Efficient *K*-Means Clustering Using Accelerated Graphics Processors

S.A. Arul Shalom [1], Manoranjan Dash[1], and Minh Tue[2]

[1] School of Computer Engineering, Nanyang Technological University, 50 Nanyang Avenue, Singapore
[2] NUS High School of Mathematics and Science, Clementi Avenue 3, Singapore
{sall0001,asmdash}@ntu.edu.sg, h0630082@nus.edu.sg

Abstract. We exploit the parallel architecture of the Graphics Processing Unit (GPU) used in desktops to efficiently implement the traditional *K*-means algorithm. Our approach in clustering avoids the need for data and cluster information transfer between the GPU and CPU in between the iterations. In this paper we present the novelties in our approach and techniques employed to represent data, compute distances, centroids and identify the cluster elements using the GPU. We measure performance using the metric: computational time per iteration. Our implementation of *k*-means clustering on an Nvidia 5900 graphics processor is 4 to 12 times faster than the CPU and 7 to 22 times faster on the Nvidia 8500 graphics processor for various data sizes. We also achieved 12 to 64 times speed gain on the 5900 and 20 to 140 times speed gains on the 8500 graphics processor in computational time per iteration for evaluations with various cluster sizes.

Keywords: *K*-means clustering, GPGPU, Computational efficiency.

1 Introduction

Commodity Graphics Processors in today's PC world are highly parallel with extreme computational powers. Modern GPU is capable of processing tens of millions of vertices per second and rasterize hundreds of millions of fragments per second. The schematic in Figure 1 shows the vertex transformation and fragment texturing and coloring stages in a typical graphics pipeline. Due to the fact that these processors are economically affordable and programmable for implementing iterative algorithms there is high possibility that desktop computers with such GPUs will soon be capable of performing fast and efficient computing.

1.1 Graphics Processors for General Purpose Computations

The factors that enable the processing power of GPUs are the inherent parallel architecture, peak memory bandwidth, high floating-point operations and the various hardware stages with programmable processors. High-end graphics processors such as Nvidia GeForce7900 GTX and GeForce8800 Ultra have peak memory bandwidth of about 51.2GB/sec and 103.68GB/sec respectively. The floating-point operations

I.-Y. Song, J. Eder, and T.M. Nguyen (Eds.): DaWaK 2008, LNCS 5182, pp. 166–175, 2008.

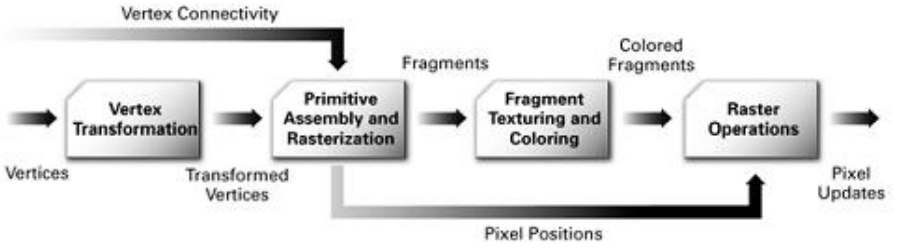

Fig. 1. The Graphics Hardware Pipeline

possible are over 200 GFlops in Nvidia GeForce7900 GTX and 345 GFlops in Nvidia GeForce8800 Ultra. The performance of the GPU in computing general-purpose algorithms depends heavily on how the algorithms are arranged so as to exploit the parallel data processing power of the GPU. In our study, we have used Nvidia's GeForce FX 5900 XT processor and a GeForce 8500 GT processor. In graphics processing, the GPU receives commands for display in the form of vertices and connectivity details from the CPU. Today's GPUs have very high memory bandwidth and parallel internal processors, which are capable to process streams of incoming data. These processing is performed in either the vertex or the fragment processor in the GPU using specific shader programs. Computations in the GPU processors are data independent, which means that the processing that occurs in each processor, is independent of the data used by other processors. Currently, there is lot of research focus in the arena of implementing general-purpose computations in the GPU (GPGPU) to leverage on speed w.r.t unit cost function [9]. Within the GPU, the fragment processors support parallel texture operations and are able to process floating-point vectors. Implementations of GPGPU are challenging and mostly utilize the texture processing abilities of the fragment processor of the GPU.

1.2 An Approach to Implement *K*-Means Algorithm in GPU

In this paper we present an efficient implementation of the *k*-means clustering algorithm completely in the GPU. We realize this by using the multi-pass rendering and multi-shader capabilities of the GPU. This is done by maximizing the use of textures and minimizing the use of shader program constants [3]. In this implementation we have minimized the use of GPU shader constants thus improving the performance as well as reducing the data transactions between the CPU and the GPU. Handling data transfers between the necessary textures within the GPU is much more efficient than using shader constants. This is mainly due to the high memory bandwidth available in the GPU pipeline. Since all the steps of *k*-means clustering could be implemented in the GPU, the transferring of data back to the CPU during the iterations is avoided [13]. The programmable capabilities of the GPU have been thus exploited to efficiently implement *k*-means clustering in the GPU.

2 Existing *K*-Means Clustering Methods Using Graphics Processors

Two existing *k*-means implementations are analyzed in this section [2, 3]. We find out how the current GPU hardware architecture could be further exploited to overcome some of the limitations faced in these implementations [2, 3, 13].

2.1 The *K*-Means Implementation with Textures and Shader Constants

The *k*-means iterative clustering method and few of its variants have been implemented in the GPU [3]. The result is a speed-up in clustering between 1.5 to 3 times compared to the CPU implementation. In this implementation each input point is stored in a single texel of a texture as a float and mapped over several textures. The cluster data is stored in fragment shader constants. The fragment processor accesses the textures to obtain the data points and the fragment constant to obtain the cluster information. The fragment processors and depth tests are used in this implementation. The functioning of the depth buffer is programmed in such a way that the resultant distance is written into the depth buffer and the cluster ID (label) of the cluster that is closest to the data point is written into the color value. The cluster data in the fragment shader constants are updated after the iteration. After all the iterations are complete the cluster IDs are read back to the CPU. In the above implementation it is notable that it is faster to read and to write the cluster data in the fragment shader constants than accessing the data in the textures. But this limits the performance when there are many clusters. Moreover, in this implementation only one clusters' data is stored in the fragment shader constants at any one time. This leads to the fact that the fragment shader constants are to be accessed very frequently for distance computations, comparisons and centroid updating during the iterations, which restrains the possible parallelism in updating clusters. The use of the depth buffer is limited to fixed-point values; which requires the distance metric to be scaled in a range of 0 to 1.

2.2 The *K*-Means Implementation Using Stencil and Depth Buffers

The implementation of *k*-means clustering on commodity GPUs has been presented in [2]. The motivation is to reduce the number of operations in the fragment processor. Data points to be clustered are stored in textures. The distance of each data point to every centroid is computed in parallel. The cluster label of each data point is identified after the computation of distances of each data point to all centroids is done. The computed distances are assigned to the depth buffer by the fragment program. The current distance in the depth buffer is compared with the distance that arrives from the fragment program. Writing the distances into the depth buffer continues until the distance computations to all the centroids are completed. Stencil buffers are defined and maintained for each cluster to keep track of the nearest label. The fragment program is written in such a way that the stencil buffer of each centroid contains the label of the nearest data points and the depth buffer has the corresponding distance values. In this implementation multiple depth buffers and stencil buffers are required to match the number of initial cluster centroids. This becomes a bottleneck when the

number of clusters is high. Moreover the use of the depth buffer is limited to fixed-point values; thus the distance metric needs to be scaled and rescaled back for the iterations.

3 Harnessing the Power of GPU in Clustering

Our implementation is done using OpenGL as the Application Programming Interface (API) [4], and the operational kernels are invoked via shader programs, using the Graphics Library Shading Language (GLSL) [11]. The Single Instruction Multiple Data (SMID) technique is employed to achieve data or vector level parallelism in the fragment processor.

Few technical concepts that harness the GPU into a usable parallel processor are:

- Independence of texture elements (texels) that can be accessed by fragment shaders
- Mapping of the texture into a required geometric shape
- Shader as a set of instructions (kernels) that modify mapped texels
- Shader execution with draw command to output to the memory buffer.

3.1 Efficient Use of GPU Hardware for Parallel Computation

The input data sets are stored in 2 dimensional (2D) textures of size $\sqrt{N}$ x $\sqrt{N}$, where N is the total number of observations. In this approach we use the Luminance format of the texture elements. Luminance texture format allows the texture to store a single floating-point value per texel. The initial centroids, the distances and the other relevant information are stored in the textures. The new cluster centroids are transferred back to the CPU and the stop condition is checked after the iteration.

An important concept in GPGPU is stream programming or parallel programming model. In this model, all data is represented as a stream or "an ordered set of data of the same data type" [10]. The key to maximize parallel processing is to allow multiple computation units to operate on the data simultaneously and to ensure

Table 1. GLSL codes for distance computations

```
char* shader3 = \
"#extension GL_ARB_texture_rectangle : enable\n" \
          "uniform sampler2DRect textureX;" \
          "uniform sampler2DRect textureY;" \
          "uniform float cx;" \
          "uniform float cy;" \
          "void main(void) { " \
"  float x = texture2DRect(textureX,
gl_TexCoord[0].st).x;" \
"  float y = texture2DRect(textureY,
gl_TexCoord[0].st).x;" \
"  gl_FragColor.x = sqrt((x-cx)*(x-cx)+(y-cy)*(y-
cy));"\
          "}";
```

that each unit operates efficiently. This model of parallelism can be achieved easily in the GPU hardware data path [10]. A kernel is defined as an instruction or computation that operates on the entire stream. The kernel is applied to each element of data in the hardware path. The result is dependent only on that particular element and independent of the rest of the stream. We implement the distance metric calculations in parallel and the GLSL fragment codes are listed in Table 1.

The code listing in Table 1 works as follows: The first line is an OpenGL extension that allows the 2 dimensional (2D) texture targets to support data sizes, which are "non-power-of-twos" (NPOT). The next two lines are used in GLSL to access the NPOT 2D rectangular texture sources in parallel. The codes within the main are used to read the data from each texel in the textures and the code line with gl_FragColor performs the actual computation simultaneously on data that is read from each of the texel. The resultant parameter is stored in the target textures.

4 Efficient Implementation of *K*-Means Clustering in the GPU

In our implementation of *k*-means clustering we have kept the use of fragment shader constants to a very minimum. Identifying the data point that belongs to each cluster and maintaining the identity of that cluster is achieved by using individual textures. These textures are labeled so as to know the data points that belong to the cluster. In this way, we are able to keep track of the cluster elements, execute cluster operations in parallel and minimize use of shader constants. No practical issues were noticed by defining sufficient textures to handle a large number of textures, say up to 32 clusters. No programmable limitation is foreseen to extend the implementation to handle large number of clusters.

4.1 Distance Computation in GPU

The data that needs to be clustered are stored in 2D textures of size $\sqrt{N}$ x $\sqrt{N}$. The kernels are applied to all the elements in the textures using fragment programs. Figure 2 shows that the distances of each data point to the clusters are stored in individual textures. The Euclidean distances of the *k*-clusters to each of the data point is calculated and is stored in k distance textures. Every corresponding texel coordinate in each of these distance textures has the distance from the same point to the k^{th} cluster. To compute the minimum distance, all the distance textures are simultaneously loaded as read textures. The *Dmin* texture (minimum distance texture) is defined and initiated as the write texture. The minimum of each corresponding texel is executed via a fragment shader program and the resultant is written into the *Dmin* texture. The *Dmin* texture has the distance of each data point to its nearest centroid.

4.2 Clusters Identification, Labeling, Centroid Computations, Updating in GPU

To identify and label the data point, which is closest to the cluster centroid, each k^{th} cluster distance texture is compared with the *Dmin* texture. The kernel, which is executed as a fragment shader program, compares the value in the cluster distance texture

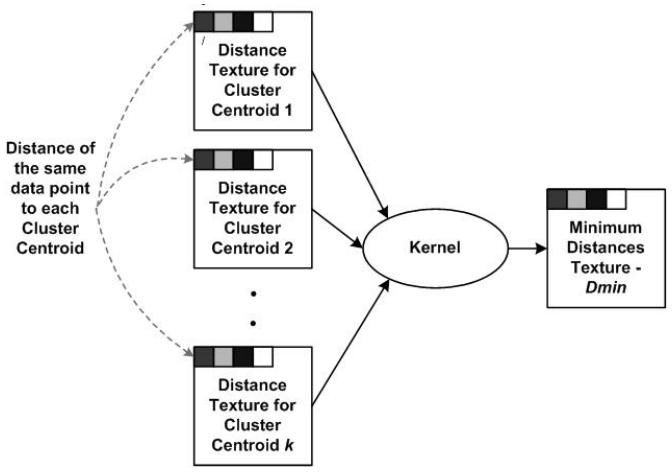

Fig. 2. Cluster Distance Textures and Minimum Distance Texture

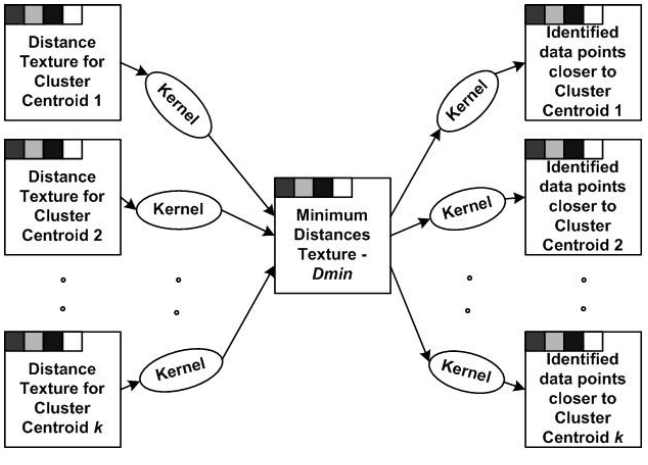

Fig. 3. Identification of Cluster Groups from the Distance Textures

to the corresponding texel value in the *Dmin* texture. Figure 3 shows kernel operations on the textures. If the distance values in both the textures are same, the output buffer is written with a value of "1" if not, "0". This kernel is repeated for all the *k*-cluster distance textures and that results in *k*-group textures. The new cluster centroids are computed based on the elements in each of the cluster group textures. In order to compute the new centroid the number of elements in the group is counted followed by the sum of the data values of the coordinates and subsequently the average is computed. Figure 4 shows the kernel operation on the cluster group textures and the texture with the input data points.

These operations are accomplished by executing a kernel, which effectively performs reduction operation on the textures. As a result each cluster has its own output

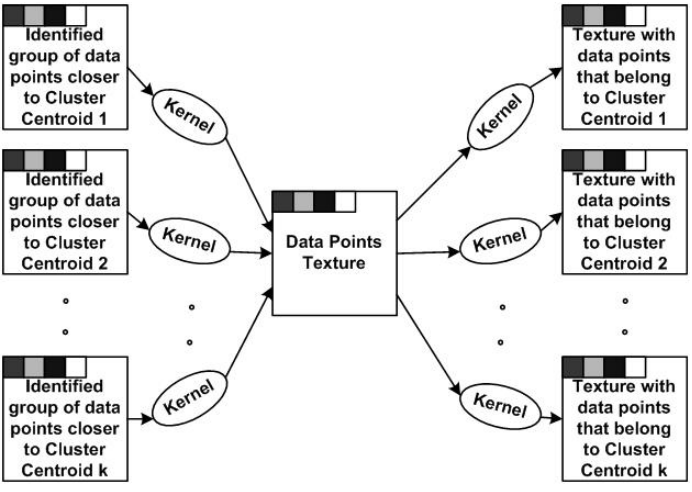

Fig. 4. Labeling of Data Points using a Texture for each Cluster

texture and the data points in that texture belong to the corresponding cluster. The number of passes required to compute each new centroid is given as the $\text{Log}_2\sqrt{N}$, where N is the number of data points. To make these computations efficient the read and write textures are swapped after every execution of the kernel.

The new centroids are computed in three steps using the texture reduction technique [5, 6]. The centroid for each cluster is then calculated. At the end of the kernel operation, each new centroid will be stored in a texture. After all the iterations are complete the centroids and the final data points of each cluster are transferred to the CPU.

5 Experimentations and Evaluations

5.1 The Experimental Setup for GPU Computations

A set of clustering data from the "Intelligent data storage for data-intensive analytical systems" [12] was used to test the efficiency of the implementation. The timings taken by the CPU and GPU were measured for every run. A 1.5 GHz Pentium IV CPU with a nVIDIA GeForce FX 5900 XT processor and a 3 GHz Pentium IV CPU with a nVIDIA GeForce 8500 GT processor were used in the experiment. The 8500 GT GPU has 16 textures 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 5900 XT is the older version of the 7900 GTX GPU. The 5900 engine has only 8 textures processing texels to pixels at a memory clock rate of 700 MHz and 256MB of video memory. The peak memory bandwidth is 22.4 GB/sec. The 8500 processor has a PCI E -16x interface with the CPU for data transfer whereas the 5900 have an AGP 8X communication slot. The clustering algorithms were implemented using the OpenGL API with embedded shader programs. The shader programs were developed using GLSL.

5.2 The Analysis of Results: CPU w.r.t GPU Computational Execution Time

The CPU and the GPU computational execution time were measured to compare the speed. The GPU execution time included the steps such as loading the data into the GPU textures, complete computations, assigning data observations to the cluster textures and transferring back the cluster information and centroids to the CPU. Using the "covtype" [12] dataset, we varied the number of clusters and the data size. We discuss the computational time in seconds / iteration as a performance metric for comparison, computed based on measured computational time and the number of iterations. Figure 5 shows the performance w.r.t the number of data points for the various processors used.

Tremendous gain in performance (time per iteration) is noticed while using the GPU over the CPU. It is quite evident that the CPU performance is affected by the data size whereas GPU shows little or no drop in performance with increased data size. The implementation in the 5900 GPU gains about 4 to 12 times in speed than its CPU counterpart. This is at least 3 times faster than the previous implementation [3]. The implementation of the same algorithm in the 8500 GPU gains speed by about 30 times than its CPU counterpart.

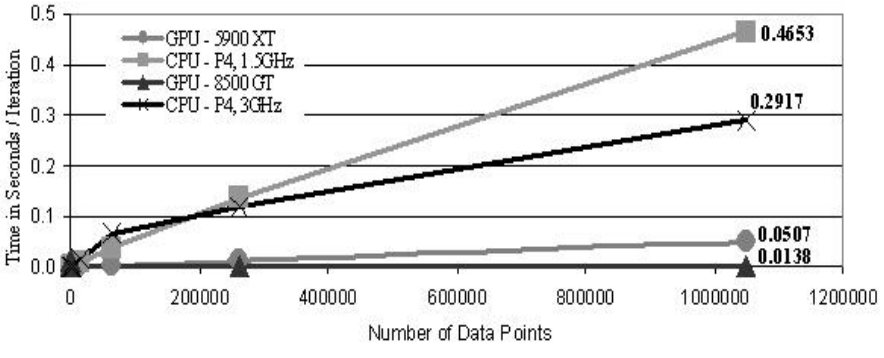

Fig. 5. Performance of *K*-means Clustering based on Data Size

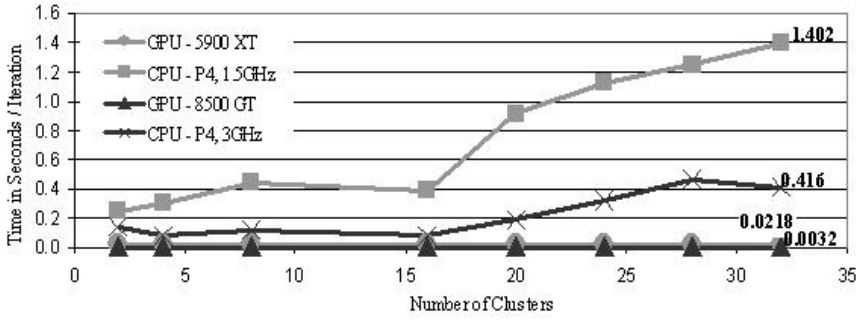

Fig. 6. Performance of *K*-means Clustering based on number of Clusters

Figure 6 shows the performance w.r.t the number of clusters. It is obvious that when the number of clusters increases the performance of the implementation in the GPU shows no or very little drop. For number of clusters less than 20, the implementation in the 5900 GPU gains about 10 to 20 times in computational speed than its CPU counterpart. The gain in computational time is more than 50 times when there are more than 20 clusters. When there are 32 clusters in the data ser, the 8500 GPU is faster about 130 times than its CPU counterpart.

6 Conclusions and Future Work

In this paper, we have presented an effective implementation of *k*-means algorithm in the GPU. The performance of this implementation has surpassed the CPU implementations by few tens to about a hundred. It has also succeeded the previous implementation of the *k*-means clustering in the GPU with significant gains in computational resources. By efficient use of textures in the fragment processor, the use of constants to update cluster data has been eliminated. Moreover all cluster centroids can be updated in parallel using textures after the iterations. Thus the necessity to transfer data and results between the GPU and CPU during the computations has been avoided. The results are encouraging and have made the *k*-means clustering algorithm much more efficient.

The parallel processing capabilities of the GPU will be further exploited to implement clustering of Gene expressions. This will involve the scaling of the *k*-means implementation to accommodate more dimensions. Similar approach in identification of data points in the clusters and computing cluster centroids using fragment shader and textures will be applied to Hierarchical clustering methods. Clustering techniques such as Fuzzy c-Means and variants of Hierarchical agglomerative clustering algorithm will be implemented to show higher computational efficiencies. There is a physical limitation in the size of the texture and the maximum number of textures available for simultaneous computation. These hardware limitations will have to be considered in future implementations of computational algorithms in GPU.

Recently, NVIDIA has released CUDA (Compute Unified Device Architecture), which is a technology that allows programmers to code algorithm into the 8000 series of GeForce graphics processors directly; an advantage for non-graphics programmers.

References

1. Fluck, O., Aharon, S., Cremers, D., Rousson, M.: GPU histogram computation. In: International Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH (2006)
2. Cao, F., Tung, A.K.H., Zhou, A.: Scalable Clustering Using Graphics Processors. In: Yu, J.X., Kitsuregawa, M., Leong, H.-V. (eds.) WAIM 2006. LNCS, vol. 4016. Springer, Heidelberg (2006)
3. Hall, J.D., Hart, J.C.: GPU Acceleration of Iterative Clustering. In: The ACM Workshop on GPC on GPU & SIGRAPH (2004)
4. Richard, W.S., Lipchak, B.: OpenGL SuperBible. Sams Publishing (2005)

5. Göddeke, D.: Basic Math Tutorial. Retrieved from GPGPU (2007), http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html
6. Göddeke, D.: Reduction Tutorial. Retrieved from GPGPU (2007), http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial2.html
7. NVIDIA. GPU Gems 2. ADDISON-WESLEY (2006)
8. Zhang, Q., Zhang, Y.: Hierarchical clustering of gene expression profiles with graphics hardware acceleration. Pattern Recognition Letters 27, 676–681 (2006)
9. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A Survey of General-Purpose Computation on Graphics Hardware. In: Eurographics (2006)
10. Owens, J.D.: Streaming Architectures and Technology Trends. In: GPU Gems2, ch. 29, pp. 457–470 (2004)
11. Rost, R.J.: OpenGL® Shading Language, 2nd edn. Addison Wesley Professional, Reading (2006)
12. Intelligent Data Storage for Data-Intensive Analytical Systems. Real Data Set covtype Downloaded from D·Star (2007), http://uisacad2.uis.edu/dstar/data/clusteringdata.html
13. Takizawa, H., Kobayashi, H.: Hierarchical parallel processing of large scale data clustering on a PC cluster with GPU co-processing. J. Supercomputing 36, 219–234 (2006)