Volume 5, Issue 2, February 2015 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

K-Means for Parallel Architectures

Rahul Ghudji, B Padmavathi
Department of Computer Sciense,
G.H.Raisoni College of Engineering
Wagholi, Pune, Maharashtra, India

Abstract— Data Mining is the field where useful information is retrieved from raw data. Tremendous data is generated daily due to social networking sites or online internet data. Processing of this huge data is tedious task. Various shared and distributed architectures are used to process data. We used combination of CPU and GPU to achieve significant speedup for K-Means algorithm. GPU (Graphics Processing Unit) is shared memory low cost architecture used to run parallel applications with CUDA (Compute Unified Device Architecture). Implementation of K-Means on hybrid architecture is more energy efficient and cost effective.

Keywords— Clustering, K-Means, GPU, CUDA, Data Mining, Hybrid architecture, Hybrid programming

I. INTRODUCTION

Graphics processor unit (GPUs) becomes attractive because they offer extensive resources like massive parallelism and high memory bandwidth even for general-purpose computations, including support for both single- and double-precision IEEE floating point arithmetic. In fact, GPU contain hundreds of processing elements in "manycore" processor architecture. Some applications are preferred to use general purpose computing on GPUs over conventional multicore CPUs. As GPUs and multicore CPUs emerges as advanced parallel computing platforms have come to dominate the market [7]. It's important to revisit parallel programming models and find the best balance between programming model according to convenience and hardware implementation cost. While performance is key factor which gives inspiration to our work, efficiency is another leading driver. Data center consumes enormous amount of energy during processing task.

Data Center consists of rack servers occupied by computers with inter-node communications addition to present spatial, performance with backup electrical generation overhead. Significant energy savings can be obtained by transferring this processing from power-hungry CPUs to multiple powers efficient GPUs on a single node. CUDA technology gives speedup for computationally intensive general purpose applications with the tremendous processing power of the GPUs through a C, C++ and FORTRAN like programming interface [6].

Data mining is study to discover interesting, meaningful, and understandable information from massive raw data sets [10]. In recent years, it has become a hot research domain due to important application areas such as e-commerce, business intelligence, scientific simulation, customer relationship management, WWW, bioinformatics, and many more. However, due to architectural style of GPUs, there are certain limits in coding strategies to gain significant speedups [11].

The paper provides efficient CUDA implementation along with hybrid implementation of K-Means algorithm. Section II briefly describes different parallel programming frameworks and basic K-Means algorithm. Section III briefly describes both the implementation of K-Means on GPUs. Section IV shows how both implementations gives significant speedup with results. Finally, Section V concluded with the findings of this work and future scope for improvement.

II. BACKGROUND REVIEW

A. CUDA Programming

CUDA is an extension to simple programming languages, based on a few easily-learned abstractions for parallel programming and a few corresponding additions to programming language syntax. CUDA run in a hierarchy of grid, block and threads that can run on the coprocessor as a device with large number of threads. The grid launch as a CUDA kernel which contains thousands of thread mapped as parallel tasks in an application. The parallel threads share memory and synchronize threads at block level using synchronization API. Data is processed by GPU after coping from CPU memory to graphics board's memory with memory copy API's of CUDA. This Data transfer is asynchronous and takes place concurrently with several memory copies [1].

Once copied, data on the GPU is not change unless it is de-allocated or overwritten and available for subsequent kernels. The GPU is composed of multiple streaming multiprocessors ideally suited for massively data-parallel computations. The GPU executed multiple blocks composed of multiple threads on set of multiprocessors in parallel. CUDA works in SIMD (Single Instruction Multiple Data) i.e. threads are grouped into thread blocks and all threads execute the same instruction with different data in parallel. Within one block, threads can be synchronized at any instance of execution. Threads within a block does not execute in order.

Blocks are grouped into a grid. Within grid synchronization and communication among blocks is not possible. Blocks and threads can be organized in one, two and three dimensions. A thread is identified with an id showing its position in the

block similarly a block is also identified with id showing its position within a grid. GPU contains various kinds of memory. All GPU threads have access to these memories. Each thread has very fast private local registers and local memory. Each block contains local shared memory which shared by all the threads within block [4].

Registers, local memory and shared memory are fast but and are limited resources. Threads also access general purpose device memory or global memory. Global memory is huge but slowest memory present in GPU. Addresses in memory accessed by multiple threads in global memory simultaneously should be arranged so that memory access can be continuous called as coalesced memory access. This is often referred to as memory coalescing. Occupancy defines at execution point how many threads within blocks are actually running in parallel.

Efficient use of coalescing memory access and occupancy properties reduces execution time of an application. As shared memory and registers are limited resources, it is mandatory to allow run as many blocks and threads in parallel by optimizing the usage of shared memory and registers. The CUDA SDK (Software Development Kit) comes with tools that fully integrate with various C++ compilers. Code for the GPU is written in a subset of C, C++, and FORTRAN like languages with some extensions and can coexist in the same source file. The host code is containing configuration settings for blocks and threads executing data to the GPU. Device code is debugging in an emulation environment. Emulation environment runs the kernel on the CPU in heavyweight threads [4].

B. OpenMP Programming

Open Multi-Processing (OpenMP) is shared memory architecture which provides a multithreaded capacity to the multiprocessor CPUs. Loop can be parallelized easily by invoking OpenMP directives and subroutine calls from OpenMP thread libraries. In this way, the threads can access data independently for processing and performing assigned tasks. Threads can share data from local shared memory during iterations of loop [3].

OpenMP is a fork-join model for shared memory parallelism. The basic idea behind OpenMP is SIMD for data-shared parallel execution. It consists of a set of compiler directives and subroutines callable from runtime library routines also environment variables extending to FORTRAN, C and C++ like programming languages. OpenMP is portable and scalable across shared memory architecture. The execution unit of OpenMP is threads executed by workers. Every thread can access a variable through shared cache or RAM. The OpenMP is an API (Application Programming Interface) that supports multi-processing programming on multi-platform shared memory architecture [3].

C. K-Means Algorithm

K-Means is a most commonly used clustering algorithm in data mining. Clustering is a means of distributing n data points into k clusters where two clusters not share common one data point. Each cluster has maximal similarity between data points included in cluster. Union of all clusters contains all n points. The algorithm assigns each data point to the cluster based on Euclidean distance between data point and centroid of cluster. The centroid is the average of all the data points in the cluster [6]. The algorithm steps are:

- Choose the number of clusters, k.
- Randomly generate k clusters and determine the cluster centroids, or directly generate k random points as cluster centroids.
- Assign each point to the nearest cluster centroid.
- Re-compute the new cluster centroid.
- Repeat the two previous steps until some convergence criterion is met.

The result of this algorithm changes with every run. Results depend on initial random assignments of cluster centroids. Usually the convergence criteria is that assignments has not change between clusters in a iteration. If the initial cluster assignments are chosen heuristically around the final point then one can expect convergence to correct values. The first and second steps of the algorithm take Θ (k) time, while the third step takes Θ (nk) to complete. Finally, the fourth step takes order of Θ (n + k) execution time. In a typical application n > k, so the execution time of the algorithm is bound by the third step. This observation guided us to concentrate our parallelization efforts on the third step of the k-means algorithm [6].

III. PARALLEL IMPLEMENTATION

A. CUDA Implementation

K-means CUDA implementation has the following four stages:

K-means CUDA implementation has the following four stages:

1. Compute new centroids (except at first iteration)
2. Assign data points to clusters
3. (Optional) Sort data points according to assignments by:
   - **a.** A complete sorting step if more than half of all data points are unsorted, or
   - **b.** Updating a previous sorting operation
4. Compute score, or break at maximum iteration

The functions have computational complexity O (K*D*N) (stage 2), and O ((K+D)*N) (stages 1 and 4). Then, optional sorting or updating in stage 3 reduces this to O (D*N) for stages 1 and 4. Also, sorting stage fulfills occupancy property which ensures that all threads within a warp are equally occupied, gives 32-fold speedup when compared to processing unsorted data. We have summarized our K-Means CUDA implementation strategy as a flowchart in Fig. 1 [2].

Centroids computed by averaging all data points assigned to each cluster. Before sorting all assignment of N data points in all K clusters have to be checked. D is the number of attributes for every data point has to add it to the average. K blocks of TPB (Thread per Block) threads are assigned as K centroids of clusters. One data point has processed by one thread in consecutive segments of TPB size. Moving along the array, all the data points are covered in (N/TPB) iterations. Such a way each cluster used the number of threads is larger by a factor of TPB and for small K a high level of parallelism is achieved. Data points are stored in column-major order and therefore transfers of data point coordinates of TPB data points from global to shared memory are coalesced.

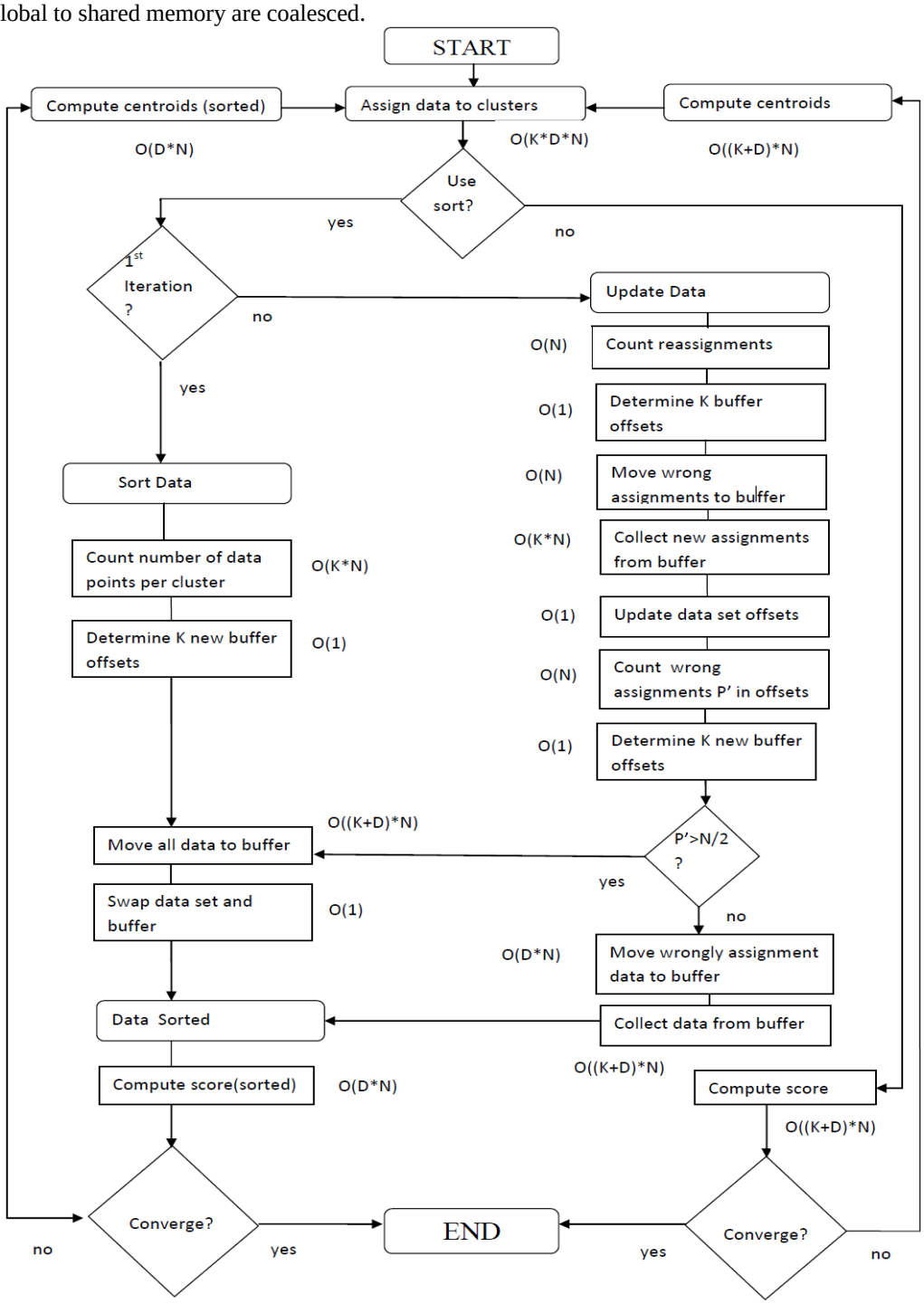

Fig. 1 Flowchart of K-Means CUDA implementation

Assigning all data points to K clusters requires checking each data point with all K centroids. This is done in (N/TPB) iterations. Centroids are stored in row major order, first min (TPB, D) coordinates of the first centroid are copied from global to shared memory. Hence, these reads are coalesced. For each dimension the distance is calculated between a data point and the centroid by Euclidean distance method. Upon completion each thread keeps track of the closest centroid. The TPB threads write the final assignments to membership array in global memory (coalesced) [9].

The score is convergence criteria computed by summing all distances between data points and their respective centroids. K blocks of threads are used. During the score computation, we do not keep track of number of elements, but compute distances between centroids and their respective assigned data points. These distances are computed with

Euclidean distance as in the assignment stage and summed up in TPB shared memory. One partial score per block is stored to global memory. Those K values are summed up by the host with a negligible overhead. With this implementation any number of data points with any number of dimensions can be processed with fully coalesced accesses [5].

In sorting data set represented as a key-value pairs by cluster assignment and a D-dimensional vector. Sorting is just placing N data points into K bins in O (D*N) time and O (D*N) space. At first count the assignments by traversing the membership array in global memory with K blocks of TPB threads each. Each of the K blocks checks for all assignments in K cluster. Each block initializes shared-memory integer array with zero of size TPB. Each thread reads one membership from global memory and increments shared memory counter by one if that data point present in that cluster K. After traversal we got the total number of element present in that cluster K which gives us the segment size that we have to reserve in the global memory for storing sorted array of data points for that cluster [2].

The next step is determining K buffer offsets with K segment sizes, which we store in global memory. The CPU determines offsets of each block for sorted data within the buffer array, by storing intermediate result of a running sum. This step takes order of O (K*N) time. These offsets are used as input for K thread blocks. In next step all the wrong assignments are moved to buffer. Finally all threads within the block save with single coalesced global memory.

An update step is more efficient than sorting step when number of reassignments between clusters is dropped i.e. significant portion of the data remains sorted. Updating step used in place of sorting, as very few data has to be moved during updating. At first we have to count the reassigned data points for each cluster, i.e., data points for which the new cluster is not equal to old cluster. Segment sizes of this cluster for the data are updated by subtracting this reassigned count. This is done by K blocks of threads of TPB threads each in (SK/TPB) iterations, where SK is the segment size of cluster K.

After counting reassigned data, we know the size of K segments need to be reserved for temporarily store those data points. K offsets for the buffer segments are determined by updating segment sizes in the data array. It is sufficient to move the misassigned data points to the buffer, instead of moving all the data points. K blocks have starting position of its buffer offset and traverses with TPB threads to its assigned data segment. During collecting new assignments from buffer all K blocks of threads traverse in the buffer and increases data segment sizes by the number of reassigned data points to them. Hence we obtain segment sizes for the new, sorted data array. Next step is to update data set offsets by the final segment offsets of the data points for the current iteration is computed. The new segment boundaries are updated relative to the old ones and therefore previously occupied data from other clusters may be overlapping with segments. This overlap region may contain wrongly assigned data. We have to traverse membership array to find wrongly assigned data points in new data segments also compute final updated buffer segments. Count wrong assignments P` in offsets is done with new data offsets using the same operation as for counting reassignments [2].

K new buffer offsets are determined from counted reassignment. These new buffer segments are larger than those computed in previously because they may contain data from adjacent segments in addition of misassigned data points. The sum of the P` shows data points have to be moved in total. Updating requires moving misassigned data from the data array to the buffer and back to the correct position in the data array.

If full resort is triggered, program returns from updating function and using already determined cluster offsets calls resort to the buffer as shown previously. Move wrongly assigned data to buffer. If full resort was not triggered, data is copied from the data array to the buffer in the same way that the assignments were compacted. Data points that have been copied may leave gaps in the data array. Collect data from buffer takes largest time complexity, O ((K+D)*N), but for a small data set, all K blocks of threads run through the P' data points in buffer and collect those that are assigned to it. The number of data points in buffer assigned to cluster K equals to the gaps in the data segment for cluster K. To collect data from the buffer and store it in gaps, match the locations of gaps with the locations of data points in the buffer.

B. Hybrid Implementation

The OpenMP-CUDA programming model is used for heterogeneous hybrid architecture. In general, the program shows combined parallelism of both OpenMP and CUDA at various stages of K-Means implementation. The serial part of the program is executed by the master thread on the host (CPU). Then, various worker threads launch GPU kernels and do the remaining parallel part of the program. Specifically, multiple CPU worker threads are created by an OpenMP instruction, and each worker thread launch one CUDA kernel [3].

When a kernel is launched, a large number of hardware threads i.e. GPU threads are generated to exploit data parallelism. Those threads are one-dimensional entity used to access data point attribute arrays with each one-dimension block consist of TPB threads. All threads in a block generated by the kernel will execute the same instructions as it is SIMD architecture. After every kernel finishes its parallel task, the CPU will collect final results do the remaining serial work. GPUs specific hierarchical architecture and memory bandwidth gives significant computational efficiency. The lifetime of variables in CUDA for the global memory is the entire application unless the programmer freed it. GPUs are normally well-suited for the application where massive data parallelism is required and it is weak at executing logical instructions. The CPU code also parallelized with OpenMP where huge data parallelism does not require [3].

IV. EXPERIMENTAL RESULTS

Experiments are performed intel xeon machine with Tesla C1060 gpu card. The synthetic data is created by uniform synthetic data generator on CentOS 5.9. CUDA 5.5 is used along with gcc 4.2 for compilation and computation purpose. The performance behavior is studied depending on change in number of clusters, dimensions and data point. The performance of K-Means is also recorded for various parallel implementations.

A. Change in dimension

For change in dimension D, the speedup over the CPU reference version for 100,000 data points and 90 clusters remains at around 15 fold for values of D between 1 and 300. Table 5.6 shows speedup varies with change in number of dimensions, it goes upto 90 fold speedup for D = 600, 900 and 1200. This speedup is maintained for larger numbers of dimensions.

Fig. 2 shows graphical representation of Table 1 for change in number of dimensions. Speedup remains constant on 10 fold for dimensions 1 to 300. It gradually increases upto 90 fold for 300 to 600 dimensions and remains constant for larger numbers of dimensions.

TABLE 1 PERFORMANCE FOR CHANGE IN DIMENSIONS

DATA SET	NO. OF DIMENSIONS (D)	TIME (SEC)		SPEEDUP
		CPU	HYBRID	
N=100K K=90	1	34.84	3.74	9.84x
	300	65.043	5.94	10.95x
	600	648.540	7.24	89.64x
	900	1115.50	12.63	88.38x
	1200	1316.78	14.56	90.56x

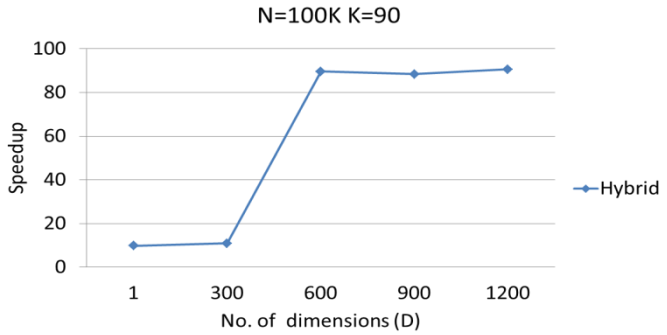

Fig. 1 Chart for change in dimension

B. Change in clusters

For variable clusters K, Table 5.7 shows speedup variation with N = 100,000 and D = 1000, the 45 fold speedup roughly from K = 5 to K = 10 and again from K = 10 to K = 100 peaking at about a 100-fold speedup. This means that peak performance is achievable for a number of clusters well at 100.

Fig.3 shows speedup performance for change in number of clusters, it remains stable after cluster increases beyond 100.

TABLE 2 PERFORMANCE FOR CHANGE IN CLUSTERS

DATA SET	NO. OF CLUSTERS (K)	TIME (SEC)		SPEEDUP
		CPU	HYBRID	
N=100K D=1000	5	13.80	5.63	2.45x
	10	231.984	5.11	45.50x
	100	1776.608	18.82	94.40x
	1000	10522.548	113.61	92.62x
	10000	40154.3010	419.80	95.65x

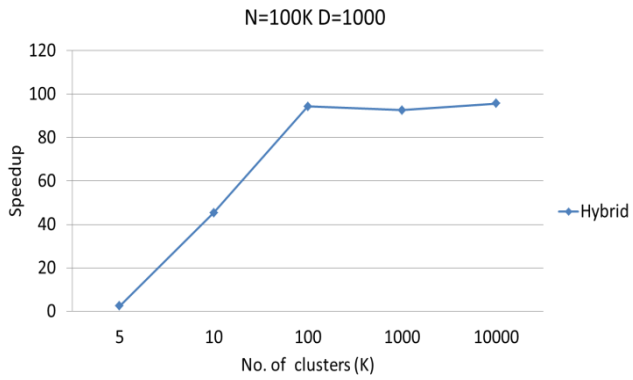

Fig. 2 Chart for change in clusters

C. Change in data point

For the change in number of data points, with K = 90 and D=1000, Table 5.8 shows rapid increase in performance from roughly 1 fold to 40 fold is observed between N = 1,000 and N=10,000, before the speedup increases more slowly from N=10,000 to N = 100,000, where it peaks at about 200 fold. For N = 150,000, a decrease to 170-fold speedup is observed.

Fig. 4 shows variation in speedup for change in number of data points, it reaches to the peak value at 100,000 and decrease upto 70 fold for 150,000.

TABLE 3 PERFORMANCE FOR CHANGE IN DATA POINTS

DATA SET	NO. OF DATA POINTS (N)	TIME (SEC)		SPEEDUP
		CPU	HYBRID	
D=1000 K=90	1000	2.10	3.3530	0x
	10000	147.61	3.42	43.17x
	50000	370.72	6.42	57.91x
	100000	1672.20	17.87	93.58x
	150000	1994.62	28.56	69.48x

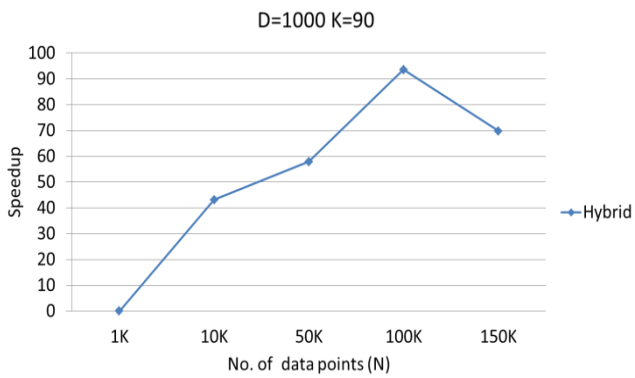

Fig. 3 Chart for change in data points

D. Comparison for various implementations

The proposed work mainly shows four implementations i.e. sequential, OpenMP, CUDA and hybrid of K-Means algorithm on various architectures. Table 4 shows performance analysis of these four implementations with different data sets of different parameters. These are synthetic data sets generated by synthetic data generator. These results are calculated with four different data sets. First data set consists of 50K data object with 50 dimensions distributed in 50 clusters. Second data set is having 50K data object with 1000 dimensions and distributed in 50 clusters. Third data set shows 100K data object with 1200 dimensions and 90 clusters. Fourth consists of 150K data object with 1000 dimensions and 1000 clusters.

The result shows that hybrid execution is the fastest execution while sequential is the slowest execution. This result shows significant time difference between all the execution times.

TABLE 4 COMPARISONS OF VARIOUS IMPLEMENTATIONS

DATA SET	TIME (SEC)			
	SEQUENTIAL	OPENMP	CUDA	HYBRID
N=50K D=50 K=50	15.864	3.8609	3.3541	3.3202
N=50K D=1000 K=50	78.22771	19.1306	5.0995	4.7977
N=100K D=1200 K=90	335.3584	81.0008	16.0543	14.5270
N=150K D=1000 K=1000	10522.548	1478.3951	114.8749	113.6127

V. CONCLUSIONS

We have implemented various versions of K-means on CPU, GPU and hybrid architectures. These are efficient regardless of the size and dimensionality of the input data set. As the number of cluster increase it gives better performance compared to the increase in dimensions and data object. Our implementation wok effectively for energy efficient and low cost system enabled with GPU card for high speedup. We have efficiently used all the key feature of CUDA to gain significant performance from CUDA implementation. We also get speedup from the combination of CUDA and OpenMP i.e. hybrid compared to the sequential execution.

ACKNOWLEDGMENT

We express our sincere thanks to the all authors, whose papers in the area of parallel data mining, K-Means algorithm and GPGPU published in various conference proceedings and journals.

REFERENCES

[1] Liheng Jian, ChengWang, Ying Liu, Shenshen Liang,Weidong Yi, Yong Shi, "Parallel data mining techniques on Graphics Processing Unit with Compute Unified Device Architecture(CUDA) ", The Journal of Supercomputing Springer , Volume 64, Issue 3, June 2013.

[2] Kai J. Kohlhoff, Vijay S. Pande, and Russ B. Altman,"KMeans for Parallel Architectures Using All-Prefix-Sum Sorting and Updating Steps", IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 24, NO. 8, AUGUST 2013.

[3] C. Yang, C. Huang and C. Lin,"Hybrid CUDA, OpenMP and MPI parallel peogramming on multicore GPU," ELSEVIER transaction in Computer Physics Communications 182 (2011) 266-269.

[4] R. Wu, B. Zhang and M. Hsu, "Clustering Billions of Data Points Using GPUs," Proc. Combined Workshops Unconventional High Performance Computing Workshop Plus Memory Access Workshop (UCHPC-MAW '09), 2009, doi: 10.1145/1531666.1531668.

[5] M. Zechner and M. Granitzer, "Accelerating K-Means on the Graphics Processor via CUDA," Proc. First Int'l Conf. Intensive Applications and Services (INTENSIVE '09), pp. 7-15 , 2009, doi: 10.1109/INTENSIVE.2009.19.

[6] S.A.A. Shalom, M. Dash, and M. Tue, "Efficient K-Means Clustering Using Accelerated Graphics Processors," Proc. 10th Int'l Conf. Data Warehousing and Knowledge Discovery I. Song, J. Eder, and T. Nguyen, eds., pp. 166-175, 2008, doi: 10.1007/978-3-540-85836-2_16.

[7] R. Farivar, D. Rebolledo, E. Chan, and R. Campbell, "A Parallel Implementation of K-Means Clustering on GPUs," Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications, 2008.

[8] S. Che, J. Meng, J.W. Sheaffer, and K. Skadron, "A Performance Study of General Purpose Applications on Graphics Processors," Proc. First Workshop General Purpose Processing on Graphics Processing Units, 2007.

[9] H. Bai, L. He, D. Ouyang, Z. Li, and H. Li, "K-Means on Commodity GPUs with CUDA," Proc. WRI World Congress Computer Science and Information Eng., vol. 3, pp. 651-655, 2009, doi:10.1109/CSIE.2009.491.

[10] Wu R, Zhang B and Hsu MC,"Cl W.D. Hillis and G.L. Steele Jr., "Data Parallel Algorithms," Comm. ACM, vol. 29, no. 12, pp. 1170-1183, Dec. 1986, doi:10.1145/7902.7903.

[11] You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu,"Speeding up K-Means Algorithm by GPUs".

