15-418 Final Project Report

Yu Dong (yudong)

Summary:

The k-means algorithm is one of the most popular unsupervised machine learning algorithms for clustering. In this project, we explored GPU-based k-means algorithms with the support of map reduction, triangle inequality, and dynamic task pool model. We parallelized these K-means clustering algorithm on both GPU and multi-core CPU platforms. In addition, we performed a detailed analysis of performance characteristics, especially focusing on the trade-off between load balance and memory access coalescing through data layout management.

Background:

The K-Means clustering algorithm is an iterative algorithm that is widely used in the data mining area. It aims to partition dataset into K predefined distinct non-overlapping subgroups, called clusters. In there, each data point belongs to only one group. The algorithm assigns data points to a cluster to ensure that the sum of the squared distance between the data points and the cluster's centroid is at the minimum.

However, as the dataset becomes larger, the computation challenge that the K-Means algorithm facing becomes more obvious. For this reason, we decide to implement the K-Means algorithm in parallel using both the Cuda and OpenMp model to overcome problems of the simple K-Means algorithm model. We would also evaluate and compare the performance of the two models in terms of cluster quality, number of iterations, load balance, memory coalescing, and elapsed time.

A. **The k-means algorithm and Benchmarks**:

Mathematically, the clustering problem the k-means algorithm is designed to solve can be summarized as: given a set of d-dimensional data points $\{P_1 \ldots P_k\}$, partition the n points into k clusters $\{k < n\}$ with c centroids to minimize the sum of squares of distances in each cluster.

$$E = \sum_{i=1}^{k} \sum_{x \in C_i} |x - m_i|^2$$

Equation 1. k-means clustering

The inputs of our k-means algorithm are a number of clusters K and the data set. The dataset is a collection of features for each data point. It contains the number of data points and their two-dimensional data coordinates labeled of x-axis and y-axis. The output of our program is a datapoint output file that indicates the cluster to which each point is allocated and a center output file, which lists a matrix of cluster centers in x and y coordinates.

The dataset we used to test our algorithm performance is the six clustering benchmark datasets from researcher P. Fänti and S. Sieranoja at the University of Eastern Finland. We would use the A sets to test correctness of our algorithms and the Birch-sets to test

the algorithm's general performance on large datasets, where N = 100,000. In the end, we would analyze the load balance of our programs using the unbalanced datasets. The dataset in the unbalanced folder has eight clusters in two well-separated groups. The first three clusters are dense with 2000 points each (st.dev= 2043). The five other clusters are sparse with 100 points each (st.dev= 6637) as figure 1 shows.

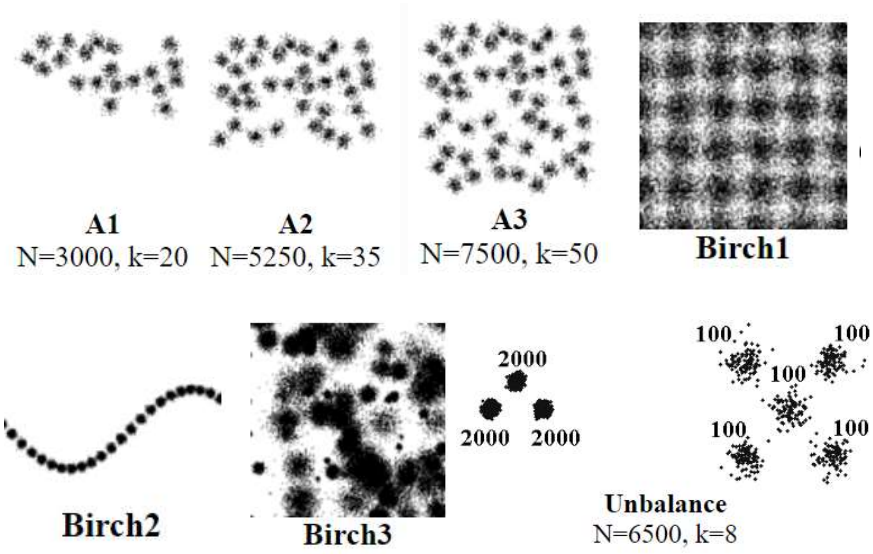

Figure 1. Benchmark Datasets

B. Data Structure and Operations:

a. The data structures we used to implement the k-means algorithm can be found in the kmeans.h file. As the algorithm aims to categorize a set of datapoints into user specified number of clusters, we decided to use typedef struct in c programming to store points and centroid information. The point_t structure is used to store a point information, it contains a point's x and y coordinate value, the cluster id it belongs to, and its distance to the nearest cluster. The centroids_t structure is used to store a centroid information. It summarizes the centroid's x and y coordinate information, its previous coordinate information, the total number of data points it contains, and the sum of distances in both coordinates.

b. In general, the k-means algorithm is an iterative method. It assigns each point to the nearest cluster. Then, the centroids are recalculated by the average of all the points in the same clusters. This process repeats with the new centroids and the updated membership of a data point. The algorithm terminates when the number of memberships changed is less than some thresholds. The iterative k-means algorithm is outlined as below.

Algorithm 1. Standard iterative k-means algorithm

1. Set up the number k of clusters to assign.
2. Randomly initialize k centroids.
3. Repeat
    1. Assign each point to its closest centroid.
    2. Compute the new centroid of each cluster.
4. Until convergence. The centroid positions do not change.

C. Optimization:

a. As the standard k-means algorithm demonstrates, we can see that the k-means algorithm can be divided into two main parts, the membership assignment (labeling step) and centroid recalculation (the updating step). The time complexity of the updating step is close to O(dkn), where d is for dimensional space, k is for the number of clusters, and n is for the number of data points. The time complexity of the updating step is about O(nd) following the same syntax. Since the centroid recalculation updating requires less computation power than the labeling step, we could conclude that the membership labeling step is computationally expensive and could benefit the most from parallelization.

b. Our major dependence the code has is determining the best centroid in each cluster so that all the points in such cluster can have the minimized distance to the centroid. Thus, no matter how fast we calculate the value of the nearest cluster of an input data point, we must wait until all the data points in that cluster are computed before we can determine the updated centroid. In other word, all the distance computation must be completed, and this poses a synchronization barrier for our algorithm control flow. Our implementations are data-parallel because each data point can be labeled independently and then compared at the end. The program does not have much cache locality because threads can access indices that are not close to each other. Our implementations are amenable to SIMD execution if we transform data structure from the array of structure to the structure of arrays.

Approach:

In this project, we explored GPU-based k-means algorithms with the support of map reduction, triangle inequality, and dynamic task pool model. The reason we decided to use GPU is because GPU with Compute Unified Device Architecture (CUDA) can be used as a highly programmable massively parallel computing device. It can achieve a massive parallelism through launching a general-purpose shared-memory and a large number of lightweight threads. In this section, we would provide implementation details of each of our parallel k-means algorithms. The parallelism is performed on NVIDIA's Graphics Processing Units with CUDA.

1. CUDA-based k-Means algorithm on MapReduce Approach:

Our k-means algorithm on MapReduce can be divided into two sections, data partitioning and centroid calculation. For data partitioning, we chunk the input files of data points information and store them as a point_t array. The size of the array is equal to the number of data points and each array element contains a point's coordinate information and its cluster information. In the initialization stage, all the parameters of a point except the coordination value would be set to zero. In the centroid calculation stage, we share the centroid among all the threads since MapReduce with CUDA is a shared-memory model. To share the centroids information, an array of type centroid_t can be created in the main function to include the initial K centroids and the updated centroids in each iteration.

Algorithm 2. K-Means Algorithm on MapReduce

1. Randomly initialize k centroids
2. Repeat MapReduce parallel execution of k-means
   a. Map Phase:
      i. Chunk data points based on the thread number
      ii. Compute the nearest centroid and assign the point to it.
   b. Combiner Phase:
      i. Use exclusive scan and prefix sum to only store the points whose assigned cluster is changed.
   c. Reduce Phase:
      i. Recalculate the centroids by iterating over the intermediate centroid lists.
3. Until convergence, the centroid values do not change

The Algorithm. 2 illustrates the pseudocode of k-Means algorithm on MapReduce. Compared with Algorithm. 1, three phases are added. In the map phase, the mapper gets the data input and gets the centroids from either the last iteration or the initial iteration. The data points are chunked into map function (on GPU) using data parallel strategy. Each thread will take a couple of data points based on the blocks number and the thread number per block. Then, the thread will find the nearest centroid and assign the point to it. In the combiner phase, we would use a combiner to minimize the size of data before sending it to the reducer. In this way, we can minimize the number of points to be processed by the reduce phase. It would output the cluster id, number of updated instances in each cluster.

2. CUDA-based k-Means algorithm using Triangle Inequality

While applying the MapReduce results in some improvements and speedup of the labeling stage, we found that the standard algorithm may perform unnecessary distance calculations. This issue can become problematic when there is a large number of clusters we need to categorize. For this reason, the second optimization we decide to make is applying triangle inequality as a reinforcement. This idea is inspired by the Xueyi Wang who has proposed a new algorithm kMkNN for the nearest neighbor searching problem and considered implementation of K-means clustering with triangle inequality.

Algorithm 3. K-Means Algorithm on MapReduce using Triangle Inequality

1. Randomly initialize k centroids
2. Repeat parallel execution of k-means
   a. Calculate inter-centroid distances matrix
   b. Sort each row of the ICD matrix to get the ranked index RID matrix in ascending order
   c. Copy centroid lists, ICD, and RID matrix to GPU device
   d. Launch GPU to label data points to the nearest centroid using ICD, RID, and the support of triangle inequality.
   e. Copy updated memberships back to host.
   f. Calculate the mean for each cluster and update the centroid lists.
3. Until Convergence.

To summarize, we add two steps to the previous k-means algorithm. The first step we added is calculating the inter-centroid distances. The ICD matrix is a k × k matrix storing the distance between two centroid points. The distance between every two centroid points is calculated using Euclidean distance calculations. The second step we added is sorting the inter-centroid distances and storing the results into the RID matrix. The triangle inequality is performed as follows: for a data point P, instead of looping through the entire centroid list to find the nearest cluster, the labeling step will loop through the centroids following the sequence stored in a row of RID, the row index is determined by the previous cluster assignment. The loop will terminate when we find the first centroid j, which the distance between the centroid I and j is larger than the two times distance between the Point P and its previous centroid I, i.e., d (Cj, Ci) > 2d (P, Ci).

On the GPU side, the labeling work of n data points are distributed to a total number of GridSize× BlockSize threads, and each thread will compute the membership update using triangle inequalities for a subset of data points. The BlockSize is set to 128 threads, and the GridSize is calculated based on the total number of data points.

3. CUDA-based k-Means algorithm using Dynamic Load Assignment

After the implementation of the triangle inequality, we find that this approach lowers our speedup and hurts the performance. The main reason is that even though we reduce the number of unnecessary distance calculations, it introduces a new set of problems. Now, the GPU threads are more likely to operate on diverged execution paths and follow some irregular memory access patterns. For instance, suppose we have 100 data points in total, with 10 data points need 20 calculations and the rest of data points need 1 calculation. In such case, the earlier finished threads have to be stalled and executed in the idle state until all the threads are complete. To solve this computation loss due to path divergence, we proposed a dynamic load assignment approach in hope of having better load balance and memory locality.

Algorithm 4. K-Means Algorithm on MapReduce using Triangle Inequality and Load Balance

1. Randomly initialize k centroids
2. Repeat
   a. Algorithm 3 and compute the number of distance calculations of each point.
3. Until the change of the average of distance calculations is less than 1%.
4. Sort the Point_t array to make sure the distance calculations of each point is in descending order.
5. Copy the updated point_t data points to GPU device
6. Repeat parallel execution of k-means
   a. Calculate inter-centroid distances matrix
   b. Sort each row of the ICD matrix to get the ranked index RID matrix in ascending order
   c. Copy centroid lists, ICD, and RID matrix to GPU device
   d. Launch GPU to label data points to the nearest centroid using ICD, RID, and the support of triangle inequality.
   e. Copy updated memberships back to host.
   f. Calculate the mean for each cluster and update the centroid lists.
7. Until Convergence.

The detailed implementation shown in Algorithm. 4 is inspired by Jiadong Wu's paper that published in IEEE international parallel & distributed processing symposium []. The basic idea is to solve the problem of load imbalance and memory locality introduced from path divergence, we would compute the workload of each thread dynamically and process data points in the decreasing order of their workloads. The approach can ensure the data layout in GPU's global memory matches the thread's processing order. As a result, the memory locality can be improved.

Results:

During our evaluations, we conducted extensive experiments on both sequential and the parallel implementations of the k-means algorithm. In these experiments, we ran the tests of algorithms with C/CUDA on NVIDIA 1080 GPU using GHC machines. The dataset as we introduced in the background section are synthetically created for the experiments. All attributes for each instance in the dataset are an integer value using 2-dimensional coordinates.

For the sequential implementation, the whole algorithm runs as a single thread on a single processor. For the parallel implementation, the maximum number of threads can be set by CUDA kernel is 1024 threads per block. For the tests we performed, thread number = 128 is used.

In this section, we would first analyze the performance of our algorithms for varying data size. The experiment is conducted for datasets of size 3000, 5250, and 7500 instances using the A-sets. These data sets have increasing number of clusters (k). There are 150 vectors per cluster. The performance is summarized in the table below (in appendix). Each row represents a dataset we ran the tests on. For each dataset, we perform the tests using k = 10, 20, 35, and 50.

Moreover, we break the total computation time into distinct execution time of each component. From top to bottom, i stands for the time to initialize. C stands for the total computation time, Fn is the time takes to update the membership, Fc stands for the time to recompute the centroid list. In the parallel algorithms, we use Copy to represent the time takes for data transformation between host and device. For the algorithm with load balance, we also record the iteration of both epochs.

Figure. 2 shows the performance of different algorithms with respect to the data size. All the computations are performed using k= 20 as the input parameters. The speedup is computed based on the total computation time. Our results show that when the size of dataset increases the execution time also increases proportionally. For the mapReduce algorithm, when the total number of instances is less than 5000, the parallel algorithm perform is almost constant. When the total number of instances is greater than 5000, the execution time start to increase linearly for all the three parallel algorithms, and the slope of mapReduce function is the greatest. Figure 3 presents the speedup of our parallel algorithms with respect to data size. It shows that the mapReduce algorithm is experiencing a significant drop of speedup when data set. The algorithm with triangle inequality and the algorithm with load balance increase about linearly as data size becomes larger. The reason that the mapReduce algorithm follows such a pattern is that as the data size becomes larger, all the threads in mapReduce become saturated, and there will be an increasing number of threads need to wait before the computation of instances getting processed. Following the trends in the speedup graph, we can conclude that the parallel k-means clustering can get greater speedup benefits as the number of data size becomes larger or the number of clusters becomes larger.

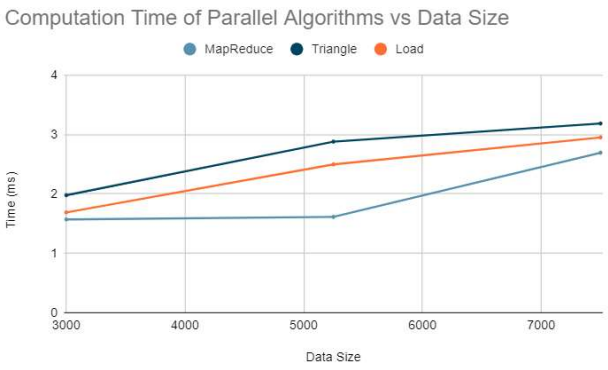

Figure 2 Computation time of k-means algorithm

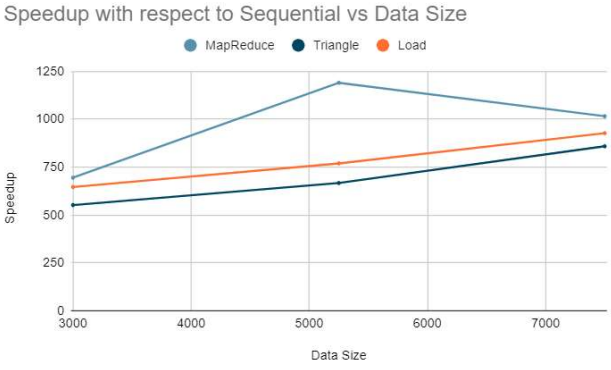

Figure 3. Speedup of k-means algorithm with respect to data size with respect to data size

The reason that the MapReduce algorithm has a better speedup than the others is because the MapReduce algorithm is executed based on same paths. The time take for all the threads compute the distance to the nearest centroid and update its centroid list is about same. The algorithm with triangle inequality has the minimum speedup among all the three proposed parallel algorithms. This is because we are experiencing an execution path divergence when we use the triangle inequalities to skip unnecessary distance calculations. In this algorithm, the

divergence is caused by imbalanced workload. Even though a thread does not need an execution path, it will still be stalled and run in an idle state, which results in a significant waste of resources.

Figure 4 and Figure 5 show the performance of parallel algorithms for varying cluster sizes. This experiment is conducted for cluster size 10, 20, 35, 50. The graph is generated using the A3 dataset with 7500 instances. It shows that the performance of the parallel k-means algorithm also depends on the number of clusters. In addition, compared with the Figure 2, we can conclude that the execution time increase rate is lesser than that of varying number of instances. While the MapReduce algorithm increases linearly as the cluster size becomes larger, the speedups of the other two algorithms are saturated as the cluster size is larger than 30. This is because the triangle inequality and dynamic load balanced algorithms result additional overheads to the computation. As the cluster size is larger, the overheads can be more significant impacts of the algorithm performance. For instance, the computation time takes to generate the ICD and RID matrix and sort data points will increase proportionally and compromise the benefits we can get from skipping unnecessary calculations.

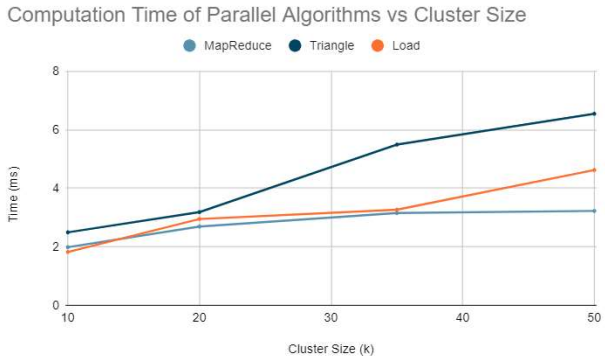

Figure 4 Computation time of k-means algorithm to cluster size

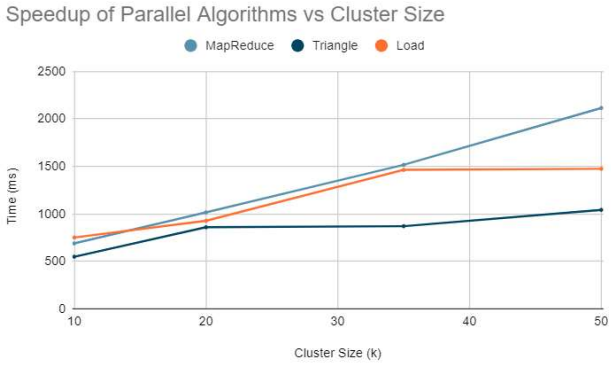

Figure 5. Speedup of k-means algorithm with respect to with respect cluster size

Figure 6 and Figure 7 show the computation time of different components of each algorithm. By comparing the percentage of execution time of each algorithm, we can find that our k-means algorithm 4 with dynamic load balance shows the most balanced time distribution between the data transformation stage and the labeling stage. As the figure shows, it takes 47% computation on data transformation and 43% on the labeling step. This improvement is mainly achieved by calculating the workload (number of distance calculations) dynamically and assign tasks to each thread based on the workload distribution in descending order. This modification helps us stabilize workload distribution after first few iterations. After the converging behavior of workload, we can rearrange the data points to guarantee optimal workload balancing.

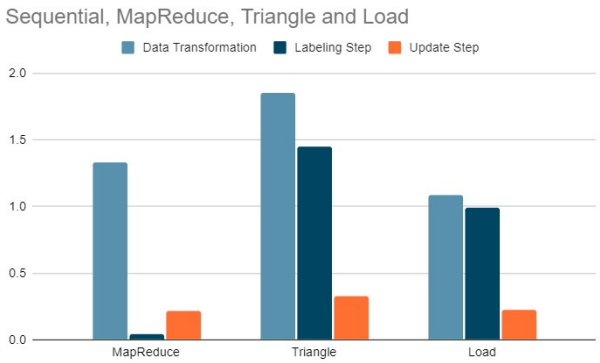

Figure 6 Computation time of k-means algorithm components of each type.

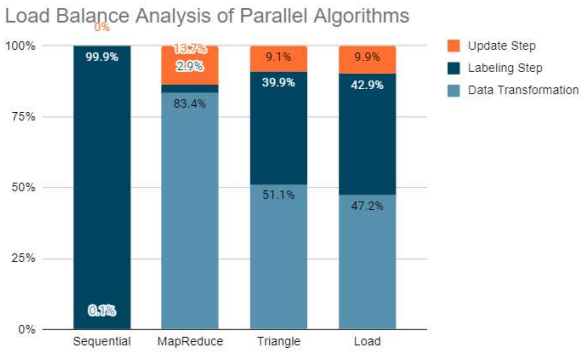

Figure 7. Percentage of k-means algorithm with respect to each type.

To summarize, the computational bottleneck of this k-means clustering algorithm is at step 2, the labeling step, where each thread computes a distance for all instances assigned to itself and all cluster centroids. For a k number of cluster, the total computation cost of computing Euclidean distance will be 2k + c, where the first term shows the cost of calculating distance and the second term represents the cost of finding the closest centroid.

Appendix:

		Sequential	MapReduce	Triangle	Load
A-sets 1 Iteration speedup	K = 5, 10, 20, 35, 50	K:10 I=1.918 C= 544.748819 Fn = 542.957 Fc = 0.5839 K20 I=2.33 Fn=1089.198 Fc = 0.6559 C = 1091.083 K35 I = 2.2321 Fn = 1906.54 Fc = 0.75588 C = 1908.531 K50 I= 2.087	K:10 I: 0.9566 Fn 0.04 Fc 0.184 C = 1.25 K20 I 1.219 Fn 0.0474 Fc 0.233 C 1.57 K35 1.414 0.0494 0.15 1.686 K50 1.5527	K10 Copy 0.408 0.735 0.15 1.78 K20: 0.6 0.784 0.139 1.977 K35 0.971 0.681 0.098 2.099 K50 1.940	K10: Iters 16, 4, 20 0.238 0.574 0.121 1.46 K20: 22, 1, 23 0.36 0698 0.115 1.688 K35 15, 2, 17 0.687 0.57 0.081

		Fn = 2734.976 Fc = 0.862 C = 2737.0611	0.053961 0.159 1.835ms	0.853 0.1 3.25	1.766 K50 11, 3, 14 1.206 0.586 0.065 2.243
A2 5250		10: I= 2.77 Fn=956.43 Fc = 0.58 C = 958.21 20: I =13.729 Fn =1919.158 Fc = 0.67 C=1921ms 35: 2.258 3344.5 0.799 3346.53 50: 2.68 4781.128 0.8966 4783.246	10: I 1.1138 Fn 0.043519 Fc 0.218 C 1.445 20: 1.2485 0.05 0.2455 1.613 K35 1.33 0.045853 0.2182 1.665 K50 1.688 0.0595 0.28 2.099	K10 0.411 0.873 0.254 2.019 20: 0.806 1.149 0.328 2.879 35: 1.855 1.447 0.329 4.239 K50: 2.76 1.103 0.245 4.578	K10 20, 4 0.244 0.707 0.203 1.76 K20 14, 16, 30 0.474 1.068 0.253 2.496 K35 12, 14, 28 1.089 0.988 0.228 2.937 K50 18, 1, 19 1.636 0.746 0.157 3.067
A3 7500		10: 13.935 1366.8 0.581 1368.6 20: 3.022 2734.21	10: 1.49 0.0528 0.37 1.986 20: 2.033 0.08	K10 0.437 1.135 0.396 2.494ms K20 0.803 1.29	K10 17, 4, 21 0.214 0.73 0.251 1.822 K20. 33, 1, 34

		0.664 2736.11 35: 3.02812 4778.2 0.776 4780.265 50: 2.976 6817.327 0.8973 6819.433	0.509 2.693 35: 2.467 0.084 0.532 3.153 50: 2.5527 0.0826 0.5187 3.224	0.479 3.185 K35 2.396 1.719 0.603 5.492 K50 3.925 1.459 0.509 6.546	0.537 1.176 0.406 2.95 K35 10, 18, 28 1.183 1.001 0.343 3.266 K50 6, 22, 28 2.499 1.036 0.338 4.623

References:

1. Fränti, P., Sieranoja, S. K-means properties on six clustering benchmark datasets. *Appl Intell* **48,** 4743–4759 (2018). https://doi.org/10.1007/s10489-018-1238-7
2. Kumar, R. Sinha, V. Bhattacherjee, D. S. Verma, S. Singh, "modeling using K-means clustering algorithm", IEEE 2012, 1 st international conference on recent advances in information technology(RAIT).
3. B. K. Mishra, N. R. Nayak, A. Rath, S. Swain, "far efficient K-means clustering algorithm", Proceedings of the 2012 ACM's International Conference on Advances in Computing, Communications and Informatics
4. D. Napoleon, P.G. Lakshmi, "An efficient K-Means clustering algorithm for reducing time complexity using uniform distribution data points, " Trendz in Information Sciences & Computing (TISC), IEEE 2010, vol., no., pp.42, 45, 17-19 Dec. 2010.
5. E. Kijsipongse, S. U-ruekolan, "Dynamic load balancing on GPU clusters for large-scale K-Means clustering, " 2012 IEEE International Joint Conference on Computer Science and Software Engineering (JCSSE), vol., no., pp.346, 350, May 30 2012-June 1 2012.
6. H. Xiuchang, SU Wei, "An Improved K-means Clustering Algorithm", JOURNAL OF NETWORKS, VOL. 9, NO. 1, JANUARY 2014.
7. C. N. Vasconcelos A. SÁ P. C. Carvalho and M. Gattass "Lloyd's algorithm on GPU" ISVC '08: Proceedings of the 4th International Symposium on Advances in Visual Computing. pp. 953-964 2008.
8. M. Zechner and M. Granitzer "Accelerating k-means on the graphics processor via CUDA" pp. 7-15 apr. 2009.

9. Lin Wang, Bo Yang, Yuehui Chen, Zhenxiang Chen, Hongwei Sun, "Accelerating FCM neural network classifier using graphics processing units with CUDA", *Applied Intelligence*, vol. 40, pp. 143, 2014.
10. K. Alsabti, S. Ranka, V. Singh, "An efficient k-means clustering algorithm", 1998 Proceedings of IPPS/SPDP Workshop on High Performance Data Mining.(date location).
11. E. Kijsipongse, S. U-ruekolan, "Dynamic load balancing on GPU clusters for large-scale K-Means clustering, " 2012 IEEE International Joint Conference on Computer Science and Software Engineering (JCSSE), vol., no., pp.346, 350, May 30 2012-June 1 2012.
12. J. Feng; Z. Lu; P. Yang; X. Xu, "A K-means clustering algorithm based on the maximum triangle rule, " 2012 IEEE International Conference on Mechatronics and Automation (ICMA), vol., no., pp.1456, 1461, 5-8 Aug. 2012.
13. X. Wang, "A fast exact k-nearest neighbors algorithm for high dimensional search using k-means clustering and triangle inequality", The 2011 International Joint Conference on Neural Networks (IJCNN),, vol., no., pp.1293, 1299, July 31 2011-Aug. 5 2011.

Distribution of Total Credit:

100%