Graphics Hardware based Efficient and Scalable Fuzzy *C*-Means Clustering

S.A. Arul Shalom[1] **Manoranjan Dash**[2] **Minh Tue**[3]

[1, 2] School of Computer Engineering,
Nanyang Technological University,
Block N4, Nanyang Avenue, Singapore 639798

[1] sall0001@ntu.edu.sg [2] asmdash@ntu.edu.sg

[3] NUS High School of Mathematics and Science,
20 Clementi Avenue 1
Singapore 129957

[3] h0630082@nus.edu.sg

Abstract

The exceptional growth of graphics hardware in programmability and data processing speed in the past few years has fuelled extensive research in using it for general purpose computations more than just image-processing and gaming applications. We explore the use of graphics processors (GPU) to speedup the computations involved in Fuzzy *c*-means (FCM). FCM is an important iterative clustering algorithm, and usually performs better than *k*-means. But for large data sets it requires substantial amount of time, which limits its applicability. FCM is an iterative algorithm that involves linear computations and repeated summations. Moreover, there is little reuse of the same data over FCM iterations (i.e., the centre of the clusters change in each iteration) and these characteristics make it a good candidate to be mapped to the parallel processors in the GPU to gain speed. We look at efficient methods for processing input data, handling intermediate results within the GPU with reusability of shader programs and minimizing the use of GPU resources. Two previous implementations of FCM on the graphics-processing unit (GPU) are also analysed. Our implementation shows speed gains in computational time over two orders of magnitude when compared with a recent generation of CPU at certain experimental conditions. This computational time includes both the processing time in the GPU and the data transfer time from the CPU to the GPU.

Keywords: Fuzzy *c*-means, GPGPU, Clustering, Parallel Computation

1 Introduction

Clustering finds out hidden patterns in the data set by grouping similar data objects together. It does not require any prior knowledge of the data objects and about the groups they belong to. Typically there are three types of clustering algorithms: partitioned, hierarchical and density based. (Jain, Murty and Flynn 1999). In the partitioned clustering algorithm (MacQueen 1967) the number of clusters is specified and the clustering algorithm uses similarity measure to determine the clusters. Hierarchical clustering (Guha, Rastogi and Shim 1998) can be divisional or agglomerative. In the agglomerative hierarchical clustering algorithm, each data object is considered as a cluster and the closest pair of clusters is merged in iteration repeatedly until there remains only one cluster, thus producing a dendrogram. The dendrogram can be used to obtain the clusters as per the required number of clusters. Density based clustering (Ester, Kriegel, Sander and Xu 1996) is based on density parameters. Data objects are considered connected to a cluster or disconnected depending on the density parameters. In this paper we focus on an important partitioned clustering algorithm called fuzzy *c*-means (FCM) (Bezdek 1981).

The *k*-means clustering method or the hard *c*-means algorithm groups *n* objects in a data set into *c* clusters. To begin this iterative process, the initial *c* cluster centres are predetermined. In hard clustering, data is divided into distinct clusters, where each data element belongs to exactly one cluster. In fuzzy clustering, such as FCM, data elements can belong to more than one cluster. Each data element is associated with a set of membership values. These indicate the strength of the association between that data element and a particular cluster.

As will be shown in later section, FCM is based on the standard least squared errors model. FCM is very popular due to several reasons. It can be generalized in many ways. Arguably it is much easier to generalize FCM than the hard *c*-means clustering. For example, the memberships are generalized to include possibilities; the distance used has been generalized to include Minkowski (non-inner product induced) and hybrid distances; there are versions of FCM for very large data sets that utilize both progressive sampling and distributed clustering; there are many techniques that use FCM clustering to build fuzzy rule bases for fuzzy systems design; and there are numerous applications of FCM in virtually every major application area of clustering (Wiki 2008). The various high volume data visualization applications of FCM include image

segmentation, multi-spectral image compression, remote sensing, object recognition, biological sequence analysis, clustering co-expressed genes, and to hybridise various other data mining algorithms.

Clustering large amounts of data takes a long time. To cluster these large data sets, either sampling is required to fit the data in memory or the time will be greatly affected by disk accesses making iterative clustering (e.g., *k*-means, fuzzy *c*-means) an unattractive choice for data analysis. In this paper we focus on how to ease the computational bottleneck of FCM on large data sets using graphics processors (GPU).

Many researchers are able to use GPU for data mining algorithms over large data sets (Owens, Luebke, Govindaraju, Harris, Krüger, Lefohn and Purcell 2005). This area of research is known as GPGPU (General Purpose Computations using GPU). Although GPUs are quite powerful due to their internal architecture they favour algorithms that can be structured as streaming computations often realizing notable performance gains (Fatahalian, Sugerman and Hanrahan 2004). Streaming computations can be characterized as being highly parallel and numerically intensive. One such suitable application is FCM. It is streaming in nature but its data (the centroids) change from iteration to iteration. Hard *c*-means (*k*-means) which is the primate of the FCM is efficiently implemented in the GPU (Arul, Dash and Tue 2008).

In Section 2 we briefly discuss the GPU hardware features that enhance its application in GPGPU. Section 3 describes briefly the previous FCM implementations on the GPU and their results. In Section 4 our proposed implementation and its novelties are discussed addressing scalability issues. Section 5 discusses experimental setup, shows the analysis and results. Section 6 is the conclusion with a brief discussion on future expansions on FCM and other clustering methods using GPU.

2 Exploiting the Modern Graphics Hardware for General-Purpose Computations

The GPU has tremendous image processing capabilities such as vertex transformation, lighting computations, clipping and culling of images using its highly parallel hardware pipeline. For instance, the massively parallel GPU, Nvidia's GeForce 8800 GTX, consists of 128 individual stream processors each running at 1.35 GHz clock frequency, with very high memory bandwidth of 86.4 Gigabytes per second. (NVIDA: GeForce 8800 Architecture Technical Brief 2008). The GeForce 8800 GPU's shader architecture is designed for extreme 3D graphical performances, producing near reality image quality for delighted gaming performances, which is its traditional forte. Figure 1 shows the block diagram of the GeForce 8800 GPU, which shows the various stages of the parallel programmable processors.

2.1 GPU as a Low Cost High Power Computational Processor

In Figure 1 the host forms the interface block between the CPU and the GPU. In graphics processing, the host receives the commands from the CPU, geometric data and other display data. The input data from the CPU is assembled and formatted before the next stage of graphics processing. Each of the GPU's internal processors could be assigned to a specific shader program. Shaders are short lines of codes that run on the stream processors which process incoming stream data and send the computed data to output buffers or textures. The stream processors are grouped in a manner so that computational resources can be efficiently mapped to these processors. The processed data can be sent as stream data to other stream processors for further processing. Such computations are possible due to data independency in graphics processing. This also permits multiple shader programs to run on the processors, each shader accessing data in parallel that is linked to the stream processor.

The stream processing capabilities of the GPU makes it highly applicable to implement general-purpose computations. Computations to be implemented in the GPU will need to be mapped appropriately using the hardware resources such as textures and frame buffers.

The programmability of the stream processor is achieved using shader programs. Various general-purpose computations such as physical simulations, image processing and data mining algorithms have been implemented on GPU, harnessing its computational power and programmability to improve computational efficiency as compared to the CPU (Owens, Luebke, Govindaraju, Harris, Krüger, Lefohn and Purcell 2005). The GPU thus has become a low cost commodity processor with high computational power, for which the growth is heavily driven by the gaming industry. The cost of the speed gained from using such a GPU is much lower than the CPU based massive parallel processors. We intend to efficiently implement the FCM computations using GPU.

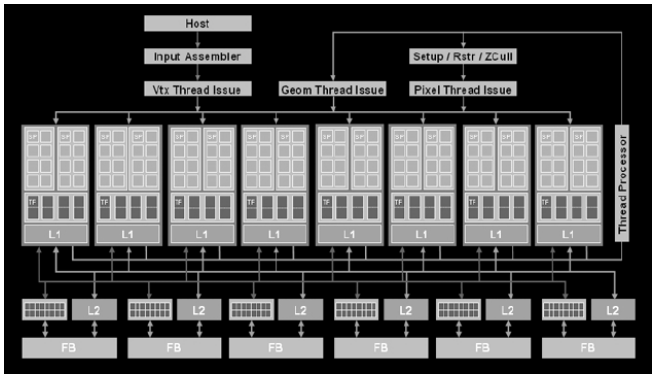

Figure 1: GeForce 8800 GTX Block Diagram

3 FCM Algorithm and Existing GPU based Implementations

The FCM algorithm can be summarized using the following simple steps:

1. Initialise cluster memberships
2. Calculate cluster centres
3. Update cluster memberships
4. Check stopping condition, else go to Step 2.

The FCM algorithm partitions a set of feature vectors x_i into c clusters by minimizing the objective function given by $J(U_{ij}, C_j)$ in equation 0, where m is a real integer greater than 1, U_{ij} denotes the degree of membership of the d-dimensional vector x_i in the cluster j and C_j is the centre

of that cluster. The norm ||*|| expresses the closeness of the vector to its cluster centre.

$$J(U_{ij}, C_j) = \sum_i \sum_j U_{ij}^{\ m} \| x_i - c_j \|^2 \quad (0)$$

In this iterative fuzzy partitioning optimization process on the data set of size *n*, the cluster memberships U_{ij} of each observation *i*, to the *c* clusters is computed by equation (1).

$$U_{ij} = 1 \Big/ \sum_{k=1}^{c} \left[(x_i - c_j) \ / (x_i - c_k) \right]^{2/(m-1)} \quad (1)$$

This equation is also used for cluster centroid updates, where *m* is the fuzzifier. The value *m* determines the amount of fuzziness. The value of *m* can be chosen from (1, ∞). A value of *m*=1 produces a hard clustering. As *m* approaches ∞ the solution approaches its maximum degree of fuzziness. It is often chosen on empirical grounds to be equal to 2. In FCM, the fuzzy centroids depend on the current membership values and all the individual observations *i*. The fuzzy centroid C_j is computed using the equation (2).

$$C_j = \sum_{i=1}^{n} U_{ij}^m x_i \Big/ \sum_{i=1}^{n} U_{ij}^m \quad (2)$$

This iteration will stop when the termination criterion given in equation (3) is fulfilled, ε is a termination criterion between 0 and 1.

$$Max_{ij} \{ | U_{ij}^{k+1} - U_{ij}^{k} | \} < \varepsilon \quad (3)$$

The steps of the FCM algorithm are further briefly explained here. The first step involves the initialisation of the initial cluster memberships and also includes the initialisation of the clustering variables. The value for *m* is chosen to be 2, the number of clusters *c* is set as predefined for the *n* number of observations in the data set. The distance between the *c* initial clusters and the individual observations are computed, which is the inner product norm between the vectors. To end step 1, the values of the cluster memberships for each observation is computed using equation (1).

In the second step the centre of the clusters is calculated using equation (2). The fuzzy centroid C_j represents the vector location of the centre of the i^{th} cluster. The cluster centres are thus computed for the all the *c* clusters.

In the third step the memberships are updated with the new values based on the distances of each observation to each of the cluster centre. Equation (1) is again used for this computation.

In the last step, the algorithm is checked for its stopping condition using equation (3). The stopping condition should be predefined. If the error between the current cluster centres and the corresponding previous centres are less than 0.00001, the computations of the algorithm ends.

In fuzzy *c*-means iterations, the utilization of computational resources is high and is mainly contributed by:

1. Distance computation between the objects and the cluster centres.

2. Computing the degree of membership for all objects in every cluster.

3. Computing new cluster centres as a function of the degree of membership.

The implementation of FCM in the GPU will reduce the computational time, utilizing the computational resources of the Graphics hardware.

3.1 Previous GPU Implementations of FCM

There are two previous works where FCM has been implemented in the GPU (Harris and Haines 2005), (Anderson, Luke and Keller 2007). Reduction in computational time in the order of 2x times has been achieved from a non-iterative GPU implementation when compared to the CPU FCM (Harris and Haines 2005). This implementation is able to handle huge number of observations, but not scalable in terms of dimensions and the number of clusters. In the second work, the authors present a FCM with non-Euclidean distance computation metric and have demonstrated processing time gains of over two orders of magnitude for certain configurations of FCM, where different combinations of data size, dimension size and clusters are used.

While implementing the FCM on the GPU the following considerations are to be carefully made so as to avoid the drawbacks of the previous FCM implementation (Anderson, Luke and Keller 2007). (1) Limitation on the number of textures that can be fetched by the fragment programs: For instance, if the GPU has 16 fragment processors, the maximum number of textures that can be accessed at any one time is limited to 16. (2) Minimum use of textures per cluster to avoid memory constraints: For instance, while computing large number of clusters, if enough care is not taken, the number of textures required for handling cluster membership values will be large. (3) Maximum reuse of shader programs to increase portability.

4 Efficient and Scalable Implementation of FCM on the GPU

In our GPU-based FCM implementation, the various iterative components of the algorithm are executed in the fragment processor using shaders. Textures are memory locations in the GPU, which are used to store the distance and the membership matrices. Multiple dimensions in the incoming data are handled by using partial sum of squared distance computations and stored in the distance textures. All textures use '*Luminance*' as internal data format. A speed gain over two orders of magnitude has been achieved for a 79 dimensional yeast gene expression dataset which has about 64k observations. Figure 2 shows the FCM scheme that is implemented on the GPU. The CPU provides the control on the execution of the algorithm; the required number of iterations, control loop branching and the checking of stop condition. The inherent parallelism of the GPU is exploited and used for the iterative computations in the FCM algorithm such as distance computations, membership computations, and computation of cluster centres. The execution steps of FCM in GPU are quite similar to the implementation in CPU. In the next section we discuss the steps involved in our GPU based FCM implementation briefly.

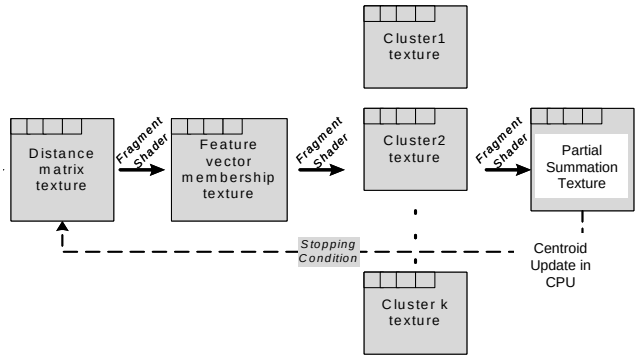

Figure 2: GPU Implementation Scheme of Fuzzy *c*-means Clustering Algorithm

4.1 GPU based FCM Functions

Parts of the algorithm are computed in a way that the parallelism of the GPU hardware can be exploited to make it efficient. The major steps in our implementation of the FCM algorithm on the GPU are stated below and further discussed.

1. Create initial membership matrix for all *n* data observations with respect to each cluster.
2. Initialize the *c* cluster centers from the *n* data vectors.
3. Compute sum partial deviations between the *c* cluster centers and the *n* data vectors.
4. Compute the ratio between the sum partial deviations of the cluster being compared to each other cluster.
5. Store ratio of sum partial deviations in textures, one per cluster.
6. Compute exponentiation of all the deviation textures.
7. Compute partial memberships for all observations per cluster.
8. Compute the membership values via summation of partial memberships.
9. Transfer the summed membership values to CPU.
10. Compute new cluster centers in the CPU.

The initial membership matrix U_{ij} is randomly generated for all the *n* observations with respect to each cluster. The initial U_{ij} is made the same for both the GPU and CPU implementations by using the same seed in the random generation of membership values. Initial cluster centres (C_j) are identified. The deviations between these cluster centres and each of the data vectors are computed and summed. The deviations between the cluster centre and data vectors are computed partially. The partial computation of deviations is repeated *d* times and summed, where *d* is the number of dimensions in the data vector. Texture reduction technique is employed for all summations. The ratio of the deviations between each cluster centre being compared with each individual data point and the other deviations as in equation (1) is computed. These ratios are stored in textures one per cluster. After the computations are complete for *d* dimensions, all these textures are simultaneously raised to the power of $2/(m-1)$. The inverse of the resultant texture will produce the iterated membership texture matrix U_{ij}. Using the membership values the new cluster centres are computed. For this operation, the summation of the product of membership texture and the input data objects and the summations of the membership textures are obtained in the GPU. These summations per cluster per iteration are transferred to the CPU and repeated for *d* dimensions. So the number of transfers is in the order of $d * c *$ number of iterations. In the CPU the new cluster centres (C_j) are compared with the previous cluster centres (C_k) and the decision is made whether to continue or stop the iteration based on the stopping condition. The error between the current cluster centres and the previous should be less than 0.00001. Table 1 lists the shader programs used to accomplish these computations and the major purpose of each shader program.

4.2 Scalability in the GPU based FCM

In data mining scalability means to take advantage of the existing parallelism and design solutions to solve a wide range of problems without needing to change the underlying implementation. We realize scalability via (1) data representation in the GPU memory, (2) operational flexibility on data dimensions and (3) ability to accommodate data sets with higher dimensions. Data representation is handled by accessing individual dimensions across all data objects. So it is easier to perform computations on huge data sets. Moreover, using our GLSL implementation it is easier to perform various operations on the dimensions. It is also simpler to reconfigure the shaders for higher dimensions, since partial computations are done across the data vectors and large number of clusters, thus being more adaptable and flexible, compared to previous implementations. Most notably, scalability is achieved since there is no necessity to define huge textures and redesign the fragment shader codes, as was the case for the earlier algorithms.

No.	FCM Functions	Function call	Fragment Shaders	Purpose of the steps in GPU based FCM
I	Distance Computations (GPU)	Computation0( )	glslProgram0( )	Sets initial textures with zeros
			glslProgram1( )	Computes the distances; summation of partial distances
II	Calculating the exponential (GPU)		glslProgram4( )	Computes the exponentials of the distance deviations in the membership matrix
III	Partial Summations (GPU)	Computation1( )	glslProgram0( )	Sets initial textures with zeros
			glslProgram2( )	Computes partial summation across all textures
IV	Partial membership computations (GPU)	Computation2( )	glslProgram3a( )	Computes partial memberships based on distance and partial sums
			glslProgram3b( )	Multiplies the memberships with coordinates to obtain the membership * cluster member product
			glslProgram3c( )	Computes the summation of the membership values
V	Update of new cluster centroids (CPU)	-	-	Computes the new centroids by dividing the membership * cluster member product by the summed membership values to obtain fuzzy centroids

Table 1: Summary of the FCM Steps and the Fragment Shaders used for Computations

5 Experimentations on GPU based FCM

The objective of this experiment is to implement the traditional FCM algorithm on a GPU to form fuzzy clusters. Compare its performance with an equivalent implementation of the same algorithm on a desktop CPU. In both implementations initial membership values were randomly generated and made the same using a common random generated seed. The detailed experimental setup and the evaluation of the results are discussed in the next three sections below. The novelties of our implementation and the challenges are discussed subsequently.

5.1 The Experimental Setup

The algorithm is executed on 2 Nvidia's GPUs; viz. GeForce 8500 GT, which is considered as a mid-range graphics processor, and a GeForce 8800 GTX, which is considered as a high-end graphics processor. The results obtained are compared with that obtained from their corresponding CPU counter parts, which are Pentium4 (D), 3.0 GHz CPU and a Pentium(R), 1.5 GHz CPU respectively. The performance of the GPU on the computations heavily depends on the hardware characteristics and hence the GPU configurations are described in this section. The 8500 GPU has 16 fragment shaders processing texels to pixels at a memory clock rate of 800 MHz and 512MB of video memory. The peak memory bandwidth is 12.8 GB/sec. The 8800 GPU has 128 total stream processors with a memory clock rate of 900 MHz and 512MB of video memory. The peak memory bandwidth is 86.4 GB/sec. The experiments will involve the following:

1. Complete the GPU based FCM iterations until the stopping criterion is satisfied and measure the computational time (GPU processing time + data transfer time) for various combinations of n, d and c. Repeat the same on the corresponding CPU and measure the computational time.

2. Use synthetic data to conduct efficiency studies. The size of data, size of dimensions and the cluster numbers will be varied in order to understand the computational efficiency, and the GPU to CPU data processing time ratio.

3. Use the yeast gene expression data set, which has 79 dimensions with about 65k genes (Arul, Dash and Tue 2008), to compare the performance of both the GPUs over their CPU counterparts.

4. Analyse the data transfer time (CPU to GPU) and computational time (time/iteration).

5. Perform regression on the data summarized from the 8800 GPU studies to understand the effect of the cluster parameters n, d and c on computational efficiency.

From the above experiments we intend to analyse the following performance metrics to compare and understand the benefits and challenges in using GPU for the FCM computations, which can be then generalized to other general-purpose computations.

1. Accuracy of the cluster centres formed by the GPU as compared to the centres from the CPU.

2. Raw computational time (C_t) comparison between the GPU and the CPU.

3. Comparison of the computational efficiency which is obtained from the ratio of (CPU computational time per iteration) / (GPU computational time per iteration).

4. Comparison of the processing speed gain, (P_t ratio) which is the ratio of the (CPU computational time) / (GPU computational time – Data transfer time, T_t)

5. The influence of the FCM factors such as c clusters, input data size n and d dimensions on the GPU computational time per iteration.

5.2 Experimental Results Evaluation

The accuracy of the GPU cluster centres as compared to that from the CPU was determined using Mean Square Error (MSE) between the cluster centres from both the CPU and the GPU. With data sizes more than 65k, dimensions up to 10 and for 4 clusters the MSE was in the order of 10^{-8} to 10^{-11}. This insignificant error is due to the lower floating-point precision in GPU compared to that of the CPU. This insignificant error shows that the clusters formed by the GPUs are reliably accurate.

The results obtained from the GeForce 8800 GTX GPU as compared with the Pentium(R), 1.5 GHz CPU are substantial. In our implementation we show that the computational efficiency (CPU computational time per iteration) / (GPU computational time per iteration) ranging from 20x to 94x times. Note that number of iterations required satisfying the stopping criterion may vary between CPU and GPU because of GPU's limited precision. So, if we consider the total time ignoring the difference in number of iterations, then we get the following equation: overall speed gain = (total time taken by GPU/time taken by CPU) which is almost of the same order: 20x to 95x times based on various combinations of factors such as d, n and c.

The results obtained from the mid-range GeForce 8500 GT GPU as compared with the Pentium4 (D), 3 GHz CPU are also promising. Results show that the computational efficiency ranges from 14x to 43x times and overall speed gain is almost of the same order: 14x to 53x times.

The results of the various experiments are graphically summarized, shown and discussed in the next section. Figure 3 shows the comparison of the computational time (C_t) between the two GPUs and their corresponding CPU counterparts.

5.3 Discussion on the Results

When the data size is small (say, 2048), the GPU seems to be slower or just the CPU is as good as the GPU in computing as seen in Figure 3. This comparison shows the raw computational time taken by both the CPU and the GPU ignoring the number of iterations. It can be seen that as the data size increases, there is tremendous speed gain in the GPU computation. When the data size is small, many parallel processors and memory resources are left unused, but when the data size is large, the utilization of the GPU resources is maximized. The GeForce 8800 GPU is able to complete the tasks of forming 4 fuzzy clusters from 1 million 4 dimensional data objects within 0.91 seconds where as the corresponding CPU could take up to 87.8 seconds. The GeForce 8500 GPU is able to complete the same task in 6 seconds when its CPU takes 313.8

seconds to complete the task. It can also be noted that almost 77% of computational time (C_t) of GPU is taken up by the data transfer time (T_t) from the GPU to the CPU. Figure 4 shows the comparison of the computational efficiency between the two GPUs. The processing time ratio is also compared to show how fast it is to process data within the GPU.

As the data size increases, the computational efficiency of the GPUs in implementing the FCM algorithm increases ranging from 20x to 94x times. The computational efficiency takes the GPU to CPU data transfer time and the actual processing time into account. To have a fair comparison with the implementation in (Anderson, Luke and Keller 2007) we also compare the processing time ratio from the two GPUs. It can be noticed in Figure 4 that depending on the GPU used, the processing time can be as fast 924x times when the data size is over 1 million. Figure 5 compares the computational efficiency and the processing time ratios of the GPUs in implementing the FCM algorithm for various sizes of clusters and dimensions. This comparison shows that as the size of the dimensions increase, the performance of the GPUs drop slightly, still being about 19x to 31x times faster than the CPUs. This drop is attributed to the scheme of our implementation, where we minimize the use of distance textures for any size of dimensions. By doing so, the implementation becomes more generic and scalable to any number of dimensions. We also used the complete set of yeast gene expression data, which had 79 dimensions and 65k observations, and the results are summarized in Table 2. The P_t ratio which compares the CPU to GPU processing times shows that our implementation outperforms the results from the previous implementation (Anderson, Luke and Keller 2007), for both low and high dimensional data.

Cluster Size	GeForce 8800 GTX		GeForce 8500 GT	
	Efficiency	P_t Ratio	Efficiency	P_t Ratio
4	19.5	112.5	12.5	24.4
8	20.8	129.4	26.8	60.6
16	21.7	141.8	20.7	41.2
32	21.9	137.9	34.2	73.1
64	23.5	131.0	22.1	35.2

Table 2: Comparison of Computational Efficiency and Processing Time between the Two GPUs

5.4 Novelties in our GPU-FCM Implementation and Comparisons

The following are the novelties in our FCM implementation:

1. For any size of *d*, we use only two input textures for data transfer and distance computations. So the number of texture sizes depends only on the data size and not on the dimensions. Thus the implementation is scalable and there is no need to expand the number of textures and change the fragment shader codes due to increase in *d*.

In the previous implementation (Anderson, Luke and Keller 2007), individual textures are used to pack the data sets thus limiting the number of dimensions allowed in a texture, while increasing the number of data objects. In this scheme, computation on any data object would require to access a number of textures. For instance, if there are 64 dimensions then in the previous implementation they would divide the dimensions into groups of 4 each, perform partial computations on each group followed by the final computation on these partial computations. But in our scheme, we store each data object in its entirety in a single texture, thus avoiding the extra computation required to perform the final computation from partial computations. The fragment program for such a scheme is complex.

2. During membership computation in FCM effectively only one cluster centre is involved at a time, so there is no need to maintain a huge membership texture of size $c * n$. In our implementation we use an $\sqrt{n} * \sqrt{n}$ texture per cluster centre per membership computation and we repeat this *c* times. It helps in better management of GPU memory (textures).

3. In the membership computation step, the summations of the ratio of deviations of each data point to the cluster centre and the deviation of each data point to the previous cluster centre are stored in an $\sqrt{n} * \sqrt{n}$ texture per cluster centre. The novelty is that all the *c* textures are simultaneously raised to the power of $2/(m-1)$. This helps in speeding up of the computations in the GPU.

5.5 Research Challenges

In our implementation of the FCM in GPU, we find the following research challenges:

1. The fuzzy cluster partial sums are transferred from the GPU to the CPU and the fuzzy centroids are updated in the CPU. This GPU to CPU data transfer time is about 70 to 77% of the total computational time taken by the GPU, which is a potential issue for further research and shader optimization.

2. The computational efficiency varies with the data size-*n*, number of dimensions-*d* and number of clusters-*c*. We intend to determine which of these factors influence the GPU efficiency the most, so that textures and shaders could be rearranged to maximize efficiency. From the studies conducted so far using GeForce 8800 GTX, we use regression analysis to identify the significance and contribution of each of these factors to the computational efficiency of GPU. Regression analysis showed that all the four factors including the intercept are significant with 95% confidence and the R^2adjusted is 97.4%. The coefficients of the factors are plotted in Figure 6 for relative comparison.

From Figure 6 it can be noted that the dimension size *d* and the cluster size *c* have more influence on the GPU efficiency. The positive coefficients denote that as *d* and *c* in a given data set increases, the GPU is expected to be more efficient in performing the computation.

3. The use of very large sets of data may pose a limitation on execution speed depending on the size of the graphical memory. The speed also depends on the graphics processor hardware being used. The Nvidia 8800 used in our implementation supports textures of sizes up to 8192 x 8192; which means data sizes as big as 67 million could be handled. More care should be taken while implementing shaders to handle huge data sets which may exceed the size of the textures, which can be further explored. It is also vital to note the number of textures that could be simultaneously used by the fragment shader depends on

the number of independent stream processors available in the GPU used.

These issues will lead us into the next stage of research to identify and generalise suitable implementation architectures for FCM and other clustering problems. The optimisation of FCM by selecting optimal number of clusters using standard FCM performance indices is one such application. Experiments will be conducted to explore the limitations posed by the GPU fragment processors and size of textures vs. the size of input data.

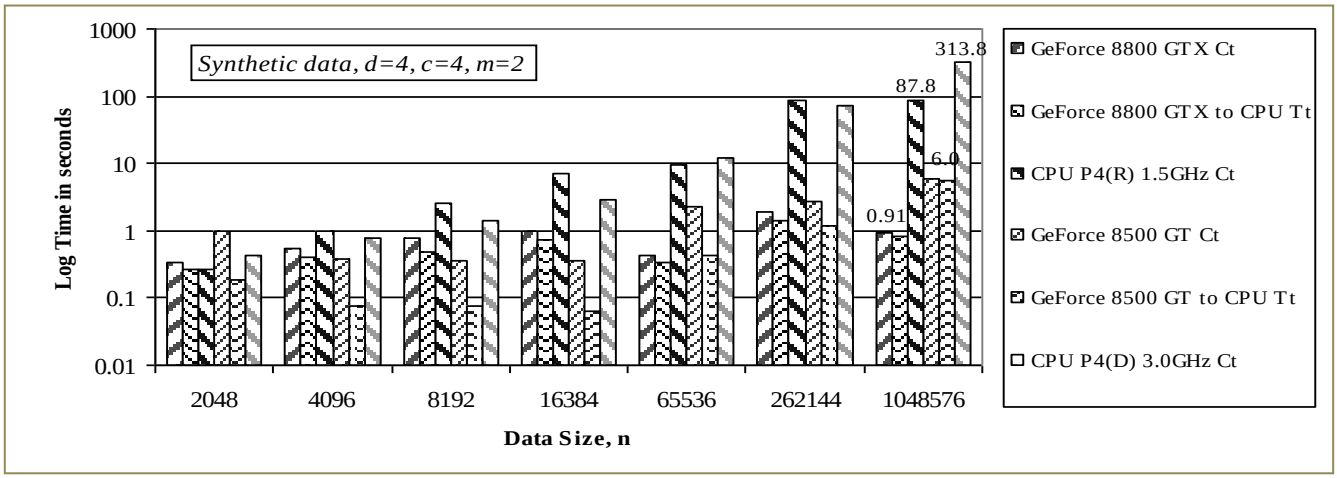

Figure 3: Comparison of Raw Computational Time between the GPU and CPU in Implementing FCM

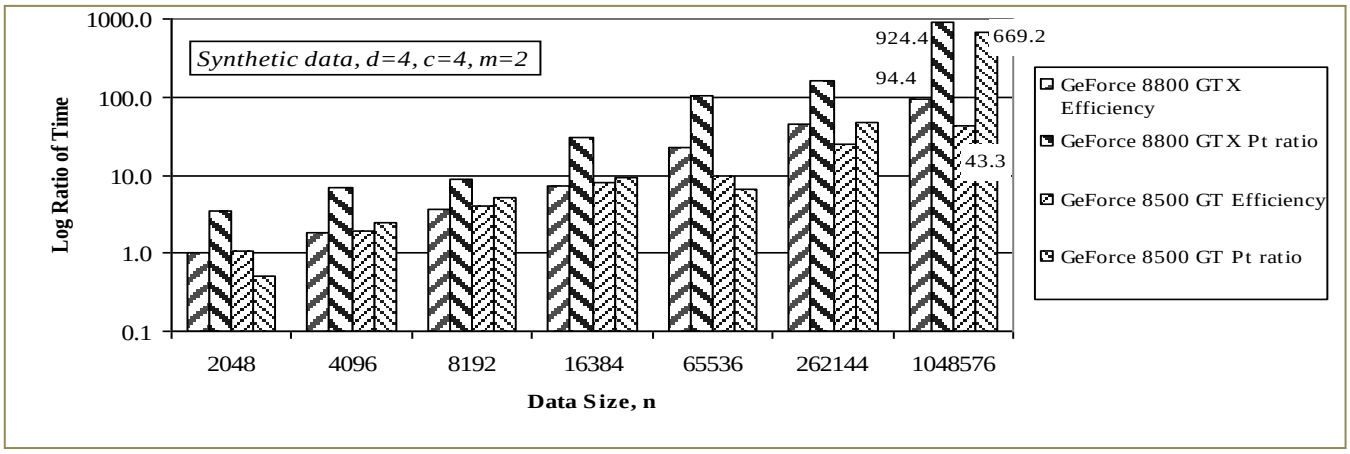

Figure 4: Comparison of Computational Efficiency and Processing Time between the two GPUs used in Implementing FCM

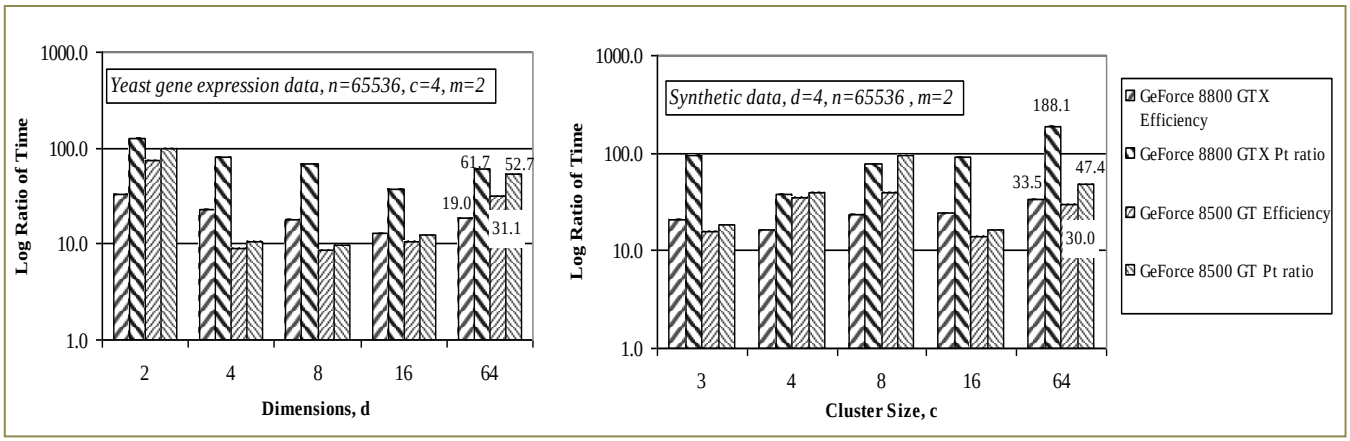

Figure 5: Comparison of Computational Efficiency and Processing Time between the two GPUs used in implementing FCM with Various Dimension Sizes and Cluster Sizes.

6 Conclusion and Future Extendibility

The scheme we have devised for implementing the FCM algorithm in GPU is giving very significant gains over its counterpart CPU. Speed gains up to 140x times on GPU 8800GTX and up to 73x times on GPU 8500GT is realized. In our analysis with the earlier implementations we found that for FCM in high dimensional large data sets, many factors need to be considered very carefully to improve GPU effectiveness. For instance, if the number of dimensions is large then it benefits by keeping all the dimensions in one texture rather than splitting it into many textures. We are able to handle any number of dimensions and clusters without the need for defining new textures and change of fragment programs. Thus we make the implementation scalable.

We effectively use the GPU resources for membership computations by exponentiation all the textures per cluster centre simultaneously by using a single execution of the shader. This helped in performance boosting. Using our implementation scheme any distance metrics such as Manhattan and other non-Euclidean distances can be implemented easily in our program, without the need to change other fragment programs which do involve in distance computations.

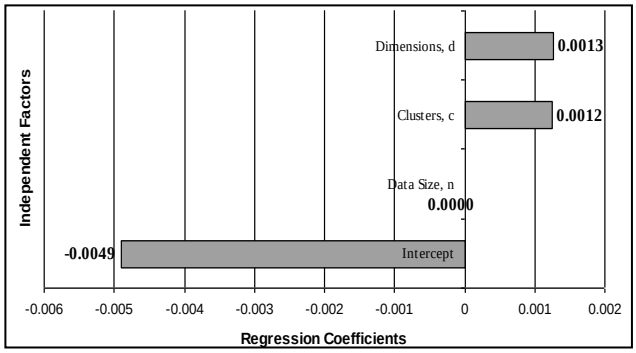

Figure 6: FCM Factors Influencing Computational Efficiency of GPU

Transfer of cluster summation data in iterations to the CPU for computing the new cluster centres takes about 77% of the total time while performing the computation in the GPU. Instead, this computation can also be implemented in the GPU so as to reduce this heavy overhead due to data transfer, especially when handling high dimensions and possibly large number of clusters.

The shader programs we have used for this implementation of FCM are not optimised for the latest GeForce 8800 GTX and beyond. The decoupling of mathematical operations and the texture operations of the latest GPU architectures will be utilized to further improve the efficiency, leveraging on CUDA. Performance index computations on GPU to identify optimal number of fuzzy clusters will also be investigated.

7 References

Jain, A.K., Murty M.N., and Flynn P.J. (1999): Data Clustering: A Review, *ACM Computing Surveys*, Vol 31, No. 3, 264-323.

MacQueen, J. B. (1967): Some Methods for classification and Analysis of Multivariate Observations, In *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability*, Berkeley, University of California Press, 1:281-297.

Guha, S., Rastogi, R., and Shim, K. (1998): CURE: An efficient clustering algorithm for large databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 73--84, New York.

Ester, M., Kriegel, H., Sander, J., Xu, X., (1996): A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise, In *Proceedings of 2nd International Conference on KDD.* AAAI Press.

Bezdek, J. C. (1981): *Pattern Recognition with Fuzzy Objective Function Algorithms*. Plenum Press, New York.

Free Encyclopaedia: GNU Free Documentation, Wiki Software, http://www.wikipedia.org/. Accessed 20 May 2008.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., Purcell, T. J. A (2005): Survey of General-Purpose Computation on Graphics Hardware. *Eurographics*.

Fatahalian, K., Sugerman, J., Hanrahan, P. (2004): Understanding the efficiency of GPU algorithms for matrix-matrix multiplication, In *Proceedings of the ACM SIGGRAPH/ Eurographics conference on Graphics hardware*.

Arul, S., Dash, M., and Tue, M. (2008): GPU-based fast *k*-means clustering of gene expression profiles", In *Proceedings of 12th Annual International Conference on Research in Computational Molecular Biology (RECOMB.* Singapore.

Arul, S., Dash, M., and Tue, M. (2008): Efficient *K*-means Clustering Using Accelerated Graphics Processors, Accepted for *International Conference on Data Warehousing and Knowledge Discovery (DAWAK)*.

NVIDA: GeForce 8800 Architecture Technical Brief, http://static.tigerdirect.com/pdf/NVIDIA_GeForce8800_GPU_Architecture_Technical_Brief.pdf. Accessed 12 January 2008.

Harris, C. Haines, K. (2005): Iterative Solutions using Programmable Graphics Processing Units, In *Proceedings of the 14th IEEE International Conference on Fuzzy Systems*, pages: 12- 18.

Anderson, D., Luke, R. H., Keller, J. M. (2007): Incorporation of Non-Euclidean Distance Metrics into Fuzzy Clustering on Graphics Processing Units, *Analysis and Design of Intelligent Systems using Soft Computing Techniques*.