

Scalable Clustering Using Graphics Processors

Feng Cao¹, Anthony K. H. Tung², and Aoying Zhou¹

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, but it reorders the way of determining object labels, and exploits the high computational power and pipeline of graphics processing units (GPUs). The core operations in clustering algorithms, i.e., distance computing and comparison, are performed by utilizing the fragment vector processing and multi-pass rendering capabilities of GPUs. We introduce two strategies to retrieve data from GPUs, taking into account low bus bandwidth. 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 for faster clustering.

Assuming that data sets are stored in the hard disk, the scalability of a clustering algorithm is often achieved by reducing the number of disk I/O. However, after such successful effort to reduce the number of scans on data sets into one or two, it is difficult to further enhance scalability by reducing I/O cost. Meanwhile, computation cost is no longer an ignorable factor for scalability improvement in clustering algorithms (see Figure 1). In data stream applications, computation cost becomes more important because of their real time processing requirement. Therefore, new techniques for reducing computation 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 ability of common GPU is becoming very 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 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] and stream frequency and quantiles approximation [6]. However, as far as we know, the computational power of GPUs have not been well exploited for scalable clustering yet. The main contributions of this paper are:

1. We analyze that distance computation and comparison are among the most expensive operations in clustering algorithms. We propose a new distance computation technique which utilizes the fragment vector processing and *multi-pass rendering* capabilities of GPUs. We further apply multi-texturing technology to deal with high-dimensional distance computing.
2. We present novel algorithms of clustering via the GPU. Our basic method is based on k-means, but it reorders the way of determining object labels, and utilizes the high computational power and the pipelined architecture of GPUs. It minimizes the data transmission between CPU and GPU by taking into account low bus bandwidth. Furthermore, it is extended to stream clustering.
3. We implemented our algorithm in a PC with a Pentium IV 3.4G CPU and a NVIDIA GeForce 6800 GT graphics card. A comprehensive performance study proves the efficiency of our algorithms. The GPU-based algorithm for stream clustering reduces about 20 times of clustering cost as compared to prior CPU-based algorithms. The basic k-means-based algorithm obtains 3 – 8 times speedup over CPU-based implementations. Overall, 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 a brief overview of GPUs. Section 4 presents our novel GPU-based clustering algorithm. Section 5 presents the performance study. Section 6 briefly surveys related work on GPU-based computing. Section 7 concludes the paper.

2 Analysis of Existing Clustering Algorithms

Typically, existing clustering algorithms could be classified into partitioning [11], hierarchical [8, 16], density-based [4] and streaming methods [1, 2, 7], etc. Since multiple I/O scans of data sets is often a bottleneck for clustering, many methods have been proposed to reduce the number of scans on large data sets into one pass or two. The examples include: the random sampling technology in CURE [8] to handle large data set, the R*-Tree adopted in DBSCAN [4] to load a block of data nearby, the divide and conquer strategy in STREAM [7] to process large data set chunk by chunk, and the CF-tree in BIRCH [16] to perform preclustering.

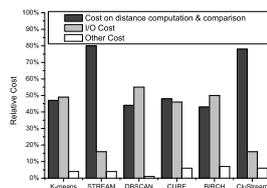


Fig. 1. Relative costs in clustering methods

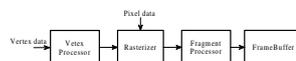


Fig. 2. Graphics pipeline overview

However, when the I/O cost was successfully reduced by these methods, the computation cost becomes an unignorable factor. Specifically, after proper data fitting in main-memory, distance computation and comparison often become very expensive operations in existing clustering algorithms, because of the widely existence in execution.

The typical partitioning method – k-means contains three steps: (1)Initialization: Choosing k points representing 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 judgement: Repeating Steps 2 and 3 until the centroids no longer move. After data loading into memory, the most time consuming computing is assignment, i.e., distance computation and comparison (see Figure 1). The number of distance computation and comparison in k-means is 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 in CURE without considering the final labeling process. Because these operations widely exist in methods of finding the nearest cluster and merging two clusters. The number of distance computation and comparison 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 on these clusters. The merge step could also be accelerated by applying our GPU-based k-means algorithm.

In order to determine the density of a given point p , density-based methods (such as DBSCAN [4]) need to compute the distance from a 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 in DBSCAN (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 technique on these data blocks, 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 getting

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 in *textures*. A *texture* is an image, usually a 2D array of values, which often contains multiple channels. For example, an RGBA texture has red, blue, green and alpha four channels. To apply the clustering algorithm 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 offscreen frame buffer) support the 32-bit IEEE single precision floating-point.

The term *multi-texturing* refers to the applications of more than one texture to 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. In software-based clustering, distances to k input centroids are commonly evaluated for a single output object label at one time, which is illustrated in Figure 3(a).

Instead of focusing on computation of the label of a single object at any one time, we calculate the distances from a single input centroid to all objects at one time, as shown in Figure 3(b). The distances to a single input centroid can be

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);
```

actives Pbuffer. Line 2 enables the fragment program fComDist in the course of rasterization. Line 3 renders a textured quadrilateral using the fragment program FComDist. SUB and DOT are hardware optimized vector subtract and dot product instruction, respectively. Finally, the distance array is stored in the depth component for each fragment. In case of very large databases, we can swap textures in and out of video memory by out-of-core techniques.

The ARB.fragment_program OpenGL extension allows depth values to be assigned by the fragment program. We exploit this feature to accelerate the comparison step described in Section 4.2. Otherwise, the distance array should be stored into a texture, and reloaded from the texture into depth buffer to compare different distance arrays, which would introduce additional cost of texture writing and reading.

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

Our algorithm uses multi-texturing technology to compute high-dimensional data. Although current GPUs only support eight simultaneous texture units. That is, there will be at most 32 dimensions in one pass. We believe it will be improved in the next generation of GPUs. At the current stage, we adopt *multi-pass rendering* in case of $d > 32$. Let a given graphic process support m simultaneous texture units, the number of passes equals $\lceil \frac{d}{4m} \rceil$.

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 nearest centroid's label. Finally, 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

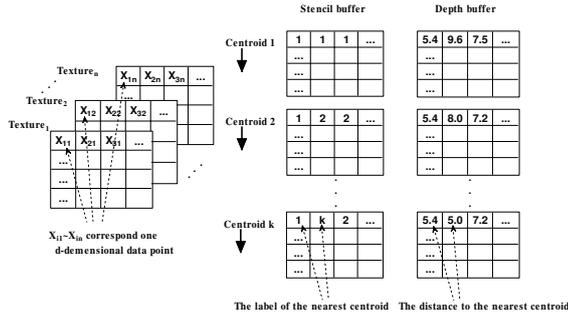


Fig. 4. Labeling

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 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 of points in the stencil buffer.

4.3 Generating New Centroids

It is a bottleneck of current hardware to retrieve data from GPU to CPU (sending data from CPU to GPU is much faster, generally ten times of retrieving bandwidth). According to the data retrieved, we design two strategies to generate new centroids:

1. **Retrieve Centroids** One way is computing centroids in the GPU and retrieving them from it. 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 of the mipmap contains the average of all the values in the lowest level. A group of *occlusion queries* must be called in order to get the number of points in each cluster. An occlusion query returns the number of fragments that pass variance tests. In our case, it is stencil test. The procedure is as 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 in communication cost, as our experiments in Section 5.4 will show. We therefore introduce another strategy.
2. **Retrieve the Label Array** In this strategy, we retrieve the label array directly from the stencil buffer where it is stored. To reduce communication

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 GetCentroids ($tex_{in}, tex_{out}[i]$)

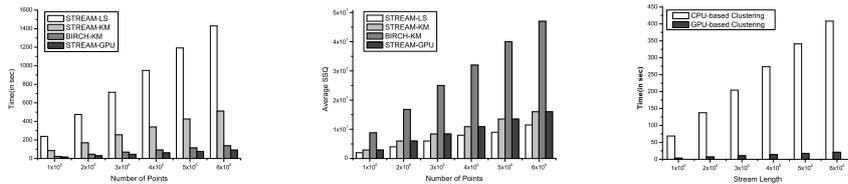
```
1: for  $i = 1; i \leq k; i++$  do
2:   Set stencil reference value as  $i$ ;
3:   Set stencil test to pass if stencil value is equal to the reference value.
4:   Enable Occlusion query  $i$ ;
5:   RenderTexturedQuad( $tex_{in}$ ); { generate a frame of fragments which correspond
   to all the points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

cost, the label array is retrieved from the stencil buffer in an impact mode GL_BYTE. Although 8-bit value constraint exists in this mode (that is the upper bound of k is 256), the requirements of most real applications could be met. We generate the new centroids in the CPU by adding the points with the same label in the label array.

4.4 Clustering Data Stream

We extend our GPU-based method to data stream clustering. Specifically, we demonstrate entire data stream clustering [7] and sliding window clustering [2].

1. **Entire Data Streams Clustering.** We adopt divide-and-conquer methodology [7] and our GPU-L method (abbr. STREAM-GPU) to cluster a data stream. As STREAM-GPU processes data stream block by block, video memory consumption equals block size $m = n^\epsilon$. We compare STREAM-GPU with three CPU-based algorithms: BIRCH-KM, STREAM-KM [7] and STREAM-LS [7]. Figures 5(a)(b) show STREAM-GPU achieves the fastest



(a) Efficiency comparison in entire stream (b) Effectiveness comparison in entire stream (c) Efficiency comparison in sliding window

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

processing speed with competitive SSQ (sum of square distance). Although STREAM-LS achieves the finest SSQ, its processing speed is 15 times slower than STREAM-GPU. STREAM-GPU is more efficient than BIRCH-KM with 200% effective 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, at any time only the N most recent points contribute to the results. We adopt the algorithm in [2]. The basic operation in combination procedure is implemented by our GPU-L. Figure 5(c) shows the clustering costs with window size = 100,000. GPU-based clustering is always better than an optimized CPU-based implementation by about 19 – 20 times.

Streaming clustering benefits greatly from the pipe-line architecture and parallel processing of the GPU, which are well suited for stream processing [14].

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 processor. To generate 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 GPU 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 were: (1) $tc = cc + I/O \text{ cost}$, where tc denotes total cost; cc denotes clustering cost. (2) $cc = pt * m$, where pt denotes the cost of one iteration; m denotes the number of iterations. (3) $pt = dc + gc$, where dc denotes the cost of distance computation and comparison; gc denotes the cost of generating new centroids. The costs of the GPU-based algorithm were: (1) $cc_{gpu} = pt_{gpu} * m + m2g$, where $m2g$ denotes the cost of sending data from main memory to GPU. (2) $pt_{gpu} = dc + gc + g2m$, where $g2m$ denotes the cost of retrieving data from GPU to main memory.

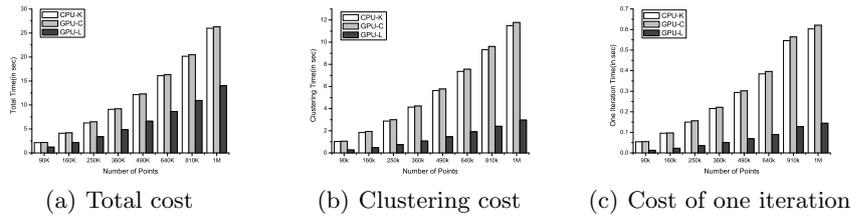


Fig. 6. GPU-based vs. CPU-based clustering

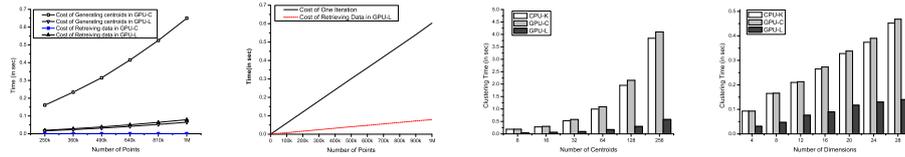


Fig. 7. Costs of **Fig. 8.** Cost of **Fig. 9.** Clustering **Fig. 10.** Clustering
generating centroids retrieving data vs. cost vs. k cost vs. d
and retrieving data cost of one iteration

Unless otherwise mentioned, the experiments adopted $d = 8$, $k = 8$ normal distributed synthetic data set. GPU-L denotes GPU-based clustering with retrieving the label array, GPU-C denotes GPU-based clustering with retrieving centroids, and CPU-K denotes k-means implemented by CPU.

5.2 Total Cost

Figure 6(a) shows that the total cost 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 iteration time increases. The performance improvement of GPU-L and GPU-C relative to CPU-K become 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 of CPU-K's. 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 we compare distance via depth test, no branch mispredictions exist in the GPU implementation, which further improves performance. 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. GPU-L constantly outperforms CPU-K by four times.

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 texture writing, which is 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 the 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 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.

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] and 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 provide a GPU-based iterative clustering [9]. As being designed for geometry processing, it is not suit for large scale data set let alone data stream. Furthermore, it doesn't well utilize the pipeline of GPU. 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. [13] used GPUs for spatial selection and join operations. Govindaraju et al. [5] presented algorithms for predicates and aggregates on commodity 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. Baciú, 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.