International Information and Engineering Technology Association
Advancing the World of Information and Engineering

Ingénierie des Systèmes d'Information
Vol. 26, No. 1, February, 2021, pp. 59-66
Journal homepage: http://iieta.org/journals/isi

Parallelization of the K-Means++ Clustering Algorithm

Sara Daoudi*, Chakib Mustapha Anouar Zouaoui, Miloud Chikr El-Mezouar, Nasreddine Taleb

RCAM Laboratory Dept of Electronics, Djillali Liabès University, Sidi Bel Abbes 22000, Algeria

Corresponding Author Email: sara.daoudi@univ-sba.dz

https://doi.org/10.18280/isi.260106

Received: 18 November 2020
Accepted: 27 January 2021

Keywords:
clustering, K-means clustering, K-means++, GPGPU, OpenCL, parallel K-means++, streaming SIMD extension (SSE)

ABSTRACT

K-means++ is the clustering algorithm that is created to improve the process of getting initial clusters in the K-means algorithm. The k-means++ algorithm selects initial k-centroids arbitrarily dependent on a probability that is proportional to each data-point distance to the existing centroids. The most noteworthy problem of this algorithm is when running happens in sequential mode, as this reduces the speed of clustering. In this paper, we develop a new parallel k-means++ algorithm using the graphics processing units (GPU) where the Open Computing Language (OpenCL) platform is used as the programming environment to perform the data assignment phase in parallel while the Streaming SIMD Extension (SSE) technology is used to perform the initialization step to select the initial centroids in parallel on CPU. The focus is on optimizations directly targeted to this architecture to exploit the most of the available computing capabilities. Our objective is to minimize runtime while keeping the quality of the serial implementation. Our outcomes demonstrate that the implementation of targeting hybrid parallel architectures (CPU & GPU) is the most appropriate for large data. We have been able to achieve a 152 times higher throughput than that of the sequential implementation of k-means ++.

1. INTRODUCTION

Clustering is one of the fundamental descriptive data mining tasks. Clustering is the unsupervised learning procedure of grouping a set of objects into the same class [1]. Cluster analysis depends on the assignment of a set of objects into subsets (named clusters) so that objects within a similar cluster are identical according to pre-designated criteria [2]. Various applications are found in ref. [3, 4].

The k-means clustering algorithm developed by McQueen in 1967 [5], one of the simplest unsupervised clustering algorithms, assigns each point in a cluster whose center (centroid) is closest. In general, this algorithm includes three stages: initialization, computation, and convergence.

The center is the mean of all the objects in the cluster; its coordinates are the mean for each feature separately from all the objects in the cluster, where each cluster is depicted by its centroid. However, what is important to note is that the authors in ref. [6] have determined that the K-means implementation will converge to a local optimum; the selection of the first cluster centers has been crucial when implementing the K-means algorithm. The challenge is to ensure a better initialization of the centroids. There are several approaches to achieve this goal. In ref. [7], it was proposed to use a smart technique to seed the initial centroids for k-means, leading to a combined algorithm called k-means++. k-means ++ has rapidly proved to be one of the most popular, with implementations in areas such as clustering geographic information [8], summarizing microblogs [9], social network analysis [10], and image compression [11]. The k-means++ method chooses the starting centroids arbitrarily based on a probabilistic approach that is proportional to each data point distance to the existing centroids. To select the k initial centroids, K-means++ needs k-iterations, and because of that; it might not be an efficient method for the dataset with a large number of clusters. Bahmani et al. [12] present another K-means initialization method, nevertheless, they do not furnish time comparisons between their implementation and K-means or even K-means++. The most noteworthy problem of this algorithm is when running happens in sequential mode, as this reduces the speed of clustering. In our work, we choose to exploit the Graphics Processing Units (GPUs) since they guarantee higher performance. In addition, the GPUs are going to be progressively utilized for real-time implementations of the k-means algorithm. For a quick overview please refer to [13-16].

In this paper, we propose a new parallel implementation of the k-means ++ algorithm using GPUs. The Open Computing Language (OpenCL) [17, 18] platform is used as a programming environment in conjunction with the use of the Streaming SIMD Extension (SSE) technology [19]. The focus is on optimizations directly targeted to this architecture to exploit the most of the available computing capabilities. K means++ is an iterative algorithm in which each iteration includes two phases: data assignment and k-centroids up-date. To accelerate the compute-intensive parts of k-means++, the initialization step to select the initial k-centroids is performed in parallel using SSE instructions while the data assignment step is loaded to the GPUs in parallel. Only the K- centroids recalculation and the convergence test steps are performed by the CPU.

So our contributions are outlined as follows:

1. We present a new initialization step algorithm to select the initial centroids in parallel using the SSE technology, because the initialization phase to select the initial centroids must be calculated with precision

2. We present a new point assignment algorithm for GPUs, the OpenCL platform is used while using the Local memory and Private memory to compute the distance.

3. We will show how the parallel K-means++ implementation scales to large datasets and we will examine its performance compared to that of the sequential k-means++.

The rest of the paper is organized in various sections. Section 2 provides an outline of related works on accelerating the k-means++ algorithm. Section 3 will briefly introduce the sequential K-Means++ algorithm. Section 4 will present a detailed description of our original parallel K-Means++ using OpenCL and SSE technology. Section 5 will focus on reporting the synthesis results and discussions. Finally, Section 6 concludes the paper.

2. RELATED WORKS

Cluster analysis has been an important research topic for data mining researchers for many years. In the past, under the condition of small data size, researchers mainly focused on the optimization problem of the k-means algorithm itself.

And then several methods have been invented to parallelize the clustering algorithms [20-22] including k-means [13-16, 23-25] to further improve the efficiency of them.

In our previous article reported in ref. [15], we have compared three different approaches dealing with a parallel implementation of k-means: OpenMP, Pthread, and OpenCL. The results have demonstrated that all three parallel techniques showed a significant increase in speed, with the best results being achieved by OpenMP for smaller datasets and by OpenCL for larger datasets.

There are however few works reported on the parallelization of K-means++. The earliest work is a Master thesis by Karch [11], and then the work of Maliheh et al. [26]. In [11], the thesis examines the parallelization of both k means++ and k means over GPU utilizing CUDA. They only parallelized the computation of distances of data points to each seed and none of the other portions of the k-means++ implementation. They have used a GPU Nvidia GeForce 9600M GT card obtaining a maximum speed-up that is 5 times faster than that of the CPU implementation. In ref. [26], the authors aim to parallelize the most time-consuming steps of the k-means++ method, the initialization step is offloaded to the GPUs in parallel using CUDA. The initial centers are picked consecutively by a probability that is proportional to the distance to the nearest center using the GPU GeForce GTX 1070 card. Utilizing this approach, they were able to obtain a speed-up that is 36 times faster than that of the CPU implementation. In our present work, we have parallelized the data assignment phase on GPU using OpenCL while the initialization step to select the initial centroids has been performed in parallel using SSE instructions in the CPU because the initialization phase to select the initial centroids must be calculated with precision, so it is an excellent choice to make optimizations using SSE technology. The K-centroids recalculation and the convergence tests phases have been performed by the CPU because these steps do not consume a lot of time compared to the other phases.

To sum up, we have adapted in this work our Kmeans++ parallelization strategy to handle large datasets. To perform this implementation, we have been using the OpenCL platform and the SSE technology. Finally, we have compared this parallel Kmeans++ implementation with the sequential Kmeans++ one.

3. PROPOSED METHOD

3.1 K-means ++

The K-means algorithm [5] picks the first centers randomly. Since they depend on pure luck, they can be chosen really badly. The K-means++ algorithm [12] tries to solve this issue, by spreading the first centers evenly. The serial implementation of K-means++ is shown in pseudocode Algorithm 1.

Algorithm 1: The Serial k-means++ Algorithm

Input: X;Setof data points (N), number_of_clusters(k), k>0

Output: A data pointsN partitioned into k clusters

1: Select centroids ***c_1, c_2, …, c:***

1-a. Choose one center c_i uniformly at random from among the data points X.

1-b. Take a new center c_i, for every data-point x, calculate **D(x),** the distance among x, and the nearest center that has just been picked. choosing x € X with probability $\frac{D(x)^2}{\sum_{x \in X} D(x)^2}$.

1-c. Repeat Steps 1-b until k centers have been chosen.

O(#N x D x K)

2: Iterate until a convergence condition is satisfied

3: For each data point ***x_i, i=1..N:***

4: For each data cluster c_j, *j=1..**K:***

5: Compute the distance to c_j, *given all **D dimensions***

6: Assign the membership of data-point ***x_i to nearest cluster j***

7: For each data cluster c_j, *j=1..**K:***

8: Compute the centroid: ***c_j =the mean of all data-points whose membership is J***

O(N x D x Kx #iterations)

We appeal the weighting utilized in step 1-b <D^2 weighting>.

Algorithm 1 shows that, after the selection of each centre, the distance of each point to the nearest cluster is updated. It is detailed as follows:

1-a Choose one center uniformly at random from among the data points.

1-b For every data-point x, calculate D(x), the distance among x and the nearest center that has just been picked. The Euclidean distance is commonly used as a measure of cluster scatter for K-means++ clustering. This metric is preferred because it minimizes the mean distance between points and the centroids [27]. In this article, the Euclidean distance is applied.

What's more, pick one new data-point at arbitrary as a new center utilizing a weighted probability distribution where a point x is picked with probability proportional to $D(x)^2$ (see formula (1)). k-means++ uses distance $D(x)^2$ weighting method to cull the next opportune centre. This algorithm reduces the instability effect occurring in K-Means and provide better stable clustering results [28].

$$\text{Probability } p(x)=\frac{D(x)^2}{\sum_{x \in X} D(x)^2} \quad (1)$$

1-c Repeat Steps 1-b until k centers have been chosen.

2- K-Means++ is an amendment to the K-Means algorithm in order to surmount the arbitrary selection of initial cluster centre. We can continue with the standard K-Means algorithm after initializing the centroids. Using K-Means++ to initialize the centroids tends to improve the clusters. Although it is computationally costly relative to random initialization, subsequent K-Means often converge more rapidly. For every iteration, the k-means clustering executes two phases; firstly, the data assignment phase, which associates every data-point with its nearest centroid relying on a distance metric (here the Euclidean distance metric is used). The result of the data assignment phase is a membership vector indicating the new cluster center for every data-point. Secondly, after the data assignment phase, k-means execute another phase which is updating the centroids. This phase calculates the new centroids by computing the average of all the data-points belonging to the same cluster. Assuming that the number of objects (points) in cluster i is defined as s_i the formula for the cluster update step is shown here:

$$c_i = \frac{1}{s_i}\sum_{j=1}^{s} X_j \qquad (2)$$

In the initialization step, to select the initial K-centers, k-means++ has to make a full pass through the data for every cluster center sampled. This leads to the maximum computational complexity of (N*K*D), where N is the number of data-points, D the dimensionality of the data and K the number of cluster. The most computationally-intensive part of this implementation is the distance calculation step. After the initialization step we proceed to utilize standard k-means. For each iteration of the k-means clustering implementation, the maximum computational complexity is (D*N*K +N*K + N*D). The most computationally part of this implementation is the distance computation step. The most computationally-intensive part of this implementation is the distance calculation step that requires one subtraction, one addition, and one multiplication to compute the distance partial sum per each data point. In general, the number of operations is approximately equal to *D*N*K*3 corresponding to the number _of iterations.

3.2 The parallel K-means++ implementation

The K-Means++ algorithm is an iterative method. Thereby, the computation of a distance between each point and the various clusters shows a lot of parallelism within each iteration, since the For loops of lines 3 to 6 (Algorithm 1) are time consuming for both the smallest and the biggest datasets. Therefore, the 3 to 6 steps part of pseudocode Algorithm 1 is an excellent candidate to explore parallelization. As we previously referenced, we aim to parallelize the most time-consuming phase of the initialization step to select the initial centroids of the K-means++ algorithm, namely the 'For' loop of line 1-b (Algorithm 1). The distance of all points to the chosen center should be determined and afterward, the clusters are picked based on a probability formula. These are the most testing parts of parallelizing the K-means++.

A parallel implementation is shown in pseudo code Algorithm 2 using OpenCL and The SSE technology. In the following, the main stages are described in more details. Figure 1 indicates an overview of the proposed approach.

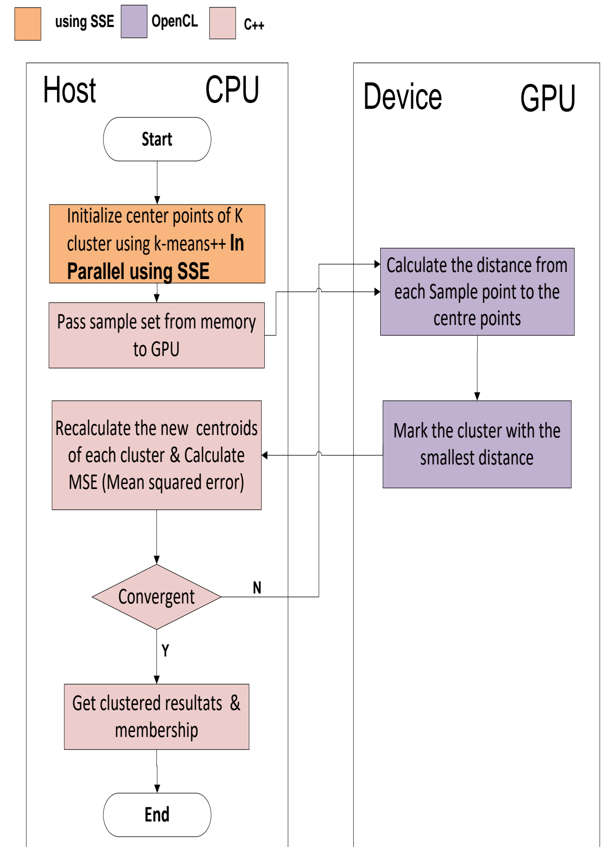

Figure 1. Steps of a parallel K-means++ implementation

The initialization phase to select the initial centroids is performed in parallel using the SSE instructions (The detailed code for optimization is shown in Algorithm 2), then the distance of all points to the chosen center should be determined, and afterward the clusters are picked based on a probability formula (1).

In addition, the SSE instruction allows us to compute four floating-point numbers with precision [19, 29]; we have already mentioned previously that the initialization phase to select the initial centroids must be calculated with precision, so it is an excellent choice to make optimizations using SSE technology.

This work will investigate the possibility of utilizing the SSE technology to accelerate the initialization step of kmeans++ algorithm to select the initial centroids by executing multiple computations with a single instruction. In 1999 with Pentium III, Intel introduced SSE to the 'x86 architecture' [19]. In [19], the authors have given a lot of details on the abilities and motivations of SSE. SSE adds eight 128-bit wide registers to the architecture [19]. With the arrival of the Pentium 4, Intel presented the second generation of SSE, usually known as SSE2 [30].

The SSE instructions allow in our implementation from the initialization phase where we will calculate the distances between each pair of data-points and data-clusters requiring "N" floating point operations, say in matrix distance, a similar number of the distances may be executed in N/4 cycles, instead of N cycles. This is because four floating point distances should be done in a single cycle. Nevertheless, the rearrangement of data in a format that is acceptable for SIMD computation is required. This consumes a few clock cycles.

The OpenCL [31] platform is used as a programming environment to calculate the assignment phase of the k-means ++ algorithm in parallel on GPU (see Algorithm 2). OpenCL

is utilized to accelerate different applications in various areas like computational chemistry, bioinformatics, and other fields of research [31, 32].

When using OpenCL, we think about several distinct methodologies for parallelization including:

- Global Memory
- Local memory
- Private memory

The steps to implement parallel K-means++ method utilizing OpenCL and SSE (see Figure 1) are shown below:

Step 1: We choose one cluster uniformly at random from among the data. After that, we use The SSE instructions to calculate the distance of all points to the chosen center which should be determined and afterward, the clusters are picked based on a probability formula. Then, we go on and read the user load clusters and data, and then set the number of clusters. The predefined number of iterations is utilized as a convergence criterion.

Step 2: We copy the user load datasets and the initial K -clusters from the memory to the GPU.

Algorithm 2: A parallel K-Means++ Algorithm

```
Input:X; Set of Data points(N), Clusters(K), Distance (d), k > 0
Output: A data points N, partitioned into K clusters

// C, the list of k data points (vectors) that should be used as initial seeds
for k-means++ clustering
Function Parallel-K-means++_initialization_phase (X, k)
    for i from 0 to n − 1 do        In Parallel using SSE
        D[i] := ∞
    —The initial seed is uniformly selected at random —
    C[0] := rand x ∈ X
    —The rest of the seeds are selected probabilistically by distance—
    for j from 1 to k − 1 do
        for i from 0 to n − 1 do     In Parallel using SSE
            D[i] := min(||X[i] − C[j − 1]||, D[i])
            W[i] := D[i]
        i := WEIGHTED_RAND_INDEX (W)
        C[j] := X[i]
  return C
//calculate the distance and membership in parallel using OpenCL
workgoup_size = NUM_CLUSTERS * NDIMS;
number_threads= NUM_POINTS ;
Function kmeans_kernel_assignment_phase (points, clusters,
membership)
        g_id = get_global_id(0);
clusters_local[NUM_CLUSTERS*NDIMS] ;
        l_id=get_local_id(0) ;
clusters_local[l_id]—> clusters[l_id] ;
barrier(CLK_LOCAL_MEM_FENCE)
        index=0
    for (int c=0 ;c< NUM_CLUSTERS ; c++) do
            min_dist = ∞;
        for (int d = 0; d< NDIMS; d++) do
                distance =0;
                ans=0 ;
                ans = (points[d* NUM_POINTS + g_id] −
clusters_local [c * NDIMS + d]);
                distance+ = (ans * ans) ;
            end
        if (distance <min_dist) then
                            min_dist = distance;
                            index = c;
            end

    end
            membership[g_id] = index;
  End
```

Step 3: In this step, we utilize the OpenCL platform to calculate the Euclidean distance. The N-threads correspond to the N-load data, and the workload of each thread is as expressed in formula (3). Thus, each cluster is loaded by its workgroup only once into the local memory, then gets shared among all the threads in the workgroup. Afterward, each thread computes the distance among the point of the dataset and cluster. We then save the index of the cluster that is the nearest to the point of the dataset.

$$\text{Thread} = \text{Ceil}\left(\frac{\text{Numberofobjects}}{\text{Threads}}\right) \quad (3)$$

Step 4: The load data are arranged depending on the closest center, and then the partial results are copied to the global memory of the GPU.

Step 5: In CPU, we recalculate the new centroid of each cluster and calculate the Mean Squared Error (MSE). If the maximum iteration number is attained, we will continue to the next step, and if not, we will return to the preceding step.

Step 6: The end of the algorithm, we get the clustered results.

4. EXPERIMENTAL RESULT

In this section, we will explain the evaluation part and analysis of the performance of the Kmeans++ algorithm in both parallel mode (GPU) and serial mode (CPU). Our implementation was performed on windows 7 and exploited our heterogeneous computing environment, that integrated an Intel(R) Xeon(R) X5650 @2.67GHz, 12GB of RAM, a GPU Nvidia GeForce GTX 1060 with 6 GB memory, and Nvidia OpenCL 1.2 for the GPU code to run K-means++.

We have used randomly generated datasets for running our experiments for random floating-point among data between −1 and +1 [33].

Datasets with variables N (number of Objects) vary from 2048 to 4194304 data points which are computed with different dimensions. Furthermore, these datasets are computed for various clusters.

We have studied the performances of respectively GPU and CPU implementations when processing data with various cluster sizes K, object sizes N, dimension sizes D, and iteration sizes.

In this work, the number of iterations defined previously is utilized as a convergence criterion because this way, it is much easier to identify the acceleration factor. For a better interpretation, the performances are presented by both the throughput in gflops and the execution time in milliseconds.

We have looked at the performance of the parallel implementation of the k-means ++ algorithm using GPU and that of the sequential implementation of the k-means ++ on a CPU. Then, for both GPU and CPU implementations runs, the same datasets were utilized.

1. We have compared the computation time only in the initialization phase to select the initial k-clusters in parallel using SSE instructions with the one in the initialization phase to select the initial k-clusters in series without using SSE of the k-means ++ algorithm, where the number of iterations here is the number of k-clusters.
2. We then compared the clustering time, which is the global time for all iterations, which includes the computation and communication time between the CPU and the GPU and does not include the time obtained for the initialization of the k-clusters, where the pre-defined number of iterations is used as a convergence criterion. When the number of iterations is fixed, it is much easier to identify the acceleration factor.

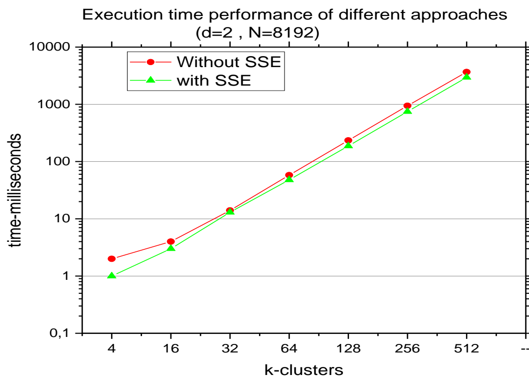

Figure 2. Execution time for the initialization step with and without using SSE instructions of k-means++ with varying cluster sizes when dataset=8192

Figure 3. Execution time for the initialization step with and without using SSE instructions of k-means++ with varying cluster sizes when dataset=65536

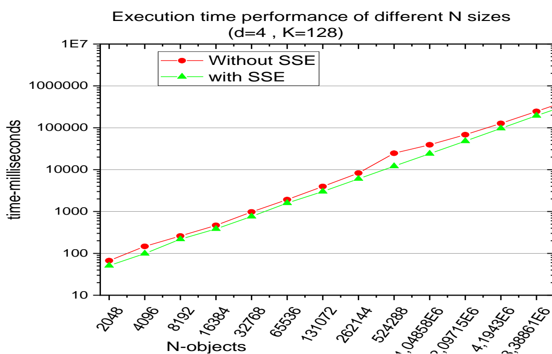

Figure 4. Execution time for the initialization step with and without using SSE instructions of k-means++ with varying object sizes when K-cluster =4

We have applied SIMD SSE instructions to the initialization step (The detailed code for optimization is shown in algorithm 2) and got the optimization results which are shown in Figures 2, 3, and 4.

Figures 2, 3 shows the Execution time for the initialization step to select the initial centroids when using SSE and without using SSE of the k-means++ algorithm with different cluster sizes on CPU when data-points were set to 8192 and 65536, respectively. A conclusion might be drawn that whatever the number of clusters, the parallel implementation using the SSE instructions surpasses the traditional implementation (without using the SSE).

Figure 4 shows the Execution time for the initialization step for selects the initial centroids when we used the SSE and without used the SSE, of k-means++ algorithm with different object sizes. A conclusion might be drawn that whatever the number of objects the version parallel using the SSE outperforms significantly the one without the use of SSE.

From these figures, we notice the performance gain when using SSE. In Figure 4, we take a scale dataset of 524288 points with 4 features to calculates the 128 initials centers, the performance gain due to our parallel implementation of the initialization step using the SSE instructions is up to 50.86% compared with no optimization (the traditional implementation of the initialization step without using SSE for the k-means ++ algorithm).

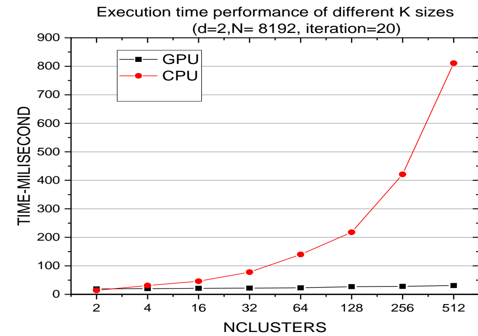

Figure 5. Execution time for k-means++ with varying cluster sizes on GPU and CPU when dataset=8192

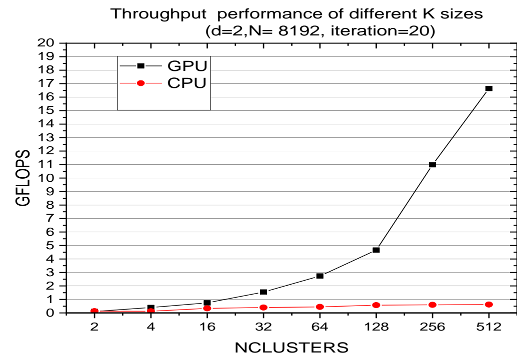

Figure 6. Throughput for k-means++ with varying clusters sizes on GPU and CPU when dataset=8192

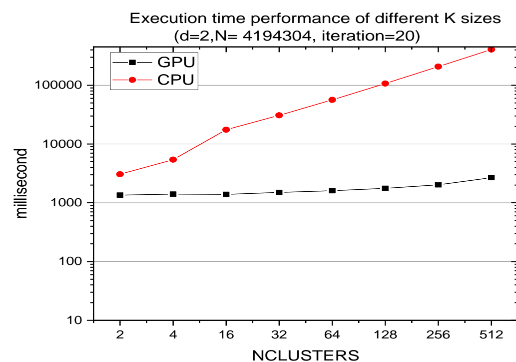

Figure 7. Execution time for k-means++ with varying cluster sizes on GPU and CPU when dataset=4194304

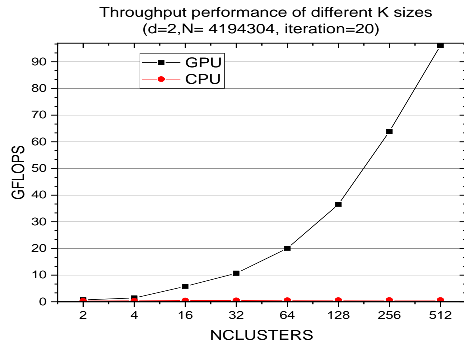

Figure 8. Throughput for k-means++ with varying cluster sizes on GPU and CPU when dataset=4194304

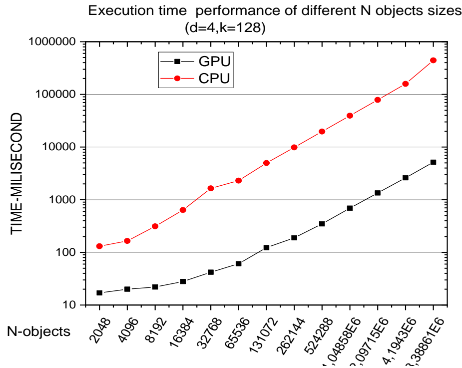

Figure 9. Execution time for k-means++ with varying object sizes on GPU and CPU

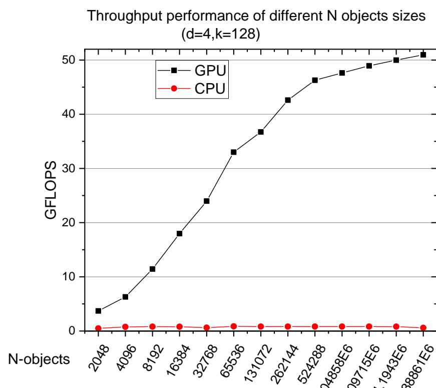

Figure 10. Throughput for k-means++ with varying object sizes on GPU and CPU

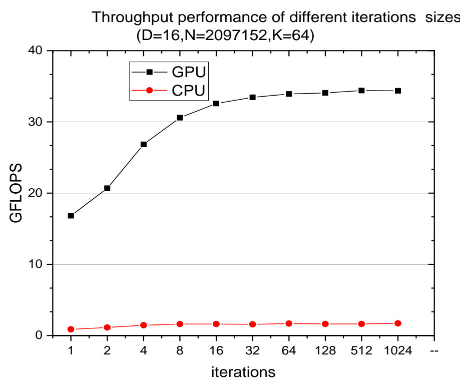

Figure 11. Throughput for k-means++ with varying iterations sizes on GPU and CPU

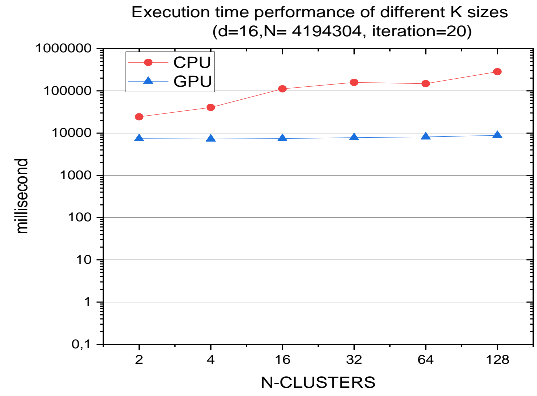

Figure 12. Execution time for k-means++ with varying cluster sizes on GPU and CPU when dataset=4194304

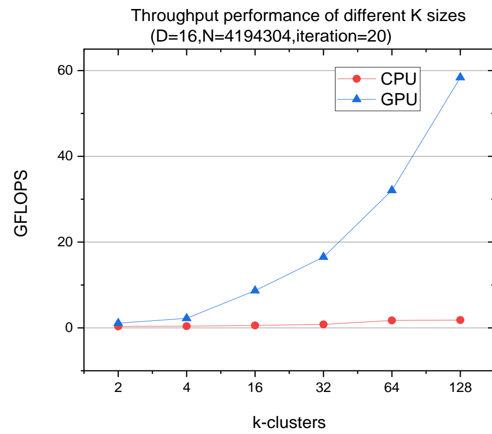

Figure 13. Throughput for k-means++ with varying cluster sizes on GPU and CPU when dataset=4194304

The throughputs of running different dimensional data with different clusters, objects and iteration sizes are shown in Figures 6, 8, 10 and 11, 13.

Figures 5, 7, 9 and 12 show the execution time for k-means++ implementation with different cluster sizes and with varying object sizes on GPU and CPU. More exactly, the CPU surpasses the GPU implementation when the data size is small, the number of dimensionalities is less than or equal to 2 and the number of clusters is less than 4 ($k<4$ and $d\leq 2$). Moreover, the GPU implementation increases the data transmission because the datasets are copied from the memory of CPU to the GPU and then the partial results are copied from the GPU to the memory of CPU.

The GPU implementation of k-means++ achieves better results at large cluster and medium feature sizes, as demonstrated in the figures. We will use Local memory and Private memory to compute the distance. When the cluster size is large and because the clusters are saved in local memory, more data could be reutilized enhancing more the efficiency of the kernel.

Some conclusions could be drawn regardless of the number of objects, clusters and iterations, in that the GPU outperforms the CPU significantly except when the data size is small and the number of dimensionalities is less than or equal to 2 and the number of clusters is less than 4. The GPU performs consistently across any cluster, object or iteration sizes.

The speedup of the parallel implementation of the k-means ++ algorithm utilizing GPU over the sequential version of k-means ++ could reach up to 152 times.

Table 1. k-Means++ GPU versus CPU implementation: Peak throughput results

Data size	Feature Dimension	K-clusters	Peak Throughput GPU (GFLOPS)	Peak Throughput CPU (GFLOPS)	Speed-Up
4194304	**2**	512	96,0843	0,635007	151,32
	4	256	85,928	0,934129	91,987
65536	**16**	64	26,8435	1,21758	22,047
	32	32	13,7424	1,36585	10,062
	64	16	7,80336	1,49241	5,2287
	128	4	4,88862	1,53684	3.181

For this GPU implementation, the peak throughput for different datasets, dimensions, and k sizes is summarized in Table 1. The peak throughput for the GPU implementation is consistently high from the two -dimensional feature to the 16-dimensional one, but starts to decrease after the dimension exceeds 16. This is because it is no longer possible to completely use the GPU resources (local memory) at big dimension sizes. As it is indicated in Table, the maximum speedup of 152 times is reached when processing two-dimensional data.

4.1 Comparison with among the proposed work and the literature

In this section, we compare our best performance obtained against other best state-of-the-art performances, As shown by Table 2, we obtain an acceleration of more than X152 compared with the Sequential k-means++. we tested a maximum of data size of 16.777.216 while the best-published result to date is obtained by ref. [26] using CUDA that achieve an acceleration of X36 compared with the Sequential k-means++. They tested a maximum of data size of 10 million.

Table 2. Comparison with among the proposed work and the literature

Paper	Technology	Speed-Up
[11]	GPU Nvidia GeForce 9600M GT	5
[26]	GeForce GTX 1070	36
Our work	GPU NVIDIA GTX 1060	152

5. CONCLUSION

In this paper, we provide a comprehensive study over the parallelization of the K-means++ algorithm. We propose a new parallel implementation of the k-means ++ algorithm using the graphics processing unit (GPU). The Open Computing Language (OpenCL) platform is used as a programming environment with the use of the Streaming SIMD Extension (SSE) technology. The focus is on optimizations directly targeted to this architecture to exploit the most of the available computing capabilities. This algorithm includes three stages: Initialization, computation, and convergence. K means++ is a compute-intensive iterative technique; each iteration includes two phases: data assignment and k centroids up-date. To accelerate the compute-intensive parts of k-means++, the initialization step is calculated in parallel using the SSE technology and the data assignment step is loaded to the GPU in parallel. Only the K centroids recalculation and the convergence test steps are performed by the CPU. Our results show that the implementation of targeting hybrid parallel architectures (CPU & GPU) is the most appropriate for large data. We have achieved a 152 times higher throughput than that of the sequential version of k-means ++.

ACKNOWLEDGMENTS

The research being reported in this publication was supported by the Algerian Directorate General for Scientific Research and Technological Development (DGRSDT).

REFERENCES

[1] Han, J., Kamber, M. (2006). Data Mining: Concepts and Techniques. Second Edition, Elsevier Inc., Rajkamal Electric Press.

[2] Estivill, C.V. (2002). Why so many clustering algorithms-a position paper. In: Newsletter ACM SIGKDD Explorations Newsletter Homepage Archive, 4(1): 65-75. https://doi.org/10.1145/568574.568575

[3] Eisen, M.B., Spellman, P.T., Brown, P.O., Botstein, D. (1998). Cluster analysis and display of genome-wide expression patterns. Proc Natl Acad Sci., 95(25): 14863-14868.

[4] Cuomo, S., Michele, P., Pragliola, M. (2017). A computational scheme to predict dynamics in IoT systems by using particle filter. Concurr Comput, 29(11): 4101. https://doi.org/10.1002/cpe.4101

[5] MacQueen, J.B. (1967). Some methods for classification and analysis of multivariate observations. Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability. Berkeley, University of California Press, 1: 281-297.

[6] Selim, S.Z., Ismail, M.A. (1984). K-means type algorithms: A generalized convergence theorem and characterization of local optimality. IEEE Transactions on Pattern Analysis and Machine Intelligence, 6(1): 81-87. https://doi.org/10.1109/TPAMI.1984.4767478

[7] Arthur, D., Sergei, V. (2007). K-means++: The advantages of careful seeding. Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, 1027-1035. https://doi.org/10.1145/1283383.1283494

[8] Lee, S.S., Won, D., McLeod, D. (2008). Tag-geotag correlation in social networks. In Proceedings of the 2008 ACM Workshop on Search in Social Media, 59-66. https://doi.org/10.1145/1458583.1458595

[9] Inouye, D. (2010). Multiple post microblog summarization. REU Research Final Report, 1: 34-40.

[10] Velardi, P., Navigli, R., Cucchiarelli, A., D'Antonio, F.

(2008). A new content-based model for social network analysis. In 2008 IEEE International Conference on Semantic Computing, pp. 18-25. https://doi.org/10.1109/ICSC.2008.30
[11] Karch, G. (2010). GPU based acceleration of selected clustering techniques. Department of Electrical and Computer Engineering and Computer Sciences, Silesian University of Technology in Gliwice, Silesia, Poland.
[12] Bahmani, B., Moseley, B., Vattani, A., Kumar, R., Vassilvitskii, S. (2012). Scalable k-means++. Proceedings of the VLDB Endowment, 5(7): 622-633.
[13] Zhang, J., Wu, G., Hu, X., Li, S., Hao, S. (2011). December. A parallel k-means clustering algorithm with MPI. In Fourth International Symposium on Parallel Architectures, Algorithms and Programming, 60-64.
[14] Soua, M., Kachouri, R., Akil, M. (2018). GPU parallel implementation of the new hybrid binarization based on Kmeans method (HBK). Journal of Real-Time Image Processing, 14(2): 363-377. https://doi.org/10.1007/s11554-014-0458-2
[15] Daoudi, S., Zouaoui, C.M.A., El-Mezouar, M.C., Taleb, N. (2019). A comparative study of parallel CPU/GPU implementations of the K-Means Algorithm. In 2019 International Conference on Advanced Electrical Engineering (ICAEE), pp. 1-5. https://doi.org/10.1109/ICAEE47123.2019.9014783
[16] Clemens, L., Sebastian, B., Steffen, Z., Volker, M., Tilmann, R. (2018). Efficient k-Means on GPUs. publication rights licensed to the Association for Computing Machinery. ACM.
[17] Khronos, G. (2017). OpenCL - The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/.
[18] Stone, J.E., Gohara, D., Shi, G. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in Science & Engineering, 12(3): 66. https://doi.org/10.1109/MCSE.2010.69
[19] Thakkur, S., Huff, T. (1999). Internet streaming SIMD extensions. Computer, 32(12): 26-34. https://doi.org/10.1109/2.809248
[20] Raymound, N.T., Han, J. (1994). Efficient and Effective clustering methods for spatial data mining. In Proceedings of VLDB: 144-155.
[21] Zhang, T., Ramakrishnan, R., Livny, M. (1996). BIRCH: an efficient data clustering method for very large databases. ACM Sigmod Record, 25(2): 103-114. https://doi.org/10.1145/235968.233324
[22] Wang, M., Zhang, W., Ding, W., Dai, D., Zhang, H., Xie, H., Xie, J. (2014). Parallel clustering algorithm for large-scale biological data sets. PloS One, 9(4): e91315. https://doi.org/10.1371/journal.pone.0091315
[23] Tang, Q.Y., Khalid, M.A. (2016). Acceleration of K-means algorithm using Altera SDK for OpenCL. ACM Transactions on Reconfigurable Technology and Systems (TRETS), 10(1): 1-19. https://doi.org/10.1145/2964910
[24] Cuomo, S., De Angelis, V., Farina, G., Marcellino, L., Toraldo, G. (2019). A GPU-accelerated parallel K-means algorithm. Computers & Electrical Engineering, 75: 262-274. https://doi.org/10.1016/j.compeleceng.2017.12.002
[25] Li, Y., Zhao, K., Chu, X., Liu, J. (2013). Speeding up k-means algorithm by GPUS. Journal of Computer and System Sciences, 79(2): 216-229. https://doi.org/10.1016/j.jcss.2012.05.004
[26] Maliheh, H.S., Reza, T. (2019). Parallelization of Kmeans++ using CUDA. (in press).
[27] Singh, A., Yadav, A., Rana, A. (2013). K-means with Three different Distance Metrics. International Journal of Computer Applications, 67(10).
[28] Zhang, M., Duan, K.F. (2015). Improved research to k-means initial cluster centers. In 2015 Ninth International Conference on Frontier of Computer Science and Technology, pp. 349-353. https://doi.org/10.1109/FCST.2015.61
[29] Diefendorff, K. (1999). Pentium iii= pentium ii+ sse. Microprocessor Report, 13(3): 1-6.
[30] Sager, D. (2001). Desktop platforms group, and intel corp. The microarchitecture of the Pentium 4 processor. Intel Technology Journal.
[31] Kussmann, J., Ochsenfeld, C. (2017). Employing OpenCL to accelerate ab initio calculations on graphics processing units. Journal of Chemical Theory and Computation, 13(6): 2712-2716. https://doi.org/10.1021/acs.jctc.7b00515
[32] Cadenelli, N., Jaksić, Z., Polo, J., Carrera, D. (2019). Considerations in using OpenCL on GPUs and FPGAs for throughput-oriented genomics workloads. Future Generation Computer Systems, 94: 148-159. https://doi.org/10.1016/j.future.2018.11.028
[33] http://www.cs.virginia.edu/~kw5na/lava/Rodinia/Packages/Current/rodinia_3.0/data/kmeans/.