Accelerating K-Means Clustering with Parallel Implementations and GPU computing

Janki Bhimani
Electrical and
Computer Engineering Dept.
Northeastern University
Boston, MA
Email: bhimani@ece.neu.edu

Miriam Leeser
Electrical and
Computer Engineering Dept.
Northeastern University
Boston, MA
Email: mel@coe.neu.edu

Ningfang Mi
Electrical and
Computer Engineering Dept.
Northeastern University
Boston, MA
Email: ningfang@ece.neu.edu

***Abstract*—K-Means clustering is a popular unsupervised machine learning method which has been used in diverse applications including image processing, information retrieval, social sciences and weather forecasting. However, clustering is computationally expensive especially when applied to large datasets. In this paper, we explore accelerating the performance of K-means clustering using three approaches: 1) shared memory using OpenMP, 2) distributed memory with message passing (MPI), and 3) heterogeneous computing with NVIDIA Graphics Processing Units (GPUs) programmed with CUDA-C. While others have looked at accelerating K-means clustering, this is the first study that compares these different approaches. In addition, K-means performance is very sensitive to the initial means chosen. We evaluate different initializations in parallel and choose the best one to use for the entire algorithm. We evaluate results on a range of images from small (300x300 pixels) to large (1164x1200 pixel). Our results show that all three parallel programming approaches give speed-up, with the best results obtained by OpenMP for smaller images and CUDA-C for larger ones. Each of these approaches gives approximately thirty times overall speed-up compared to a sequential implementation of K-means. In addition, our parallel initialization gives an additional 1.5 to 2.5 times speed-up over the accelerated parallel versions.**

I. Introduction

Clustering is the unsupervised classification of patterns such as feature vectors, observations or other data items into groups [1]. Each cluster aims to consist of objects with similar features. Clustering has broad appeal and has been used to address many contexts like feature extraction, data compression, dimension reduction and vector quantization. The notion of quality of clustering depends on the requirements of an application. There are several methods for finding clusters using techniques of neural networks, splitting-merging and distribution based clustering. Among different clustering formulations, K-Means clustering is one of the most popular centroid-based clustering algorithms due to its simplicity. We investigate parallelization of K-Means clustering on three different platforms: OpenMP, MPI and CUDA.

The three major contributions of this paper are:

- A K-Means implementation that converges based on dataset and user input.
- Comparison of different styles of parallelism using different platforms for K-Means implementation.
- Speed-up the algorithm by parallel initialization.

The rest of this paper is organized as follows. In Section II, we briefly introduce the K-Means clustering algorithm and explore the possible parallelism in it. In Section III, we explain the deployments of K-Means with OpenMP, MPI and CUDA, respectively. Section IV explains the effect of the initialization step on the K-Means clustering algorithm and deployment of an improved parallel initialization of K-Means on shared memory platforms. Section V presents results and speed-up analysis. We describe the related work in Section VI and summarize the paper in Section VII.

II. KMEANS

A. K-Means clustering

K-Means is an unsupervised machine-learning algorithm widely used for signal processing, image clustering and data mining. K-means clustering aims to partition n observations into K clusters. Each observation is assigned to the cluster with the nearest mean, with the mean value of a cluster serving as a prototype for each cluster. This results in a partitioning of the data space into Voronoi cells. The algorithm consists of three stages: initialization, computation and convergence.

Algorithm 1: K-Means clustering algorithm

```
1  Input: Dataset, K, Tolerance
2  Output: Mean table, Assignment of each datapoint to a
   cluster
3  Initialize
4  Assign data to nearest cluster
5  Calculate new means
6  if Converged then
7      Output assignment of data
8  return;
9  else
10     Continue from step 4 with new means
```

Algorithm 1 explains the basic K-Means clustering algorithm, where K is the desired number of clusters to be formed. There are a number of approaches to initialize K means, the most commonly used is random initialization.

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

The computation includes two major steps: 1) computing the distance between points and means and 2) computing the means. Euclidean is the most commonly used distance metric, but other norms can also be used according to user requirements. The mean calculation is the summation of all points belonging to a cluster divided by the total number of members belonging to that cluster. The convergence check determines when to stop iterating. This is done by keeping track of the number of points changing their clusters in a given iteration compared to the prior iteration. Tolerance is defined as the permissible range of variation in a quantity. We use tolerance to keep track of how many pixels may change clusters between the previous iteration and the current iteration in order to converge. For example, if the tolerance is 0, then we expect that no point should change its cluster compared to the previous iteration. Tolerance is computed by counting the number of points that have changed clusters in two consecutive iterations and dividing this count by the total number of points.

The three major challenges in implementing K-Means are:

- Initializing centroids
- Determining number of centroids (K)
- Determining number of iterations

The K-Means clustering algorithm has a problem of converging to a local minimum of the criterion function depending upon the selection of initial centroids. Thus, selection of proper initial centroids is important. This algorithm expects the total number of desired clusters to be formed and in some cases, the total number of iterations to be performed as inputs. Predicting K and iterations may be difficult as these may vary with different input datasets. The above discussed challenges are addressed in our implementation.

B. Parallel feature extraction

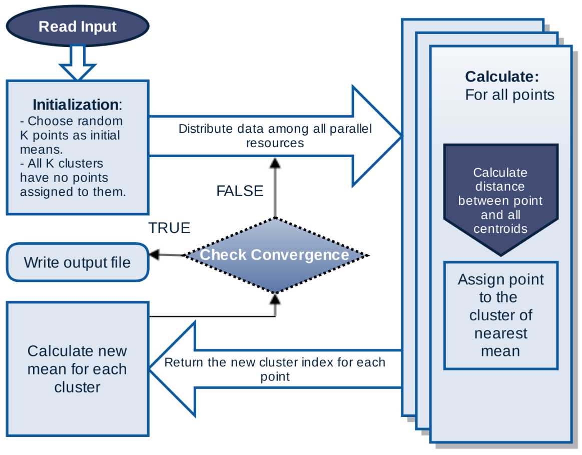
Read Input
Initialization:
- Choose random K points as initial means.
- All K clusters have no points assigned to them.
Distribute data among all parallel resources
FALSE
TRUE
Check Convergence
Write output file
Calculate new mean for each cluster
Return the new cluster index for each point
Calculate:
For all points
Calculate distance between point and all centroids
Assign point to the cluster of nearest mean

Fig. 1: K-Means parallel implementation

K-Means is an iterative method where in each iteration, calculation of the distance of each data point to all the means is independent of other data points. Thus, the computation of clusters exhibits a lot of parallelism within each iteration. The algorithm spends 85% to 90% of its execution time in computing. Thus step 4 in Algorithm 1 is an excellent place to explore parallelism.

Figure 1 gives the overview of K-Means on a parallel platform. The steps of initialization and convergence check are done sequentially while the computationally heavy step of finding the nearest mean to a point is done in parallel. The major goal is to explore different parallel implementations for K-Means without compromising its correctness.

III. Parallel implementations

We applied three forms of parallelism, including shared memory using OpenMP, distributed memory using Message Passing Interface (MPI) and NVIDIA's Graphics Processing Units (GPUs) programmed with CUDA-C. Each of these are described below.

A. OpenMP

OpenMP implements a shared memory model and a multiple thread program to spawn parallel computation. Random points are picked as initial means to start the algorithm, then the parallel region is entered. Each point is assigned to its nearest cluster by a group of threads operating in parallel, followed by a sequential step to compute new means. If the algorithm has not converged then the parallel threads are spawned again with the new means.

B. Message Passing Interface

MPI is the implementation for a distributed memory platform. The deployment is a Master-Slave model where sequential steps are performed by the master process and the slaves do the parallel work. The major difference in the distributed memory implementation from the shared memory implementation is that the master needs to perform explicit send and receive calls to distribute the data among available processes and to collect resulting assignments from the processes. In every iteration, the slaves send the index array containing the cluster assignment for each point and the master broadcasts new means to all slaves after updating means and checking for convergence. The master decides to terminate when the desired convergence is reached.

C. CUDA

This is a heterogeneous implementation consisting of CPU as host and GPU as device. The initialization is done on the host, then the host copies all the points to the GPU's global memory. The GPU performs the required calculations to form the index array containing the nearest cluster for each point in parallel. The index array is copied back to the host memory. The host then calculates new means and copies them back to the device if convergence is not reached.

IV. Improved parallel initialization

We exploit parallelism during initialization to find a good set of initial centroids. The K-Means clustering algorithm may converge to a local minimum of the criterion function depending upon the selection of initial means. The time taken

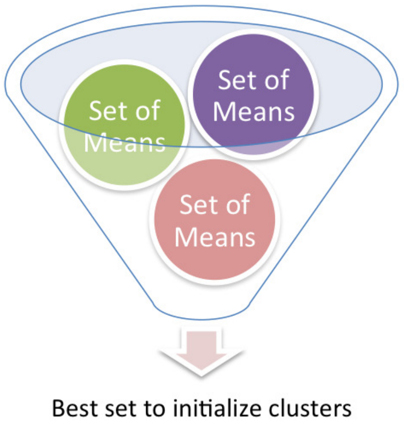

Fig. 2: Parallel initialization of Means.

by K-Means clustering and the number of iterations required depend strongly on the initialization of the means, so this step needs to be handled properly. As shown in Figure 2, for initialization of means in parallel, we pick different sets of means and give each to one OpenMP thread.

Each set of means is operated on each thread independently for 5 iterations, on a portion ($\sqrt{\#points}$) of the dataset. The quality of clustering from each set of means is compared using their respective partial results. The quality of clustering is the best for the set of means which results in the minimum intra-cluster distance and maximum inter-cluster distance. The set with better quality of clustering will converge faster and in fewer iterations. This set is selected as the initial means for all the implementations. It is not required to wait until convergence to compare, but only 4 to 5 iterations suffice, because approximately 60% of the points are assigned proper clusters in the first 4 to 5 iterations. After obtaining the best set of means, the whole dataset is divided among the available parallel resources and a copy of the best set of means is given to all these parallel resources. Once an initial set of means is computed, parallel computation proceeds as before.

The advantage of the parallel initialization method is that the possibility of getting a set of initial points which can reach a global minimum rather than a local minimum is increased. This also helps in reducing the number of iterations to converge. The disadvantage is that we are introducing an additional computational step to select the best set of random means. This trade-off between parallel and random initialization is explored in the experiments.

V. Evaluation

A. Experiments

Our clustering implementation tackles all three major challenges of K-Means as explained in Section II. It takes images as input datasets. We consider a total of five features including the three RGB channels as well as the x, y position of each pixel in the image. The x, y position is used to identify clusters that are close spatially as well as in color. We choose

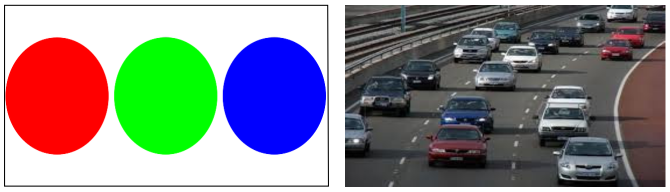

Fig. 3: Content of input image (a) Simple content (left), (b) Complex content (right).

random points to initialize each cluster center but keep them the same over all parallel implementations for comparison purposes. We also implement parallel initialization as explained in Section III for tackling the challenge of centroids initialization. We use Euclidean distance for calculation of the nearest cluster to each data point. On parallel platforms, this step is done simultaneously on multiple data points. The convergence check is done using $tolerance$ as explained in Section II. As K-Means is an iterative algorithm, the total number of iterations required for a given input dataset depends on the input size and contents. Figure 3 gives examples of simple (left) and complex (right) images. A bigger but simple image may require fewer iterations than a smaller but complex content image. Thus our implementation does not require the total number of iterations as input. Instead, the total number of iterations is decided at runtime, depending on input size, contents and the tolerance provided by the user. We use the *Drop-out* technique for estimating the proper number of clusters (K). In this technique, we initially give an upper limit of K as input. After every iteration the clusters which have no points assigned will be dropped. Experiments were performed on several sample images, with different image sizes, resolutions and complexity to evaluate the performance of our implementations.

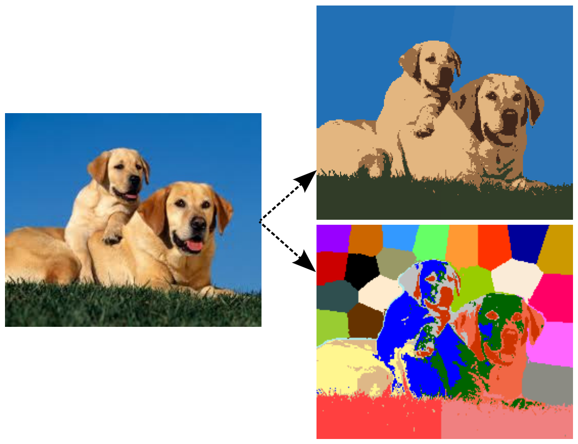

Fig. 4: Reconstructed clustered output of input image (left) with (a) Each cluster having the color of its mean(top right), (b) Cluster colors chosen to show contrast (bottom right).

To visualize the output, we create an image where each pixel is represented by its cluster. We do this in two ways:

1) Assign the *color of the centroid to each cluster*. In this

way, we obtain an output image which resembles the input image (e.g., top right image in Figure 4).

2) Assign *random colors to each cluster*. Using the first method of visualization, it is not easy to identify those clusters that have their means very close to each other. To get a better visualization of image features, we implement this second method, to observe the different clusters that K-means came up with (e.g., bottom right image in Figure 4).

All the parallel versions including OpenMP (with different number of threads), MPI (with different number of processes) and CUDA (with different block size) result in the same output image. We conclude that the quality of clustering by K-means is not dependent on the platform and implementation if the algorithm is kept the same.

B. Setup

We implement parallel versions for K-Means and compare the end-to-end execution time. In our experiments, we vary size of an image, number of clusters, tolerance and number of parallel processing units (including number of threads for OpenMP, number of processes for MPI and block dimensions in a grid for CUDA) to observe the performance of K-Means. We experiment with a range of the above mentioned features, but report here on results for two images, four different numbers of clusters depending on the input image and four different values of tolerance. We show the best speed-up obtained under the given number of parallel processing units. The images used have 300x300 pixels (1.8MB) and 1164x1200 pixels (0.5GB) where each pixel is 20 bytes consisting of five double floating point values. Evaluation was done using the Discovery cluster [2] at Northeastern University. Two types of architecture are considered in our evaluation. The compute nodes have dual Intel E5 2650 CPUs @ 2.00 GHz and 128 GByte of RAM. These nodes have 16 physical and 32 logical compute cores and a 10Gb/s TCP/IP backplane. The GPU nodes have NVIDIA Tesla K20m GPU containing 2496 computing CUDA cores and 10Gb/s TCP/IP backplane. The OpenMP experiments were performed on one of the compute nodes with different numbers of threads. The MPI experiments were performed on a group of compute nodes ranging from one to ten out of which the best performing combination was reported. We does not use the SIMD capabilities of the Intel processors. The CUDA experiments use one of the GPU nodes launched with different sizes of block dimensions as powers of two, as the GPU performs better if the block size is a power of two. It was observed that a block dimension having similar aspect ratio to that of image dimension performed best. For example, a block of 32x16 is better suited for a landscape image having more pixels in the horizontal direction when compared to the vertical direction.

C. Results

Tables I and II show the total end-to-end execution time and speed-up for each implementation with the 300x300 pixel image when clustered into 100 clusters, and the 1164x1200 pixel image when clustered into 240 clusters, respectively. In both tables, the tolerance is set to 0. We observe that all parallel versions including OpenMP, MPI and CUDA perform better than sequential C. The multi-threaded C version using OpenMP performs best with 16 threads and outperforms the rest with a speed-up of 31x for the 300x300 pixel image. The 16 threads performed best as all 16 cores of a node had enough load and computations to deal with, that there was a slow down due to context switching on any further increase in number of threads. CUDA performs the best for the 1164x1200 pixel image when a block of dimension 32x32 is launched, with a speed-up of 30x. For the 1164x1200 pixel image, the number of pixels on the horizontal axis is almost equal to that on the vertical axis, so the block dimension of 32x32 performs the best.

From the range of experiments on different size of images, we observe that GPUs can be beneficial when working with large datasets while a shared memory platform is good for small and medium sizes of input. For the larger images there exist lots of computations that can be done in parallel. As GPUs have more cores and are better suited for embarrassingly parallel calculations, the CUDA implementation performs best in these cases. GPUs are not well suited to process smaller datasets as the communication overhead between device and host is large compared to the benefit derived from faster computing. The shared memory platform of OpenMP is better suited for K-Means than the distributed platform of MPI as the communication overhead between physical nodes in a distributed setting is expensive. The downlink communication to broadcast the new means to all the workers is the dominant time consuming factor. Use of shared memory for broadcast is faster than on the distributed memory platform.

In the MPI implementation, distribution of processes among fewer physical nodes is advantageous for small images as a few nodes are sufficient for the required computation. Any additional increment in the number of nodes will give only network overhead rather than improvement. In the case of larger images, distribution of processes among a larger number of nodes is better as there exist enough computations to be performed by distinct nodes.

Figure 5 shows the time taken by K-Means to operate on the 300x300 pixel image with K = 30 for OpenMP with 16 threads and varying tolerance values. We observe that as we decrease the tolerance, the number of iterations as well as the speed-up compared to sequential C increase.

Figure 6 shows the effect of parallel improved initialization, as explained in Section IV for both images with (a) 300x300 pixel and (b) 1164x1200 pixel. We set tolerance to 0 and vary the number of clusters (K) from 10 to 240. We observe that parallel initialization helps in reducing the total number of iterations, leading to an overall reduction in processing time. Additional speed-up of 1.5x to 2.5x depending on the number of iterations is obtained by implementing parallel initialization. As K increases, a proper assignment of initial means becomes more important. If a small feature in an image does not contain an initial point then it may get merged with

TABLE I: Time and Speed-up (SU) for 300x300 pixels input image

K	Iter.	Seq.(s)	OpenMP(s)	OpenMP SU	MPI(s)	MPI SU	CUDA(s)	CUDA SU
10	4	1.8	0.1	16.51	0.13	13.84	0.16	10.77
30	14	5.42	0.21	25.8	0.32	16.93	0.47	11.48
50	63	30.08	1.28	23.5	1.45	20.74	2.06	14.56
100	87	43	1.39	30.93	1.98	21.71	2.68	16.01

TABLE II: Time and Speed-up (SU) for 1164x1200 pixels input image

K	Iter.	Seq.(s)	OpenMP(s)	OpenMP SU	MPI(s)	MPI SU	CUDA(s)	CUDA SU
30	21	1187.08	49.54	23.95	60.51	19.61	46.48	25.53
60	49	2651.28	98.77	26.84	115.68	22.91	93.68	28.29
120	92	4595.96	159.97	28.72	170.22	26.99	154.36	29.77
240	166	8897.72	300.54	29.60	315.15	28.23	294.01	30.26

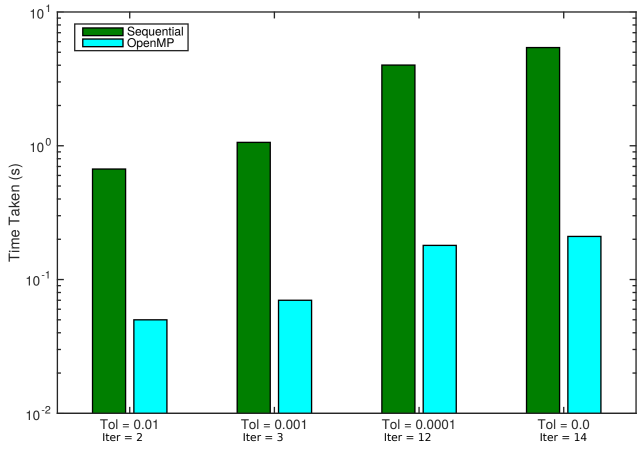

Fig. 5: Time with varying tolerance - Input image of 300x300 pixels

surrounding clusters. Also, the time taken by the algorithm increases with increasing number of clusters. As a result, parallel initialization has more room to improve the quality of clustering as well as showing speed-up for larger number of desired clusters.

In summary, our implementation of K-Means, which addressed all three basic challenges, is performed on three different parallel platforms and achieves significant improvements in speed-up for both small and large input images. In addition, We evaluate K-Means clustering algorithm on different parallel computing platforms to obtain the best one for a given input dataset.

VI. Related Work

Clustering is required in diverse applications in several fields. K-Means is a popular for clustering algorithm because it works well with large datasets and is a simple algorithm. K-Means clustering has previously been modified for speed-up on sequential platforms by following some statistical approaches like using dormant bins, skip frequency and hierarchical structure of k-d tree data structure [3]–[5]. These techniques give good speed-up on sequential platforms but as parallel architectures are not suited for operations including conditional statements and flow control, these methods do not give the desired improvement for parallel implementations. We explore and compare the methods suited for three different parallel platforms in this paper.

K-Means has three major challenges. First, the total number of iterations and quality of clusters formed depends on initial selection of centroids. Second and third, the number of the desired clusters to be formed and total number of iterations to be performed is expected to be given as input to the algorithm [6]. Previous work has tried to overcome the challenge of initialization on a sequential platform [7]–[9]. The basic idea is to selecting more than one set of centroids initially and pick the best one to operate further. In this work, we obtain better initial centroids by utilizing shared memory parallel platform with OpenMP and then use this set of centroids to initialize all other platform implementations. We tackle the second problem of giving the number of clusters as input by providing an upper limit to K and dropping out clusters when no points are assigned to them. The third challenge is that many implementations of the K-Means algorithm require the total number of iterations to be provided as input. Since the total number of iterations required to cluster the given dataset not only depends on the size of the dataset but also on the complexity of the contents, a small dataset might take more iterations than a larger dataset. Thus, assignment of a fixed number of iterations based on input size is not sufficient. In this paper, we implement K-Means based on convergence tolerance and thus automatically takes into consideration the size and the content of the input dataset to decide the total number of required iterations.

Machine learning algorithms have iterative behaviour and are often computationally demanding. This makes them good candidates for implementation on parallel computing platforms. K-Means clustering algorithm has previously been explored for speed-up on shared and distributed memory platforms individually [10], [11]. For very big datasets to be

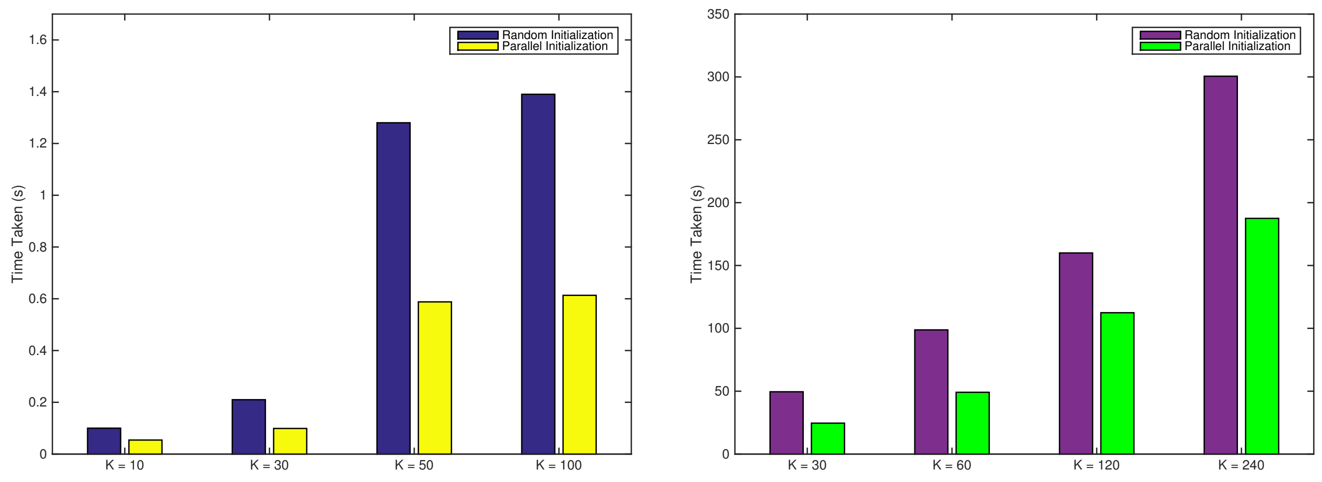

Fig. 6: Parallel Initialization - (a) Input image of 300x300 pixel (left). (b) Input image of 1164x1200 pixel (right).

clustered, GPUs were used for speed-up [12]–[14]. Working with GPUs, global memory was found to be better when compared with implementing K-Means using texture memory and constant memory [13]. Most of the parallel implementations explore only a single platform and either limit the number of iterations to a small number (around 20-30) [14], or use a small number of desired clusters (around 30-40). This paper is the first study that tackles all the major challenges of K-Means and evaluates performance across three parallel implementations.

VII. Conclusions and Future Work

We have presented a K-Means clustering implementation which tackles the three challenges of K-Means and deployed it on different computing platforms. We evaluate and compare our K-Means implementation across three parallel programming approaches. Our experimental results show around 35x speed-up in total. We also observe that the shared memory platform with OpenMP performs best for smaller images while a GPU with CUDA-C outperforms the rest for larger images. In the future, we plan to investigate using multiple GPUs as well as combining approaches using, for example, OpenMP-CUDA and MPI-CUDA. We also plan to adapt our implementation to handle larger datasets.

VIII. Acknowledgements

We would like to thank Professor Jennifer Dy for her guidance and Prajwal Shetty for his contributions and help in running the experiments. This work was partially supported by AFOSR Grant FA9550-14-1- 0160, NSF CAREER Award CNS-1452751 and NSF CCF Award 1218075.

References

[1] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM computing surveys (CSUR)*, vol. 31, no. 3, pp. 264–323, 1999.

[2] "Discovery cluster overview," May 2015. [Online]. Available: http://nuweb12.neu.edu/rc/?page_id=27

[3] T. Ishioka, "Extended K-Means with an efficient estimation of the number of clusters," in *Intelligent Data Engineering and Automated LearningIDEAL 2000. Data Mining, Financial Engineering, and Intelligent Agents*. Springer, 2000, pp. 17–22.

[4] D. Pelleg and A. Moore, "Accelerating exact K-Means algorithms with geometric reasoning," in *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1999, pp. 277–281.

[5] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient K-Means clustering algorithm: Analysis and implementation," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 24, no. 7, pp. 881–892, 2002.

[6] G. Hamerly and C. Elkan, "Learning the K in K-Means," *Advances in neural information processing systems*, vol. 16, p. 281, 2004.

[7] M. Yedla, S. R. Pathakota, and T. Srinivasa, "Enhancing K-Means clustering algorithm with improved initial center," *International Journal of computer science and information technologies*, vol. 1, no. 2, pp. 121–125, 2010.

[8] C. Zhang and S. Xia, "K-Means clustering algorithm with improved initial center," in *Knowledge Discovery and Data Mining, 2009. WKDD 2009. Second International Workshop on*. IEEE, 2009, pp. 790–792.

[9] K. A. Nazeer and M. Sebastian, "Improving the accuracy and efficiency of the K-Means clustering algorithm," in *Proceedings of the World Congress on Engineering*, vol. 1, 2009, pp. 1–3.

[10] J. Zhang, G. Wu, X. Hu, S. Li, and S. Hao, "A parallel clustering algorithm with MPI-MK-means," *Journal of computers*, vol. 8, no. 1, pp. 10–17, 2013.

[11] R. S. Yadava and P. Mishra, "Performance analysis of high performance K-Mean data mining algorithm for multicore heterogeneous compute cluster," *International Journal of Information*, vol. 2, no. 4, 2012.

[12] R. Wu, B. Zhang, and M. Hsu, "Clustering billions of data points using GPUs," in *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*. ACM, 2009, pp. 1–6.

[13] L. Jian, C. Wang, Y. Liu, S. Liang, W. Yi, and Y. Shi, "Parallel data mining techniques on graphics processing unit with compute unified device architecture (CUDA)," *The Journal of Supercomputing*, vol. 64, no. 3, pp. 942–967, 2013.

[14] J. Wu and B. Hong, "An efficient K-Means algorithm on CUDA," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 1740–1749.