Speeding up Generalized Fuzzy k-Means Clustering Using m Nearest Cluster Centers Algorithm on GPU

Dinh-Trung Vu1, Jim Z. C. Lai2, Wei-Chung Lee3

Department of Computer Science and Engineering

National Taiwan Ocean University, Keelung 202, Taiwan, R.O.C.

1 Email: [trungvdt49@gmail.com](mailto:trungvdt49@gmail.com)

2 Email: [zclai@mail.ntou.edu.tw](mailto:zclai@mail.ntou.edu.tw)

3 Email: [10357004@mail.ntou.edu.tw](mailto:10357004@mail.ntou.edu.tw)

***Abstract*―The graphics hardware is becoming increasingly more powerful and programmable with the introduction of Graphics Processing Units (GPU) like the NVidia GeForce series. The GPU’s exceed the ordinary general purpose CPU’s ability to do ﬂoating point operations due to the massively parallel architecture in the GPU’s. With the newest GPU’s one actually have enough programmable freedom to do other computations than computer graphics processing. This project will take advantage of this in order to get high performance implementations of data clustering algorithms. In this project we will implement a data clustering algorithm, which is Generalized Fuzzy k-Means Clustering Using m nearest Cluster Centers (GFKM) [4], on a GPU. We also make comparisons with CPU based implementations and analysis the pros and cons of using GPU’s. Our experimental results show that our GPU-based GFKM algorithms are about three to eighteen times faster than the optimized CPU code-based GFKM algorithms.**

***Index* *Terms*―** **GFKM, GPU, CUDA, data clustering.**

I. INTRODUCTION

The GFKM algorithm, which is developed from the fuzzy k-means clustering (FKM) algorithm, uses a data point’s M nearest cluster centers to partition the data set. The experimental results of method GFKM shown that it has the less computing time and the better clustering quality than method FKM. The optimal value of M, which used to obtain the better clustering result for the method GFKM, is 2 [4].

In this paper, we design a parallel GFKM algorithm for GPUs by using a general-purpose parallel programming model, namely Compute Unified Device Architecture [2]. Our first contribution is the observation that the size of the data set, which is number of data points, is an important factor to be considered. However, the GPU-based GFKM have not yet fully exploited the computing power of GPU. Thus, we apply the GPU-based parallel reduction algorithm to reduce data at the steps that they are difficult to be fully parallelized, then we leave the left small data blocks for executing effectively on CPU.

This paper is organized as follows. Section II presents the related works. In Section III, the proposed method is presented. Some experimental results are given in Section IV and Section V presents the final conclusions and future works.

II. RELATED WORK

1. The GPU architecture

The following information is from [2] and [8]. The GPU is specially designed such that more transistors are devoted to compute-intensive, and highly parallel computation rather than data caching and flow control, as schematically illustrated by Fig 1. To archive highly parallel performance, GPUs use the Single-Instruction Multiple-Thread (SIMT) architecture with a very large number of thread processor cores executing the same instruction sequence on different data in parallel. GPUs need many that active parallel threads or smaller cache on each active thread to hide memory latency.

We take NVIDIA GeForce GTX 760 GPU as an example GPU architecture. GTX 760 has up to 6 independent multiprocessors called Streaming Multiprocessors (SMs) with 192 Scalar Processors (SPs), so up to 1152 thread processor cores can run in parallel. Each SM has four different types of on-chip memory, namely registers, shared memory, constant cache, and texture cache, as shown in Fig 2. The properties of the different types of memory have been summarized in [2]. In general, the scarce registers and shared memory should be carefully utilized to amortize the global memory latency cost.

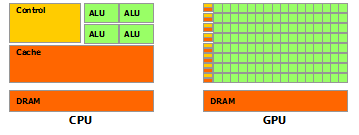


Fig 1.The GPU devotes more transistors to data processing

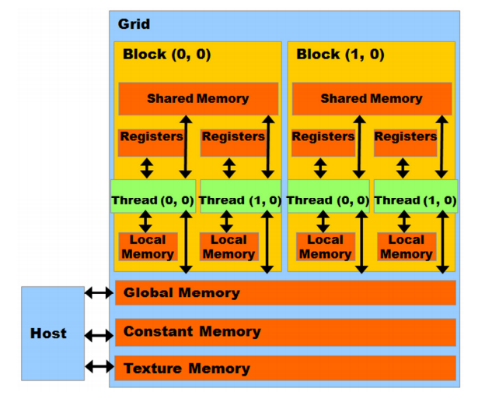


Fig 2. Hardware architecture of the GPU

In CUDA model, GPU is regarded as a coprocessor which is 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 GPU. Compute-intensive and data-parallel tasks have to be implemented as kernel codes so as to be executed on GPU. GPU 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 native synchronization mechanism for different thread blocks except by 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; second, if the number of threads in a block is not a multiple of warp size, the remaining instruction cycles will be wasted. Besides, when accessing the memory, half-warp executes as a group, which has 16 threads. If the half-warp threads access the coalesced data, the data access operation will perform within one instruction cycle. Otherwise, the access operation will occupy up to 16 instruction cycles.

In addition to, if many threads in a warp want to read the same address in global memory, this data is broadcasted. Also if many threads in a warp want to write to the same address in global memory, there is a serialization as the write accesses are done by an atomic primitive. Otherwise one (undefined) thread will win and get the write through, the others will be lose.

2. GFKM algorithm [4].

(1) Input an initial set of cluster centers *SC*0 = {**C***j* (0)} and the values of ε and *M*. Set *p* = 0. Let *dij,* , and *NNTi*, responding to the squared Euclidean distance between data point **X***i* and centroid **C***j*, the squared Euclidean distance between X*i* and the *l*th nearest neighbor of **X***i*, and the set of *M* nearest cluster centers for **X***i*.

(2) Given the set of cluster centers *S*C*p*, calculate *dij* for *i* = 1 to *N* and *j* = 1 to *k*, and update and *NNTi*. Then, update membership  using equation (1). If C*j*∈*NNTi* is the *l*th nearest neighbor of X*i*, set  = ; otherwise let  = 0.

 =  (1)

, for *i* = 1 to *N* and *l* = 1 to *M*, where *q* is the fuzzifier, which is shown in reference [7]. The optimal value of the fuzzifier *q* can be estimated using equation (2).

*q* = 1+(+22.05)*d*-2+(+0.243)*d*-0.0406ln(*N*)-0.1134 (2)

, where *N* is the number of data points in a data set and *d* is the data dimension.

(3) Calculate the center for each cluster using equation (3) to obtain a new set of cluster representatives *SCp+*1 = {**C***j*(*p*+1)}.

**C***j*(*p*+1) =  (3)

, for S*j* = {**X***i*: **C***j* ∈ *NNTi*, *i* = 1 to *N*} (3)

Note here that, at step (2), values  instead of  are updated using in equation (3).

(4) If  < ε, for *j =* 1 to k, or p > maximum iteration, then stop, where ε > 0 is a very small positive number, and maximum iteration > 0 is the maximum number of iterations. Otherwise set *p = p + 1* and go to step (2).

The computational complexity of GFKM is also O(*Nkt*), where *t* is the number of iterations.

III. Parallel GFKM algorithm on GPU

1. Updating membership

On CPU, the updating membership has two levels of loop, one for each data point and other for each centroid. On GPU, there only has one level of loop, because the loop for *N* data points has been dispatched to *N* threads working in parallel. It is worth pointing out that the key step of achieving high efficiency is loading the data points into the on-chip registers, which ensures that reading the data point from global memory happens only once when calculating the distances between the data point and *k* centroids. Reading from register is much faster than reading from global memory. Besides, coalesced access to the global memory also decreases the reading latency.

In addition to, threads read the same centroids in global memory, so the centroids are broadcasted to threads. As the size of the centroids can fit inside the shared memory available to each thread block, about maximum 48 KB, each thread will write at least one value to shared memory. Then, all threads inside the thread block can use the same centroids in shared memory. However, the low latency of the shared memory will not hide the high reading latency of the global memory when each thread reads too many values from the global memory. Generally, in our experimental condition, the performance was only improved as each thread writes less than five values to shared memory.

2. Calculating new centroids

The parallel algorithms are designed for three different strategies as shown below.

Strategy (1): When the number of data points *N* is relatively small, the number of active threads in parallel is small. The algorithms based on GPU are inefficient, so the algorithms based on CPU are still used in this case. In other words, the parallel computational power of GPU cannot be fully exploited.

Strategy (2):When *N* is relatively large, and *M* and *k* are close to each other, we design algorithms using equation (4) with the computational complexity O(*Nkd* + *kd*) of FKM. Note here that, equation (4) uses and do not need *NNT*, so must be updated not and *NNT* is not used at step (1). However, after the previous steps, we can see that data are not coalesced as shown in Fig 3. Hence, we transpose data points from *N*×*d* to *d*×*N*, and memberships from *N*×*k* to *k*×*N* using cuBLAS library in CUDA Toolkit [1]. This helps achieving coalesced access to the global memory as shown in Fig 4.



C*j*(*p*+1) = (4)





Fig 3. Un-coalesced data access



Fig 4. Coalesced data access

After the data is transposed, we customize the final optimized kernel version of the parallel reduction algorithm developed by Mark Harris [5] into the reducing kernel for the *j*th cluster and the reducing kernel for the *j*th cluster and the *x*th dimension. Above kernels may be executed concurrently or interleaved in different streams. The first kernels is executed by stream #1, and the others are executed by other streams. After all streams run on GPU completed, the final small output “block sums” of as well can reduce very fast on GPU or CPU.



Strategy (3): When *N* is relatively large, and *k* is significantly greater than *M*, the GPU-based algorithms is still designed using equation (3) of GFKM. Thus, and *NNT* must be updated at step (1). In equation (3), the data points accessing is discrete because all the data points do not belong to the same cluster. In this strategy, the data points are assigned into each correct cluster to achieve coalesced access to the global memory. However, this will hurt performance due to the big time-consuming on the large data sets. Instead of that, data points are transposed from *N*×*d* to *d*×*N* first to reduce the number of un-coalesced data point accesses. Then, to hide un-coalesced membership accesses, we sort array *NNT* by cluster indices as keys, point indices and memberships as values. For sorting array NNT, we design algorithm for two different scenarios.



Scenario (1): the parallel counting sort for sorting array *NNT* is divided into four steps. The initializing histogramstep can run very fast using “cudaMemset” function on GPU. The calculating exclusive prefix sums (scan from histogram) can run very fast on both GPU and CPU. The calculating histogram as well final sorting (gathering) step apply the same parallelization technique, atomic operations. The read-modify-write atomic operations in the sense that it is guaranteed to be performed without interference from other threads [2]. In other words, no other thread can access this address until the operation is complete. The atomic operation, which is supported by CUDA as device runtime component, is necessary since lots of threads with the same cluster index increase the histogram array conflict with each other as shown in Fig 5. Note here that, the using atomic functions for counting sort will cause the crossover accesses of threads to data points in the global memory and reduces the performance, as shown in Fig 6.



Fig 5. Histogram write conflicts



Fig 6. Crossover access

Scenario (2): We use the stable sort by keys of Thrust library in CUDA Toolkit for sorting *NNT* array with keys and values are cluster indices and *NNT* indices, respectively. The stable sort helps avoid cross accesses in reduction functions later as show in Fig 7. After sorting, the Thrust binary search (upper bound) function [6] is used for finding the end of each bin of clusters (the cumulative histogram). Next, we use the Thrust adjacent difference function [6] for computing the histogram by taking differences of the cumulative histogram. Final, the sorted *NNT* indices array is used for gathering memberships and point indices.



Fig 7. Un-crossover access

After all two scenarios above, we have the sorted point indices array, sorted memberships array, histogram array. Also same as in Strategy (2), we use the parallel reduction algorithm for the reducing for the *j*th cluster as well for the *j*th cluster and the *x*th dimension. The kernels may be also executed concurrently or interleaved in different streams.



3. Checking convergence

Checking convergence step occupies a very small quota. This step is performed quickly on both CPU and GPU; as a result it does not affect the overall performance. On GPU, each thread executes one dimensional of centroid and each thread block executes for one centroid as shown in Algorithm 20.

IV. EXPERIMENTAL RESULTS

The parallel GFKM algorithm is implemented using CUDA version 7.0. The experiments are conducted on a PC with an NVIDIA GeForce GTX 760 GPU and an Intel(R) Core(TM) i5-4690 CPU. GTX 760 has six SIMD multi-processors, and each one contains 192 processors and performs at 980 MHz. The memory of the GPU is 2GB with the peak bandwidth of 192.2 GB/s. The compute capability of GPU is 3.0 with 48 KB maximum amount of shared memory per multiprocessor (CUDA Wikipedia, 2015). The CPU has four cores running at 3.50 GHz. The main memory is 8 GB with the peak bandwidth of 25.6 GB/s. To show the speedup effect more clearly, the time of the application is measured after the file I/O and the speedup are compared to the optimized CPU codes. For each test, the proposed method and GFKM are executed 10 times. The same set of initial cluster centers is used by both GFKM and the proposed method for each execution. As designed above at the calculating new centroids step, it generated four corresponding versions: (1) on CPU; (2) on GPU using equation of FKM; (3) on GPU using counting sort algorithm; (4) on GPU using stable sort by key of Thrust library.

Test 1: A medium data set generated from three real images: “Lena,” “Baboon,” and “Peppers.” In this test, the data set consists of *N =* 49,152 data points with *d* = 16. The values *M* = 2, *k =* 8, ε = 10-8, and ­maximum iteration = 200are used for the test. The speedup of four GPU-based GFKM versions is shown in Table 1. The updating memberships step on GPU is about twelve times faster than on CPU. The calculating new centroids step on GPU is about 1.5 times faster than on CPU. Checking convergence does not affect the overall performance. In this test, the running on GPU is about six to seven times faster than on CPU.

Table 1: The speedup of four GPU-based GFKM versions on a real data set “LBP”

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Steps | Speedup of method parallel GFKM | | | |
| Ver. 1 | Ver. 2 | Ver. 3 | Ver. 4 |
| 1 | 10.5 | 9.95 | 11.38 | 11.41 |
| 2 | 1.14 | 1.35 | 1.52 | 0.98 |
| 3 | 1.00 | 1.00 | 1.00 | 1.00 |
| Total | 6.14 | 6.40 | 7.32 | 5.92 |

Test 2: In this test, we perform testing speedup of our proposed methods with various number of data points *N* on the “poker” data set as shown in Fig 8. The “poker” data set consists of *N =* 1,025,010 instances, with *d* = 10. The values *M* = 2, *k =* 10, ε = 10-8, and ­maximum iteration = 200are used for this test. From Fig 8, we can see that the speedups of our proposed methods increase when the number of data points increases. The maximum speedup is about twelve times faster of Ver. 3 and Ver. 4 in this test.

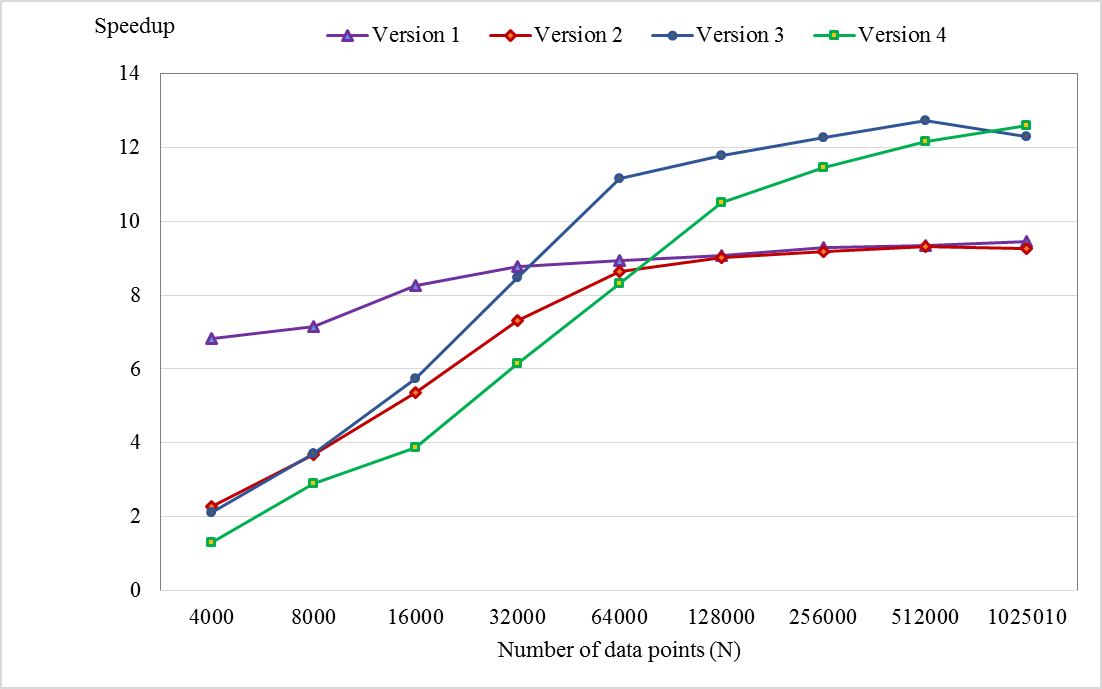


Fig 8. Changing speedup with various number of data points

Test 3: On the “poker” data set also with 1025010 data points, a speedup test is performed for our proposed methods with various number of cluster representatives *k*. From Fig 9, as value *k* is much greater than value *M*, the speedup of version 2 is dramatically reduced. When value *k* is smaller than 5, version 2 becomes more efficient than others, while value *k* is larger than 64, versions 1 remains stable and becomes the most efficient. With values *k* from 5 to 8, version 4 is the best case scenario. Version 3 is better to use where *k* is from 8 to 64.

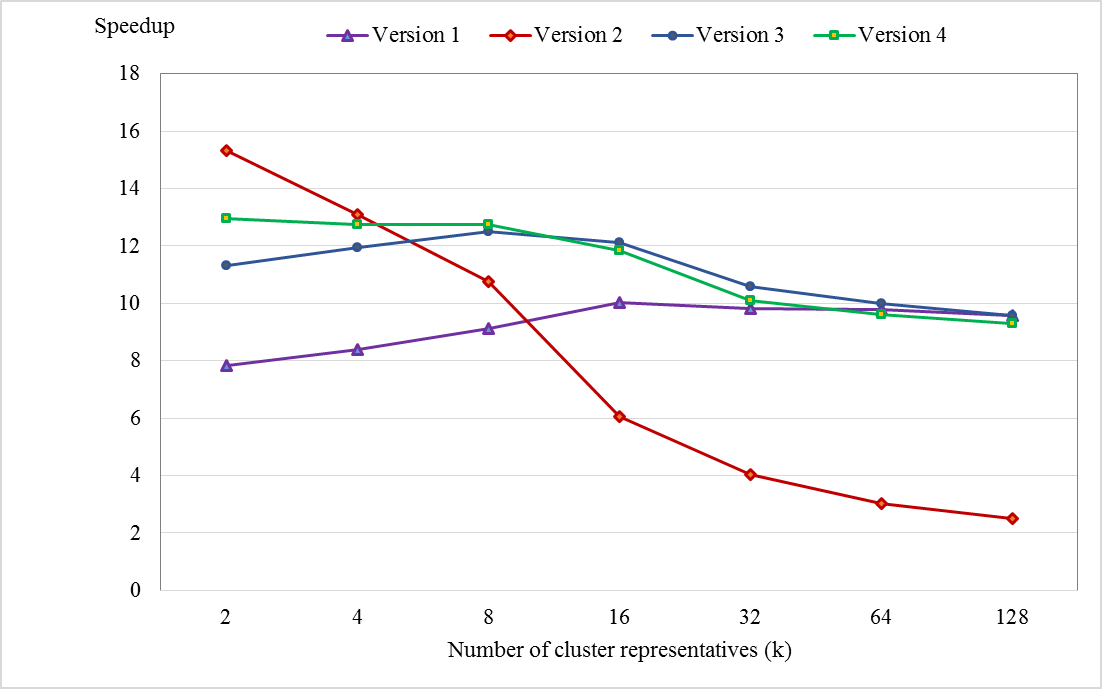


Fig 9. Changing speedup with various number of cluster representatives

Test 4: An evaluation of influence of the data dimension to the speedup of our proposed methods is performed, as shown in Fig 10. The “Synthetic” data set with *N* = 491520, *M* = 2, *k =* 8, ε = 10-8, and ­maximum iteration = 200are used for this test. From Fig 10, we can see that the speedups of our proposed methods are rapidly decreasing as number of dimension *d* is increasing and *d* is smaller than 16. Speedups are more slowly decreasing as *d* becomes larger and larger. In this test, the lowest speedup archived about 4 to 5.5 times faster, in which, speedup of the version 4 is better than the others’.

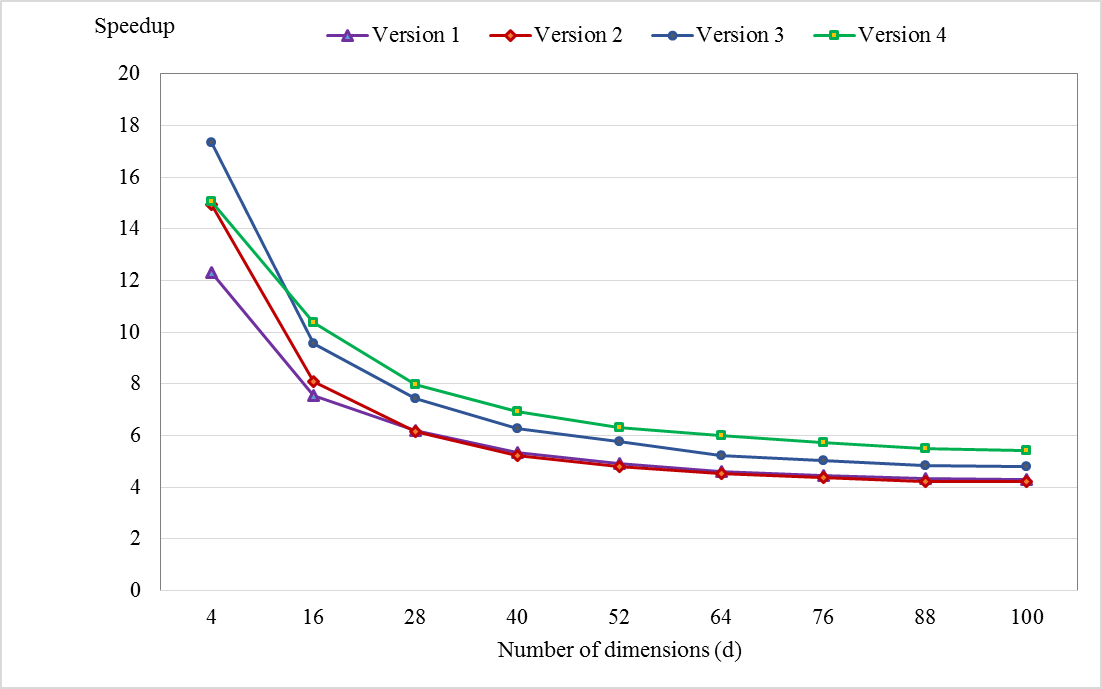


Fig . Changing speedup with various number of dimensions

V. CONCLUSIONS AND FUTURE WORK

In this thesis, we have presented the GPU-based parallel algorithms for speeding up GFKM algorithm. From the experimental results, we can find that the proposed method can reduce significantly the computing time compared with the CPU-based GFKM algorithm. The better speedups are archived with the larger number of data points, smaller number of dimensions and cluster representatives. In general, the speedups have archived about three to eighteen times, especially about six to twenty four times in updating memberships step. We also attempted to design algorithms for the non-fully parallel problem of calculating new centroids step effectively for the different data sets. In particular, we apply calculating new centroids on CPU with data sets having the small number of data points, on GPU using the equation of FKM with data sets having *M* and *k* are close to each other, on GPU using the counting sort or Thrust stable sort by key with the larger data sets.

Our proposed method only achieves high performance when the data point can be loaded into registers. Otherwise, data will be stored in local memory, which will increase the reading latency and decrease performance significantly. Hence, in future, we can utilize registers and the shared memory together by loading the data into the shared memory, tile by tile, and then adopt the idea of matrix multiplication to calculate the temporary distance and store in the on-chip registers [8]. This new proposed method may help increase performance significantly on high-dimensional data sets.

ACKNOWLEDGEMENTS

This work is supported by Prof. Jim Z. C. Lai from Lab 308, Department of Computer Science and Engineering National Taiwan Ocean University, Keelung 202, Taiwan, R.O.C.

REFERENCES

1. *cuBLAS Library.* (2015, March). Retrieved from CUDA Toolkit Documentation: http://docs.nvidia.com/cuda/pdf/CUBLAS\_Library.pdf
2. *CUDA C Programming Guide.* (2015, March). Retrieved from CUDA Toolkit Documentation: http://docs.nvidia.com/cuda/pdf/CUDA\_C\_Programming\_Guide.pdf
3. *CUDA Wikipedia*. (2015). Retrieved from Wikipedia, the free encyclopedia: https://en.wikipedia.org/wiki/CUDA
4. Franklin J. C. Lai, Eric Y. T. Juan, and Jim Z. C. Lai. (2013). Generalized Fuzzy k-Means Clustering Using m Nearest Cluster Centers.
5. Harris, M. (2007). Optimizing Parallel Reduction in CUDA. *NVIDIA Developer Technology*.
6. *Thrust Quick Start Guide.* (2015, March). Retrieved from CUDA Toolkit Documentation: http://docs.nvidia.com/cuda/pdf/Thrust\_Quick\_Start\_Guide.pdf
7. V. Schwammle and O. N. Jensen. (2010, September). A simple and fast method to determine k-means cluster analysis. *Bioinformatics, Vol. 26*, 2841-2848.
8. You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu. (2010). Speeding up K-Means Algorithm by GPU.