Assignment 1 - Research and Planning: K-MEANS CLUSTERING

Luke Testa[1]

URN: 6119412

K-means clustering is an unsupervised learning algorithm that performs similarity grouping. Data points are grouped into clusters. The algorithm recursively queries a cluster's centre position until its position converges to a local minimum. This report evaluates a serial K-means solution for a 2D data set with a complexity of $O(log(N).N^{dk+1})$. The performance of K-means deteriorates with an increasing number of data dimensions, d, number of paritions, k, and the data set size, N. This document proposes parallising K-means by treating data points as independent asynchronous threads. The constraints and advantages of the parallel proposal are discussed.

I. INTRODUCTION: K-MEANS CLUSTERING

IN image processing, objects are represented as a set of data points in an N-dimensional space. Data points with similar geometric properties are grouped into clusters/subsets that represent object categories. Unsupervised learning utilises K-means to partition the feature space into object categories [1].

II. Q1: ALGORITHM DESCRIPTION

The algorithm first defines K random points as subset centres (also known as centroids). The algorithm generates subsets by assigning data points to their nearest centroid. On each iteration, data points compute their Euclidean distance to K centroids. Data points are assigned to the centroid with the lowest distance [2]. Once all data points are grouped into clusters, the clusters's mean position, μ_j, is computed from its data points, x_j. The centroid is moved into the mean position [2].

$$\mu_j = 1/N_j \sum_{n \epsilon S_j}^{N} x_n \tag{1}$$

K-means repeats until all centroids have converged to their local minimum. The program terminates when the total distance between data points, x_j and their centroid, μ_j, is at a minimum. This equation is defined as the clustering function in equation 2.

$$f(t) = \sum_{i=1}^{K} \sum_{n \epsilon S_j}^{N} ||\sqrt{x_k^2 - \mu_j^2}|| \tag{2}$$

A. Procedural Operations

Figure 1 proposes a serial solution to a 2D K-means problem. The data set is stored in a text file. Each line contains a data points [X Y] position.

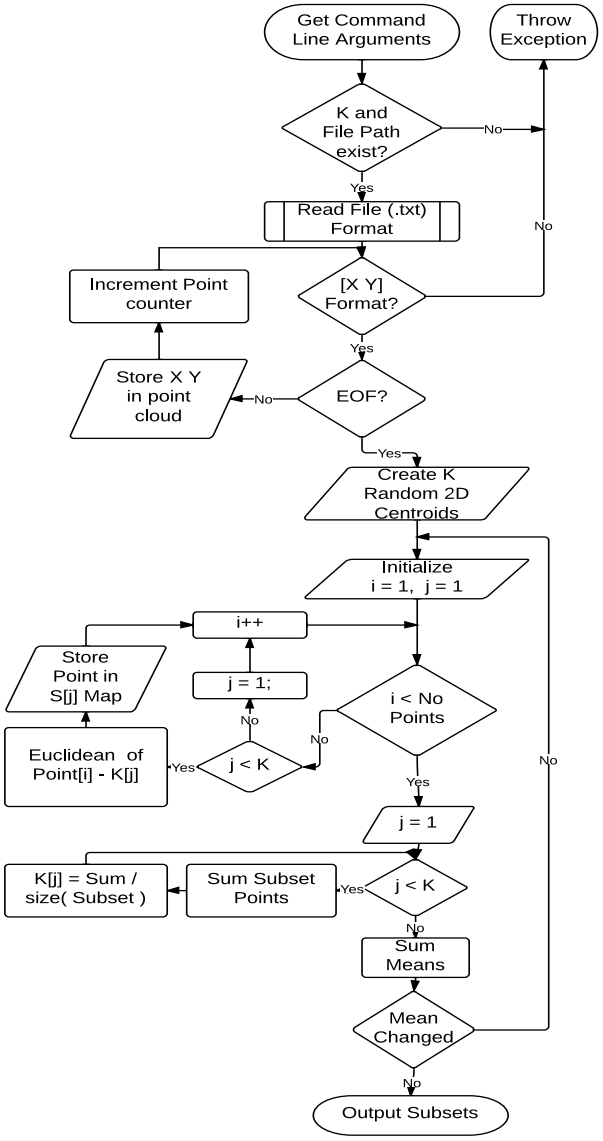


Fig. 1: Program level description of serial K-means.

The serial program has three command line arguments: The data set, output directory and number of partitions required. The first loop verifies the file format. Correctly formatted data point are appended onto a dynamic array. The array represents the point cloud. The second loop fits data points to a centroid using the Euclidean measure. Once all points are fitted to a subset, centroids converge to the clusters's mean position. The program terminates when the local minimum is achieved.

B. Object-Oriented Design

The proposed object-oriented design contains four classes. It can be assumed an exception class is present to handle errors appropriately. The FileHandler class opens and edits files in the local directory. This object has read and write capabilities. Data is transferred between the fileHandler and another object using stream objects and an operator interface.

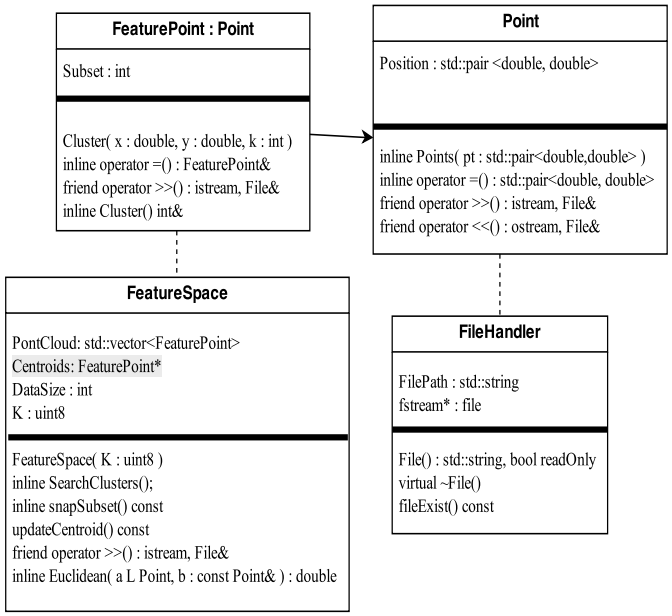


Fig. 2: Object hierarchy for serial K-means.

A FeatureSpace object contains the centroid positions and the data set. The parameter constructor takes in the number of clusters and instantiates K centroids with random positions. All numerical functions are in-lined to reduce function call overheads. Similarly function parameters are referenced to eliminate copy time. The squared Euclidean distance is computed to avoid square root operations. The FeatureSpace's SnapSubset member function fits all data points to a subset. UpdateCentroid computes the mean position of a subset's point cloud. SnapSubset and updateCentroid are called automatically by CreateClusters until the K-means terminating condition is reached.

C. Constraints

In addition to user-defined parameters, K-means' random nature may reduce efficiency. The compactness of centroid's initial positions vary the required number of iterations for convergence. Compactness is inversely proportional to execution time. These positions may also result in biasing if centroids are generated outside the data boundaries. Furthermore MacQueen implies the algorithm's execution speed is dependent on the properties of the data set - the distribution of data points and the overlap of clusters further influence complexity [2].

III. Q2: Justifying K-means Parallisation

Learning algorithms may update their data sets with successfully classified test points. As a result, the data set's size has a linear increase with time. This also requires K-means to re-partition the feature space into an unknown number of clusters with each updated point.

A. Complexity

K-means can be generalised as an numerical problem for N data points with D-dimensions. K-means' execution time is sensitive to these user-defined parameters. The second loop in figure 1 fits data points to their nearest centroid. This loop computes $N*K$ distance calculations and comparisons. Each Euclidean calculation requires $O(4D)$ arithmetic operations (D subtractions, additions and 2D multiplications). Therefore the total number of operations for the second loop is $O(N^{dk})$. The third loop moves centroids to their subset's mean position. The mean calculation requires N additions and 1 division per subset with a complexity of $O(Nlog(n))$. Both loops are repeated i times. The total complexity is $O(N^{dk+1} * log(n))$ makes real-time use impossible. Parallising K-means may improve the execution time to an acceptable complexity.

B. Existing Parallel Solutions

The CUDA SDK does not contain a readily available solution. K-means can be parallised by considering:

- Data points execute independent vector operations - Data points can operate as asynchronous threads.
- An algebraic operation is performed in D-dimensions - Vectorisation can reduce the required number of instruction cycles per calculation.
- Data points require read-only privileges to centroid data - Global memory access is sufficient.

IV. Q3: Parallel K-means Implementation

In this section the author will identify the parallelisable aspects of the serial K-means solution proposed in figure 1, and will discuss the difficulties of parallisation.

A. Q3a: Parallel Operations

Figure 1 decomposes the K-means algorithm into 4 tasks. Selecting the centroids, fitting centroids to subsets, converging the centroids and the clustering condition. In terms of parallization, K-means is a compute bound problem requiring $O(N^{dk+1}log(n))$ numerical operations.

Zachner and Fariver have created a parallel K-means solution [3,4]. Data points perform independent distance calculations. Furthermore data points require only read privilages to shared centroid data. Therefore data points numerical operations can be executed asynchronously on parallel threads. Each thread, p, will be responsible for N/p data points, decomposing the subset fitting problem to N^{kd}/p operations. Zachner suggests storing centroid data globally in the DRAM and using mutexing localise data control [3]. Global memory access is slow. Furthermore localising control is difficult. Fariver overcomes this by copying the centroid data from the global memory to the shared memory [4]. Shared memory access is significantly faster than global access. Also memory bandwidth is increased for large data sets due to threads having faster memory access. However, threads must use indexing to access shared data.

The author will implement an adaptation of these methods. Threads will locally control shared memory access. Each thread will be assigned N/p data points. CUDA handles dynamic scheduling implicitly when the number of cores is less than the number of points. A master-slave architecture will be utilised to manage threads. The master thread will be responsible for initialising and synchronising blocks, and managing memory transfer between the CPU and GPU. The random selection of centroid positions will be run sequentially on the CPU. For the second task, threads will run within parallel kernels. Loop unrolling will improve the subset fitting complexity to $O(N * K/p)$. As centroid data is accessed locally, the master thread must manually copy memory from the CPU to GPU, then request blocks to copy DRAM memory locally.

The author will vectorise data to improve the number of arithmetic operations per instruction cycle. SIMD architectures perform four parallel arithmetic operations

```
Sj ← Random (Xj, Yj)
Initialize Nx16 threads per block
DeviceSync
copymem( host → device )
ThreadSync
repeat
    for each thread
        Move centroid local
        replicate data size == centroid size
        compute squared euclidean
        lock shared
            store subset data
            increment point counter
        unlock shared
    end
    copymem( device → host )
    find mean
    copymem( host → device )
    for each thread
        compute euclidean
    end
    sum euclidean
    cluster function comparison
until convergence
```

Fig. 3: K-means Parallized Methodology.

on aligned memory per instruction cycle. Vectorisation may return to a scalar operation for other architectures. The overhead introduced by memory alignment will be neglectable for large values of K, D and N. Aligned memory will be stored within thread registers to free shared memory space. The third task computes the mean position of subsets. This requires data points to know their subset's index prior to computing the mean position. As a result, some solutions do this sequential on the CPU [3, 4]. The master thread will use the subset reference index returned by data points in task two to compute subsets mean positions. The master thread will broadcast the updated centroid positions on the GPU.

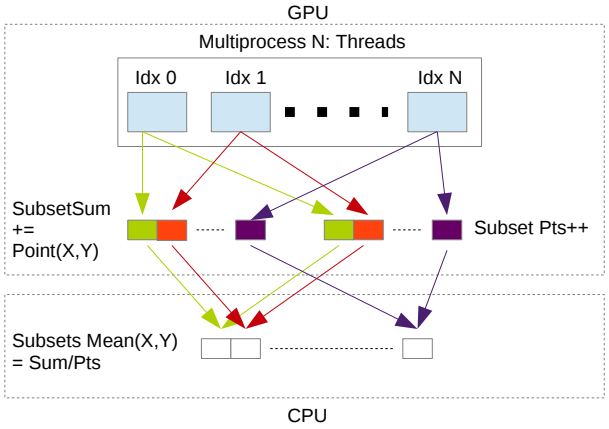


Fig. 4: Decomposing mean into summation and division.

Alternatively, a mean calculation can be separated into a summation followed by a division. This has an ILP of 3/2. The summation can be executed in parallel and the division in series. This is shown with two arrays in figure 4. This will require localising shared memory control. A synchronisation barrier is also required prior to aggregating the summation output. Two empty arrays will exist in global memory. One array identifies the point counter of subsets while the other contains the [X Y] sum of subset's point clouds. Threads will know their subset's index on finishing task two. Each thread will increment their subset's point counter and add their X Y data onto their subset's sum. When all threads terminate, the master thread will copy the 2 arrays from the device to host. The new centroid positions will be computed on the CPU. The master thread will broadcast the results on the GPU. CUDA supports atomic functions. These allow threads to perform read-modify-write operations on shared memory without the risk of race conditions [5]. Threads implicitly lock data before modification and unlock data after storage. Therefore localising shared memory control for the summation step is possible. However Atomic functions only support 32-bit and 64-bit operations - This method sacrifices double precision. Type casting from a double to 64-bit data types have an overhead. The performance for the speculative mean is $(8N + k)/p(6K + N)$ faster than the sequential mean including the overhead.

The final K-means task computes the cluster function. This is similar to fitting data points to their nearest subset. The master thread can either retrieve the Euclidean distances for each subset sequentially or retrieve the cluster function output from shared memory by assigning N/p data points to p parallel threads after populating the GPU with the updated centroid and subset information. The master thread will query the termination condition.

B. Q3b: Synchronisation

Blocks are launched asynchronously in task two. Synchronisation points are required to manage GPU memory transfers. The device is synchronised with cudaDeviceSynchronize() after launching blocks to acknowledge GPU memory has been allocated correctly by the host. Threading operations will contain a syncthreads() barrier to pause threads until host to device memory transfers are complete. Threads will continue to align data points for vectorisation. Each block will manage barrier synchronisations independently. After launching kernels, control returns to the CPU. synthreads() is also required to stop the CPU from terminating while threads are running. No explicit synchronisation for read-modify-write operations is required. In step three, the host processes data cached on the GPU. All threads must be synchronised on completing their tasks. On synchronising threads, blocks will copy shared memory to global memory. The host will create a local copy of the DRAM memory on the CPU with cudaMemCpyFromSymbol() for further processing. The final task reinitialise blocks with a new centroid position. This task's synchronisation requirements are similar to task two. Memory transfers and kernel launching will require the same synchronisation points discussed earlier.

C. Q3c: CUDA Architecture

Instructions and memory transfers are handled differently within the CUDA architecture. Figure 5 describes the memory transfers between master and slave threads.

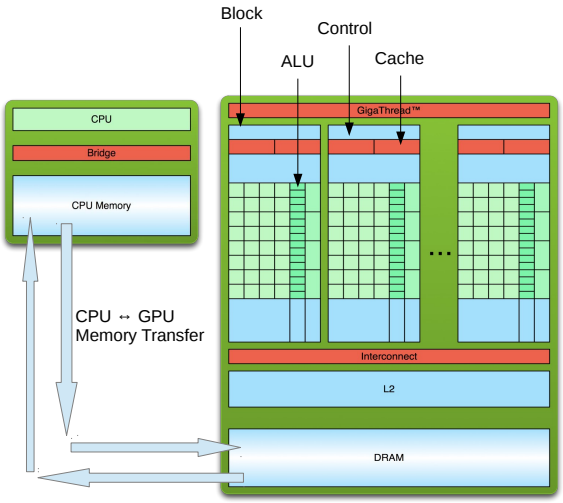


Fig. 5: CPU and GPU memory transfer [6].

GPU memory is manually allocated by the master thread running sequentially on the host. Host memory is then copied from the CPU to the GPU's DRAM. All kernels have read-write access to the DRAM memory, but DRAM memory is not shared between grids. Each block manages a group of threads. The number of threads contained per block is set by the hardware specifications. Threads contain a local cache. Cache memory contains a configurable 16kB of memory [7]. Threads can copy global data to shared memory to improve memory bandwidth as shared memory access is faster than global access. Thread's also contain registers for storing fetched and local scope data. Feature data will be copied from the DRAM to local memory to improve access speeds. Centroid data will be accessed

from the shared memory by indexing array elements.

Blocks contain a hierarchy of threads. Blocks are grouped into grids. Each block manages shared memory. When threads are initialised, the host, or master thread, states the number of threads to be allocated per block. Block are asynchronous, thus do not share memory. This allows threads to compute the Euclidean distance for data points independently. Shared memory is only accessible within blocks. Each block must contain a copy of the centroid information. The master thread will distribute a copy centroid positions to launched blocks after confirming all blocks are setup. For data collection, blocks will contain a segment of an array's information. The master thread will be responsible for reconstructing data from asynchronously kernels into a 1D array at the synchronisation barrier.

The CPU instructs the GPU to begin processing instructions. The master thread may over launch multiprocessors if the number of points exceeds the number of GPU cores. In this case, the GPU will automatically handle dynamic scheduling on the Fermi architecture's Execution Queue as shown in figure 6.

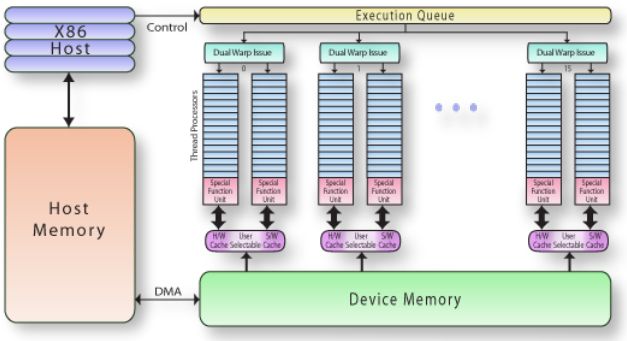


Fig. 6: Fermi Block Diagram [7].

SIMD architectures support vectorisation. A thread's ALU performs parallel arithmetic operations on aligned vector elements. This reduces the number of instruction cycles required to complete vector operations. Blocks contain stream processors. These operate on parallel elements without explicit synchronisation. The Fermi architecture supports up to 16/32 stream processors per block. Blocks contain a group of 32 threads called warps [7]. The stream processors execute one instruction for each warp element in four clock cycles. Blocks are launched with a thread size equal to a multiple of the warp size. User defined array sizes must also be a multiple of the warp size although this is handled implicitly.

V. Q5: Serial Verification and Testing

A serial solution has been written in C++ and tested in the Penguin labs. MATLAB has been used for data mining. The following three tests were completed.

- Verify K-means is partitioning data points correctly.
- Measure the required execution time for an increasing data size.
- Measure the required execution time for an increasing number of partitions.

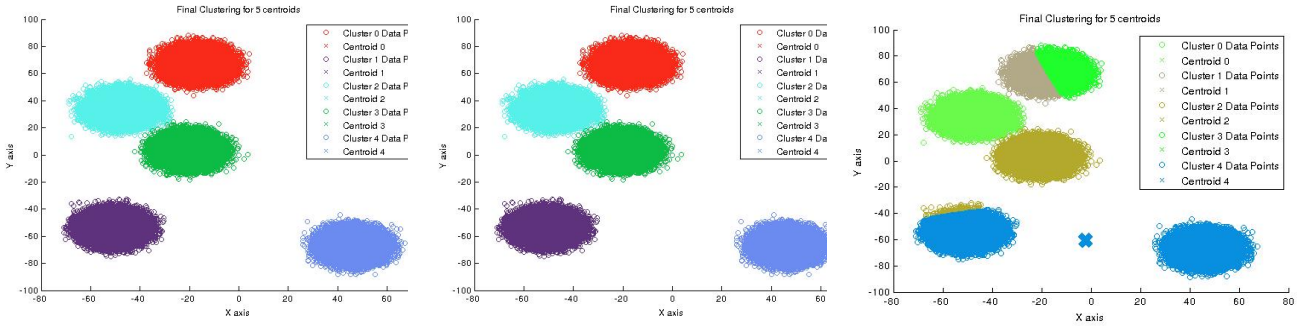
A. Training Set

A synthetic data set has been generated in MATLAB. The data set contains 5 clusters, each cluster containing 100K points with a distribution of 5. The spread of cluster points is constant to avoid biasing partitions. The feature space is two dimensional. K-mean's performance at other dimensions can be approximated by scaling the experimental complexity by a factor of D. Centroids are generated within the limits displayed in figure 7.a. Finally the distance between clusters is varied to replicate unideal data sets. Tests were repeated 10 times.

B. Results

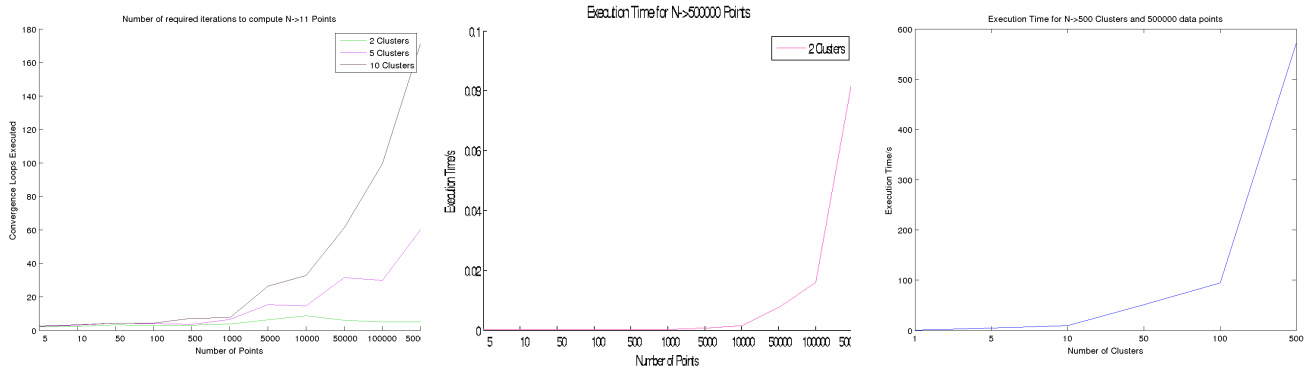
The first test validates whether data is partitioned correctly. Figure 7.b and 7.c show the results of two runs with 5 clusters. Data points are grouped by colour. Fig 7.b and 7.c concludes centroids are randomly generated within the data bounds and converge to a local minimum as expected. Figure 8.a and 8.b shows the influence of the data size on the complexities growth for under fitting, perfect fitting and over fitting cases. The number of data points is varied for all paritions. figure 8.a shows the iterations required for convergence. Over fitting results in compact centroid positions. Theur rate of convergence is therefore dependent on the separation between neighbouring centroids. For under fitting, centroids are generated in their final cluster bounds. This increases their rate of convergence, maintaining a constant number of iterations for an increasing data size. The influence of data size on K-means' complexity was derived from K = 2 due to the constant number of iterations undertaken. A new data set requires $f(N) - f(N-t)$ additional calculations. Under fitting can conclude the data size has a growth of $O(N * log(N))$.

Figure 8.c varies the number of partitions for 500k points. Data points oscillate between subsets until the separation between centroids is sufficient. This causes



(a) Synthetic Ground Truth (b) Serial Data Partitions (c) Serial Data Partitions

Fig. 7: Data set partitioned into 5 clusters



(a) Required Iterations for K = 2 (b) Execution time for K = 2 (c) Execution time for K tends to oo

Fig. 8: Influence of N and K on K-means complexity

the complexity to grow at $O(N^k)$. Each iteration performs a D-dimensional Euclidean calculation. The general complexity is $O(N^{kd})$. The K-means experimental complexity is $O(N^{k*d}Nlog(N)) = O(N^{k*d+1}log(N))$.

VI. Q5 continued: Conclusion

K-means may be used to update unsupervised learning models in real-time. A general solution is computationally heavy, and is significantly dependent on the properties of the data set and is worsened by the number of partitions having a growth of N^k. Parallisation proposes loop unrolling the Euclidean and mean calculations by assigning N/p data points to p parallel threads. This may improve the data size growth to $O(N^{kd}/p)$. Speculative execution can decompose the mean into a summation and division. An ILP of 3/2 has a computational improvement of $klog(n)/Np$. K-means can therefore benefit from the proposed parallel solution, which may improve the overall performance by $k/(N * p^2)$, making K-means applicable for intraframe processing. Synchronization barriers and localising memory control are key requirements to a successful parallisation. The next document will compare the execution time of a parallised and serial K-means solution.

References

[1] C.M. Bishop, *Neural Networks for Pattern Recognition*, 3rd ed, pp 187-188. 1995.

[2] J. MacQueen, *Some methods for classification and analysis of multivariate observations, Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability,*, Volume 1: Statistics, pp. 281-297. 1967.

[3] M. Zechner, M. Granitzer, *Accelerating K-Means on the Graphics Processor via CUDA, Intensive Applications and Services, 2009*, First International Conference on , vol., no., pp.7,15, 20-25. 2009.

[4] R. Fariver, *Implementing Parallel K-means clustering using CUDA*, Presented at the Systems Research Group Reading Group, Univsity of Illinois. 2009.

[5] Nvidia, *CUDA C Programming Guide 5.0*, Internet: http://docs.nvidia.com/cuda/cuda-c-programming-guide/ 2013.

[6] C. Zeller, Nvidia *Nvidia, CUDA C/C++ Basics: Supercomputing 2011 Tutorial* , Presented at SC10, New Orleans, 2011

[7] M. Wolfe *Understanding the CUDA data parallel threading model*, Internet: www.pgroup.com/lit/articles/insider/v2n1a5, PGI Insider Feb 2010