Speeding up Generalized Fuzzy k-Means Clustering Algorithm by GPUs

Dinh Trung Vu

Department of Computer Science and Engineering

National Taiwan Ocean University

Keelung, Taiwan 202

R. O. C.

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 image analysis algorithms.

In this project we will implement an image analysis algorithm, which is Generalized Fuzzy k-Means Clustering Using m nearest Cluster Centers (GFKM) [1], 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 two to eight times faster than un-optimized CPU code-based GFKM algorithms, and two to five times faster than optimized CPU code-based GFKM algorithms.

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. However, the running time of the GFKM algorithm as well the method FKM algorithm grows with the increase of the size and also the dimensionality of the data set. Thus, the parallelizing fuzzy *k*-means is a promising approach to overcoming the challenge of the larger computational requirement.

In this paper, we design a parallel GFKM algorithm for GPUs by using a general-purpose parallel programming model, namely Compute Unified Device Architecture (CUDA) [4]. 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 GPUs. 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 CPUs.

This paper is organized as follows. Section II introduces the GPU architecture and review the GFKM algorithm. Section III presents our design of parallel GFKM algorithm on GPUs. Some experimental results are given in Section IV and concluding remarks are presented in Section V.

II. RELATED WORK

We first briefly introduce the GPU architecture, and then review the GFKM algorithms.

A. The GPU architecture

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 Figure 1 [4].

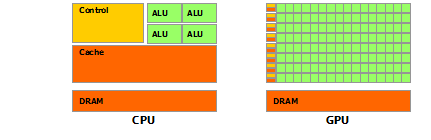


Figure 1. The GPU devotes more transistors to data processing

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. Constant cache and texture cache are both read-only memories shared by all SPs. Off-chip memories such as local memory and global memory have relatively long access latency, usually 400 to 600 clock cycles [4]. The properties of the different types of memory have been summarized in [4]. In general, the scarce registers and shared memory should be carefully utilized to amortize the global memory latency cost.

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.

B. The GFKM algorithm

1. Input an initial set of cluster centers *SC*0 = {**C***j*(0)} and the values of ε and *M*. Set *p* = 0. Let, *NNTi*, and *DNNTi* responding to the squared Euclidean distance between **X***i* and **C***j*, the set of *M* nearest cluster centers for the data point**,** and the set of *M* corresponding shortest distances for the data point****. Then, we calculateand initialize *NNTi* and *DNNTi*.
2. Given the set of cluster centers *SCp*, update membership **** using equation (1). If **C***j*∈*NNTi* is the *l*th nearest neighbor of **X***i*, set  = ****; otherwise let  = 0.

**** = , for *r* = 1 to *N* and *s* = 1 to *M* (1)

1. Compute the center for each cluster using equation (2) to obtain a new set of cluster representatives *SCp+*1 = {**C***j*(*p*+1)}.

**C***j* =  , for S *j* = {**X** *i*: **X** *i*∈ *NNTj*, *i* = 1 to *N*} (2)

1. Calculate, update *NNTi* and *DNNTi* for *i* = 1 to *N*, and calculate distortion value *J* using equation (3).

*J* =  (3)

1. If < ε, then stop, where ε > 0 is a very small positive number. 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. The pseudocode of algorithm as follows:

Algorithm 1: CPU-based GFKM

//

//

//

//

//



III. Design of GPU-based parallel GFKM algorithm

The steps of the GFKM algorithm are: (1) calculating  andinitializing *NNTi* and *DNNTi*, (2) updating memberships, (3) computing the new center for each clusters, (4) calculate  andupdating *NNTi* and *DNNTi*, (5) calculating distortion value *J*. The GPU-based parallel GFKM algorithm is designed as follows:

* Step (1), (2), and (4): We utilize the GPU on-chip registers to minimize the latency of data access [2].
* Step (3): The first, we use the GPU-based counting sort algorithm for sorting array *NNT* with keys and values are cluster indices and point indices, respectively. To reduce  and points for each centroid, we use the GPU-based parallel reduction algorithm [3]. Each kernel processes each dimension of each centroid, and the kernels may be executed concurrently or interleaved in different streams.
* Step (5): We use the GPU-based parallel reduction algorithm for this step [3].

A. Calculating  andinitializing *NNTi* and *DNNTi*

Algorithm 2: Calculating  andinitializing *NNTi* and *DNNTi* based on CPU

//

//

//

//



Algorithm 3: Calculating  andinitializing *NNTi* and *DNNTi* based on GPU

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//



The CPU-based algorithm of calculating  andinitializing *NNTi* and *DNNTi* is shown in Algorithm 2. The first method parallelizes computing the distance between each data point and each centroid in Algorithm 2. One data point is dispatched to one thread, and then each thread calculates the distance from a corresponding data point to *K* centroids, and then initializes *NNTi* and *DNNTi*, as shown in Algorithm 3. Line 1 and 2 show how the algorithm designs the thread block and gird. Line 3 to 6 calculate the position of the corresponding data point, NNT, and DNNT for each thread in global memory. Line 7 loads the data point into the register. Lines 8-13 calculate the distance and initialize *NNTi* and *DNNTi*.

Algorithm 3 only has one level of loop instead of two levels in Algorithm 2, because the loop for *N* data points has been dispatched to N threads, which decreases the time consumption significantly because many threads are 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. Obviously, reading from register is much faster than reading from global memory. Besides, coalesced access to the global memory also decreases the reading latency.

B. Updating memberships

We apply the design as described in the section A. Note here that *NNTi* and *DNNTi* were initialized and updated after step (1) and step (4), respectively.

Algorithm 4: Updating memberships based on CPU

//

//

//

//

//

//



Algorithm 5: Updating memberships based on GPU

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//



C. Computing the new center for each clusters – Updating centroids

Algorithm 6: Updating centroids based on CPU

//

//

//

//

//

//



The sequential code of the updating centroids step is shown in Algorithm 6 with the computational complexity O(NMd+kd). Each data point ****belonging to *M* nearest cluster centers determined by *NNTi*, and the data points belonging to the same centroid constitute one cluster. It is difficult to be fully parallelized. If we assign each element in *NNT* to a thread, it will generate write conflict when adding the data to the shared memory. On the other hand, if we assign each centroid to a thread, the computing power of the GPU cannot be fully utilized. However, if we sort array *NNT* first with keys and values are cluster indices and point indices, respectively, then the updating centroid can be executed effectively using the parallel reduction algorithm and the concurrent kernels on GPU. The counting sort algorithm for sorting array *NNT* is divided into four steps as in Table 1.

Table 1: The sequential code of counting sort for sorting array *NNT*

|  |  |  |
| --- | --- | --- |
| Steps | Function | Pseudocode |
| 1 | Initialize histogram | **for** *i* = 1 **to** *k* **do** *histogram*[*i*] = 0; |
| 2 | Calculate histogram | **for** *i* = 0 **to** *N\*M* **do** *histogram*[ *NNT*[*i*] ]++; |
| 3 | Calculate exclusive prefix sums (scan) | *scan*[1] = 0;  **for** *i* = 1 **to** *k-*1 **do** *scan*[*i+*1] *= histogram*[*i*] *+ scan*[*i*]; |
| 4 | Sort | **for** *i* = 0 **to** *N\*M* **do** *ouput*[ scan[ NNT[*i*] ]++] = *i*/*M*; |

The step 1 can run very fast using **cudaMemset** function on GPU. The step 2 and 4 apply the same parallelization technique, atomic operations, as shown in Algorithm 7 and 9, respectively. The read-modify-write atomic operations in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete [4]. The atomic operation, which is supported by CUDA as device runtime component [4], is necessary since lots of threads with the same cluster index increase the histogram array conflict with each other as shown in Figure.1. The step 3, the exclusive prefix sums operation, is commonly known as scan. The scan step is sequential, for which there are some effective parallel algorithm. However, we still use sequential code since the number of cluster is small enough to run it effectively on GPU as shown in Algorithm 8, this also avoids the slow memory copy operation from device to host and vice versa.



Figure 1. Histogram write conflicts

Algorithm 7: Calculating the histogram of *NNT* using atomicAdd operation based on GPU

//

//

//

//

//

Algorithm 8: Calculating the starting index for each cluster based on GPU

//

//

//

//



Algorithm 9: Counting sort based on GPU

//

//

//

//

//

//

After the NNT array is sorted, we use the parallel reduction algorithm, which is developed by Mark Harris through seven different versions [3]. In this paper, we use the seventh version, which is the final optimized kernel, and customize for the reducing ****for the *j*th cluster as well ****for the *j*th cluster and the *l*th dimension, as shown in Algorithm 10, 11, respectively.

Algorithm 10: Reducing ****for the *j*th clusterusing the parallel reduction algorithm based on GPU

//

//

//

//

//

//

//

//

//

1. ;

4. s
6. s
8. s
9. s
10. s
11. s
12. s

Algorithm 11: Reducing ****for the *j*th cluster and the *l*th dimensionusing the parallel reduction algorithm based on GPU

//

//

//

//

//

//

//

//

//

//

//

1. ;


5. s
7. s
9. s
10. s
11. s
12. s
13. s

The above Algorithm 10 and 11 may be executed concurrently or interleaved in different streams as shown in Algorithm 12. The kernels of Algorithm 10 are executed by stream #1, and the kernels of Algorithm 11 are executed by other streams, which of indices are determined by the formula: *x* mod (*Nstream* - 1*)* + 1, where *NStream* is the number of streaming multiprocessors (SMs) supported on GPU. After all streams run on GPU completed, we will have the output block sums of **** as well ****, then the calculating new centroids can run very fast on CPUs as shown in Algorithm 13.

Algorithm 12: Running Algorithm 10 and 11 concurrently in different streams on GPU

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//



Algorithm 13: Calculating new centroids on CPU

//

//

//

//

//

//

//

//

//



D. Calculating, updating *NNTi* and *DNNTi*

Algorithm 14: Calculating, updating *NNTi* and *DNNTi*, and also calculating distortion value *J* based on CPU

//

//

//

//

//



Algorithm 15: Calculating and, updating *NNTi* and *DNNTi*, based on GPU

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//



E. Calculating distortion value *J* using the parallel reduction algorithm based on GPU

Algorithm 16: Calculating distortion value *J* using the parallel reduction algorithm based on GPU

//

//

//

//

//

//

//

//

1. ;

4. s
6. s
8. s
9. s
10. s
11. s
12. s

IV. EXPERIMENTAL RESULTS

The 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 1.5 GHz. The memory of the GPU is 2GB with the peak bandwidth of 192.2 GB/s. 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. In Test 1, Test 2, and Test 3, speedups are compared to un-optimized CPU code. In other tests, speedups are compared to optimized CPU code.

Test 1: The data set generated from three real images: “Lena,” “Baboon,” and “Peppers,” abbreviated as LBP data set.

In this test, the data set consists of *N =* 49,152 data points with *D* = 16. The values *M* = 2, *K =* 8, ε = 1e-8, and ­*max\_iter* = 300,are used for the test. The average computing time each step of GFKM algorithm and total running time after 208 iterations on CPU and GPU is shown in Table 2. The updating memberships step on GPU is about eight times faster than on CPU. The updating centroids step on GPU is about two times faster than on CPU. The updating *NNT* and *J* step on GPU is about fifteen times faster than on CPU. In this test, the running on GPU is about seven times faster than on CPU.

Table 2: The average computing time each step and total running time of the GFKM algorithm based on CPU and GPU using the data set generated from three real images: “Lena,” “Baboon,” and “Peppers”

|  |  |  |  |
| --- | --- | --- | --- |
| GFKM method | CPU  (4 cores, un-optimized code) | GPU  (1152 cores) | Speedup |
| Initializing NNT | 20.850 | 1.089 | 19.1 |
| Updating memberships | 10.530 | 1.360 | 7.7 |
| Updating centroids | 4.295 | 2.064 | 2.1 |
| Updating NNT and J | 20.703 | 1.421 | 14.6 |
| Total time after 208 iterations | 7410.650 | 1008.997 | 7.3 |

As mentioned above, the number of data points *N* is an important factor to be considered. Thus, we perform testing our method with different values of *N* on the “LBP” data set as shown in Table 3.

Table 3: The changing speedup with different values of *N* on “LBP” data set

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *N* | 6144 | 12288 | 24576 | 49152 |
| Speedup | 4.4 | 6.3 | 6.8 | 7.2 |

From Table 3, we can see that the speedup of our method increases with the increase of number of data points. On the “LBP” data set also, we perform testing our method with different values of factors *K* and *M* to evaluate the influence of these factors to the speedup of method as shown in Table 4. The values ε = 1e-8, and ­*max\_iter* = 100,are used for this test.

Table 4: The changing speedup with different values of *K* and *M* on the “LBP” data set.

|  |  |  |  |
| --- | --- | --- | --- |
| GFKM method | Speedup | | |
| *K* = 4 | *K* = 8 | *K* = 16 |
| *M* = 2 | 5.3 | 7.2 | 7.8 |
| *M* = 3 | 4.9 | 6.1 | 6.7 |
| *M* = 4 | 4.7 | 5.6 | 6.0 |

From Table 4, we can see that the speedup of our method increases with the increase of value *K* and decreases since value *M* increase.

Test 2: We perform evaluating the influence of dimension of data to the speedup of our method as shown in Table 5. We use the “LBP” data set and the “Landsat Satellite” data set to compare. The values *N* = 6435, *M* = 2, *K =* 6, ε = 1e-8, and ­*max\_iter* = 100,are used for this test.

Table 5: The influence of dimension of data to the speedup

|  |  |
| --- | --- |
| Data set | Speedup |
| LBP: *D* = 16 | 4.4 |
| Landsat Satellite: *D* = 36 | 3.6 |

From Table 5, we can see that the speedup of our method decreases since dimension of data increase.

Test 3: In this test, we evaluate the performance of our proposed method on the “Image Segmentation” data set, a small data set, as shown in Table 6. The values *M* = 2, *K =* 7, ε = 1e-8, and ­*max\_iter* = 50,are used for this test.

Table 6: The performance of our proposed method on a small data set “Image Segmentation”

|  |  |  |  |
| --- | --- | --- | --- |
| GFKM method | CPU  (4 cores, un-optimized code) | GPU  (1152 cores) | Speedup |
| Initializing NNT | 1.028 | 0.139 | 7.4 |
| Updating memberships | 0.479 | 0.173 | 2.8 |
| Updating centroids | 0.235 | 0.641 | 0.4 |
| Updating NNT and J | 0.968 | 0.161 | 6.0 |
| Total time after 50 iterations | 85.133 | 48.940 | 1.7 |

From Table 6, we can see that the proposed method is ineffective in the updating centroids step on a small data set. Thus, to achieve a better overall performance on the small data sets, we use the CPU-based method in the updating centroids step. In this test, after applying that change, the speedup is significantly improved to 3.0 times.

Test 4: In this test, we use LBP data set with the same paramenters in Test 1. However, the speedup is compared to an optimized CPU code-base version or release version, as shown in Table 7.

Table 7: The speedup of the GPU-based GFKM algorithm on the “LBP” data set, compared to optimized CPU code-based version

|  |  |  |  |
| --- | --- | --- | --- |
| GFKM method | CPU  (4 cores, optimized code) | GPU  (1152 cores) | Speedup |
| Initializing NNT | 5.672 | 0.984 | 5.8 |
| Updating memberships | 9.425 | 0.449 | 21.0 |
| Updating centroids | 1.393 | 2.243 | 0.6 |
| Updating NNT and J | 4.986 | 1.332 | 3.7 |
| Total time after 200 iterations | 3116.253 | 805.974 | 3.9 |

From Table 7, in the updating centroids step, we can see that the proposed method is ineffective when compared to optimized CPU code-based version. Thus, to achieve a better overall performance, we use the CPU-based method in the updating centroids step. In this test, after applying that change, the speedup is significantly improved to 4.9 times.

V. CONCLUSIONS

REFERENCES

[1] Franklin J. C. Lai, Eric Y. T. Juan, and Jim Z. C. Lai, Generalized Fuzzy k-Means Clustering Using m nearest Cluster Centers, 2013.

[2] You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu, Speeding up K-Means Algorithm by GPUs, 2010.

[3] Mark Harris, Optimizing Parallel Reduction in CUDA, NVIDIA Developer Technology, 2007.

[4] CUDA Programming Guide Version 6.5, Technical report, NVIDIA Corporation, 2014.