Incorporation of Non-Euclidean Distance Metrics into Fuzzy Clustering on Graphics Processing Units

*# Derek Anderson[1], *# Robert H. Luke[2], and * James M. Keller[3]

* Department of Electrical and Computer Engineering
University of Missouri-Columbia

# Biomedical and Health Informatics Research Training Program,
National Library of Medicine

[1]dtaxtd@missouri.edu, [2]rhl3db@missouri.edu, and [3]kellerj@missouri.edu

Abstract. Computational tractability of clustering algorithms becomes a problem as the number of data points, feature dimensionality, and number of clusters increase. Graphics Processing Units (GPUs) are low cost, high performance stream processing architectures used currently by the gaming, movie, and computer aided design industries. Fuzzy clustering is a pattern recognition algorithm that has a great amount of inherent parallelism that allows it to be sped up through stream processing on a GPU. We previously presented a method for offloading fuzzy clustering to a GPU, while maintaining full control over the various clustering parameters. In this work we extend that research and show how to incorporate non-Euclidean distance metrics. Our results show a speed increase of one to almost two orders of magnitude for particular cluster configurations. This methodology is particularly important for real time applications such as segmentation of video streams and high throughput problems.

1. INTRODUCTION

Many pattern recognition algorithms, such as clustering, can be sped up on platforms that utilize multiple processing cores, operate efficiently for large amounts of floating point operations (FLOPS), and natively support numerically expensive linear algebra instructions. Graphics Processing Units (GPUs) are relatively new general-purpose stream processing hardware that are well suited for these types of problems. When presented with a collection of input data, such as pixels in an image, stream processing computes a function on the entire collection, much like a kernel operation, where each stream element is assumed to be independent of all other elements. This framework naturally lends itself to image processing, but there has recently been a trend of converting many pattern recognition algorithms to GPU programs for computational speedup. However, the conversion process is typically not trivial. In order to be transferred to a GPU, most algorithms must be reformulated. In addition, GPU hardware is changing at a fast rate and programming languages are not as mature as desired. This all equates to a slight GPU programming learning curve.

Improving the computational performance of clustering is not a new concept. Shankar and Pal presented a progressive subsampling method called fast fuzzy c-means (FFCM) [1]. FFCM generates a sequence of extended partitions of the entire data set by applying the original FCM to a nested sequence of increasing size subsamples. FFCM terminates when the difference between successive extended partitions is below a single threshold value. Speed is always a concern, but so is the size of the data set. In [2] Pal and Bezdek developed the extensible fast fuzzy c-means clustering (eFFCM) algorithm for the segmentation of very large digital images. In [3] Hathaway and Bezdek discuss an extension of the eFFCM method, geFFCM, to non-image object data. It should be made clear that we are presenting a procedure to transfer fuzzy clustering to specialized hardware, which is different from many previous attempts that look to find algorithmic or mathematical reformulations. As improvements are made to the clustering algorithms, these concepts can be converted to a GPU implementation. Our research goal is to present a different computational platform for clustering.

Harris and Hanes conducted the first known work in the area of offloading clustering to a GPU in [4]. In their work, a speedup of 1.7 to 2.1 times over a CPU is reported for a NVIDIA GeForceFX 5900 Ultra.

Their method is designed to handle three linearly separable clusters with a dimensionality of three. The benefit is that they can handle a large number of data points. A problem is that the proposed formulation is not scalable. As the method is presented, it is not capable of extension with respect to either the feature dimensionality size or number of cluster centers. They acknowledge this and state that it would be a large undertaking. They instead proposed to look into increasing efficiency before dealing with the problem of control over the various fuzzy clustering parameters.

In [5] we presented a generalized method for offloading fuzzy clustering, in particular, the Fuzzy C-Means (FCM) to a GPU, allowing for full control over the various clustering parameters. A computational speedup of over two orders of magnitude was observed for particular clustering configurations, i.e. variations of the number of data points, feature dimensionality, and the number of cluster centers. The metric used was the Euclidean distance. Here, we extend that research and show how to incorporate non-Euclidean distance metrics. This changes the parameter representational scheme some, but the algorithmic formulation has the biggest modifications.

The focus of this paper is the FCM, but many high throughput problems, like protein sequence structure search, are excellent candidates for GPU enhancement. We are currently using GPUs to speed up image processing. In particular, we employ GPUs for human silhouette segmentation for eldercare activity analysis and fall detection. Silhouette segmentation requires image pre-processing, feature extraction (color and texture) in various color spaces, data fusion, shadow detection and removal, and post-processing (such as morphology). Image processing is significantly sped up as long as the problem is formulated as one such that the image data remains on the GPU, i.e. avoid CPU to GPU memory transfers. In the following sections we (1) present an overview of the FCM, (2) provide an introduction to GPUs, (3) discuss the generalized algorithm for computing fuzzy clustering on a GPU with a non-Euclidean distance metric, (4) present the experiments, (5) display the results, and (6) discuss extensions to this work.

2. CLUSTERING

Clustering is an unsupervised learning procedure that can be used to reveal patterns in a collection of data, denoted by $X = \{\bar{x}_1, \ldots \bar{x}_N\}$. Each sample vector contains K features, represented as $\bar{x}_i = (f_{i1}, \ldots f_{iK})^T$. Each cluster can be represented by a set of parameters, θ_j $(1 \leq j \leq C)$. In the simplest case, θ_j is a K-dimensional vector representing the jth cluster center. In the standard approach [6], the clustering algorithm alternately estimates the collection of cluster centers, $\theta = \{\theta_1, \ldots \theta_C\}$, and a membership matrix, U, where the membership of the ith sample in the jth cluster is denoted by $u_{(i,j)}$. In the Hard C-Means (HCM) clustering algorithm, cluster membership values are crisp, i.e. $u_{(i,j)} \in \{0,1\}$. Fuzzy clustering allows $u_{(i,j)} \in [0,1]$, i.e. each element can be shared by more than one cluster. Fuzzy clustering follows the principle of least commitment, which is the belief that one should never over commit or do something which might have to be later undone [7]. The Fuzzy C-Means (FCM) cost function for a C cluster problem, originally proposed by Bezdek [6], is defined as

$$J_{FCM}(\theta, U) = \sum_{j=1}^{C} \sum_{i=1}^{N} u_{(i,j)}^{q} d\left(\bar{x}_i, \theta_j\right)$$

where $\sum_{j=1}^{C} u_{(i,j)} = 1$ for $i = 1, \ldots N$

$\bar{x}_i \in X$ and $|X| = N$.

In the J_{FCM} equation, $d(\bar{x}_i, \theta_j)$ is a distance metric and $q > 1$ is a parameter called the fuzzifier, typically $q = 2$. The update equations, found as necessary conditions to minimize J_{FCM}, for the membership values, $u_{(i,j)}^{FCM}$, cluster centers and covariances, $\theta_j = (\bar{\mu}_j, \Sigma_j)$, are

$$u_{(i,j)}^{FCM}(t) = \frac{1}{\sum_{k=1}^{C} \left(\frac{d(\bar{x}_i, \theta_j(t))}{d(\bar{x}_i, \theta_k(t))} \right)^{\frac{1}{q-1}}} \tag{1}$$

$$\bar{\mu}_j(t+1) = \frac{\sum_{i=1}^{N} \left(u_{(i,j)}^{q}(t) \bar{x}_i \right)}{\sum_{i=1}^{N} u_{(i,j)}^{q}(t)} \tag{2}$$

$$\Sigma_j(t+1) = \frac{\sum_{i=1}^{N} \left(u_{(i,j)}^{q}(t)(\bar{x}_i - \bar{\mu}_j(t+1))(\bar{x}_i - \bar{\mu}_j(t+1))^T \right)}{\sum_{i=1}^{N} u_{(i,j)}^{q}(t)} \tag{3}$$

The update equations depend on the current iteration, shown as t above. The membership values and the data points are used at each iteration to compute the next cluster centers (equation 2). The new cluster centers and the memberships are then used to update the covariances (equation 3). When we initially presented the FCM on a GPU, we used the Euclidean distance metric. In this paper we use both the Mahalanobis distance, $d_M(\bar{x}_i, \theta_j)$, and the Gustafson-Kessel (GK) distance, $d_{GK}(\bar{x}_i, \theta_j)$. At the current time, we restrict the covariance matrices to be diagonal to simplify the storage and computation.

$$d_M(\bar{x}_i, \theta_j) = \left((\bar{x}_i - \bar{\mu}_j)^T \Sigma_j^{-1} (\bar{x}_i - \bar{\mu}_j) \right)^{1/2} \tag{4}$$

$$d_{GK}(\bar{x}_i, \theta_j) = \left(|\Sigma_j|^{1/K} \left((\bar{x}_i - \bar{\mu}_j)^T \Sigma_j^{-1} (\bar{x}_i - \bar{\mu}_j) \right) \right)^{1/2} \tag{5}$$

3. GRAPHICS PROCESSING UNITS

Traditionally, graphics operations, such as mathematical transformations between coordinate spaces, rasterization, and shading operations have been performed on the CPU. GPUs were invented in order to offload these specialized procedures to advanced hardware better suited for the task at hand. Because of the popularity of gaming, movies, and computer-aided design, these devices are advancing at an impressive rate, given the market demand. The key to programming for a GPU is in the design of the CPU driver application, understanding of the graphics pipeline, and ability to program for GPU processors. Figure 1 shows the process of rending a triangle in the graphics pipeline.

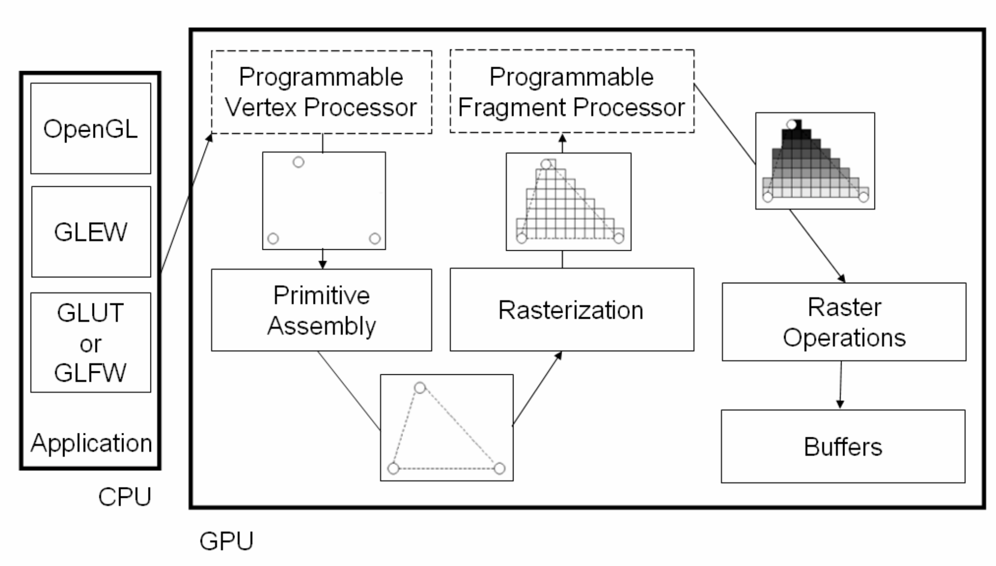

Fig. 1. Graphics pipeline stages for rendering a triangle. Programmable GPU components shown as dashed boxes.

The traditional rasterization and shading process, shown in Figure 1, begins with the specification of vertices. The vertices form a primitive that undergoes rasterization. Rasterization results in a set of fragments, which are then subject to shading. Currently there are three programmable components on a GPU. The first programmable component is the vertex processor. Vertex processors are traditionally responsible for mathematical transformations between coordinate spaces and texture coordinate generation. The next programmable unit, introduced in DirectX 10 and the Shader Model 4, is the geometry processor, which takes the transformed vertices and allows for per-primitive operations. The last programmable unit, the fragment processor, also sometimes called the pixel shader, is traditionally responsible for sampling and applying textures, lighting equations, and other advanced graphics and image processing effects. There are traditionally more fragment processors, because a few vertices are responsible for a larger number of fragments. A good GPU introduction can be found in [8, 9].

GPUs are increasing at a faster rate, in terms of computational power, than CPUs. The annual growth in CPU processing power, as empirically noted by Moore's law, is approximately 1.4, while vertex processors are growing at a rate of 2.3 and pixel processors are growing at a rate of 1.7 [10]. GPUs also support native operations such as matrix multiplication, dot products, and computing the determinant of a matrix. GPUs are also capable of executing more floating point operations per second. A 3.0 GHz dual-core Pentium4 can execute 24.6 GFLOPS, while an NVIDIA GeForceFX 7800 can execute 165 GFLOPS per second [10]. The new NVIDIA GeForce 8800 GTX has 128 stream processors, a core clock of 575 MHz, shader clock of 1350 MHz, and is capable of over 500 GFLOPS [11].

4. FUZZY CLUSTERING ON A GPU

The quintessential concept required to perform general-purpose computing on a GPU is that arrays are equivalent to textures. This means that pattern recognition algorithms must be structurally converted into a suitable texture format. Textures are combinations of red, green, blue, and alpha (RGBA) values. In [5] we presented the initial fuzzy clustering parameter texture packing scheme, where the metric was the Euclidean distance. We also provided a detailed section that describes implementation of GPU algorithms discussed below. Figure 2 shows our format for fuzzy clustering on a GPU with the Mahalanobis and GK distance metrics, which requires that the cluster covariances are stored, in addition to the cluster centers.

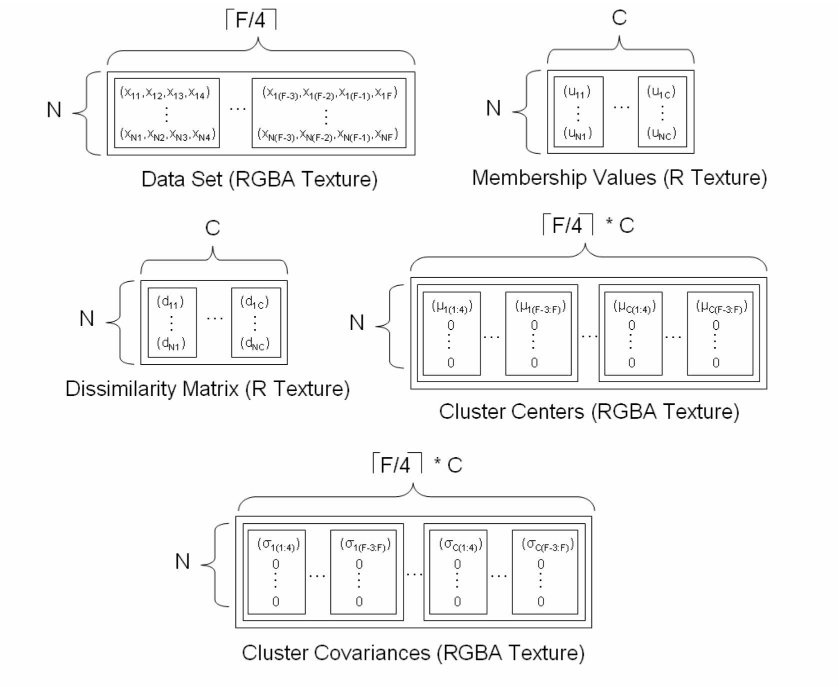

Fig. 2. Fuzzy clustering parameters packed into texture memory on a GPU. Textures above are a combination of red, green, blue, and alpha (RGBA) data. In this proposed format, used to simplify the representation and resulting computation, only the diagonal of the covariance matrices are stored.

The basic idea is that the data set is packed into a single RGBA texture that has N rows, one for each data point, and $\lceil F/4 \rceil$ columns, where $\lceil * \rceil$ represents the "ceiling" of *, i.e. the smallest integer greater than or equal to *. F is the feature dimensionality, which is divided by four because four values can be packed into each pixel (RGBA). Presently, most GPUs are limited to textures of size 4096x4096, but newer GPU models, such as the NVIDIA 8800, are starting to support textures of size 8192x8192. This means that up to 8,192 data points of up to 32,768 dimensionality can be packed into a single texture. In [5] we also presented a method to bypass this texture row size limitation, allowing for much larger data sets through the packing of elements into columns sets. The resulting data set texture will therefore have $\lceil F/4 \rceil \times S$ columns, where S is the number of column sets. Using a NVIDIA 8800 we can support profiles such as 4,194,304 data points of dimensionality 4 with 4 cluster centers or 131,072 data points of dimensionality 32 with 16 cluster centers. This column-set packing scheme is shown in Figure 3.

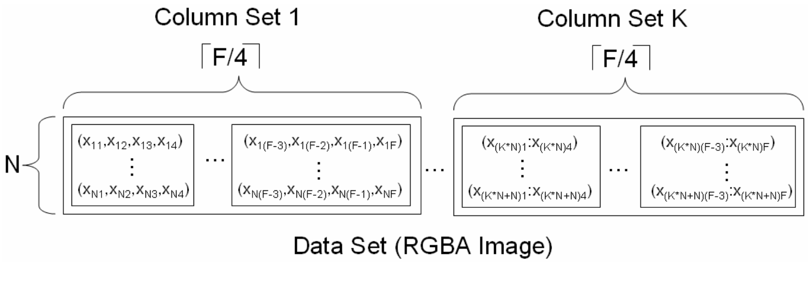

Fig. 3. Packing of samples into multiple columns in the data set texture in order to support a greater number of samples.

The membership and dissimilarity textures are N rows by C columns, where C represents the number of clusters. If multiple column sets are being used, then these matrices have $C \times S$ columns. Each element in these matrices is a scalar, which is packed into a single color channel (i.e. red color channel). The cluster centers are packed into a column format, unlike the row packing scheme for the data set. The number of columns is $\lceil F/4 \rceil \times C$, or $\lceil F/4 \rceil \times C \times S$ for the column set packing method. The number of rows is equal to the number of input vectors. This will be described shortly when we introduce GPU reduction. As mentioned above, the Mahalanobis and GK distance metrics require covariance matrices. For simplicity in representation and in computation, we only store the covariance matrix diagonals. This is the only addition to our previous packing scheme [5], but entails important changes in the ordering and number of GPU programs, shown below in Figure 4. The new FCM GPU algorithm for the Mahalanobis and GK distance metrics is shown in Figure 4.

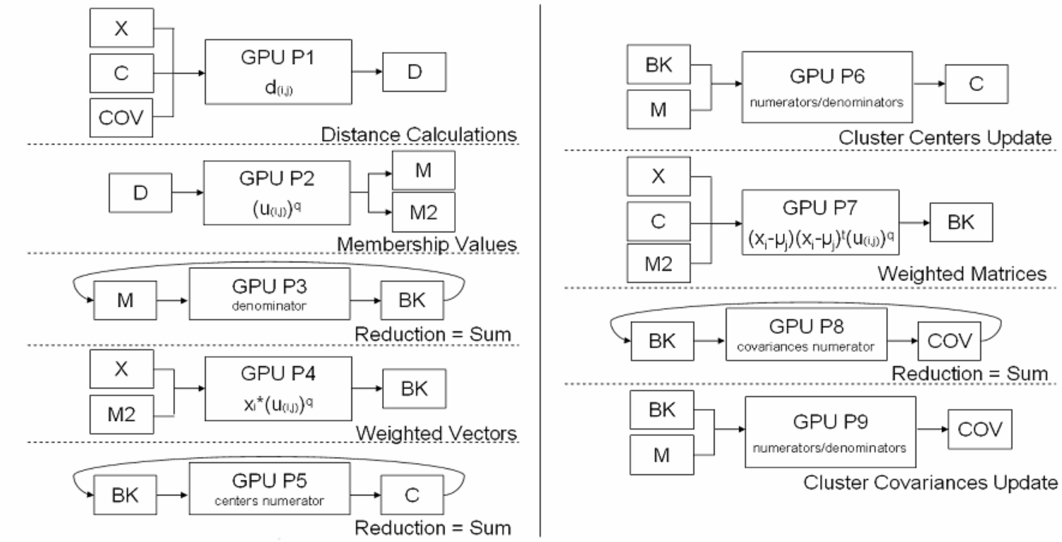

Fig. 4. Generalized GPU algorithm for fuzzy clustering with the Mahalanobis or GK distance metrics. X is the data set texture, C is the cluster center texture, M stores the membership values, COV contains the diagonals of the covariance matrices, D is the dissimilarity matrix, and M2 and BK are temporary buffers. GPU P1, program 1, calculates the dissimilarity, GPU P2 is the membership update step, GPU P3 performs a reduction on the membership matrix, GPU P4 computes the numerator terms in the cluster center update equation, GPU P5 performs a reduction to sum up the numerator terms, GPU P6 computes the new cluster centers, GPU P7 computes the numerator terms in the covariance update step, GPU P8 performs a reduction on the covariance numerator terms, and GPU P9 computes the new covariance matrices.

GPU P1, program 1, shown in Figure 4, computes the dissimilarity values. By only using diagonals of covariance matrices, the metrics are simplified and can be computed fast on the GPU, given its vector based design. The Mahalanobis distance requires (a) vector subtraction, (b) matrix inversion, which for a diagonal matrix is $\left(1/\sigma_{(r,r)}\right)$, for r=1 to K, where $\sigma_{(r,r)}$ is the rth covariance matrix diagonal term, (c) per-component vector multiplication, where the $\left(\vec{x}_i - \vec{\mu}_j\right)$ result is multiplied by the diagonal inverse terms, (d) inner product, and (e) the square root, which is a function supported by the GPU. Most of these operations can have four vector components computed in parallel, given a GPU's support for vectors of size four. The GK metric requires computing the determinant of $\sum_j$, which for a diagonal matrix is the product of the diagonals. The determinant value is raised to $(1/K)$ by the POW function on a GPU.

The following GPU passes use the memberships raised to the qth power. Thus, we compute the membership values and raise them to the qth power to avoid re-computing these values. GPU P2 computes these terms and stores the results in two textures. The reason for two textures is so that one can be used for reduction on the denominator in equations 2 and 3. A reduction in this context is the repeated application of an operation to a series of elements to produce a single scalar result. The reduction procedure here is the addition of a series of elements in texture memory, where operations are running in parallel given the number of fragments processors on a particular GPU. Reduction takes $\log_2(N)$ total passes, where each iteration, i, processes $N/2^i$ stream elements. The reduction procedure is shown in Figure 5.

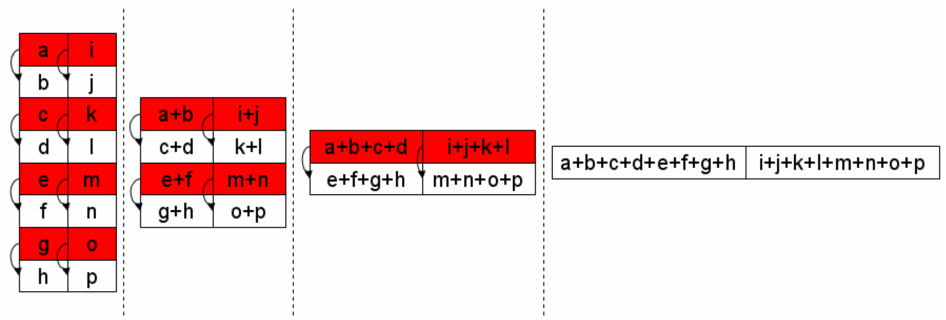

Fig. 5. GPU reduction procedure, summation operation, for a texture with two columns and 8 rows. At each step, $\log_2(N)$ total steps, $N/2^i$, where i is the iteration index, elements in a single column are pair-wised summed. Each column results in the summation of values for a respective column.

GPU P3 performs membership value reduction, i.e. the denominator of equations 2 and 3. GPU P5 and P8 use reduction to compute the sum of the numerator terms in equations 2 and 3 respectively. GPU P6 and P9 divide the reduced (summed) numerators by the reduced (summed) denominators, resulting in the updated cluster centers and covariance matrices.

The cluster centers and covariance matrices textures were shown in Figure 2 to contain zeros in all texture positions except for the first row. This is shown in order to stress the fact that only the first row stores the centers and diagonals of the covariance matrices. GPU P4 and P7 populate the cluster center and covariance matrices with the numerator terms in equations 2 and 3 respectively. The reason for all of the extra rows is to perform the reduction procedure. It is not necessary to initialize the cluster centers or covariance matrices with zeros in all rows after the first. It is also faster to not initialize the matrices past the first row. The only data that needs to be passed down to the GPU, as well as transferred back at the end of the FCM GPU algorithm, is the first row in the cluster centers and covariance matrices textures.

If using the diagonal of the covariance matrices is not acceptable, then the entire covariance matrix can be packed into texture memory and used by the GPU. Figure 6 shows a proposed texture packing scheme.

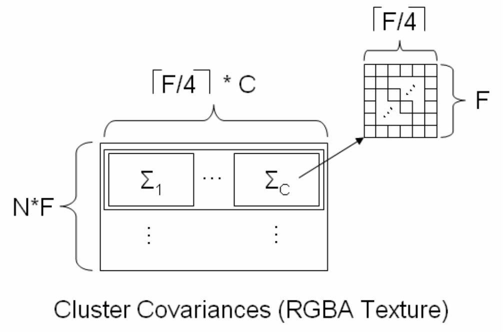

Fig. 6. Full covariance matrix packed into texture memory.

If the full covariance matrix is used then less data points can be stored. This results because of the way that we perform reduction. Each element no longer occupies a single row, but rather F rows. The following changes will also need to be made if the full covariance matrix is used. GPU P1 will need to compute the matrix inverse and the matrix determinant. If the feature dimensionality is less than 5, the GPU already has a determinant function. GPU P7 will now have to compute and store the outer product. Because a GPU can only write out to the pixel that it is currently shading, i.e. not neighboring pixels, the covariance matrix will have to be computed in parts. The reduction will also have to be varied to compute a reduction for blocks of memory, i.e. the full covariance matrices. These are the reasons that drove us to use the matrix diagonal, for computational and representational simplicity.

5. EXPERIMENTS

There is a trend in general-purpose GPU publications that implementation details are extremely vague, almost to the point of being non-reproducible. This provides little benefit to the community, and does not allow others to implement the procedures. We are making the source code, along with documentation, available at http://cirl.missouri.com/gpu/. You can refer to [5] or the web site in order to find out details regarding implementation, such as (1), how to use Cg, NVIDIA's C for Graphics GPU programming language, (2) how to use Frame Buffer Objects (FBO), which are used in order to keep the data on the GPU and make the algorithm go fast, (3) Pixel Buffer Objects (PBO), which can be used to speed up CPU to GPU memory (texture) transfers, (4) GPU numerical precision, such as 16bit and 32bit precision, (5) texture coordinate transformations on the GPU for texture data access, and (6) executing a GPU program, which breaks down to rendering a screen aligned quadrilateral.

There is a tradeoff in terms of time spent setting up the Cg programs, transferring texture memory from CPU to GPU, and managing the graphics pipeline. This means that there are points where it is more or equally efficient to implement the FCM on the CPU rather than on the GPU. GPUs are the most efficient when large batches of data are presented to them. We vary the number of data points, feature dimensionality, number of cluster centers, and distance metric to show performance results for various clustering profiles. The tables below show the speedup of clustering on a GPU, but do not reflect the fact that the CPU is freed up to perform additional computing. This means that one machine can be used for other computation at the same time or can be used for clustering on the CPU and GPU simultaneously, hence increasing the productivity of the entire platform.

Two unique CPU models and three unique GPU models are benchmarked below. The idea is to show differences as it relates to price, computing power, and manufacturer. We used two CPU's: (1) Intel Core 2 Duo T7200 and 2 GB of system RAM and (2) AMD Athlon 64 FX-55 and 2 GB of system RAM. The two GPUs were: (1) NVIDIA Quadro FX 2500M with 512 MB of texture memory, 24 fragment pipelines, and PCI Express X16, and (2) NVIDIA 8800 BFG GTX with 768 MB of texture memory, 128 stream processors, and PCI Express X16.

Our operating system is Windows XP with Service Pack 2 and we are running Visual Studio 2005. The GLEW version is 1.3.4, GLUT version 3.7.6, and Cg version 1.5. Streaming SIMD Extensions 2 (/arch:SSE2), whole program optimization, and maximize speed (/02) were enabled for the CPU in Visual Studio. Because we are not presenting any new metrics or clustering algorithms but rather a speedup method for fuzzy clustering, we use randomly generated clusters. No common data sets from the community were used. We needed many cluster configurations, so we generated elliptical clusters of different sizes with random means. In order to test the GPUs precision, we compared the GPU results to a CPU FCM implementation we wrote, and also to the MATLAB fcm function. In our C implementation we use the same 32bit floating point precision used in our GPU implementation. The C program has the same algorithmic design and final precision as our GPU program. We perform 100 iterations of the FCM on the CPU and the GPU in order to provide a fair comparison.

6. RESULTS

Tables 1 and 2 show CPU over GPU processing time ratios for a clustering task where there are 32 feature dimensions. We used a single 4,096 size row, which is the common max row texture size among the various GPUs that were used for benchmarking.

	GPU1 (2500M)	**GPU2 (8800)**
CPU1 (32bit)	8.51	76.45
CPU2 (64bit)	10.32	83.63

Table 1. CPU/GPU processing time ratio for 4096 points, 64 clusters, 32 dimensions and the Mahalanobis distance.

	GPU1 (2500M)	GPU2 (8800)
CPU1 (32bit)	8.06	74.35
CPU2 (64bit)	9.69	79.74

Table 2. CPU/GPU processing time ratio for 4096 points, 64 clusters, 32 dimensions and the GK distance.

Tables 1 and 2 show impressive computational speed improvements for a GPU versus the CPU. Depending on the particular GPU and CPU, speed improvements of one to almost two orders of magnitude are observed. The performance numbers are entirely dependent on the GPU generation and CPU that it is compared to. As each new generation of GPU emerges, performance numbers are expected to increase given the popularity of these devices and the need for stream processing. Little computational difference is noticed between the two distance metrics. Table 3 shows the performance behavior when we keep the dimensionality and distance metric fixed, but let the number of clusters and data points vary, comparing the best CPU and GPU.

		Number of Clusters		
		4	16	64
Data Points	64	0.15	0.91	8.96
	256	0.52	2.67	30.36
	512	1.19	5.16	52.11
	1024	2.26	9.96	63.37
	4096	6.79	42.66	88.21
	8192	12.29	78.25	97.55

Table 3. CPU/GPU processing time ratio trend for the 32bit Intel and NVIDIA 8800. The dimensionality is fixed at 4, the number of data points and clusters are varied for the GK distance metric.

The results in Table 3 indicate that when a small number of clusters and data points are pushed down to the GPU performance gain is minimal. In some cases the CPU is even faster. However, as the number of data points increase, the GPU becomes faster. The largest speed improvements are noticed as the number of clusters is increased.

The last performance result is for a larger number of samples. We ran the program on 409,600 samples for both the NVIDIA 8800 and the 32bit Intel CPU. The row size was 4,096, there were 100 column sets, the dimensionality was 8, and there were 4 cluster centers. The time to process the data was 0.94 seconds.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we extended our previous research related to transferring fuzzy clustering to a GPU, specifically as it relates to using different distance metrics. The texture representation change is minimal, while the order and number of GPU program passes is noticeably different. While we specifically described the FCM algorithm, the general outline can be adapted to many other heavy computational techniques. This opens up the potential to perform clustering (and other) algorithms on large data sets at low cost and in a somewhat real time environment without the need for clusters of computers or other expensive dedicated hardware, for example, segmenting a continuous video stream or for bioinformatics applications (like BLAST searches).

As stated in [5], we plan to continue this line of research and find out how a cluster of low end PCs equipped with GPUs perform for larger clustering tasks. Another area of future extension surrounds very large data sets. We intend to take the eFFCM work described by Hathaway and Bezdek and implement it on a single GPU, cluster of PCs equipped with GPUs, or multiple 8800 GPUs on a single machine connected through SLI.

8. ACKNOWLEDGEMENTS

D. Anderson and R. Luke are pre-doctoral biomedical informatics research fellows funded by the National Library of Medicine (T15 LM07089).

9. REFERENCES

1. B. Uma Shankar and N. R. Pal, "FFCM: An effective approach for large data sets," *Proc. 3rd Int. Conf. Fuzzy Logic, Neural nets, and Soft Computing*, IIZUKA, Fukuoka, Japan, 1994, 332-332
2. N. R. Pal and J. C. Bezdek, "Complexity reduction for "large image" processing," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 32, 2002, 598-611.
3. R. J. Hathaway and J. C. Bezdek, "Extending fuzzy and probabilistic clustering to very large data sets," *Computational Statistics & Data Analysis*, 51 (2006), 215-234.
4. C. Harris and K. Haines, "Iterative Solutions using Programmable Graphics Processing Units," *14th IEEE International Conference on Fuzzy Systems 2005*. May, 2005, 12-18.
5. D. Anderson, R. Luke, and J. M. Keller, "Speedup of Fuzzy Clustering Through Stream Processing on Graphics Processing Units", *IEEE Transactions on Fuzzy Systems*, 2006, in review.
6. J. C. Bezdek, Pattern Recognition with Fuzzy Objective Function Algorithms, New York, NY: Plenum, 1981.
7. D. Marr, *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. San Francisco, CA: W. H. Freeman, 1982.
8. M. Pharr and R. Fernando, GPU Gems 2, Addison-Wesley, 2005.
9. GPGPU, Nov. 18 2006, <http://www.gpgpu.org/>.
10. J. D. Owens, et al., "A Survey of General-Purpose Computation on Graphics Hardware," *Eurographics 2005, State of the Art Reports*, August, 2005.
11. Nvidia Corp., "GeForce 8800," Nov. 18 2006, <http://www.nvidia.com/page/geforce_8800.html>.