Journal of Innovations in Computer Science and Engineering
Shahid Beheshti Unversity

May 2023, Volume 1, Issue 1

Scalable Parallel K-Means Clustering on GPU and CPU Clusters

Armin Ahmadzadeh[1,2,3,&] Code Orcid: 0000-0002-2228-3297,
Saeid Rahmani[1,3, &] Code Orcid: 0000-0002-2447-1937
Omid Hajihassani[1] Code Orcid: 0000-0001-9818-3121,
Dara Rahmati[4,*] Code Orcid: 0000-0003-0104-4016,
Saeid Gorgin[5] Code Orcid: 0000-0001-5898-4872

[1] Institute for Research in Fundamental Sciences (IPM), Tehran, Iran
a.ahmadzadeh@ipm.ir, s.rahmani@ipm.ir, o.hajihassani@ipm.ir,d_rahmati@sbu.ac.ir, gorgin@irost.ir
[2] Department of Computer Engineering, Sharif University of Technology, Tehran, Iran,
[3] Department of Computer Engineering, Sharif University of Technology, Kish-Island, Iran,
[4] Faculty of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran,
[5] Department of Electrical Engineering and Information Technology, Iranian Research Organization for Science and Technology (IROST), Tehran, Iran

Abstract. *K-means clustering is a popular method for grouping data. It has many applications in different fields. Considering the widespread application, enhancing this method in the context of high-performance computing has a significant impact. In this paper, We aim to improve the scalability of k-means clustering by using parallel computing techniques and platforms. For this purpose, we utilize the available resources at a different level of parallelism. As a result, innovative approaches are proposed for various hardware platforms, which are evaluated separately on uniformly random generated datasets and with different sizes. We modify the classic two-stage Lloyd's formulation to a three-stage that utilizes various techniques for each stage separately. Besides, we use an algebraic approach to reduce the amount of calculation and lay the foundation for consequent ideas. We propose a parallel architecture in CPUs based on OpenMP and AVX2 instruction sets. In GPUs, we utilize atomic operation and shared memory without considering GPU memory and shared memory capabilities. The proposed method extends to multi-GPU. We combine these techniques and use MPI to scale it for multiple-node platforms.*

Keywords: *K*-Means, CUDA, GPU, Multi-GPU, MPI, OpenMP, AVX2

1 Introduction

With the emergence of massively parallel platforms that push the top performance of the fastest computing nodes, a paradigm change is observed in computing. The alteration simply migrates the applications from crude sequential execution to high-performance parallel execution [1]. With this trend in mind, we strive to further optimize the scalable implementation of one of the most prominent clustering algorithms on high-performance parallel platforms. Clustering has found its application in widespread domains, including *Machine Learning* [2], *Data Mining*, *Image Segmentation* [3], *Medical Imaging* [4], and *Bioinformatics* [5]. Clustering is the operation of distributing a set of multi-dimensional points into smaller disjoined groups and clusters. All the points within a cluster share similarity and are distinct from those within other clusters.

K-means is an unsupervised clustering method employed for a set of multi-dimensional data points into several clusters. Since *k*-means is in the top most prominent clustering method, numerous heuristics are applied to its implementation, each giving close to optimal clustering solutions. A popular heuristic reducing the time complexity of *k*-means is Lloyd's algorithm [6].

With the emergence of large-scale datasets, new challenges hurdled the application of *k*-means clustering in datasets that include millions or even billions of multi-dimensional data points. Since the vast volume of large-scale datasets will render the calculations taken in each step of the clustering computationally intensive, applying the previously proposed heuristics would only help a little. Hence, by utilizing the many-core and multi-core platforms in *k*-means clustering, we can overcome the previously mentioned obstacles in large-scale datasets. Moreover, due to the absence of branches and divergence in the procedures, Lloyd's algorithm is convenient for parallel implementation on many-core and multi-core platforms.

The objective of this paper is to develop an efficient and scalable implementation of the k-means clustering algorithm on many-core and multi-core standalone and distributed machines. To achieve a significant speedup in the clustering execution time over other parallel implementations, we employ a hybrid scheme of shared memory, distributed memory, and massively parallel platforms. We also use optimization techniques such as mathematical optimization and instruction-level parallelism to lay the

Submit Date: 2023-03-18
Revise Date: 2023-09-27
Accept Date: 2023-11-14
& These two authors contributed equally to this manuscript
* Corresponding author

foundation for faster execution time in k-means clustering. The proposed solutions in this work are highly scalable. We have developed a scalable, high-performance, and cost-efficient distributed multi-GPU parallelism scheme with the aid of hybrid CUDA and MPI programming. In this regard, we break down Lloyd's heuristic into finer parallel sections to boost the parallel execution of k-means clustering. Moreover, we have proposed multiple methods to utilize the intrinsic parallelism of Lloyd's algorithm for K-means clustering. These methods are customized for various hardware platforms to achieve higher performance and speedup. The proposed methods include using an algebraic equality to reduce the computation overhead based on the dataset size, reducing the main computation of Lloyd's algorithm to the dot product instead of Euclidean distance calculation, using AVX2 instructions to compute dot products based on a SIMD scheme, implementing all stages of Lloyd's algorithm in parallel based on OpenMP, using shared memory and atomic operations in the GPU solution to reduce GPU/CPU data transfers and GPU overheads, and separating points into different GPUs in the multi-GPU solution using OpenMP threads in the CPU. Finally, we demonstrate the suitability of the proposed methods by applying them to various datasets.

The rest of the paper is organized as follows: In section 2, a full overview of Lloyd's heuristic and the programming languages used in the implementation of k-means on parallel devices are provided. Section 3 outlines the related literature, which are evaluated and compared in the evaluation section. Section 4 thoroughly designates the optimization techniques used in our *k*-means clustering implementation. Section 5 discusses the evaluation results and finally section 6 concludes the paper.

2 Preliminaries

This section explains how *k*-means clustering works and how we implement it on many-core and multi-core platforms using different programming models.

2.1 *K*-means Clustering Algorithm

K-means clustering aims to divide a set of *n* data points with *d* features into *k* groups or clusters. It tries to minimize the average squared distance between each point and the center of its assigned cluster [7]. The *k*-means clustering algorithm works as follows. First, each data point calculates its distance from all the cluster centers. Then, it joins the cluster with the closest center. Next, after all the data points are assigned to clusters, the cluster centers are updated by taking the average of all the points in each cluster. Finally, we obtain a Voronoi diagram that shows the boundaries of the clusters [8]. In machine learning, *k*-means is utilized in numerous applications, including natural language processing, pattern recognition, and image processing [9].

Since obtaining the optimal solution for *k*-means clustering is an NP-hard problem, multiple heuristics are proposed to solve it. In 1957, Stuart Lloyd suggested a greedy clustering approach by estimating optimal centroids based on *k*-mean's mean-square cost function. This algorithm, one of the conventional heuristics proposed to solve *k*-means clustering, is founded on a basic observation that the best point for the cluster to be represented is the actual center of the mentioned cluster. Accordingly, the centroid of each cluster is obtained by simply averaging the values of all the data points of that cluster.

To achieve the peak performance from the parallel implementation of Lloyd's algorithm, each iteration of Lloyd's algorithm may be partitioned into three distinct phases. First, each point calculates its distance from all the centroids and is assigned to the cluster with the shortest distance from its centroid. Second, the number of data points and the total sum of all points in each cluster is calculated and stored. Finally, in the third phase, the new centroid of each cluster is computed by averaging the obtained sum and the number of data points in the previous step.

In the rest of the paper, the phases of Lloyd's algorithm (, which mentioned above,) will be used to implement the proposed clustering procedure. As previously described, to calculate the distance between points, Lloyd's algorithm employs various metrics such as *Euclidean Distance* metric, *Manhattan Distance* metric, etc. The computation complexity of the Euclidean distance metric is the downside of Lloyd's heuristic; however, it is one of the most popular machine-learning algorithms [10]. The distance between each pair of data points and cluster centroids is calculated based on the Euclidean distance formula, as indicated in Equation 1, wherein a set of n elements $d_0\ d_1\ \dots\ d_{n-1}$ is clustered into K clusters $S_0\ S_1\ \dots\ S_{k-1}$ and μ_i is the centroid of the i_{th} cluster.

$$Total\ Distance\ = \sum_{i=0}^{K-1} \sum_{d \epsilon S_i} \|d - \mu_i\|^2 \qquad (\mathbf{1})$$

The distance is calculated using the squared Euclidean distance. Here, m indicates the dimension size of the data points in the cluster.

$$\|d - \mu_i\|^2 = \sum_{j=1}^{m} \left(d_j - \mu_{ij}\right)^2 \qquad (\mathbf{2})$$

The optimal aim of the clustering is to minimize the total distance calculated by Equation 1 and to maximize the inter-cluster distance among the final clusters.

2.2 Programming Model

The programming model is a definition of programming criteria and executing procedures based on features and architecture of the low-level resources. Shared memory architecture offers hardware suitable for scalable cache access coherency. In this architecture, every processor has access to the data of other processors. OpenMP is a standard multithreading API developed to exploit the potential of the shared memory architecture in parallel system programming. OpenMP API presents routines and

directives to express portable and scalable parallel programming models [11]. MPI is a standardized message-passing programming model. There are many standard developments for the MPI model, such as OpenMPI [12] and MPICH [13], which are open-source projects supporting numerous operating systems and HPC platforms. The parallel section of each algorithm is coded using messages. Messages are handled by send and receive functions distributed between compute nodes, exploiting their distributed memories. At last, all the computation results from all the nodes are summed up together. Other than the parallel regions of the codes, other codes are executed serially.

Due to the primary purpose of the graphical processing units (cards), their architecture is designed to make them capable of performing a vast number of calculations on large-scale input data. Each graphic processing unit has multiple computational cores, each with above-average processing power. The threads spawned in GPUs are actually lightweight, making the cost of their spawning and scheduling much less than those of threads spawned in CPUs. The GPUs are designed in a way that they are suitable and capable of solving intensive computational problems by exploiting the data parallelism nature of the problems. Implementing problems meeting the aforementioned criteria on the GPU hardware allows the programmer to achieve a noticeable speedup over the previous implementation on the CPU.

In GPU architectures, the hierarchy of memory types is facilitated. Shared memory is a type of volatile, fast memory that is accessible by all of the threads in the same block. In this structure, each thread has its own set of registers, which no other thread can access their contents. In addition, one type of memory, namely global memory with big memory space and higher access time compared to the shared memory, is built into the GPUs, which can be read and written by all the blocks executed in the GPU Grid [14]. The CUDA programming model adds minimal extension to the C programming language. In this model, multiple built-in compiler directives are available for accessing and managing the GPU's internal functionalities. In CUDA programming language, GPU is accessible by name "DEVICE," and CPU is named "HOST".

HOST can read (write) from (to) GPU's global memory. The CUDA API makes these accesses possible. Global and constant memories are the only memories whose content can be accessed and manipulated by the HOST. The HOST, or in the synonym term the CPU, executes a function referred to as the "Kernel" as a program on the GPU with a set of initial data. Then, it collects the computation results from the DEVICE's memory. The kernel code should fully exploit the data parallelism nature of the computational task to benefit fruitfully from the massively parallel architecture of the GPU [30]. Threads are executed in a SIMT architecture, meaning that all the threads in a *Streaming Multiprocessor* execute the same instruction.

3 Related works

Due to the mentioned difficulties, *k*-means clustering algorithm is implemented using numerous heuristics. In 2002, Hamerly [15] scrutinized other alternatives to *k*-means clustering and determined that each aspect of these clustering methods contributes to better final cluster quality. In 2003, Elkan [16] suggested the usage of triangular inequality to accelerate the *k*-means clustering by reducing the problem search space complexity, which means that each data point examines less number of centroids. In 2012, Drake and Hamerly [17] proposed another clustering algorithm for accelerating *k*-means clustering by choosing adaptive distance boundaries for pruning unnecessary calculations.

The heap algorithm [17] suggests the heap data structure for storing the values calculated for bounds. This way, the search span for keys whose bounds are violated is significantly reduced. By introducing the 'neighbor cluster' term, the annulus algorithm reduces the search for the centroids to only the neighbors [18]. Compare-means and sort-means, which are modifications of *k*-means discussed in [19], enhance the execution time of each *k*-means iteration by reducing the number of distance calculations performed per each data point in the dataset and without altering the original clustering result. In *k*-means++ [20], a careful centroid initialization for the clustering is proposed, which boosts the convergence speed of the algorithm. We took advantage of the mentioned technique in initializing our centroids implementations.

Besides the proposed heuristics that reduce the problem's computation, a large part of the literature discusses parallelism opportunities for implementing the fast-clustering methods on many-core and multi-core platforms. Moreover, various papers regarding *k*-means implementation on diverse platforms were proposed. In [21], the parallelization solution of *k*-means for image processing with CUDA, OpenMP, and MPI is given. The CUDA performed the best on larger images and OpenMP on smaller ones and proposed the idea for parallel initialization of the centroids in the dataset, which proved to be fruitful.

In [22], researchers suggested the usage of dynamic load balancing in the GPU-based clusters to balance the distribution of the workload on different GPUs to improve the performance of the *k*-means clustering algorithm on GPUs inside a node.

In [23], the linearly scalable parallel implementation of *k*-means on multi-core distributed memory platforms with MPI is proposed. In [3], the performance of *k*-means was investigated, and it is shown that roughly all of the *k*-means costly parts can fully benefit from the massively parallel nature of the modern GPU architecture based on CUDA. The parallelization of other clustering methods on massively parallel platforms, such as the parallel implementation of the fuzzy c-means, is suggested for medical image processing in [24], and Parallel fuzzy c-means clustering for large datasets is proposed in [25].

Reference [31] tries to provide a detailed analysis of the trade-offs between CPU and GPU architectures. While the authors claim that their implementation offers equivalent performance on both architectures, they do not discuss the factors that may influence the choice of one architecture over the other. The parameters may be cost, power consumption, and scalability. Besides, the work on [32] discusses the potential limitations of the proposed algorithm, such as the impact of the initialization method on

the clustering results or the algorithm's sensitivity to the choice of hyperparameters. These limitations can affect the generality and robustness of the algorithm in real-world applications. Reference [31] evaluate the performance of their algorithm on several datasets for spectral clustering, including the MNIST dataset. On the other hand, Reference [33] evaluate their algorithm on a single dataset for image segmentation, the Berkeley Segmentation Dataset. Reference [31] uses CPU and GPU architectures, while [33] focus solely on GPU-based parallelization. This difference in architecture selection may impact the scalability and cost-effectiveness of the algorithms in different contexts.

Reference [34] provides a detailed explanation and performance evaluation of the *k*-means algorithm on GPUs and this work is useful for researchers and practitioners who want to implement *k*-means on GPUs. However, it only compares its algorithm with one CPU-based version of *k*-means. The authors compare the *k*-means implementations on a few datasets. They use six datasets to test the algorithms. This may not show how the algorithms perform on other datasets that have different features. In contrast [35] only focuses on parallel algorithm implementations and does not test sequential implementation performance. Parallel implementations are important for big datasets, but sequential ones may be better for small datasets and detailed analysis of CPU and GPU performance. Although the paper [35] compares the algorithms on both architectures, but it does not explain what factors may affect the choice between them, such as cost and scalability. Also, paper [35] does not analyze how hyperparameter selection affects the performance of the algorithms. It only mentions some hyperparameters, like the number of clusters, but it does not explore how altering the values affect the performance of the algorithms. Authors in paper [35] not promise to evaluate the scalability of the methods fully, and it only shows a limited evaluation of the hybrid MPI, CUDA, and OpenMP implementation on HPC clusters.

4 Proposed Method

In this paper, we aim to boost the computation and overcome the parallelism challenges and the data dependency problems which appear in implementing the *k*-means clustering algorithm with Lloyd's heuristic on Many-core and Multi-core standalone and distributed HPC machines. The detailed outline of the implementation of Lloyd's algorithm and the proposed optimization techniques applied to these implementations are given in this section. First, the serial code implementation of the *k*-means clustering is provided. Then, the optimizations employed in the serial code will be discussed. Later, the shared memory, CUDA, and distributed memory implementations are described in detail. Besides, the code for our implementation is available on GitHub at https://github.com/saeidrhm/kmeanshpc.

4.1 Sequential Implementation

Here, the *k*-means clustering algorithm is implemented and executed sequentially on a single CPU core. Firstly, the first phase of Lloyd's algorithm, the assignation process, is undertaken for each instance in the dataset. Secondly, the member count and total sum of all the clusters in the dataset are computed. Lastly, the new set of centroids for all the clusters is calculated by averaging all the points assigned to each cluster. At the end of each clustering iteration, the new set of centroids is passed to the next iteration of the clustering operation. Here, the code runs quite slowly on even small datasets, let alone those with millions or even billions of multi-dimensional points. To speedup the performance of the sequential implementation, several optimization techniques have been applied to the code.

Lloyd's algorithm has a computationally intensive phase of assigning data points to clusters. This phase uses the Euclidean distance metric to calculate the distance between each data point and each cluster center. We propose a novel mathematical optimization [26] that simplifies the Euclidean distance formula and avoids unnecessary and repeated calculations that slow down the clustering in this phase.

$$\sum_{i=0}^{m-1}(d_i - \mu_i)^2 = \sum_{i=0}^{m-1} {d_i}^2 + \sum_{i=0}^{m-1} \mu_i^2 - 2\sum_{i=0}^{m-1}(d_i\mu_i) \qquad (\mathbf{3})$$

Equation 3 shows the expansion of the Euclidean distance metric. In this equation, m is the number of features, d is a data point, and μ is a cluster center. We can simplify this equation by doing the following. First, we only calculate the sum of the squared data points once and ignore it in the rest of the calculations because it is constant for all the iterations. Second, we calculate and store the sum of the squared cluster centers for each iteration. Third, we only calculate the last part of the equation for each pair of data points and cluster centers in each iteration. This is our main computation. This heuristic reduces the number of summations and multiplications for each data point. We only need to do d-1 summations, one subtraction, and d multiplications for each data point. This is compared to the main algorithm with $2d$-1 summation (d summation for sum elements of μ_i with D_j and d-1 summation for dot product) and d multiplication for each data point. We also need to compute the dot product of the cluster centers in each iteration, but this is usually very fast. Algorithm 1 explains the mathematical optimization in detail.

Algorithm 1. Description of clustering iterations with the applied mathematical optimization

Algorithm. 1. *Inputs*: Centroids, Data points *Output*: Clusters

```
While //Clustering iteration
ComputeCentroidSquered(Centroids); //Calculate Part 2 of the expansion
Foreach Data point
Foreach Centroid
//Calculate Part 3 of the expansion
ComputeDotProduct(Data point x, Centroid μ);
AssignToNearestCluster( );
End
End
End
```

4.2 Instruction Level Parallelism

The Advanced Vector Extension instruction set from Intel empowers us to perform SIMD operations on CPUs shipped by Intel. This allows calculating multiple data operations at a single instance of execution time. We used the AVX2 dot product instruction (_mm256_dp_ps) to calculate the third part of the expanded Euclidean distance formula discussed earlier [27]. Therefore, if one plans to utilize the AVX2 instruction for dot product operation, one should, in advance, typecast the operands to floating point operands. This optimization technique is only possible with the aforementioned mathematical optimization approach.

4.3 Shared Memory Implementation

Based on the details above, with the emergence of high-performance parallel platforms, we can further benefit from migrating our execution onto these HPC platforms. Hence, we propose the shared memory implementation of the previously outlined sequential execution code. We took advantage of Open Multi-Processing (OpenMP) in the parallel CPU implementation of the *k*-means clustering algorithm.

In the parallel CPU implementation, the first phase of Lloyd's algorithm is handled in a parallel manner by the maximum number of threads that the system's CPU can accommodate. Then, to parallel execute the calculation of each cluster's total sum and total member count, we allocated distinct memory spaces to each thread. Hence, each thread calculates a partial sum and member count for all clusters. Then, the results from these calculations are reduced further into a total global sum and member count. At last, the new set of centroids is calculated to be fed into the next iteration of the clustering algorithm. The detailed outline of the mentioned parallel CPU implementation is given in Figure 1.

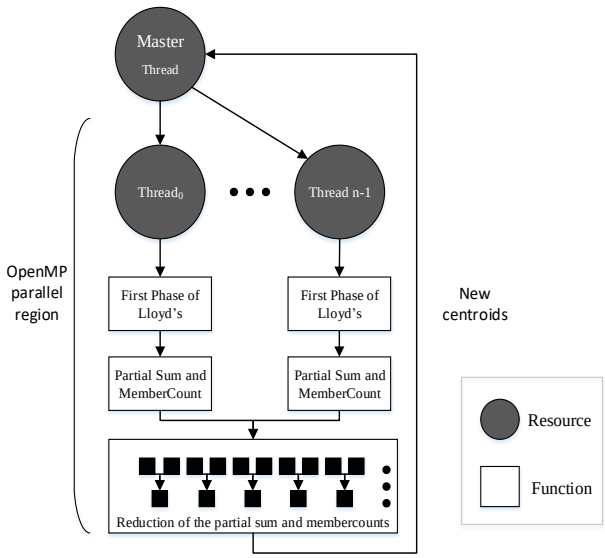

Fig 1. Multi-core CPU solution with OpenMP on shared memory platforms.

4.3 Single-GPU Implementation

In this implementation, we took advantage of the superior performance potential of the graphical processing units manufactured by NVIDIA. The extensions added to *C* in CUDA programming model and language are used to migrate the execution of *k*-means clustering to the GPU. We propose two different solutions for many-core standalone machines, Single-GPU and multi-GPU. Both of these two solutions are fully introduced in the rest of this subsection.

This implementation fully incorporates the first two phases of Lloyd's algorithm into the GPU. First, the HOST copies the data of centroids and the squared sum of centroids to the GPU's global memory. Then, a kernel with an organization of threads equal to the number of data points is launched. The organization of blocks and grids of the kernel launch depends on the number of points in the dataset. Then, each thread, representing one instance of the dataset, calculates its Euclidean distance from all of the clusters' centroids in the dataset. Later, the thread is assigned to the cluster whose distance is the smallest from its centroid. At last, each cluster's total sum and member count are computed with the atomic add operation available in CUDA. Finally, the next kernel is launched with the new set of calculated centroids for the next clustering iteration.

Here, the mentioned mathematical optimization applied to the serial execution of *k*-means is also employed in this implementation; To further optimize the execution of clustering on the GPU, we utilized the shared memory in our memory access patterns with faster access compared to the global memory. Here, the centroids are transferred to the shared memory for faster access. However, the limited size of the shared memory faces problems in cases where the centroids (dimension multiplied by a number of clusters) are bigger than the shared memory size. To resolve this issue, we need to partition the centroids into segments with a size equal to the shared memory capacity and load each of the segments individually into the shared memory.

In each of the rounds of filling the shared memory with different centroid segments from the global memory, each thread chooses the centroid that has the minimum distance from it and compares it to the centroid that was closest to it from the previously shared memory load and updated its globally closest centroid. At the end of all the shared memory loads, each thread is assigned to the cluster represented by its obtained globally closest centroid.

In summation, the first and the second phases of Lloyd's heuristic are computed with the GPU. Leaving the execution of the third phase of the clustering to the CPU. The HOST collects the results of the total sum and member count from the DEVICE's global memory and calculates the new centroids by averaging the results. Figure 2 explains this implementation of the proposed Single-GPU implementation on a standalone GPU-based machine.

By using, the atomic add operations, the first and the second phases of Lloyd's heuristic can be joined in the GPU. To do so, each thread, after finding its nearest centroid (global closest centroid), increments the member count of its centroid by using the atomic add (*atomicAdd*) and then adds itself to the cluster sum of the cluster represented by its nearest centroid by using the atomic add operation [28].

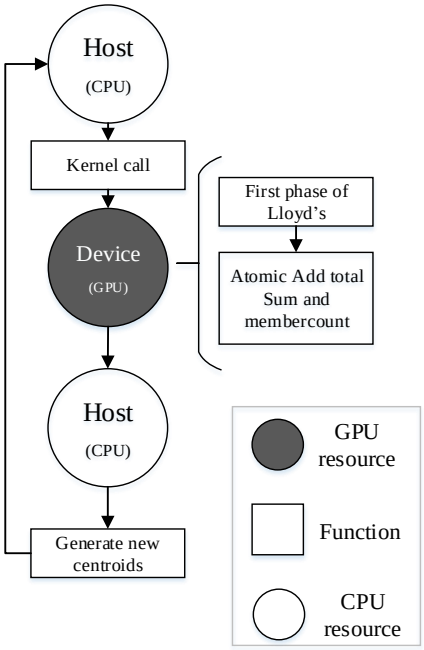

Fig 2. Single-GPU implementation with CUDA on a standalone GPU-based machine.

4.4 Multi-GPU Implementation

Because it is feasible to utilize more GPUs in a single machine, we can further scale up the implementation of the CUDA to more GPUs in a single standalone machine. Firstly, we have to distribute the dataset amongst all of the available GPUs. In the partitioning of the dataset, if the GPUs are homogenous in the sense of computational power, the dataset is equally broken down into segments of the same size. However, if the GPUs are not computationally equal, the dataset is divided among them correlative to their potential. It is important to balance the load partitioning among the GPUs with regard to their performance potential. Here, one OpenMP thread is responsible for invoking the kernel on each of the GPUs. Hence, OpenMP threads in parallel manage GPUs. The rest of the clustering operations are handled in the same way as discussed in the Single-GPU implementation section. After the GPUs have finished their computations, the HOST reduces the results of the partial sum and member count calculated by each GPU into a single total sum and member count for all of the clusters in the dataset. Then, the

newly generated set of centroids are passed to the next kernel, calling for the next iteration of clustering to take place. Figure 3 explains the implementation of the proposed Multi-GPU implementation on a standalone GPU-based machine.

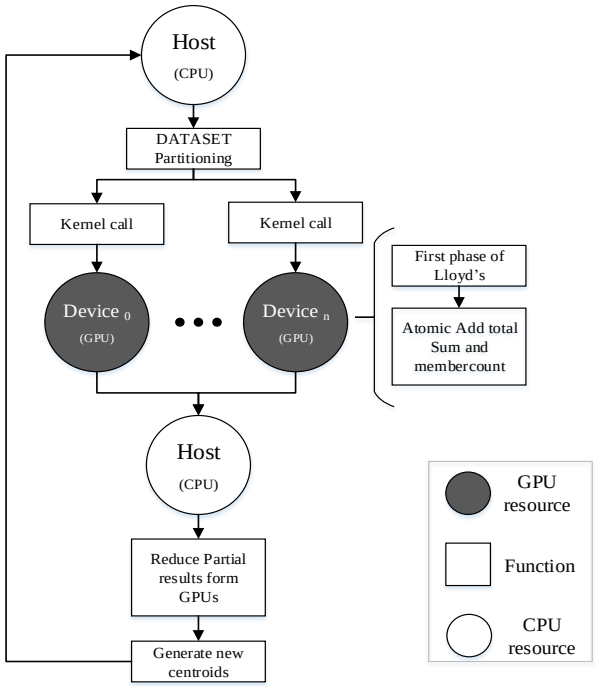

Fig 3. Multi-GPU implementation with CUDA on a standalone GPU-based machine.

4.5 Distributed Memory

In this implementation, a mixture of OpenMP and MPI is utilized in order to make the scaling of *k*-means to CPU-based clusters feasible. The OpenMP code that is employed here to parallel execute the *k*-means on the CPU shares the same setting with the implementation discussed in the Shared memory implementation section, except that now the final partial results from each machine are reduced by a single machine to obtain the total cluster sum and member count. Here, for each machine in the cluster, an MPI thread is spawned to manage and control the clustering procedures. First, the dataset is distributed amongst all of the machines existing in the cluster. Then, each machine employs the OpenMP implementation to cluster the share of the assigned dataset. We can further optimize the load balancing between the nodes so that each node gets a dataset size in correlation to its computational power. Figure 4 illustrates the scheme of the *k*-means clustering algorithm implemented on a CPU-based cluster.

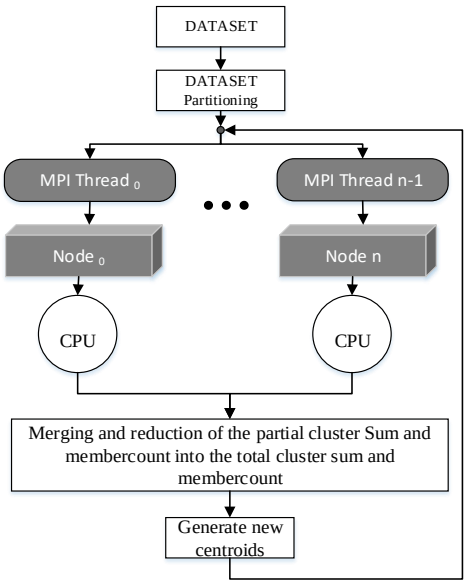

Fig 4. CPU-based cluster Implementation of *k*-means with OpenMP and MPI on parallel HPC platforms.

4.6 GPU-based Clusters

Here, we employed Message Passing Interface (MPI) to communicate information and data between our GPU-based machines. CUDA is used to parallel implement the *k*-means clustering on the Graphical Processing Units employed in our machines. The

CUDA implementation is utilized to implement the parallel execution of the *k*-means clustering on each machine, which is the same as what has been discussed in the CUDA implementation section. Identical to the CPU-based clusters, an MPI thread is spawned per each machine in our cluster. Firstly, the Dataset meant to be clustered is distributed among all of the nodes in our cluster. Then, if there is more than one GPU within a single node, the dataset is further broken amongst these nodes. Afterward, the most optimized GPU solution handles the clustering of the assigned dataset in each GPU. Finally, the results of the clustering from all of the nodes are reduced to form the final clustering operation of the whole dataset. Figure 5 outlines the GPU-based cluster implementation of the *k*-means clustering algorithm.

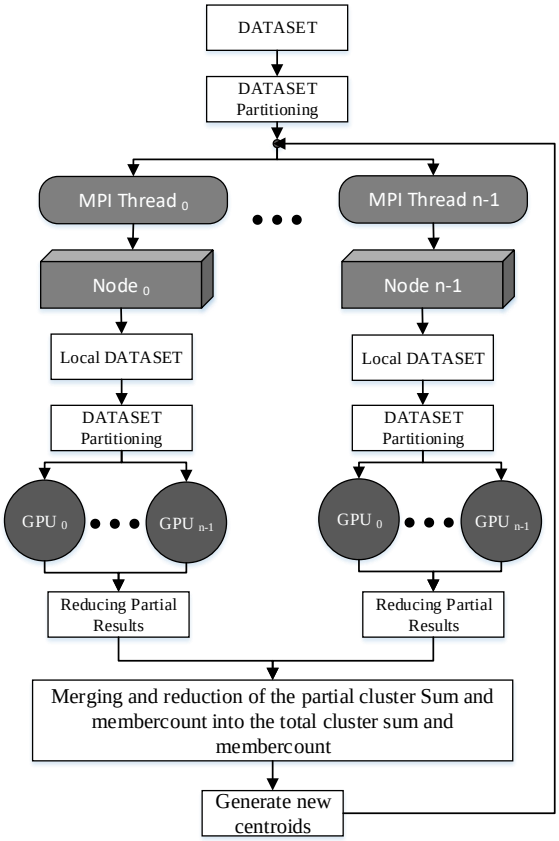

Fig 5. GPU-based cluster implementation of *k*-means with CUDA and MPI on parallel HPC platforms.

4.7 Load Balancing

Load balancing is based on partitioning the data point among our computational resources. Based on the aforementioned details, it is possible to scale the implementation of the *k*-means clustering to a greater number of machines within a cluster. However, the critical point is the way we distribute the dataset among the machines in our cluster. In the case of a cluster with homogeneous nodes, it is as simple as giving each machine the same-sized share. However, if the computational power of the machines in the cluster differs, the assigned load in the code plays a significant role in the execution time. One way to tackle this hurdle is to distribute the dataset among the machines based on their processing units' specifications. This gives us a rough speculation of how to divide the dataset among the nodes.

The most important specifications are the number of cores, threads, available memory size, and processing units within the same machine. However, the execution speed of the clustering does not linearly scale up with the number of cores or threads in a CPU or GPU device. Another way to find a clue on how to partition the dataset is to run a sample identical naïve *k*-means clustering testbench on all of the machines. This way, we can actually see the performance difference between the machines. Then, we can divide the dataset among the machines appropriately based on the evaluation results of the *k*-means clustering testbench. The size and dimension of the testbench clustering problem should be enough for the test results to be reliable.

5 Evaluation Results

In this section, the results from the evaluation of our proposed solutions are thoroughly discussed. First, the specification of the dataset employed in the evaluation of the solutions is given. Afterward, the thoroughly detailed description of the setup used to implement the proposed solutions is highlighted. This is followed by the results of the evaluation of proposed methods on the aforesaid setup. Finally, the stand-alone multi-core implementation of the previous works discussed in the related work section is indicated.

5.1 Datasets

Here, the indication of the datasets used for evaluation purposes is outlined in detail. We employed uniformly random generated datasets. Uniform datasets have three different types, having different numbers of 2^{15}, 2^{20}, and 2^{25} data points. The specifications of these different uniformly random generated datasets are shown in Table 1. Four various settings of the uniformly random generated datasets are evaluated in our results. Each of these settings has different dimensions for their data points, with 4, 8, 16, and 32 dimensions.

Table 1. Specification of the uniform datasets.

Dataset	# of data points
U1	2^{15}
U2	2^{20}
U3	2^{25}

5.2 Setup

The CPU employed to evaluate the proposed methods on multi-core platforms is Intel XEON E5 2697 V3 clocked at 2.6 GHz. Also, the GPU employed is GTX 980 Ti delivering a peak performance of 5.6 TFLOPS. Each of our computational nodes has 2X CPUs and 2X GPUs of the mentioned setting with 128GB DDR3 RAM. The connecting network among our computational nodes comprises a 1 Gb/s TCP/IP network. For shared memory programming, we used OpenMP version 3.1, released in 2011. We used CUDA version 7.5 in the programming of the GPUs. Our implementations took advantage of the MPI version 2. Following is the GTX 980 Ti graphic processing card specification used in implementing our proposed methods.

Table 2. Specification of the GTX 980 TI GPU.

CUDA cores	2048
Base clock	1.1 GHZ
Boost clock	1.2 GHZ
Memory clock	7 GHZ
Standard memory config.	6 GB

we used the network bandwidth between two nodes in our implementation, as shown in Table 3. This is the result of running MPI benchmark in a realistic situation.

Table 3. The bandwidth of inter-node connection.

Best (MB/sec)	Average (MB/sec)	Worst (MB/sec)
114.83	113.74	112.14

5.2 Single Thread CPU Solution

The evaluation results of implementing proposed solutions in clustering random datasets are specified here. Various clustering settings and different cluster counts have been applied to group the mentioned datasets. Finally, the performance gains of other proposed heuristics on multi-core CPUs, which were mentioned in the related works, are highlighted and evaluated. This is done against our multi-core implementation of the clustering on CPU. All the outlined reports in the following sections are the verified results of running the corresponding applications three times. The computation order of *k*-means clustering using Lloyd's heuristic is O*(nkdi)* that *n* is number of dataset points, *k* is number of clusters, *d* is the dimension of the dataset, and the centroid element *i* is the number of iterations until convergence or achieving the end condition threshold. In all the implementations, number of iterations is constantly 15.

This section encompasses three different evaluations. A single-threaded CPU solution that is implemented without any further optimizations is given in Figure 6(A). Datasets and centers used in CPU implementations are single floating-point variables. Several optimizations, such as the mathematical optimization (see Figure 6(B) for isolated mathematical optimization speed up) and instruction level parallelism (AVX2), are applied to the aforementioned CPU non-optimized solution. Here, a single CPU thread handles the clustering sequentially. The results from clustering the U1, U2, and U3 datasets, with three diverse cluster counts, 4, 256, and 1024 by the fully optimized (Mathematical optimization and AVX2) solution are discussed in Figure 6(C).

5.3 Optimized OpenMP Solution

The multi-threaded solution for the single-threaded non-optimized CPU implementation is given in Figure 7(A). In this multi-threaded implementation, the labeling, the first phase of Lloyd's heuristic, is handled by multiple OpenMP threads. Here, we also can merge the first and the second phases of Lloyd's heuristic by using local thread spaces that accommodate partial member count and cluster sum results. Then before the third phase, the master thread reduces these partial results into global member count and cluster-sum for all of the clusters. The results of the clustering of all three random datasets are given in Figure 7(B).

The CPU implementation here is fully optimized by applying all the aforementioned optimization techniques. Lloyd's heuristic's first and second phases are handled in parallel execution. However, the newly generated centroids are calculated by a single master thread that reduces the results of multiple parallelization cores into new centroids. The overhead of the sequential

execution of this phase can be neglected due to its trivial execution time. To take advantage of the AVX2 dot product instructions set, one needs to cast integers to floats prior to the dot product function.

5.3 Single GPU Solution

The base single GPU implementation migrates the labeling phase of Lloyd's heuristic, the distance computation, and the assignation of data points to the clusters to the GPU. The rest of Lloyd's heuristic computations are handled in the CPU. Dataset and Centers before starting timing transfer to GPU and in the duration of execution enter transfer between GPU and CPU, for reducing the effect of transferring dataset and centers using 16-bit short integer variables. The clustering results of all three uniformly random generated datasets with the base algorithm are given in Figure 8(A).

Now, to further optimize the base GPU solution's labeling phase, we applied the aforementioned mathematical optimization to the base GPU solution. The clustering result of the mathematically optimized single GPU for the random datasets on different configurations is given in Figure 8(B). In some cases, because of computing the square value of cluster centers by single thread CPU and transferring it to GPU, we encountered a speed-down ratio. Still, by increasing the size of the dataset, the problem was not considerable. In the scenario that the size of the centers compared to the size of the dataset is significant, it is suitable to calculate the square of the centers in different kernels in GPU.

To reduce the access time of each thread to the centroids stored in the global memory, we used shared memory to store the centroid values. To handle data sets that have centroids with sizes bigger than the shared memory, the centroids are managed in the same manner as explained in the single-GPU implementation section. To fully indicate the ability to handle datasets with centroids sizes bigger than the shared memory capacity, we used the base non-optimized GPU solution with datasets with 256, 4096, and 16284 centroids. The evaluation result of this implantation is given in Table 4. We clustered the same datasets with the shared memory-optimized GPU solution to highlight the shared memory optimization speedup on the base GPU solution. These results are given in Figure 9(B). The evaluation of the base GPU solution with the atomic add operation is shown in Figure 8(C) for different random datasets. By utilizing all the mentioned heuristics, we achieved the best implementation in a single GPU. The speedup ratio of this implementation compared to the conventional single GPU is shown in Figure 9(C). Besides, our evaluation results demonstrate a significant speedup (13X) compared to reference [33] for a large dataset point and 256 clusters with 32 dimensions using the same GPU.

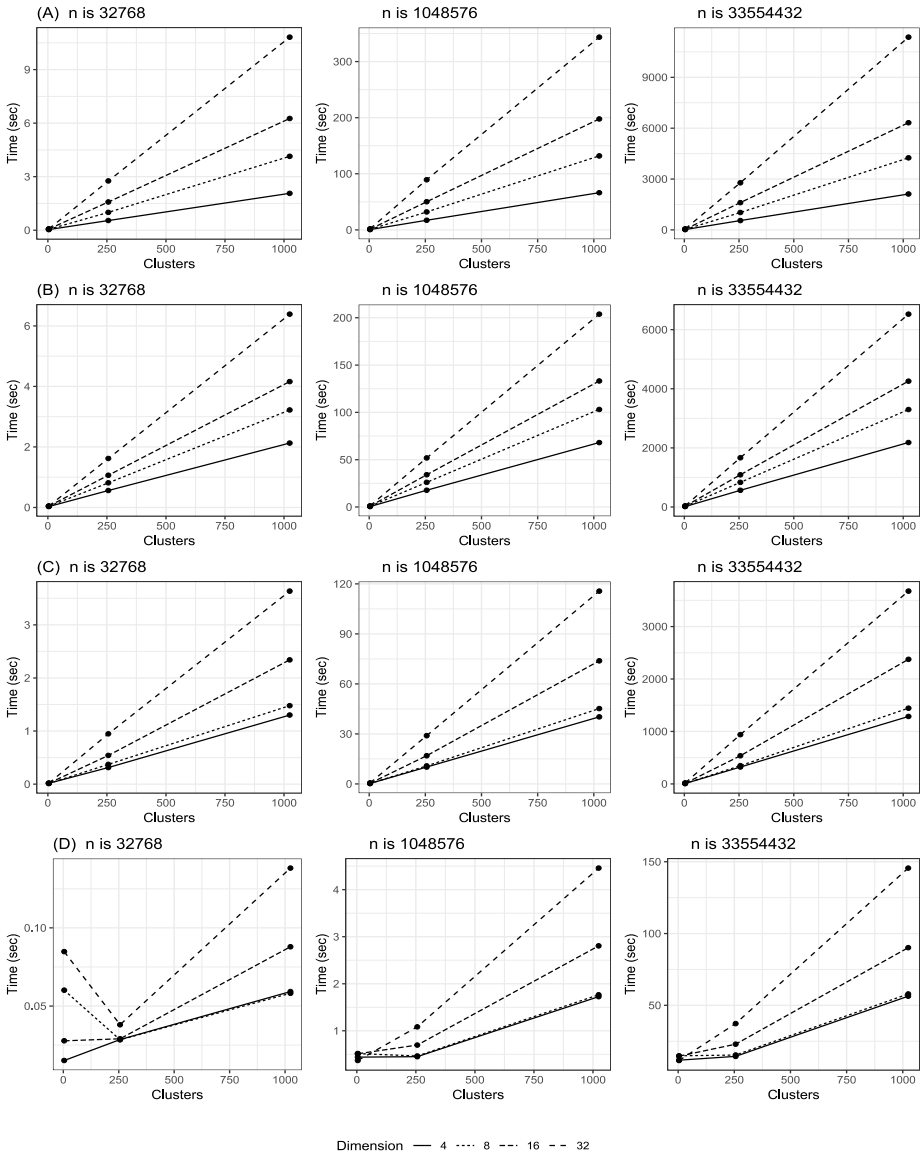

Fig 6. (A) The base Lloyd's algorithm on single thread CPU. (B) Mathematical optimization on single thread CPU. (C) Mathematical optimization implemented by AVX2 on single thread CPU. (D) Merged optimization on multi-threaded CPU implementation.

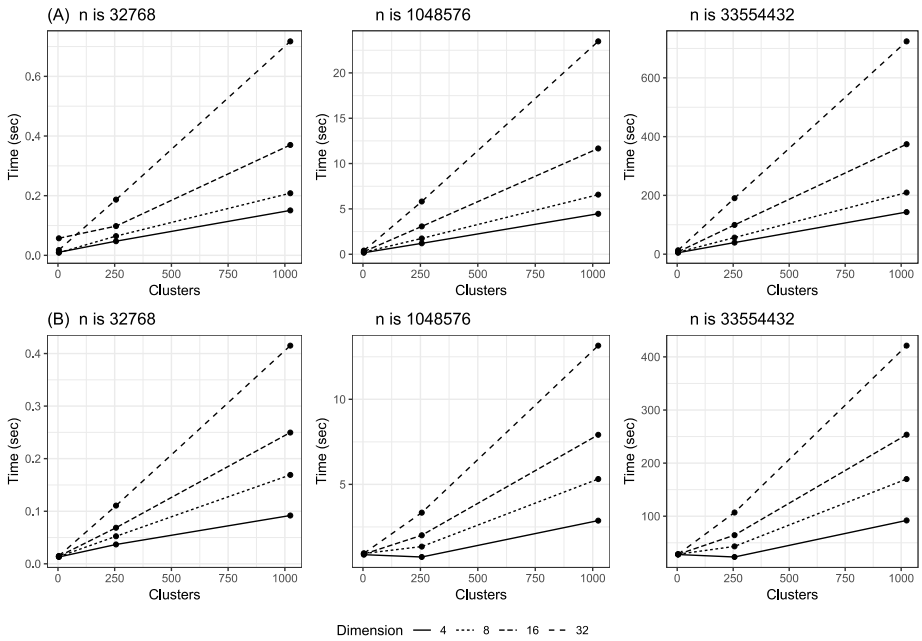

Fig 7. (A) The base Lloyd's algorithm on multi-threaded CPU using OpenMP.

(B) The base Lloyd's algorithm on multi-threaded CPU with local space using OpenMP.

5.4 Multi-GPU Solution

The Optimized Multi-GPU solution of the *k*-means clustering algorithm on the U1, U2, and U3 datasets is studied here. All the various phases of Lloyd's heuristic are handled in the GPU. We employed shared memory in order to fully maximize the GPUs throughput. The clustering is performed by a single, standalone CUDA capable GPU. Each iteration of the clustering is a kernel call from the host. The results of the cluster sum and member count of the clusters from each iteration of clustering are averaged in order to obtain the new generation of centroids. The evaluation results are depicted in Figure 9(D).

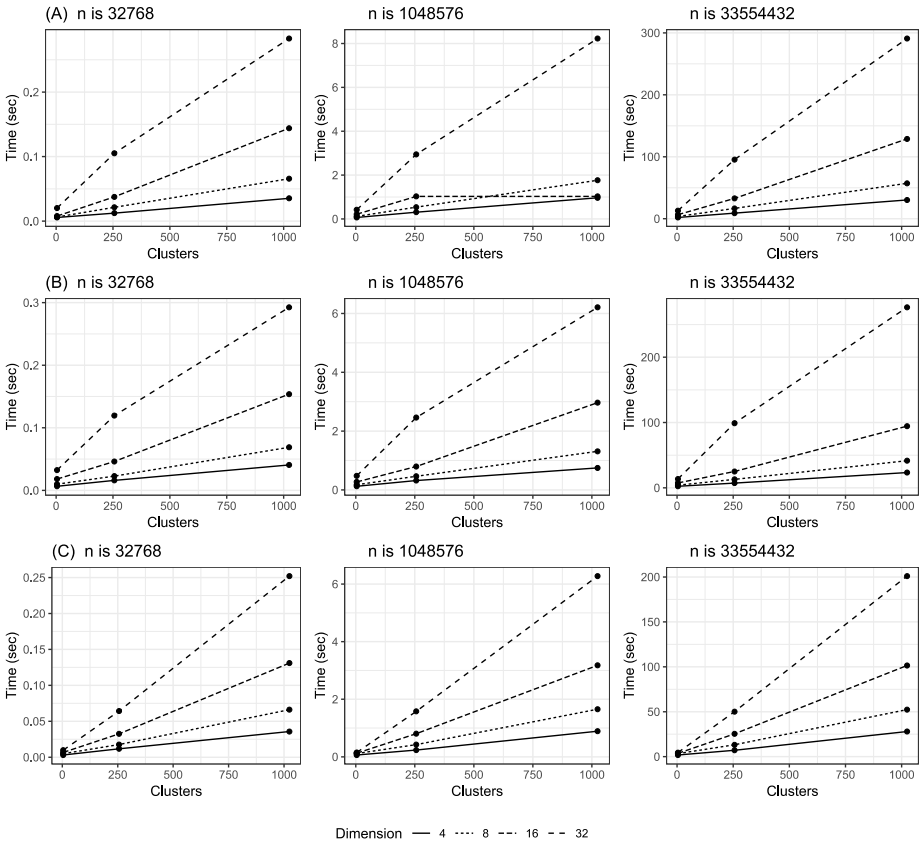

Fig 8. (A) First stage of the base Lloyd's algorithm on single GPU. (B) Mathematical optimization on single GPU. (C) Transfer second step of Lloyd's algorithm on GPU by

5.5 MPI OpenMP Solution

In this solution, the parallel clustering of Lloyd's algorithm on a CPU-based cluster is proposed. MPI is used in order to manage and distribute clustering execution on different nodes. Here, we have two nodes, each having a pair of Intel XEON processor as mentioned in the Setup subsection. In each of the machines, the most optimized OpenMP solution is applied to fully benefit from the multi-core parallelization capability of each CPU. The results of clustering of the uniformly random generated datasets are given in Figure 10.

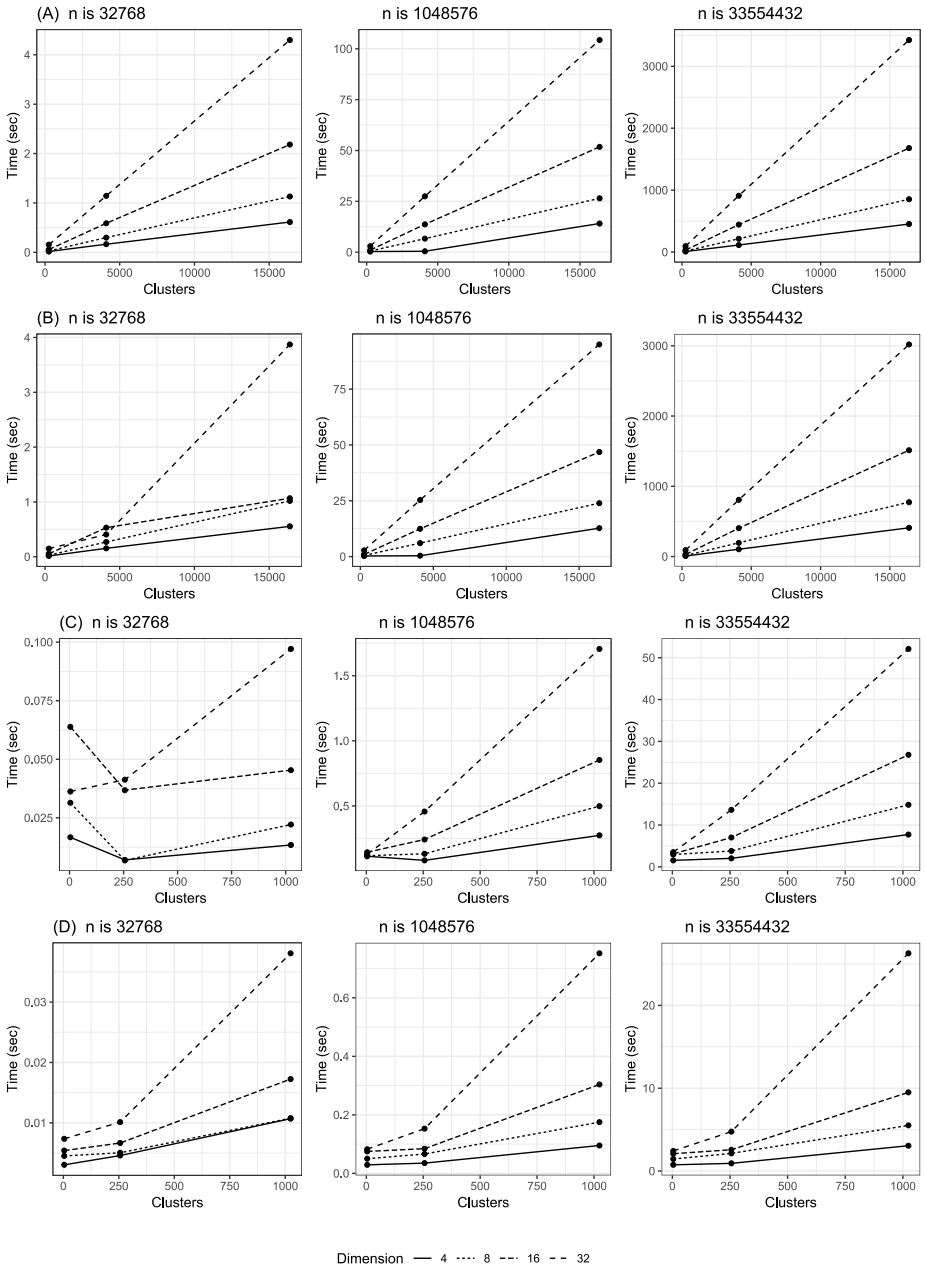

Fig 9. (A) First stage of the base Lloyd's algorithm on single GPU. Number of clusters increased because the size of the centroids array being larger than GPU shared memory capacity. (B) Utilized shared memory when centroid size is larger than the capacity of the physical shared memory on GPU. (C) Merged optimization on single GPU Implementation. (D) Extension of single GPU implementation to multi-GPU.

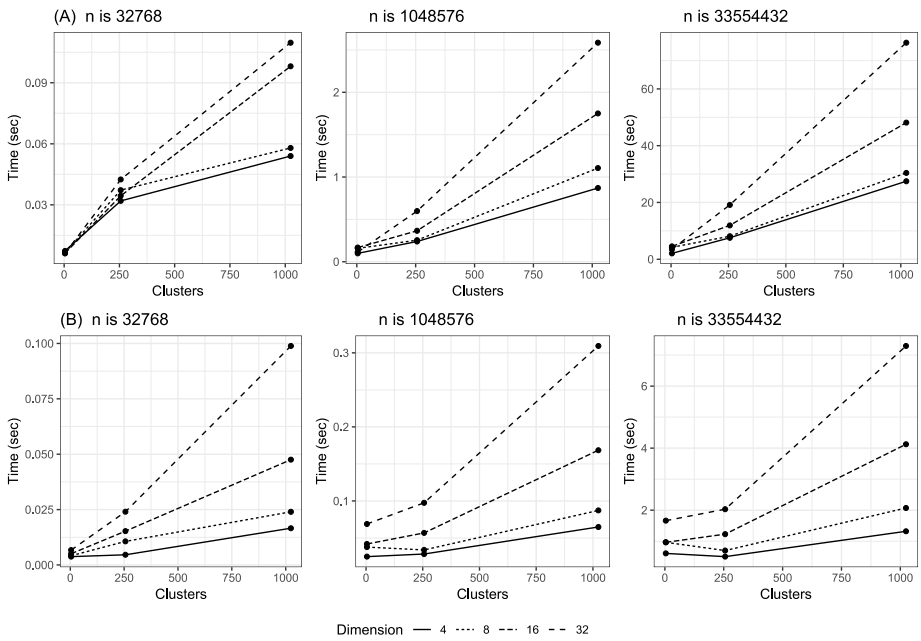

Fig 10. (A) MPI extension of single node optimized OpenMP implementation on two nodes.
(B) MPI extension of optimized multi-GPU on two nodes.

5.6 MPI CUDA Solution

In this section, the parallel execution of the *k*-means clustering Algorithm on the GPU-based clusters is thoroughly evaluated. MPI is employed in order to share the clustering operations amongst all of the nodes available in the cluster. In each node, the clustering is done using the most optimized CUDA solution developed in our work. The results of these clustering operations are outlined in Figure 10(B).

5.7 Summary

In this section, we summarize the speedup ratio and achievements of all the proposed methods and techniques (Table 4).

Table 4. Summarized results of different methods independently.

method	Plat.	Comp. by	min speedup ratio	max speedup ratio	SD speedup ratio	mean speedup ratio
Mathematical optimization	CPU	single thread CPU	0.968	1.742	0.247	1.35
AVX2 SIMD and mathematical optimization	CPU	single thread CPU	1.58	4.36	0.68	2.813
local space OpenMP multi-threading	CPU	single thread CPU	0.624	26.98	10.68	16.72
Single node Openmp	CPU	single thread CPU	4.971	149.4	53.92	67.72
Mathematical optimization	GPU	single GPU	0.346	1.37	0.273	0.971
Shared memory	GPU	single GPU	0.979	2.812	0.325	1.165
Atomic add	GPU	single GPU	0.323	2.910	0.493	1.373
Single GPU opt	GPU	single GPU	0.122	6.999	1.77	2.952
Multi-GPU	GPU	Single GPU opt	1.254	11.80	1.970	2.988
MPI multi-GPU	MPI GPU	single GPU opt	0.642	12.59	2.472	4.132
MPI OpenMP	MPI CPU	Single node OMP	0.764	13.84	2.513	2.690

Furthermore, we have evaluated the works discussed in the Background section and our OpenMP solution, and the results indicated orders of speedup over the best algorithmic approaches to handle the *k*-means clustering operation's overhead, that is, the distance calculation phase of Lloyd's heuristic. The results of the implementations and our given solution are acquired by

clustering datasets of different settings. The information regarding the datasets, their settings, and their respective clustering time are given in Table. 5. All of the algorithms are implemented in a parallel manner, with 56 CPU threads handling the clustering procedures. These results are from 15 iterations of the clustering operation. We used the implementation method that is found in [29] to achieve the results.

Table 5. Evaluation results from OpenMP optimized CPU implementation.

Name	Size	k	Dim.	Time(s)
Naïve Solution (W/O optimizations)	**U3**	**256**	**8**	**128.057**
			16	**203.9**
			32	**355.543**
		1024	**8**	**511.344**
			16	**813.002**
			32	**1424.04**
Annulus	**U3**	**256**	**8**	**89.5424**
			16	**159.025**
			32	**299.812**
		1024	**8**	**367.269**
			16	**667.463**
			32	**1229.83**
Hamerly	**U3**	**256**	**8**	**90.8453**
			16	**157.055**
			32	**287.045**
		1024	**8**	**375.576**
			16	**663.636**
			32	**1179.58**
Drake	**U3**	**256**	**8**	**45.3905**
			16	**84.6373**
			32	**164.316**
		1024	**8**	**173.326**
			16	**332.755**
			32	**669.914**
Heap	**U3**	***256***	**8**	**99.9447**
			16	**174.157**
			32	**322.528**
		1024	**8**	**409.33**
			16	**719.192**
			32	**1294.9**
Sort	**U3**	**256**	**8**	**37.3912**
			16	**206.426**
			32	**376.796**
		1024	**8**	**59.224**
			16	**676.713**
			32	**1528.72**
Compare	**U3**	**256**	**8**	**52.261**
			16	**216.727**
			32	**374.489**
		1024	**8**	**98.0183**
			16	**728.306**
			32	**1487.58**
Our work	**U3**	**256**	**8**	**15.39**
			16	**22.93**
			32	**37.26**
		1024	**8**	**57.91**
			16	**90.134**
			32	**145.56**

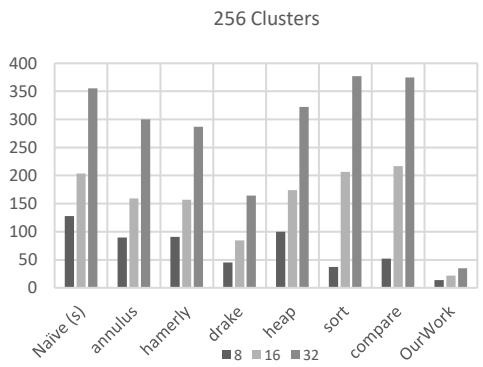

Fig 11. Comparison of the proposed multi-threaded method with a similar multi-threaded method. The number of clusters is 256.

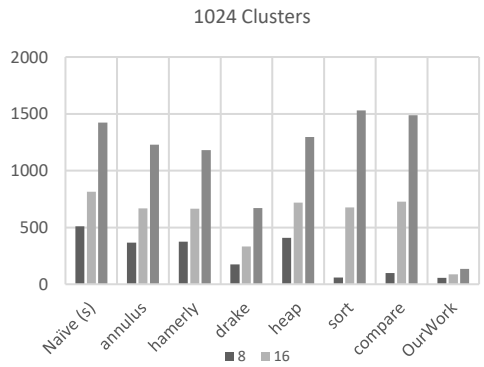

Fig 12. Comparison of the proposed multi-threaded method with a similar multi-threaded method. The number of clusters is 1024.

In a setting with 256 clusters and with 32 dimensions, our solution outperformed the fastest method by 4.7X. In the case of 1024 clusters and 32 dimensions, the OpenMP solution proposed in our work outperformed the fastest accelerating heuristic by 5X. In all previous implementations, which related results mentioned in Table 5, heuristic methods were used, and the execution time was different in each iteration. In these implementations, by increasing the number of iterations, execution time was reduced in the majority of scenarios, while in our proposed method, the execution time was almost constant in each iteration. Our suggestion to achieve higher performance was to run our proposed method during the first iterations and then cascade one of the best previous implementations, like Drake, to utilize their heuristics.

6 Conclusion

K-means clustering is one of the widely used clustering methods employed in different applications. In this paper, we proposed multiple techniques to utilize the intrinsic parallelism of Lloyd's algorithm. These techniques were customized for various hardware platforms to achieve higher performance and speedup. We proposed two main solutions for CPUs and GPUs and then, in another step, extended these solutions to multiple nodes by MPI. In the CPU solution, we utilized an algebraic equality and alleviated the computation overhead based on the dataset size. Furthermore, based on this equality, the main computation of Lloyd's algorithm was reduced to the dot product instead of Euclidean distance calculation. Also, we used AVX2 instructions to compute dot products based on a SIMD scheme. As an extension of the single-thread solution, we considered local space memory for each thread to implement all stages of Lloyd's algorithm in parallel based on OpenMP. In GPU platforms, we used shared memory without considering shared memory capacity to load centroids and also utilized atomic operations to handle reductions in the second phase of the base algorithm. By using this technique, we implemented all stages of Lloyd's algorithm in GPU to reduce GPU/CPU data transfers. In the next step, we proposed a multi-GPU solution as an extension of the single GPU solution by separating points into different GPUs. This was done using OpenMP threads in the CPU to call GPUs and finally merge results in the CPU. As a higher level of parallelism, we extended the computation to multiple nodes and then reduced the transfers between the nodes by MPI utilities. The MPI-multi-GPU solution achieved a 1558X speedup in comparison to a single CPU thread. This was implemented for the conventional Lloyd's algorithm for the U3 dataset with 1024 clusters and dimensions. In addition, MPI-multi-thread implementation achieved a 149X speedup in comparison to a single CPU thread for the conventional Lloyd's algorithm.

Acknowledgment

We are grateful to Prof. Hamid Sarbazi Azad, Head of the IPM high-performance computing center, for his support and useful guidance.

References

[1] M. Vajteršic, P. Zinterhof, and R. Trobec, "Overview–Parallel Computing: Numerics, Applications, and Trends," Parallel Computing, pp. 1-42: Springer, 2009.

[2] F. Sebastiani, “Machine learning in automated text categorization,” ACM computing surveys (CSUR), vol. 34, no. 1, pp. 1-47, 2002.

[3] B. Hong-Tao, H. Li-li, O. Dan-tong, L. Zhan-shan, and L. He, "*K*-means on commodity GPUs with CUDA," pp. 651-655.

[4] F. Masulli, and A. Schenone, “A fuzzy clustering based segmentation system as support to diagnosis in medical imaging,” Artificial Intelligence in Medicine, vol. 16, no. 2, pp. 129-147, 1999.

[5] W. Li, and A. Godzik, “Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences,” Bioinformatics, vol. 22, no. 13, pp. 1658-1659, 2006.

[6] S. Lloyd, “Least squares quantization in PCM,” IEEE Transactions on information theory, vol. 28, no. 2, pp. 129-137, 1982.

[7] J. MacQueen, "Some methods for classification and analysis of multivariate observations," pp. 281-297, 1967.
[8] Q. Du, M. Emelianenko, and L. Ju, “Convergence of the Lloyd algorithm for computing centroidal Voronoi tessellations,” SIAM journal on numerical analysis, vol. 44, no. 1, pp. 102-119, 2006.
[9] A. Jain, “Data clustering: 50 years beyond *K*-means,” vol. 31, no. 8, pp. 651-666, 2010.
[10] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, and S. Y. Philip, “Top 10 algorithms in data mining,” Knowledge and information systems, vol. 14, no. 1, pp. 1-37, 2008.
[11] L. Dagum, and R. Menon, “OpenMP: an industry-standard API for shared-memory programming”, vol. 5, no. 1, pp. 46-55, 1998.
[12] “openMPI, https://www.open-mpi.org/,” 2018.
[13] “MPICH, https://www.mpich.org/,” 2018.
[14] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture," pp. 103-104, 2021.
[15] G. Hamerly, and C. Elkan, "Alternatives to the *k*-means algorithm that find better clusterings," pp. 600-60, 2002.
[16] C. Elkan, "Using the triangle inequality to accelerate *k*-means," pp. 147-153, 2003.
[17] J. Drake, and G. Hamerly, "Accelerated *k*-means with adaptive distance bounds," pp. 42-53, 2012.
[18] G. Hamerly, "Making *k*-means even faster," pp. 130-140, 2010.
[19] S. J. Phillips, "Acceleration of k-means and related clustering algorithms," pp. 166-177, 2002.
[20] D. Arthur, and S. Vassilvitskii, "*k*-means++: The advantages of careful seeding," pp. 1027-1035, 2007.
[21] J. Bhimani, M. Leeser, and N. Mi, "Accelerating *K*-Means clustering with parallel implementations and GPU computing," pp. 1-6, 2015.
[22] E. Kijsipongse, and U. Suriya, "Dynamic load balancing on GPU clusters for large-scale *K*-Means clustering," pp. 346-350, 2012.
[23] I. S. Dhillon, and D. S. Modha, "A data-clustering algorithm on distributed memory multiprocessors," Large-Scale Parallel Data Mining, pp. 245-260: Springer, 2002.
[24] M. Al-Ayyoub, A. M. Abu-Dalo, Y. Jararweh, M. Jarrah, and M. J. T. J. o. S. Al Sa’d, “A GPU-based implementations of the fuzzy *c*-means algorithms for medical image segmentation,” vol. 71, no. 8, pp. 3149-3162, 2015.
[25] T. Kwok, K. Smith, S. Lozano, and D. Taniar, "Parallel fuzzy *c*-means clustering for large data sets," pp. 365-374, 2002.
[26] S. Rahmani, A. Ahmadzadeh, O. Hajihassani, S. Mirhosseini, and S. Gorgin, "An efficient multi-core and many-core implementation of *k*-means clustering," pp. 128-131, 2016.
[27] “AVX2, www.software.intel.com/en-us/articles/how-intel-avx2-improves-performance-on-server-applications,” 2018.
[28] “Cuda Toolkit, https://docs.nvidia.com/cuda/index.html,” 2018.
[29] “https://github.com/ghamerly/fast-kmeans,” 2018.
[30] Ahmadzadeh, Armin, and Hamid Sarbazi-Azad. "Fast and scalable quantum computing simulation on multi-core and many-core platforms," Quantum Information Processing 22, no. 5, 2023.
[31] He, Guanlin, Stéphane Vialle, and Marc Baboulin. "Parallelization of the *k*-means Algorithm in a Spectral Clustering Chain on CPU-GPU Platforms," In Euro-Par: Parallel Processing Workshops: Euro-Par International Workshops, Warsaw, Poland, 2021.
[32] He, Guanlin, Stephane Vialle, and Marc Baboulin. "Parallel and accurate *k*-means algorithm on CPU-GPU architectures for spectral clustering," Concurrency and Computation: Practice and Experience 34, no. 14, 2022.
[33] Karbhari, Shruti, and Shadi Alawneh. "GPU-based parallel implementation of *k*-means clustering algorithm for image segmentation," In IEEE International Conference on Electro/Information Technology (EIT), pp. 0052-0057. IEEE, 2018.
[34] Cuomo, Salvatore, Vincenzo De Angelis, Gennaro Farina, Livia Marcellino, and Gerardo Toraldo. "A GPU-accelerated parallel *K*-means algorithm," Computers & Electrical Engineering 75, 2019.
[35] Daoudi, Sara, Chakib Mustapha Anouar Zouaoui, Miloud Chikr El-Mezouar, and Nasreddine Taleb. "A Comparative Study of parallel CPU/GPU implementations of the *K*-means algorithm," In International Conference on Advanced Electrical Engineering (ICAEE), pp. 1-5. IEEE, 2019.

Armin Ahmadzadeh received the B.Sc. and the M.Sc. degrees in computer engineering from Qazvin Azad University, Qazvin, Iran. Currently, He is a Ph.D. candidate in computer engineering at the Sharif University of Technology, Tehran, Iran. He is a researcher in the School of Computer Science at the Institute for Research in Fundamental Sciences (IPM), Tehran, Iran. His research interest includes computer architecture, parallel and quantum computing and simulation, machine learning and big data analysis, NAND Flash memory-based storage systems. https://orcid.org/0000-0002-2228-3297

Saeid Rahmani is a computer engineer with a bachelor of science degree from Kharazmi University in Tehran, Iran, which he obtained in 2016. He also holds a master of science degree in artificial intelligence and robotics from Sharif University of Technology in Tehran, Iran, which he completed in 2020. Currently, he is a member of the School of Computer Science at the Institute for Research in Fundamental Sciences (IPM) in Tehran, Iran, where he conducts research on various topics, such as bioinformatics, machine learning, biological and medical data analysis, neuroscience, genetic and cancer biology, and computer vision.
https://orcid.org/0000-0002-2447-1937

Omid earned his MSc in Computer Engineering from the University of Alberta in 2021. Before that, he was an active member of the HPC Laboratory at the Institute for Research in Fundamental Sciences (IPM) in Tehran, Iran. Currently, he works as a Machine Learning and Client Analyst at ISAIC, a non-profit organization based in Alberta, Canada. His research interests include high-performance computing, big data, cloud computing, artificial intelligence, and MLOps. He has published several papers on these topics in reputable journals and conferences.
https://orcid.org/0000-0001-9818-3121

Dara Rahmati (M'15) is an Assistant Professor of Computer Science and Engineering at Shahid Beheshti University, Tehran, Iran. He received his B.Sc. and M.Sc. degrees in Computer Engineering from the University of Tehran, Iran, in 1998 and 2001, respectively, and his Ph.D. degree in Computer Engineering from the Sharif University of Technology, Tehran, Iran, in 2012. His research interests include Computer Architecture, Hardware Accelerators, Machine Learning and Networks-on-Chip. He is a member of the IEEE and has published several papers in prestigious journals and conferences.
https://orcid.org/0000-0003-0104-4016

Saeid Gorgin (Senior Member, IEEE) received the B.S. degree in computer engineering from Islamic Azad University, South Tehran Branch, Tehran, Iran, in 2001, the M.S. degree in computer engineering from Islamic Azad University Science and Research Branch, Tehran, in 2004, and the Ph.D. degree in computer system architecture from Shahid Beheshti University, Tehran, in 2010. He is currently an Associate Professor of computer engineering with the Department of Electrical Engineering and Information Technology, Iranian Research Organization for Science and Technology, Tehran. He is also a Visiting Scientist with the Laboratory of Computer Systems, Department of Computer Engineering, Chosun University, South Korea. His current research interests include computing systems, computer arithmetic, and VLSI design.
https://orcid.org/0000-0001-5898-4872