Speeding up K-Means Algorithm by GPUs

You Li, Kaiyong Zhao, Xiaowen Chu
Department of Computer Science
Hong Kong Baptist University
{youli, kyzhao, chxw}@comp.hkbu.edu.hk

Abstract

Clustering algorithm is always facing the efficiency challenge due to the continuously fast increasing data volume. Exploiting parallel computing is one of the most promising solutions. In this paper, we conduct systematic research on paralleling the most important clustering algorithm k-Means on GPUs. We find that data dimension is an important parameter that should be taken into consideration when designing the parallel algorithms. Particularly, two algorithms have been designed for low and high dimension data respectively to make the best use GPUs. For low dimension data, we mainly utilize GPU registers to decrease data access latency. For high dimension data, we and then design a novel algorithm which simulates matrix multiplication and exploits GPU shared memory to achieve high compute to global memory access ratio. As a result, our GPU-based k-Means algorithm is four to ten times faster than the best reported GPU-based algorithm.

I. Introduction

Clustering is a method of unsupervised learning that partitions a set of records into clusters, such that intra-cluster similarity is maximized while inter-cluster similarity is minimized [1, 2]. The *k*-Means algorithm is one of the most popular clustering algorithms [3] and is widely used in many fields such as statistical data analysis, pattern recognition, image analysis and bioinformatics [4, 5]. The running time of *k*-Means algorithm grows with the increase of the data size and data dimension. Hence clustering large-scale datasets is usually a time-consuming task. Parallelizing *k*-Means is a promising approach to overcoming the challenge of the huge computational requirement [6-8]. In [6], P-CLUSTER uses a client-server model, in which a server process partitions data into blocks and sends the initial centroid list and blocks to each of clients. P-CLUSTER has been further enhanced by pruning as much computation as possible while preserving the clustering quality [7]. In [8], the *k*-Means clustering algorithm has been parallelized by exploiting the inherent data-parallelism and utilizing message passing.

Recently, as a general-purpose and high performance parallel hardware, Graphics Processing Units (GPUs) develop continuously, and supply another platform for parallelizing *k*-Means. GPUs are dedicated hardware for manipulating computer graphics. Due to the huge computing demand for real-time and high-definition 3D graphics, the GPUs have evolved into highly parallel many-core processors. The advances of computing power in GPUs have driven the development of general-purpose computing on GPUs (GPGPU). In this paper, we use a general-purpose parallel programming model, namely Compute Unified Device Architecture (CUDA) [9, 10] to implement our parallel *k*-Means algorithm.

CUDA has been used for speeding up a large number of applications [11, 12]. Some clustering algorithms have also been implemented on the GPUs, including *k*-Means. There are three main GPU-based *k*-Means algorithms: *GPUMiner* [13], *UV_k-Means* [14], and *HP_k-Means* [15]. *UV_k-Means* achieves a speedup of ten to forty as compared with a four-threaded *Minebench* [16] running on a dual-core, hyper-threaded CPU. *HP_k-Means* claims another speedup of two to four compared with *UV_k-Means* and twenty to seventy speedup compared with *GPUMiner*. Obviously, those works reveal the high performance advantage of the GPU. However, they only investigated some general optimization rules and utilized parts. Therefore, it is still worth analyzing how to apply the optimization rules in the design of the algorithm and how to better utilize the GPU.

Thus, in this paper, we conduct systematic research on paralleling the *k*-Means using CUDA and optimizing the algorithm in detail. Particularly, considering the character of data dimension, we design two strategies for low and high dimension data respectively. For low dimension data, we adopt a relatively simple workflow and mainly utilize the register to achieve a low global memory access times and latency. For high dimension data, we present a novel idea on the relationship between *k*-Means and Matrix multiplication, and design a shared memory based *k*-Means algorithm. The experiment shows that our *k*-Means compares very favorably with *GPUMiner*, *UV_k-Means* and *HP_k-Means*, and yet achieves a speedup of 100, 10 and 5 around respectively.

The paper is organized as follows: section II introduces the existing GPU-based *k*-Means algorithms; section III presents the design strategy and implementation of our *k*-Means algorithm; section IV presents our experimental results and compares our *k*-Means algorithm with existing ones. Section V concludes this paper.

II. Related work

To the best of our knowledge, there are mainly three GPU-based *k*-Means algorithms, *UV_k-Means*, *GPUMiner*, and *HP_k-Means*, the former two of which are open source. To well understand the GPU-based algorithm, we briefly introduce the architecture first.

A. *The GPU architecture*

We take NVIDIA GTX280 as an example to show the GPU architecture. GTX 280 has 30 Streaming Multiprocessors (SMs), and each SM has 8 Scalar Processors (SPs), resulting a total of 240 processor cores. The SMs have a Single-Instruction Multiple-Data (SIMD) architecture: At any given clock cycle, each SP executes the same instruction, but operates on different data. Each SM has four different types of on-chip memory, namely registers, shared memory, constant cache, and texture cache, as shown in Fig.1. Constant cache and texture cache are both read-only memories shared by all SPs, but with very limited size. Off-chip memories such as local memory and global memory have relatively long access latency, usually 400 to 600 clock cycles [10]. The properties of the different types of have been summarized in [10, 17]. In general, the scarce shared memory should be carefully utilized to amortize the global memory latency cost.

In CUDA model, the GPU is regarded as a coprocessor capable of executing a great number of threads in parallel. A single source program includes host codes running on CPU and also kernel codes running on the GPU. Compute-intensive and data-parallel kernel codes run on the GPU. The threads are organized into thread blocks, and each block of threads are executed concurrently on one SM. Threads in a thread block can share data through the shared memory and can perform barrier synchronization. But there is no synchronization mechanism for different thread blocks besides terminating the kernel. Another important concept in CUDA is *warp*, which is formed by 32 parallel threads and is the scheduling unit of each SM. When a warp stalls, the SM can schedule another warp to execute. A warp executes one instruction at a time, so full efficiency can only be achieved when all 32 threads in the warp have the same execution path. There are two consequences: first, if the threads in a warp have different execution paths due to conditional branch, the warp will serially execute each branch which increases the total time of instructions executed for this warp; secondly, if the number of threads in a block is not a multiple of warp size, the remaining instruction cycles will be wasted.

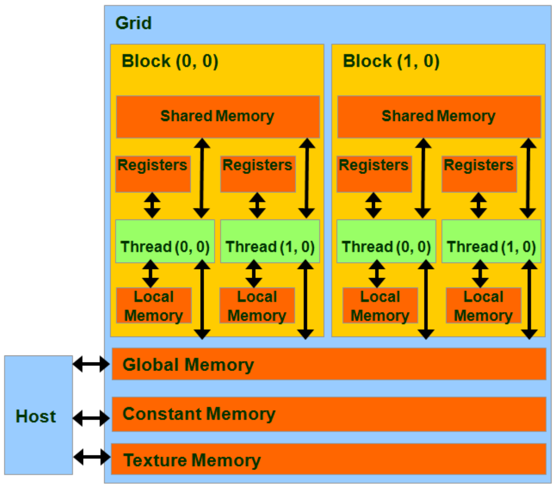

Figure 1. Hardware architecture of the GPU

B. *UV_k-Means*

In the *UV_k-Means*, in order to avoid the long time latency of global memory, they copy all the data to the texture, which has cache mechanism. Then, they use constant memory to store *k* centroids, which is also more efficient than global memory. The grid and block are organized as follows: each thread is responsible for one data point, finding the nearest centroid, and each block has 256 threads, so the grid has [*n*/256] blocks.

The work flow is straight forward: firstly, each thread calculates the distance from one corresponding data point to every centroid and finds the minimum distance and corresponding centroid. Secondly, each block calculates a temp centroid set based on several data points, and each thread calculates one dimension of the temp centroid. Thirdly, CPU copies the temporal centroid sets from the GPU to the CPU, and serially calculates the final new centroid set by adding the temporal centroid sets.

UV_k-Means has achieved a speed-up of twenty to forty through our experiment, mainly by assigning each data point to one thread and utilizing the cache mechanism to get a high reading efficiency. However, the efficiency still could be further improved by using another memory mechanism, shared memory, as well as considering not only one data point once a time, which are the two key points considered in this paper.

C. GPUMiner

GPUMiner puts all the input data in the global memory, and loads k centroids to the shared memory. Each block has 128 threads, and the grid has n/128 blocks. The workflow is also straight forward: firstly, each thread calculates the distance from one data point to every centroid, and changes the suitable bit into true in the bit array, which stores the nearest centroid for each data point; secondly, each thread is responsible for one centroid, finds all the corresponding data points from the bit array and takes the mean of those data points as the new centroids.

The main problem of *GPUMiner* is the utilization of memory in the GPU, since *GPUMiner* accesses most of the data (input data point) from global memory, which is obviously the slowest one, and thus results in a low efficiency. Besides, the main characteristic of *GPUMiner* is designing a bitmap-based algorithm, which makes it easy to find each data set. However, as *HP_k-Means* points out, bitmap approach is elegant in expressing the problem, but it is not a good method for performance, since bitmap takes more space when k is large and requires more shared memory. We will present its performance in detail in section IV.

III. Design and implementation

The k-Means algorithm is one of the most useful clustering methods. Given a set of n data points $R = \{r_1, r_2, ..., r_n\}$ in a d dimensional space, the task of k-Means is to partition R into k clusters ($k < n$) $S = \{S_1, S_2, ..., S_k\}$ such that $\sum_{i=1}^{k} \sum_{x_j \in S_i} \| x_j - \mu_i \|^2$ is minimized, where μ_i is the mean of S_i.

The k-Means algorithm iteratively partitions a given dataset into k clusters. It first selects k data points as the initial centroids. Then the algorithm iterates as follows: (1) Calculate the Euclidean distance between each pair of data point and the centroid; (2) Assign each data point to its closest centroid; (3) Calculate the new centroid by taking the mean of all the data points in each cluster. The iteration terminates when the changes in the centroids are less than some threshold or some given iteration time. The whole process is shown in Algorithm 1.

The computational complexity of a single round of k-Means is: O(nkd) in step (1), O(nk) in step(2), O(nd) in step (3). We mainly focus on speeding up step (1). Considering the parameter of data dimension d, we design two GPU-based algorithms for low and high dimension data respectively. For low dimension data, we propose to utilize register and combine Step (1) and Step (2) together. For high dimension data sets, we adopt the shared memory to parallelize Step (1) and apply the most efficient reduction method to speed up Step (2). Step (3) has a relatively low computational complexity of O(nd), and it is difficult to be fully parallelized due to write conflict. So we let GPU handle part of the task that is worthy to be performed on GPU, and then send the remaining part to CPU for execution.

Algorithm 1: CPU-based k-Means

```
// flag: shows whether it still needs to iterate;
// iter: the current round of iteration;
// Max_iter:  the maximum number of iterations;
// d(r, s): the distance between r and the cluster s;
1. while flag && iter <= Max_iter
2.    for each r in R and each s in S
3.        Compute d(r, s);
4.    Find the closest centroid based on the distance;
5.    Compute new centroids;
6.    if the changes of the centroids are less than threshol
7.        flag ← false;
8.    iter = iter + 1;
9. end of while
```

A. Finding closest centroid

The CPU-based algorithm of finding closest centroid is straightforward, as shown in Algorithm 2. Since the algorithm computes the distance between each data point and each centroid, our first method to parallelize Algorithm 2 is dispatching one data point to one thread, and then each thread calculates the distance from one data point to all the centroids, and maintains the minimum distance and the corresponding centroid, as shown in Algorithm 3. Line 1 and 2 show how the algorithm designs the block and gird; line 3 tells how to calculate the position of the corresponding data point for each thread in global memory; line 4-5 load the data point into the register; line 6-11compute the distance and maintain the minimum one.

It is worth pointing out that the key step of achieving high efficiency is loading the data point into the register, which ensures reading the data point from global memory only once when calculating the distances between the data point and k centroids. Obviously, reading from register is much faster than reading from global memory. The experiment in section IV shows the advantage of Algorithm 3 compared with the best published results. However, the problem of Algorithm 3 is the limited size of the register. In fact, users are not able to control the register right now, and could only utilize register when the data size is appropriate. When the data point cannot be loaded into the register as the dimension grows, it will be stored in local memory, which will increase the reading latency.

In fact, the input data point and the centroid could be viewed as two matrixes *data*[*n*][*d*] and *centroid*[*d*][*k*]; the result distance could be denoted as *d*[*n*][*k*]; and the distance computing process could be described as Algorithm 4, which shares the same flow as matrix multiplication. Based on this character, we design Algorithm 5 for high dimension data, adopting the idea of matrix operation and mainly utilizing the shared memory.

```
Algorithm 2: finding closest centroid based on CPU
//min_D: a temp variable, stores the minimum distance;
// index: stores the min centroid ID for each data point;
1.  for $r_i$ in $R$
2.      for $s_j$ in $S$
3.          Compute $d(r_i, s_j)$;
4.          if $d(r_i, s_j) < min\_D$
5.              $min\_D \leftarrow d(r_i, s_j)$;
6.          index[$i$] $\leftarrow j$;
7.      end of for;
8.  end of for;
```

```
Algorithm 3: finding closest centroid based on the register of the GPU
// threadDim: the dimension of the thread in each block;
// blcokDim: the dimension of the block in each grid;
// blockIdx.x: the current block ID;
// threadIdx.x: the current thread ID;
// data: the address of R;
// i: the ID of data point;
// Gdata: the address of the corresponding data point;
//S: the set of the centroid;
1.  threadDim ← 16x16;
2.  blockDim ← n/256;
3.  $i \leftarrow blockIdx.x \times blockDim + threadIdx.x \times threadDim$;
4.  $Gdata \leftarrow data + i \times d$;
5.  Load the data point from Gdata to the register.
6.  for $s_j$ in $S$
7.      Compute $d(r_i, s_j)$; // read $r_i$ the register
8.      if $d(r_i, s_j) < min\_D$
9.          $min\_D \leftarrow d(r_i, s_j)$;
10.         index[$i$] $\leftarrow j$;
11. end of for;
```

```
Algorithm 4:  distance computing
1.  for i from 1 to n
2.      for j from 1 to k
3.          for m from 1 to d
4.              $d[i][j] \mathrel{+}= (data[i][m]-centroid[m][j])^2$;
5.          $d[i][j] \leftarrow \mathrm{sqrt}(d[i][j])$;
```

```
Algorithm 5:  finding closest centroid based on the shared memory of the GPU
// THIGH: the high of the tile;
// TWIDTH: the width of the tile;
// thread: the dimensions of the block;
// grid: the dimensions of the grid;
// SMData: stores the tile in shared memory;
// TResult: stores the temp distance in shared memory;
// SR: stores the temp distance in global memory;
// Alast : is the upper bound address of data point;
1.  THIGH ← 32, TWIDTH ← 32;
2.  dim thread(THIGH, 2);
3.  dim grid (k/ TWIDTH, n/ THIGH);
4.  indexD points to the corresponding data;
5.  indexC points to the corresponding centroid;
6.  indexR points to the corresponding result;
7.  SMData[TWIDTH][THIGH] in shared memory;
8.  TResult[TWIDTH][ THIGH] in shared memory;
9.  Alast ← indexD + d;
10. do
11. {
12.     Load data from global memory to SMData;
13.     indexD is added by TWIDTH;
14.     Compute the temp distance;
15.     Add the temp distance in TResult ;
16. }while(indexD < Alast);
17. __syncthreads();
18. Write the minimum distance in TResult back to SR;
```

The main idea of Algorithm 5 is decreasing the global memory access times and latency by loading the data into the shared memory tile by tile. Thus, Algorithm 5 reads each data point from global memory only once, the same as Algorithm 3. The key point of Algorithm 5 is how to access the shared memory efficiently, which is achieved by adopting coalescing reading, accessing sixteen continuous address for the thread in a half warp to avoid the bank conflict. The details are described as follows.

As shown in Algorithm 5, each block is in charge of computing a sub result matrix *SR*[*TWIDTH*][*THIGH*]. Each block has *THIGH**2 threads, and each thread computes a column of *sr*. line 4-6 finds the right position of the data, the centroid and the result matrix for each thread; in the loop from line 11 to 16, the algorithm loads a tile data from global memory to the shared memory, and computes the temp distance saved in *TResult*; the loop ends when the whole row has been calculated, as shown in Fig.2.

After calculating the temp minimum distances from each sub matrix, we need to get the final minimum one. The computational complexity of CPU-based algorithm is O(*nk*). On GPU, if each data point is assigned to one thread, the computational complexity decreases to O(*k*). If each data point has *k* threads, the complexity will be O(log*k*). However, the cost of the parallel algorithm is (*k*/log*k*) and not efficient, which is measured by threads multiplying with complexity. In fact, according to Brent's theory, we can assign O(k/log$k$) threads for each data point, finding the minimum distance from a k array; each thread does O(log$k$) sequential work; then all O(k/log$k$) threads cooperate for O(log$k$) steps. And the computational complexity is O(log$k$) while the cost is O(k). This paper will not discuss the implementation in detail but adopt Brent's theory.

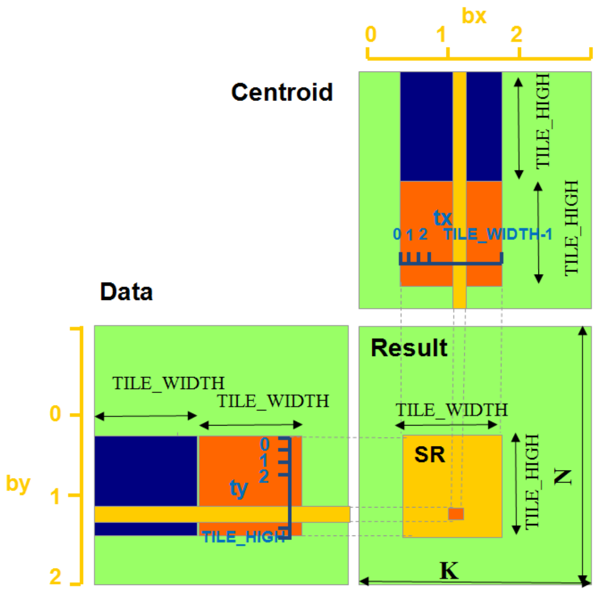

Figure 2 tile-based distance computing process

B. *Calculating the new centroid*

The result of finding the closest centroid is an array *index*[*n*], which stores the closest centroid for each data point. The data points belonging to the same centroid constitute one cluster. Calculating the new centroids is taking the mean of all the data points in each cluster. As shown in Algorithm 6, the computational complexity is O(nd+kd), and it is difficult to be fully parallelized. Since if we assign each data point to a thread, it will generate write conflict when adding the data to the shared centroid. On the other hand, if we assign each centroid to a thread, the computing power of the GPU is has limited utilization.

In this paper, we design an algorithm, which adopts the "divide and conquer" strategy: divide the data into groups; reduce each group and get temp centroids; then divide the temp centroids and reduce iteratively on the GPU until n' is smaller than M, which means the GPU has no advantage than the CPU for further computing; calculate the final centroids on the CPU, as shown in Algorithm 7. By dividing the data into groups, the write conflict decreases, since each group writes its own temp centroids and has no influence to other groups. And it is still a profitable method when we consider the additional data transfer through our experiments (see section IV). Besides, it is necessary to point out that M in Algorithm 7 line 1 should be the multiple of the number of SM, which can ensure a high schedule efficiency on the GPU.

Algorithm 6: CPU_based method for Calculating the new centroids

```
// count: stores the number of data points in each clusters;
// new_cen: the address of the new centroid;
// data: is the address of data point set R;
1.  for i from 1 to n
2.     ++count[ index[i] ];
3.     for j from 1 to d
4.        new_cen [index[i]][j] += data[i][j];
5.  for i from 1 to k
6.     for j from 1 to d
7.        new_cen [i][j] /= count[i];
```

Algorithm 7: GPU_based method for Calculating the new centroids

```
// n': is the number of groups to be divided;
1.  M is the multiple of the number of SM;
2.  n' ← n/M;
3.  Divide n data points in to n' groups;
4.  Compute n' temp centroids on the GPU;
5.  while n' > M
6.  {
7.     Divide n' temp centroids into n'/M groups;
8.     n' ← n'/M;
9.     Compute  n' temp centroids on the GPU;
10. }
11. Reduce n' temp centroids into final centroids on CPU;
```

IV. Experiments

The experiments were conducted on a PC with an NVIDIA GTX280 GPU and an Intel(R) Core(TM) i5 CPU. GTX 280 has 30 SIMD multi-processors, and each one contains eight processors and performs at 1.29 GHz. The memory of the GPU is 1GB with the peak bandwidth of 141.7 GB/sec. The CPU has four cores running at 2.67 GHz. The main memory is 8 GB with the peak bandwidth of 5.6 GB/sec. We use Visual Studio 2008 to write and compile all the source code. The version of CUDA is 2.3. We calculate the time of the application after the file I/O, in order to show the speedup effect more clearly.

The experiments contain two parts: first, we compare our results with the best published results of *HP_k-Means*, which is mainly on low dimension data. Second, we compare our *k*-Means with *UV_k-Means* and *GMiner* on high dimension data. Each of the experiments is repeated ten times and the average results are reported.

A. *On low dimension data*

Here we choose exactly the same data sets with *HP_k-Means* as follows: *n* has two values, two million and four million; *k* has two values, one hundred and four hundred; *d* also has two values, two and eight. Each dimension is a floating point number, and generated randomly.

Table 1: Speed of k-Means on low dimension data

n	*k*	*d*	*Our k-Means*	*HP k-Means*	*UV k-Means*	*GPU Miner*
2 million	100	2	0.22	1.45	2.84	61.39
	400	2	0.64	2.16	5.96	63.46
	100	8	0.24	2.48	6.07	192.05
	400	8	0.65	4.53	16.32	226.79
4 million	100	2	0.31	2.88	5.64	130.36
	400	2	1.22	4.38	11.94	126.38
	100	8	0.42	4.95	12.85	383.41
	400	8	1.26	9.03	34.54	474.83

Note: the time is in second. The hardware environment of HP is as follows: NVIDIA GTX280 GPU; Intel Xeon CPU, 2.33GHz; 4GB memory.

As is shown in Table 1, our *k*-Means is the most efficient one among the four algorithms. It is four to ten times faster than the best published results: *HP_k-Means*, ten to twenty faster than *UV_k-Means* and one hundred to three hundred faster than *GPUMiner*. Since *HP* only says some optimization rules without publishing the source code. We mainly analyze the difference between our *k*-Means and *UV_k-Means*.

The workflows of the two algorithms are very similar: each thread finds the minimum centroid for each data point. The main difference is the memory utilization: *UV_k-Means* puts the data on the texture and puts the centroids on the constant; our *k*-Means firstly loads the data on the register, and reads the data from the register each time when calculating the distance from each centroid, resulting in a low global memory access times and latency, since reading from register is by far faster than reading from other memories.

Also shown in Table 1, our *k*-Means is insensitive with dimension, since the time differs a little when the dimension changes from two to eight, which also results from the utilization of the register. On the other hand, when *k* grows, the algorithm has to access the global memory more, which is proportional to the *k*.

Through our experiment, when the dimension is larger than sixteen, the data point cannot be loaded into the register (The compiling and building information of Visual Studio can indicate the memory that the program will use), and the speed decreases sharply because of accessing the local memory. So, we use Algorithm 5, shared memory based algorithm to deal with the high dimension data.

B. *On high dimension data*

Here we use the data from the KDD Cup 1999 [10], and choose two data sets, which have 51200 and 494080 data points. Each data point contains 34 features, and each one is floating point. We compare our algorithm with *GPUMiner* and *UV_k-Means*.

The results are shown in Table 2. Our *k*-Means is four to eight faster than *UV_k-Means*, ten to forty faster than *GPUMiner*, and one hundred to two hundred faster than a CPU based *k*-Means algorithm which is also developed by us, using Algorithm 1 and Algorithm 6. We also mainly analyze the difference between our *k*-Means and *UV_k-Means*.

Table 2: Speed of k-Means on high dimension data

Data set	Our *k*-Means	UV *k*-Means	GPU Miner	CPU *k*-Means
51200	0.43	1.86	4.26	35.79
494080	1.15	8.67	40.6	224.47

Table 3: Time distribution of our k-Means algorithm

Function/data set		Find the closest centroid	Compute new centroid
51200	GPU	0.07	0.16
	CPU	33.5	2.28
494080	GPU	0.87	0.18
	CPU	207.78	16.67

When dealing with high dimension data, larger than sixteen, our algorithm loads the data tile by tile into the shared memory. Thus it accesses the global memory only once for each data point. *UV_k-Means* adopts texture to store the data point and decreases the global memory reading latency. However, it depends on the cache mechanism, and if the cache missing grows, the efficiency would lower down. On the other hand, the shared memory could perform more stably.

As shown in Table 3, *finding the closest centroid* achieves a speedup of forty to two hundred compared with our CPU-based algorithm, while *computing new centroid* achieves a speedup around ten, which further prove the advantage of our algorithm.

V. Conclusions

In this paper, we proposed a GPU-based *k*-Means algorithm. It presents mainly two novel ideas: first, based on the dimension of the data, our *k*-Means algorithm chooses two different strategies. For low dimension data, our algorithm utilizes GPU registers, and achieves a speedup of four to ten than *HP_k-*

Means. For high dimension data, our algorithm firstly observes the connections of each data point, analyzes the relationship with matrix multiplication and reduction, adopts shared memory to avoid multiple accessing the global memory, increases the number of the computing operation for each global memory access, and achieves a speedup of four to eight as compared with *UV_k-Means*.

The algorithm presented in this paper could deal with a finite scale of data set, limited by the global memory size of the GPU. Consequently, when the data set is larger than it, new strategies have to be designed. A possible method is to divide the data set into several parts, each of which could be loaded into global memory of the GPU; find the closest centroid for each data point in each part; accumulate the data point to their corresponding centroid part by part and get the new centroids. It is worth pointing out that the above method could adopt the algorithm in this paper when dealing with each part. Another feasible approach is using the GPU cluster, which is a computer cluster and each node is equipped with a GPU, and conducted as follows: divide the data into several parts, each of which is assigned to one node; on each node, find the closest centroid for each data point, using Algorithm 5; adopt the divide and conquer strategy as Algorithm 7 to calculate the new centroid.

In summary, the results of this paper prove that adopting GPUs is a promising acceleration method for clustering algorithms to deal with large scale data. Moreover, the analysis method and optimization rules presented in this paper could also be applied in speeding up other data mining algorithms.

References

[1] P.-N. Tan, M. Steinbach, and V. Kumar, Introduction to Data Mining, Addison-Wesley Companion Book Site 2006.

[2] A. K. Jain and R. C. Dubes, Algorithms for clustering data, Prentice-Hall, 1988.

[3] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg, “Top 10 algorithms in data mining,” knowledge information systems, vol. 14, pp. 1-37, 2008.

[4] X. Wang and M. Leeser, “K-means Clustering for Multispectral Images Using Floating-Point Divide,” in Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines: IEEE Computer Society, 2007.

[5] H. Zhou and Y. Liu, “Accurate integration of multi-view range images using k-means clustering,” Pattern Recogn., vol. 41, pp. 152-175, 2008.

[6] D. Judd, P. K. McKinley, and A. K. Jain, “Large-Scale Parallel Data Clustering,” in Proceedings of the International Conference on Pattern Recognition (ICPR '96) Volume IV-Volume 7472 - Volume 7472: IEEE Computer Society, 1996.

[7] D. Judd, P. K. McKinley, and A. K. Jain, “Large-Scale Parallel Data Clustering,” IEEE Trans. Pattern Anal. Mach. Intell., vol. 20, pp. 871-876, 1998.

[8] I. S. Dhillon and D. S. Modha, “A Data-Clustering Algorithm on Distributed Memory Multiprocessors,” in Revised Papers from Large-Scale Parallel Data Mining, Workshop on Large-Scale Parallel KDD Systems, SIGKDD: Springer-Verlag, 2000.

[9] NVIDIA CUDA: http://developer.nvidia.com/object/cuda.html.

[10] NVIDIA CUDA Compute Unified Device Architecture: Programming Guide, Version 2.0, June 2008.

[11] S. A. Manavski, “CUDA compatible GPU as an efficient hardware accelerator for AES cryptography,” In Proceedings of IEEE International Conference on Signal Processing and Communication, p. 4, Nov. 2007.

[12] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. Hwu, “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA,” in Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming Salt Lake City, UT, USA: ACM, 2008.

[13] W. Fang, K. K. Lau, M. Lu, X. Xiao, C. K. Lam, P. Y. Yang, B. He, Q. Luo, P. V. Sande, and K. Yang, “Parallel Data Mining on Graphics Processors,” Technical Report HKUSTCS08, 2008.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, “A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA,” Journal of Parallel and Distributed Computing, 2008.

[15] R. Wu, B. Zhang, and M. Hsu, “Clustering billions of data points using GPUs,” in UCHPC-MAW '09: Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop, Ischia, Italy, 2009, pp. 1-6.

[16] J. Pisharath, Y. Liu, W.-k. Liao, A. Choudhary, G. Memik, and J. Parhi, “NU-MineBench 2.0,” CUCIS Technical Report CUCIS-2005-08-01, Center for Ultra-Scale Computing and Information Security, Northwestern University, 2005.

[17] J. C. Shafer, R. Agrawal, and M. Mehta, “SPRINT: A Scalable Parallel Classifier for Data Mining,” VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, pp. 544-555, 1996.