Scalable Clustering Using Graphics Processors

Feng Cao[1], Anthony K.H. Tung[2], and Aoying Zhou[1]

[1] Dept. of Computer Science and Engineering, Fudan University, China
{caofeng, ayzhou}@fudan.edu.cn
[2] School of Computing, National University of Singapore, Singapore
atung@comp.nus.edu.sg

Abstract. We present new algorithms for scalable clustering using graphics processors. Our basic approach is based on k-means. By changing the order of determining object labels, and exploiting the high computational power and pipeline of graphics processing units (GPUs) for distance computing and comparison, we speed up the k-means algorithm substantially. We introduce two strategies for retrieving data from the GPU, taking into account the low bandwidth from the GPU back to the main memory. We also extend our GPU-based approach to data stream clustering. We implement our algorithms in a PC with a Pentium IV 3.4G CPU and a NVIDIA GeForce 6800 GT graphics card. Our comprehensive performance study shows that the common GPU in desktop computers could be an efficient co-processor of CPU in traditional and data stream clustering.

1 Introduction

The rapid growth of data volume in real-life databases has intensified the need for scalable data mining methods. Data warehouse and data stream applications are very data and computation intensive, and therefore demand high processing power. As a building block of data mining, clustering derives clusters which can be visualized more efficiently and effectively than the original data. Researchers have actively sought to design algorithms to perform efficient clustering.

Assuming that the data sets are in the secondary memory, effort to enhance the scalability of clustering algorithms often focus on reducing the number of disk I/O. Work in this direction have effectively reduce the scan on data sets into one or two rounds. As such, it is difficult to further enhance scalability by reducing I/O cost.

Meanwhile, CPU cost is no longer a minor factor for scalability improvement in clustering algorithms (see Figure 1). In data stream applications, CPU cost becomes more important because each data object needs to be processed in real time. Therefore, new techniques for reducing CPU cost will greatly improve the scalability of online and offline clustering algorithms.

Recently, the Graphics Processing Unit (GPU) has provided a programmable pipeline, allowing users to write fragment programs that are executed on pixel processing engines. At the same time, the computing capability of common GPU

J.X. Yu, M. Kitsuregawa, and H.V. Leong (Eds.): WAIM 2006, LNCS 4016, pp. 372–384, 2006.

is becoming increasingly powerful. For example, a NVIDIA GeForce6800 chip contains more transistors than an Intel Pentium IV 3.73GHz Extreme Edition processor. In addition, the peak performance of GPUs has been increasing at the rate of 2.5 – 3.0 times a year, much faster than the rate that Moore's law predicted for CPUs. Furthermore, due to economic factors, it is unlikely that dedicated general vector and stream processors will be widely available on desktop computers [14].

Driven by the programmability and computational capabilities of GPUs, many GPU-based algorithms have been designed for scientific and geometric computations [10][12], database operations [5], stream frequency and quantiles approximation [6], etc. However, as far as we know, the computational power of GPUs has not been well exploited for scalable clustering yet. In this paper, we will make the following contribution:

1. Having identify distance computation and comparison as the most expensive operations for clustering algorithms, we propose a new, highly parallelized distance computation technique which utilizes the fragment vector processing and *multi-pass rendering* capabilities of GPUs. We further apply multitexturing technology to deal with high-dimensional distance computing.
2. Our basic approach is based on k-means. By changing the order of determining object labels, and exploiting the high computational power and pipeline of graphics processing units (GPUs) for distance computing and comparison, we speed up the k-means algorithm substantially. We then further extend the algorithm to perform clustering on data stream.
3. A comprehensive performance study proves the efficiency of our algorithms. The GPU-based algorithm for stream clustering reduces clustering cost by about 20 times as compared to prior CPU-based algorithms. The basic k-means-based algorithm obtains 3 – 8 times speedup over CPU-based implementations. We thus bring forward the conclusion that the GPU can be used as an effective co-processor for traditional and stream clustering.

The rest of the paper is organized as follows. Section 2 analyzes existing clustering algorithms. Section 3 gives an overview of GPU. Section 4 presents our GPU-based clustering algorithms. Section 5 presents the performance study. Section 6 briefly surveys related work. Section 7 concludes the paper.

2 Analysis of Existing Clustering Algorithms

Existing clustering algorithms can be classified into partitioning [11], hierarchical [8, 16], density-based [4], streaming methods [1, 2, 7], etc. Since multiple scan of out of core data sets often create a bottleneck due to the I/Os, many methods have been proposed to reduce the number of scans on data sets into one pass or two. These include: random sampling technology in CURE [8], R*-Tree indexing approach adopted in DBSCAN [4], the divide and conquer strategy in STREAM [7] to process large data sets chunk by chunk, and the CF-tree in BIRCH [16] for performing preclustering. These methods have reduced I/O cost back to

a level in which the CPU cost become significant again. Specifically, distance computation and comparison often become the most expensive operations in existing clustering algorithms.

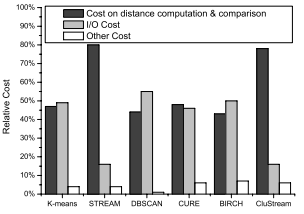


Fig. 1. Relative costs in clustering methods

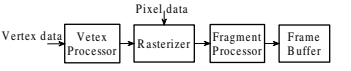


Fig. 2. Graphics pipeline overview

The popular partitioning-based method – k-means [11] contains three steps: (1)Initialization: Choosing k points representing the initial group of centroids. (2)Assignment: Assigning each point to its closest centroid. When all points have been assigned, recalculate the positions of the k centroids. (3)Termination condition: Repeating Steps 2 and 3 until the centroids no longer move. Having load the data into memory, the most time consuming computation is assignment, i.e., distance computation and comparison (see Figure 1). The number of distance computation and comparison in k-means is $O(kmn)$, where m denotes the number of iteration and n is the number of point in memory.

An effective hierarchical clustering algorithm, CURE [8], starts with the individual points as individual clusters. At each step, the closest pair of clusters is merged to form a new cluster. The process is repeated until there are only k remaining clusters. Figure 1 shows that distance computation and comparison are about 45% of the total cost. Because these operations widely exist in methods of finding the nearest cluster and merging two clusters. The number of distance operations is $O(n^2 \log n)$, where n denotes the number of points in a sampling. To find the nearest cluster, we can load the clusters to the GPU and apply our GPU-based distance computation technique to these clusters.

In order to determine the density of a given point p, density-based methods (such as DBSCAN [4]) need to compute the distance from point p to its nearby points and compare the distance with a pre-defined threshold ϵ. Therefore, the cost on distance computation and comparison becomes an important factor (see Figure 1). To determine the density for each point, we could load nearby data points into the GPU, apply our GPU-based distance computation, and compare the distance results with ϵ by GPU.

In the data stream environment, I/O cost no longer exists or could be ignored. Figure 1 shows the relative costs in STREAM [7] and CluStream [1] when accessing data from the hard disk. Ideally, we should adopt new GPU-based methods to improve the scalability of stream clustering.

3 Preliminaries of GPU

3.1 Graphics Pipeline

Figure 2 shows a simplified structure of the rendering pipeline. A vertex processor receives vertex data and assembled them into geometries. The rasterizer constructs fragments at each pixel location covered by the primitive. Finally, the fragments pass through the fragment processor. A series of tests (such as depth test) can be applied to each fragment to determine if the fragment should be written to the frame buffer. Frame buffers may be conceptually divided into three buffers: *color buffer*, storing the color components of each pixel; *depth buffer*, storing a depth value associated with each pixel; and *stencil buffer* which stores a stencil value for each pixel and can be regarded as a mask on the screen.

3.2 Data Representation and Terminology

We store the data points to be clustered on the GPU as *textures*. A *texture* is an image, usually a 2D array of values, which often contains multiple channels. For example, an RGBA texture four channels: red, blue, green and alpha. To perform clustering using the GPU, the attribute of each tuple is stored in multiple channels of a single texel (i.e., individual elements of the texture), or the same texel location in multiple textures. Several data formats are supported in textures, e.g., 8-bit bytes. In particular, the textures in *Pbuffer* (an off-screen frame buffer) support the 32-bit IEEE single precision floating-point.

The term *multi-texturing* refers to the applications of more than one texture on the same surface. *Multi-pass rendering* is a technique for generating complex scene. That is, the GPU renders several passes and combines the resulting images to create a final frame. *Stencil test* is used to restrict computation on a portion of the frame buffer. When a new fragment arrives, stencil test compares the value at the corresponding location in the stencil buffer and a reference value. The new fragment is discarded if it fails the comparison.

A group of stencil operations are provided to modify the stencil value, e.g., keeping the stencil value in the stencil buffer or replacing the stencil value to the reference value. Typically, if stencil test is passed, depending on the result of depth test, the user could define different stencil operations.

4 Clustering Using GPUs

K-means is a basic method for clustering which has wide applications. When the algorithm is implemented on the CPU, distances to the k centroids are evaluated for a single output object label at a time, as illustrated in Figure 3(a).

Instead of focusing on computation of the label for a single object one at a time, we calculate the distances from a single input centroid to all objects at one go, as shown in Figure 3(b). The distances to a single input centroid can be computed in the GPU for all objects simultaneously. In this case, the final label of a single object is only available when all input centroids have been processed.

(a) Object-centered (b) Centroid-centered

Fig. 3. Object-centered vs. centroid-centered distance computation

The rationale for the approach is as follow: GPU essentially operates by applying simple, identical operations to many pixels simultaneously. Naturally, these operations have access to a very limited number of inputs. However, in the k-means algorithms, k inputs are needed in order to calculate the label of a single data point. Furthermore, centroid-oriented distance computation allows comparison operations to be done outside each fragment, thus greatly reducing the number of operations in the fragment program.

4.1 Distance Computing

Typically, Euclidean distance is a used as a similarity measure for clustering. The Euclidean distance between 2 d-dimensional points X and Y is defined as follows: $dist(\overrightarrow{X} - \overrightarrow{Y}) = \sqrt{\Sigma_{i=1}^{d}(x_i - y_i)^2}$.

Assuming that there are a set of d-dimensional points $X_i (1 \leq i \leq N)$ and k centroids Y_j, where $1 \leq j \leq k$. We arrange X_i into an array A (named point array) as follow, where R denotes the number of rows , L denotes the number of columns, $R * L$ equals the number of points N. In the actual implementation, a point array is a texture (see Section 3.2). If the number of points is above the maximal size of one texture, the point array can be partitioned into multiple textures. In order to better utilize the parallelism of GPU, R and L are set at $\lceil\sqrt{N}\rceil$. The unused portion of the array could be simply masked by stencil.

$$A = \begin{vmatrix} X_1 & \cdots & X_L \\ \cdots & \cdots & \cdots \\ X_{(R-1)*L+1} & \cdots & X_{R*L} \end{vmatrix} \qquad D_j = \begin{vmatrix} dot2(X_1 - Y_j) & \cdots & dot2(X_L - Y_j) \\ \cdots & \cdots & \cdots \\ dot2(X_{(R-1)*L+1} - Y_j) & \cdots & dot2(X_{R*L} - Y_j) \end{vmatrix}$$

Each element $a[m][n]$ in array A corresponds to point $X_{(m-1)*L+n}$. We calculate the result array D_j (named distance array) for each centroid Y_j as above, where $dot2(X)$ is the dot product of vector X with itself. Each element $e[m][n]$ in D_j corresponds to the distance from point $X_{(m-1)*L+n}$ to centroid Y_j. Without loss of generality, we adopt squared Euclidean distance as the goodness measurement here. GPUs are capable of computing dot product on vectors in parallel giving high efficiency. Here, we propose a GPU-based method for distance computation.

ComDistance (Algorithm 1) computes the distance array for the point array in tex to centroid v_{cen}. To allow a more precise fragment, Line 1 actives Pbuffer. Line 2 enables the fragment program. Line 3 renders a textured quadrilateral

using FComDist. SUB and DOT are hardware optimized vector subtract and dot product instructions, respectively. Finally, the distance array is stored in the depth component of each fragment. In case of very large databases, we can swap textures in and out of video memory using out-of-core techniques.

Algorithm 1. ComDistance (tex, v_{cen})

```
1: ActivePBuffer();
2: Enable fragment program FComDist;
3: RenderTexturedQuad(tex);
4: Disable fragment program FComDist;
```

FComDist(v_{cen})

```
1: v_tex = value from tex
2: tmpR = SUB(v_tex,v_cen)
3: result.depth = DOT(tmpR,tmpR);
```

The ARB_fragment_program OpenGL extension allows depth values to be assigned in the fragment program. We exploit this feature to accelerate the comparison step described in Section 4.2 by avoiding the storage of the distance array in a texture which mean reloading the texture into the depth buffer.

High-dimensional Distance Computing. In case of $d > 4$, we divide every four dimensions of points into a point array, calculate each of these $\lceil \frac{d}{4} \rceil$ arrays with the corresponding section of Y_j, and sum up them to get the final Y_j.

Our algorithm uses multi-texturing technology to handle high-dimensional data. Although current GPUs only support eight simultaneous texture units resulting in at most 32 dimensions in one pass, we believe that future generation of GPU will provide more simultaneous texture units. At the current stage, we adopt *multi-pass rendering* in case of $d > 32$. Assuming a given GPU support m simultaneous texture units, the number of passes will be equal to $\lceil \frac{d}{4m} \rceil$.

4.2 Labeling

In k-means clustering, labeling is achieved by comparing the distances between the point and each centroid. We utilize *multi-pass rendering* to realize this operation. Depth test is enabled to compare the depth value of the arriving fragment to the corresponding pixel in the depth buffer. The stencil buffer is configured to maintain the label of the nearest centroid. Finally, the distance array D_j is rendered for each j ($1 \leq j \leq k$). Algorithm 2 describes this procedure in detail.

We compute and store distance array D_1 directly in the depth buffer, and initialize the stencil buffer with 1. That is, all the points are labelled to centroid 1 at first. Then, depth test is enabled and set to pass if the depth value of arriving fragment is less than the corresponding pixel. Stencil test is set to always being passed. If the arriving fragment passes depth test, the corresponding pixel is updated with the new depth value, and Line 9 replace the stencil value in

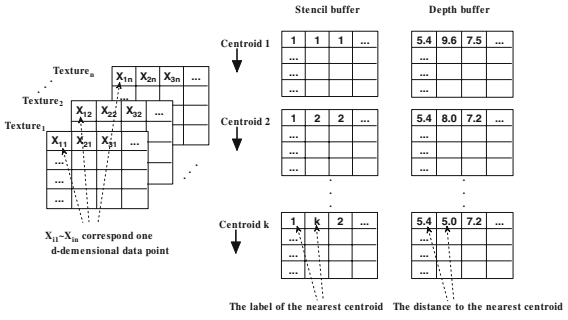


Fig. 4. Labeling

corresponding position with the new label $i+1$. Otherwise, we keep the depth and stencil values. Therefore, after each distance array D_i is generated, the stencil buffers contains the label of the nearest centroid for each point (named label array). The depth buffer contains the corresponding minimal distance value. Figure 4 illustrates this process. In the pipeline of the labeling algorithm, various operations can be processed simultaneously: the fragment program computes distance arrays; depth test compares depth value in the depth buffer; and stencil test updates the labels in the stencil buffer.

4.3 Generating New Centroids

It is a bottleneck of current hardware to retrieve data from GPU to the main memory (sending data from the main memory to GPU is much faster by say ten times). According to the data retrieved, we design the following two strategies to generate new centroids, corresponding to GPU-C (GPU-based clustering by retrieving centroids) and GPU-L (GPU-based clustering by retrieving label array) algorithms, respectively:

1. **Retrieve Centroids.** One way is to compute the centroids in GPU and retrieving them from GPU. Stencil test is utilized to filter out points in the same cluster and summarize them by mipmaps. Mipmaps are multi-resolution textures consisting of multiple levels. The highest level contains the average of all the values in the lowest level. A group of *occlusion query*s must be called in order to obtain the number of points in each cluster. An occlusion query returns the number of fragments that pass the variance tests. In our case, the test is a stencil test. The procedure is shown in Algorithm 3. In case of $d > 4$, we need to render $\lceil \frac{d}{4} \rceil$ times for each centroid. Finally, we retrieve the highest level of the mipmaps $tex_{out}[i]$ and the result of the occlusion query q_i from GPU in order to calculate the final centroid results. Although this strategy has the advantage of reducing communication cost, its computation cost overwhelms the saving on communication cost, as our experiments in Section 5.4 will show.

2. **Retrieve the Label Array.** In this strategy, we retrieve the label array from the stencil buffer directly. To reduce communication cost, the label array is retrieved from the stencil buffer by an impact mode GL_BYTE. Although 8-bit value constraint exists in this mode (that is the upper bound of k is 256), it can meet the requirements of most real applications. After retrieving the label array, we generate the new centroids in CPU by adding up the points with the same label.

Algorithm 2. Labeling (tex_{in}, $v_{centroid}[k]$)

```
1: glClearStencil(1);
2: ComDistance(tex_in,v_centroid[0]);{ generate distance array D_1 and store it in depth
   buffer}
3: glEnable(GL_DEPTH_TEST);
4: Set depth test to pass if incoming fragment is less than the corresponding value in
   depth buffer.
5: for i = 1; i < k; i++ do
6:    Set stencil test to always pass;
7:    ComDistance(tex_in,v_centroid[i]);{ generate a frame of fragments corresponding
      to distance array D_{i+1}}
8:    if depth test passed then
9:       replace stencil value with the reference value i + 1;
10:   else
11:      keep the stencil value;
12:   end if
13: end for
14: glDisable(GL_DEPTH_TEST);
```

Algorithm 3. GetControids (tex_{in}, $tex_{out}[i]$)

```
1: for i = 1; i ≤ k; i++ do
2:    Set stencil reference value as i;
3:    Set stencil test to pass if stencil value is equal to the reference value.
4:    Enable Occlusion query i;
5:    RenderTexturedQuad(tex_in);{ generate a frame of fragments which correspond
      to all the points belonging to centroid i}
6:    Disable Occlusion query i;
7:    MipMap the fragments in framebuffer into tex_out[i]
8: end for
```

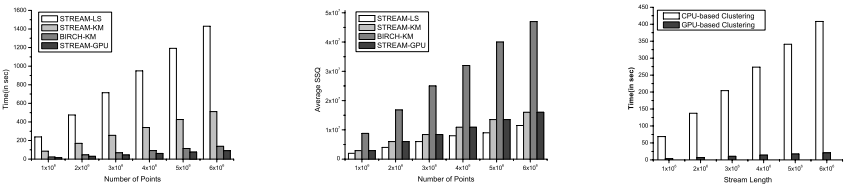
4.4 Clustering Data Stream

We extend our GPU-based method to data stream clustering, specifically, landmark window [7] and sliding window clustering [2]. The pipe-line architecture and parallel processing of the GPU are well suited for stream processing [14].

1. **Landmark Window Clustering.** We adopt the divide-and-conquer methodology [7] and our GPU-L method (abbr. STREAM-GPU) to cluster

a data stream. We compare STREAM-GPU with three CPU-based algorithms: BIRCH-KM, STREAM-KM and STREAM-LS [7]. Figures 5(a)(b) show that STREAM-GPU achieves the highest processing rate with competitive SSQ (the sum of square distance). Although STREAM-LS achieves the lowest SSQ, its processing rate is 15 times slower than STREAM-GPU. STREAM-GPU is more efficient than BIRCH-KM with 200% effectiveness gain. Considering only clustering cost, STREAM-GPU is nearly 20 times faster than an optimized CPU-based implementation.

2. **Sliding Window Clustering.** In sliding window clustering, only the N most recent points contribute to the results at any time. We adopt the algorithm in [2], and the basic operation in combination procedure is implemented by our GPU-L method. Figure 5(c) shows the comparison result with window size $N = 100,000$. GPU-based clustering is always better than an optimized CPU-based implementation by about 19 – 20 times.



(a) Efficiency comparison in landmark window

(b) Effectiveness comparison in landmark window

(c) Efficiency comparison in sliding window

Fig. 5. GPU-based vs. CPU-based stream clustering

5 Experiments and Results

5.1 Experimental Setting

We tested our algorithms on a Dell workstation with a 3.4 GHz Pentrium IV CPU and a NVIDIA GeForce 6800GT graphics card. To generate the fragment programs, we used NVIDIA's CG compiler. The CPU algorithms were compiled using an Intel compiler with hyper-threading technology and SIMD execution option. Data exchange between GPU and CPU was implemented with an AGP 8X interface. The points in synthetic data sets followed Gaussian distributions. The data sets had between 10K and 10,000K points each, varied in the number of clusters from 8 to 256, and ranged in dimensionality from 4 to 28.

Execution time was adopted to evaluate various costs. The costs of the CPU-based k-means algorithm (abbr. CPU-K) are: (1)$tc = cc + I/O\ cost$, where tc is total cost; cc is clustering cost. (2)$cc = pt * m$, where pt is the cost of one iteration; m is the number of iterations. (3)$pt = dc + gc$, where dc is the cost of distance computation and comparison; gc is the cost of generating new centroids. The costs of the GPU-based algorithm are: (1)$cc_{gpu} = pt_{gpu} * m + m2g$, where $m2g$ is the cost of sending data from CPU to GPU. (2)$pt_{gpu} = dc + gc + g2m$,

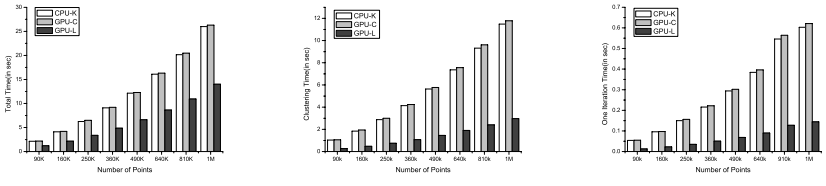
where $g2m$ is the cost of retrieving data from GPU to CPU. Unless otherwise mentioned, the experiments adopted $d = 8$, $k = 8$ normal distributed data set.

5.2 Total Cost

Figure 6(a) shows that the total costs of GPU-L, GPU-C and CPU-K increase linearly to the size of data sets. The total cost of GPU-L is about 60% of CPU-K's. However, the total cost of GPU-C almost equals to CPU-K's. We will discuss this phenomenon in Section 5.4. Because total cost includes I/O cost and the number of iteration is about 20, the influence of I/O cost on the total cost is very big. The impact of I/O cost reduces as the number of iterations increases. And the performance improvement of GPU-L and GPU-C will be greater.

5.3 Clustering Cost and Cost of One Iteration

Figure 6(b) illustrates that the clustering cost of GPU-L is about 1/4 that of CPU-K. First, the performance improvement benefits from the parallel computation of pixel processing engines. For example, a NVIDIA GeForce 6800 GT graphic processor can process 16 pixels in parallel. Second, the vector instructions in the GPU are well optimized, which greatly improves the process rate of distance computation. Third, as the distance is compared via depth test, no branch mispredictions exist in the GPU implementation, which leads to further performance gain. Branch mispredictions can be extremely expensive on modern CPUs. For example, a branch misprediction on a Pentium IV CPU costs 17 clock cycles. Figure 6(c) compares the costs of one iteration. It shows the same tendency of Figure 6(b). GPU-L constantly outperforms CPU-K by four times.



(a) Total cost (b) Clustering cost (c) Cost of one iteration

Fig. 6. GPU-based vs. CPU-based clustering

5.4 Costs of Generating Centroids and Retrieving Data

We compare the cost of generating centroids gc in GPU-C and GPU-L. Figure 7 shows the gc in GPU-C is about 10 times larger than the gc in GPU-L. This is because in order to generate centroids, GPU-C needs to perform several times of slow texture writing, which is often a relatively slow operation.

Figure 7 shows GPU-C has the advantage of retrieving data from GPU at low cost. The cost of retrieving data from GPU $g2m$ is a constant in GPU-C

because it only needs to retrieve k centroids and the number of points in each cluster. However, this advantage is overwhelmed by its great cost on generating centroids in GPU-C. Therefore, the overall clustering cost of GPU-L is much smaller than that of GPU-C.

Figure 8 illustrates the cost of retrieving data from GPU $g2m$ in GPU-L. As the number of points grow, $g2m$ increases linearly. However, as we adopt a compact mode of data retrieval, the cost of retrieving data in GPU-L is not significant compared to the cost of one iteration.

5.5 Clustering Cost vs. k and d

Because the number of centroids k and dimensions d may significantly effect the clustering cost, we test several data sets with 16,000 data points for various k and d. Figure 9 shows as k increases, the costs of GPU-L, GPU-C and CPU-K increase linearly. GPU-C has almost the same cost as CPU-K, while the cost of GPU-L is much lower than that of CPU-K. As k grows, the advantage of GPU-L becomes more obvious. This is because the larger k is, the advantage of parallelism is better utilized. Figure 10 shows that the clustering cost in CPU-K, GPU-L and GPU-C increase linearly as d increases.

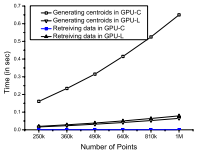


Fig. 7. Costs of generating centroids and retrieving data

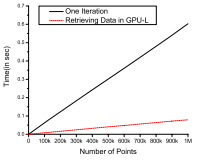


Fig. 8. Cost of retrieving data vs. cost of one iteration

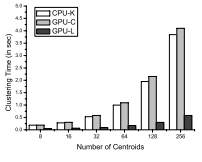


Fig. 9. Clustering cost vs. k

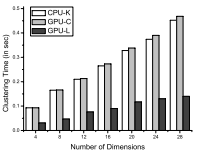


Fig. 10. Clustering cost vs. d

6 Related Work on GPU-Based Computing

High performance vertex processors and rasterization capability are utilized for certain numerical processing, including dense matrix-matrix multiplication [12], general purpose vector processing [15], etc. Different from these vertex-based methods, our algorithm achieves vector processing ability at the fragment level, which possesses higher parallel ability. Hall et al provided a GPU-based iterative clustering method [9]. As being designed for geometry processing, it doesn't fully utilize the pipeline of GPUs for mining large databases, let alone data streams. New techniques have been developed to take advantage of the highly optimized GPU hardware functions, e.g, 2D discrete Voronoi Diagrams [10] and 3D object collision detection [3]. Different from these 2D or 3D approximate algorithms, our clustering methods yield exact results for high-dimensional data points.

There has been interest in using GPUs to speed up database computations. Sun et al [13] used GPUs for spatial selection and join operations. Govindaraju et

al [5] presented algorithms for predicates and aggregates on GPUs. Another work [6] presents algorithms for quantile and frequency estimation in data streams.

7 Conclusion

In this paper, we have presented a novel algorithm for fast clustering via GPUs. Our algorithm exploits the inherent parallelism and pipeline mechanism of GPUs. Distance computing and comparison are implemented by utilizing the fragment vector processing and multi-pass rendering capabilities of GPUs. Multi-texturing technology is applied to handle high-dimensional distance computing. We have also extended our method to stream clustering. Our implementation of the algorithms on a PC with a Pentium IV 3.4G CPU and a NVIDIA 6800GT graphics card highlights their performance. Our future work includes developing algorithms for other data mining tasks such as outlier detection and classification.

References

1. C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for clustering evolving data streams. In *Proc. of VLDB*, 2003.
2. B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Maintaining variance and k-medians over data stream windows. In *Proc. of PODS*, 2003.
3. G. Baciu, S. Wong, and H. Sun. Recode: An image-based collision detection algorithm. *Visualization and Computer Animation*, 10(4):181–192, 1999.
4. M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. of KDD*, 1996.
5. N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and et al. Fast computation of database operations using graphics processors. In *Proc. of SIGMOD*, 2004.
6. N. K. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *Proc. of SIGMOD*, 2005.
7. S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams:theory and practice. In *IEEE TKDE*, pages 515–528, 2003.
8. S. Guha, R. Rastogi, and K. Shim. Cure: An efficient clustering algorithm for large databases. In *Proc. of SIGMOD*, pages 73–84, 1998.
9. J. D. Hall and J. C. Hart. Gpu acceleration of iterative clustering. In *Proc. of SIGGRAPH poster*, 2004.
10. K. E. Hoff III, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proc. of SIGGRAPH*, pages 277–286, 1999.
11. A. Jain and R. Dubes. Algorithms for clustering data. *New Jersey*, 1998.
12. E. S. Larsen and D. K. McAllister. Fast matrix multiplies using graphics hardware. In *Proc. of IEEE Supercomputing*, 2001.
13. C. Sun, D. Agrawal, and A. E. Abbadi. Hardware acceleration for spatial selections and joins. In *Proc. of SIGMOD*, pages 455–466, 2003.
14. S.Venkatasubramanian. The graphics card as a stream computer. In *SIGMOD Workshop on Management and Processing of Data Streams*, 2003.

15. C. J. Thompson, S. Hahn, and M. Oskin. Using modern graphics architectures for general-purpose computing: A framework and analysis. In *Proc. of IEEE/ACM International Symposium on Microarchitectures*, pages 306–317, 2002.
16. T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. In *Proc. of SIGMOD*, pages 103–114, 1996.