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 will also make comparisons with CPU based implementations and analysis the pros and cons of using GPU’s in image analysis.

I. INTRODUCTION

II. GFKM Clustering 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 a GPU-based parallel GFKM algorithm

The steps of the GFKM algorithm are: (1) calculate  andinitializing *NNTi* and *DNNTi*, (2) updating memberships **** and, (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 are 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): It is difficult to be fully parallelized due to write conflict, so this task is executed on CPU, this reduces the overall performance.
* Step (5): We use the 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 **** and 

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

Algorithm 4: Updating memberships **** and  based on CPU

//

//

//

//

//

//



Algorithm 5: Updating memberships **** and  based on GPU

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//



C. Computing the new center for each clusters

Algorithm 6: Computing the new center for each clusters based on CPU

//

//

//

//

//

//



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

Algorithm 7: Calculating, updating *NNTi* and *DNNTi*, and calculating distortion value *J* based on CPU

//

//

//

//

//



Algorithm 8: Calculating, updating *NNTi* and *DNNTi*, and precomputing reduction distortion value *J* based on GPU

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//

//



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

//

//

//

//

//

//

//

//

1. ;
2. \*M




8. s
10. s
12. s
13. s
14. s
15. s
16. s
17. s

IV. EXPERIMENTAL RESULTS

The GFKM algorithm is implemented using CUDA version 6.5. 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.

Example 1: The data set generated from three real images: “Lena,” “Baboon,” and “Peppers.”

In this example, the data set consists of 49,152 data points with *D* = 16. The values *M* = 2, *K =* 8, and ε = 1e-8,is used for the test. The running time each step of GFKM algorithm at iteration #1 and total running time after 268 iterations on CPU and GPU are shown in Table 1. The updating membership step on GPU is seven times faster than on CPU. The updating *NNT* and *J* on GPU is fifteen times faster than on CPU. The updating centroids step is only executed on CPU because of that it is difficult to be fully parallelized as shown before, this reduces the overall performance. In this example, the running on GPU is five times faster than on CPU.

Table 1: Run application using the data set generated from three real images: “Lena,” “Baboon,” and “Peppers”.

|  |  |  |  |
| --- | --- | --- | --- |
| GFKM | CPU  (4 processors) | GPU  (1152 processors) | Speedup |
| Initializing NNT | 20.177 | 1.130 | 17.9 |
| Update membership at iteration #1 | 10.556 | 1.468 | 7.2 |
| Update centroids at iteration #1 | 4.288 | executed on CPU | 1.0 |
| Update NNT and J at iteration #1 | 21.108 | 1.398 | 15.1 |
| Total time after 268 iterations | 9428.034 | 1916.129 | 4.9 |

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.