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.
2. Given the set of cluster centers *SCp*, compute  for *i* = 1 to *N* and *j* = 1 to *M*. Update membership **** using equation (11). If **C***j*∈*NNTi* is the *l*th nearest neighbor of **X***i*, set  = ****; otherwise let  = 0.
3. Compute the center for each cluster using equation (16) to obtain a new set of cluster representatives *SCp+*1 = {**C***j*(*p*+1)}.
4. Update *NNTi* for *i* = 1 to *N*.
5. If < ε for *j* = 1 to *k*, 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.

**Algorithm 1:** CPU-based GFKM

//

//

//



III. Design a GPU-based parallel GFKM algorithm

Analysis steps of the algorithm can be parallel implemented on GPUs: (1) Initializing, (2) Computing distance and updating memberships and, (3) Computing the new center for each clusters.

Design GPU-based parallel algorithm for each section:

* Step (1) and (2): For low-dimensional data sets, we utilize the GPU on-chip registers to minimize the latency of data access; for high-dimensional data sets, we use both registers and shared memory and apply a very efficient reduction algorithm that treats the most time-consuming part of k-Mean as matrix multiplication [2].
* Step (3): It is difficult to be fully parallelized due to write conflict, so we use GPU to speed up part of the task that is appropriate for parallelization, and leave the remaining part for CPU execute. We will design an algorithm which adopts “divide and conquer” strategy [2].

A. Initializing

The CPU-based algorithm of initializing is shown in Algorithm 2. My 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, as shown in Algorithm 3. Line 1 and 2 show how the algorithm designs the thread block and gird. Line 3 and 4 calculate the position of the corresponding data point for each thread in global memory. Line 5 loads the data point into the register. Lines 6-14 compute the distance, accumulates the total inverse distance, updates the memberships, and initializes.

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. However, the problem of Algorithm 3 is the limited size of the registers. In fact, users are not able to fully control the registers, and could only utilize registers when the data size is small enough. When the data points cannot be loaded into the registers as the data dimension grows, they will be stored in local memory, which will increase the reading latency and decrease the performance significantly.

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 Result[n][k]; and the distance computing process shares the same flow as matrix multiplication. Based on this observation, we design Algorithm 4 for high-dimensional data sets, by adopting the idea of matrix multiplication and utilizing registers and the shared memory together. The main idea of Algorithm 4 is decreasing the global memory access time and latency by loading the data into the shared memory tile by tile. Thus, Algorithm 4 reads each data point from global memory only once, the same as Algorithm 3. The key point of Algorithm 4 is how to access the global memory and shared memory efficiently, which is achieved by adopting 8 coalescing reading which accesses sixteen continuous address for the threads in a half warp to avoid the bank conflict. The details are described as follows.

The three matrixes data[n][d], centroid[d][k] and Result[n][k] are partitioned into TH×TW, TW×TH, and TH×TW tiles respectively. The resource of the GPU is partitioned as follows: the grid has (k/TW)×(n/TH) blocks, the ID of which is noted by blockIdx.y (by in Fig.2) and blockIdx.x (bx in Fig.2); each block has TH×TDimY threads, the ID of which is noted by threadIdx.y (ty in Fig.2) and threadIdx.x (tx in Fig.2). The computing task is dispatching as follows: each block calculates TDimY tiles in the Result, which is noted as SR[TH][TW×TDimY]; each thread computes a column of SR. For each thread, indexD points to the right position of the data, which contains the following three parts as shown in line 4: data is the beginning address of the data set; since the height of the data is divided by TH, blockIdx.y×TH×d is the address of the corresponding block; threadIdx.y×d adding threadIdx.x is the offset address inside the block.

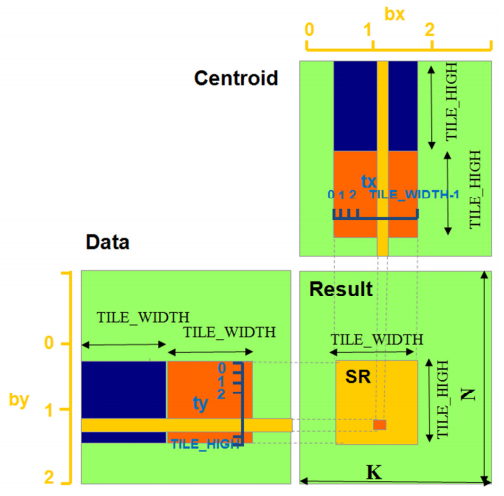


Figure 2. Tile-based distance computing process

In line 5, indexC points to the right position in centroid, which also has three parts: centroid is the beginning address of the current centroid; blockIdx.x×TW points to the corresponding block address, since the width of the centroid is divided by TW; threadIdx.y×blockDim.x adding threadIdx.x points to the address of the current thread inside the block. Obviously, the threads in one block would access centroid in continuous addresses, which is also called coalesced accessing. indexR is calculated in the same way as in line 6: the beginning address of the result, the row address, and the offset address inside the block for the current thread.

In the loop from line 11 to 16, the algorithm loads a tile of data from global memory to the shared memory, and computes the temporary distance saved in TResult which are stored in on-chip registers; the loop ends when the whole row has been calculated. Line 17-18 calculate the distance based on the results of muliplication. Line 19 waits for all the threads to finish their work. Line 20 writes the distance back from TResult to SR. The details are shown in Fig.2, and take the process of calculating a SR[TH][TW×TDimY] as an example, which is equal to data[TH][d]×centroid[d][TW×TDimY]: load the first tile (in blue color) from the data into the shared memory; multiply the blue tile in the data with the blue tile in the Centroid, which is stored in the constant memory; accumulate the temporary results into TResult, whose initial value is all zero; repeat loading the next tile (in orange color), multiplying and accumulating, until data[TH][d] and centroid[d][TW×TDimY] have been all accessed.

After calculating the distances matrix Result[n][k] between the data and the centroid, the next step is accumulating the total inverse distance, updating the memberships, and initializing. Obviously, based on the CPU, the computational complexity is O(nk). On the GPUs, each thread calculates for one data point from one row of Result, whose computational complexity is O(k).

Algorithm 2: Computing initial and initializing based on CPU

//

//

//

//



Algorithm 3: Computing initial and initializing based on GPU

//

//

//

//

//

//

//

//

//

//

//



Algorithm 4: Computing distance based on shared memory of the GPU

//

//

//

//

//

//

//

//

//

//

//

//

//

//



B. Computing distance and updating membership and

Apply the design as described in the section A.

Algorithm 5: Computing distance and updating membership and based on CPU

//

//

//

//



Algorithm 6: Computing distance and updating membership and based on GPU

//

//

//

//

//

//

//

//

//

//

//



Algorithm 7: Computing distance based on shared memory of the GPU

//

//

//

//

//

//

//

//

//

//

//

//

//

//



C. Computing the new center for each clusters

Algorithm 8: Computing the new center for each clusters based on CPU

//

//

//

//

//



Algorithm 9: Computing the new center for each clusters based on GPU

IV. EXPERIMENTAL RESULTS

Implement GPU-based parallel GFKM algorithm and make comparisons with CPU based implementations.

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.