Imaging for Brain Cancer Detection Using Multi-GPU Platforms," *IEEE Access*, vol. 8, pp. 8485–8501, 2020, doi: 10.1109/ACCESS.2020.2963939.

[8] B. Ma, "K-Means in Marketing Analysis: Clustering 210 US DMAs," *Medium*, Jun. 17, 2020. https://towardsdatascience.com/k-means-in-marketing-analysis-clustering-210-us-dmas-deb9e60e3fe5 (accessed May 16, 2022).

[9] K. Ichikawa and S. Morishita, "A Simple but Powerful Heuristic Method for Accelerating k-Means Clustering of Large-Scale Data in Life Science," *IEEE/ACM Trans. Comput. Biol. Bioinform.*, vol. 11, no. 4, pp. 681–692, Aug. 2014, doi: 10.1109/TCBB.2014.2306200.

[10] S. Lloyd, "Least squares quantization in PCM," *IEEE Trans. Inf. Theory*, vol. 28, no. 2, pp. 129–137, Mar. 1982, doi: 10.1109/TIT.1982.1056489.

[11] "Visualizing K-Means Clustering." https://www.naftaliharris.com/blog/visualizing-k-means-clustering/ (accessed May 31, 2022).

[12] "Data race." https://www.smcm.iqfr.csic.es/docs/intel/ssadiag_docs/pt_reference/references/sc_omp_anti_dependence.htm (accessed Jun. 09, 2022).

[13] L. I. Smith, “A tutorial on Principal Components Analysis”.

[14] R. M. Alguliyev, R. M. Aliguliyev, and L. V. Sukhostat, “Parallel batch k-means for Big data clustering,” *Comput. Ind. Eng.*, vol. 152, p. 107023, Feb. 2021, doi: 10.1016/j.cie.2020.107023.

[15] J. K. Parker and L. O. Hall, “Accelerating Fuzzy-C Means Using an Estimated Subsample Size,” *IEEE Trans. Fuzzy Syst.*, vol. 22, no. 5, pp. 1229–1244, Oct. 2014, doi: 10.1109/TFUZZ.2013.2286993.

[16] S. Cuomo, V. De Angelis, G. Farina, L. Marcellino, and G. Toraldo, “A GPU-accelerated parallel K-means algorithm,” *Comput. Electr. Eng.*, vol. 75, pp. 262–274, May 2019, doi: 10.1016/j.compeleceng.2017.12.002.

[17] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, “An efficient k-means clustering algorithm: analysis and implementation,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 7, pp. 881–892, Jul. 2002, doi: 10.1109/TPAMI.2002.1017616.

[18] “k-d tree,” *Wikipedia*. May 22, 2022. Accessed: May 24, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=K-d_tree&oldid=1089227535

[19] “Paper: Second Place Multidimensional Binary Search Trees Used for Associative.”

[20] M. J. Brusco, E. Shireman, and D. Steinley, “A comparison of latent class, K-means, and K-median methods for clustering dichotomous data.,” *Psychol. Methods*, vol. 22, no. 3, pp. 563–580, Sep. 2017, doi: 10.1037/met0000095.

[21] “K-means and K-medians,” *Machine learning journey*, Feb. 07, 2020. https://machinelearningjourney.com/index.php/2020/02/07/k-means-k-medians/ (accessed Jun. 09, 2022).

[22] “Manhattan Distance Calculator.” https://www.omnicalculator.com/math/manhattan-distance (accessed Jun. 09, 2022).

[23] Stephanie, “Fuzzy Clustering: Definition,” *Statistics How To*, May 21, 2022. https://www.statisticshowto.com/fuzzy-clustering/ (accessed Jun. 09, 2022).

[24] E. H. Ruspini, J. C. Bezdek, and J. M. Keller, “Fuzzy Clustering: A Historical Perspective,” *IEEE Comput. Intell. Mag.*, vol. 14, no. 1, pp. 45–55, Feb. 2019, doi: 10.1109/MCI.2018.2881643.

[25] D. Sculley, “Web-scale k-means clustering,” in *Proceedings of the 19th international conference on World wide web - WWW ’10*, Raleigh, North Carolina, USA, 2010, p. 1177. doi: 10.1145/1772690.1772862.