K-Means on commodity GPUs with CUDA

BAI Hong-tao[a,b], HE Li-li[a,b], OUYANG Dan-tong [a,b1], LI Zhan-shan [a,b] ,LI He [a,b]

(a. College of Computer Science and Technology, Jilin University, 130012, China;
b. Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Jilin University, 130012, China)

Abstract

K-means algorithm is one of the most famous unsupervised clustering algorithms. Many theoretical improvements for the performance of original algorithms have been put forward, while almost all of them are based on Single Instruction Single Data (SISD) architecture processors (CPUs), which partly ignored the inherent paralleled characteristic of the algorithms. In this paper, a novel Single Instruction Multiple Data (SIMD) architecture processors (GPUs) based k-means algorithm is proposed. In this algorithm, in order to accelerate compute-intensive portions of traditional k-means, both data objects assignment and k centroids recalculation are off-loaded to the GPU in parallel. We have implemented this GPU-based k-means on the newest generation GPU with Compute Unified Device Architecture (CUDA). The numerical experiments demonstrated that the speed of GPU-based k-means could reach as high as 40 times of the CPU-based k-means.

1. Introduction

Clustering is a search method for hidden patterns that may exist in datasets. It is a process of grouping data objects into disjointed clusters so that the data in each cluster are similar, yet different to the other clusters. K-means is one of the most famous and typical clustering algorithms and applied in many application areas such as data analyses, pattern recognition, image processing, and information retrieval [1,2]. In k-means, a data point is comprised of several values, called features. By dividing a cluster of data objects into k sub-clusters, k-means represents all the data objects by the mean values or centroids of their respective sub-clusters.

K-means has the advantages of fast convergence and ease of implementation, but it has poor performance in some applications with large dataset such as physics simulation. Some implementations use k-d trees [3] to accelerate the execution time. Other improvements leverage the high degree of task parallelism and data parallelism of k-means. For example, researchers at Northwestern University developed the Minebench using OpenMP [4]. But this approach inevitably produces too much message communication overhead.

Nowadays, most desktop computers are equipped with programmable graphics processing units (GPUs) with plenty powerful Single Instruction Multiple Data (SIMD) processors that can support parallel data processing and high-precision computation. The rapid advance in GPUs performance, coupled with recent improvements in its programmability, made it possible to parallelize k-means on personal computers.

In this paper, a novel Single Instruction Multiple Data (SIMD) architecture processors (GPUs) based k-means algorithm is proposed. In this algorithm, both data objects assignment and k centroids recalculation of traditional k-means are parallel performed on the GPU.

Recently, we noticed that Shuai Che et al. [5] published a similar work nearly simultaneously, where they used CUDA to put partial steps of k-means onto the GPU. The difference in implementation is that our approach puts new centroids recalculation step also onto GPU and algorithm performance thus becomes better.

The paper is organized as follows. Section 2 presents the concept and related works of GPGPU. Section 3 describes two key loads on GPU and the whole GPU-based k-means. The performance analysis of our approach is reported in Section 4. Finally, conclusions are drawn in Section 5.

2. General-purpose GPU

GPUs are probably today's most powerful computational hardware for the dollar. The rapid increase in the performance of graphics hardware,

[1] **Corresponding Author**

978-0-7695-3507-4/08 $25.00 © 2008 IEEE
DOI 10.1109/CSIE.2009.491

coupled with recent improvements in its programmability, have made graphics hardware a compelling platform for computationally demanding tasks in a wide variety of application domains. A lot of researches have been presented in recent years for general-purpose computing, an effort known collectively as GPGPU (for "General-Purpose computing on the GPU").

CUDA is a new hardware and software architecture for issuing and managing computations on the GPU as a data-parallel computing device without the need of mapping them to a graphics API [6]. Compared with previous programming interfaces such as Cg, CUDA provides more flexibility to efficiently map a computing problem onto the hardware architecture. CUDA applications consist of two parts. The first executes on the GPU and is called a 'kernel'. Kernels are implemented in the CUDA programming language, which is basically the 'C' programming language extended with a number of keywords. The other part executes on the host CPU and provides control over data transfers between CPU and GPU and the execution of kernels.

A kernel program is run by multiple threads that run on the GPU. We call a group of threads a block. Threads contained in the same block communicate with each other using shared memory and cannot communicate with threads in another block. Calculations on the GPU are started by specifying the number of blocks to execute and the number of threads that each block contains. The total number of threads is the product of the two.

For now, CUDA is available for the NVIDIA G80 series, the Tesla solutions, and some Quadro solutions. The NVIDIA GeForce 8800GTX hardware architecture defines a hierarchical memory structure where each level has a different size, access restrictions and access speed as illustrated by Fig.1. In general, accessing the largest type of memory is flexible but slow, while accessing the smallest type of memory is restrictive but fast. This memory structure is directly exposed by the CUDA programming framework. The challenge in mapping a computing problem efficiently on a GPU through CUDA is to store frequently used data items in the fastest memory, while keeping as much of the data on the device as possible.

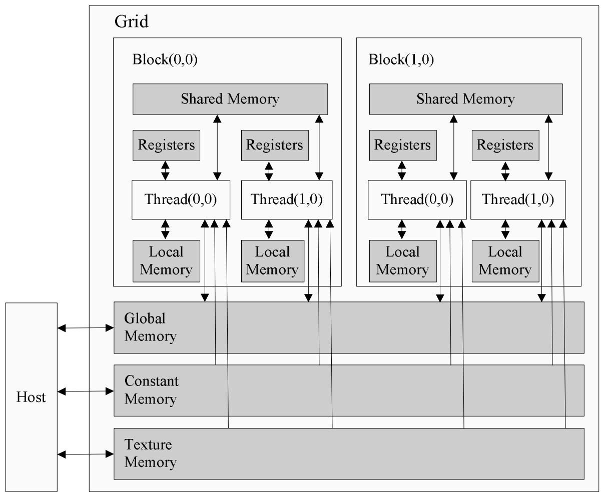

Fig.1 Hardware Architecture of G80

In fact, during a short period of one year CUDA appears, many algorithms outside the field of image rendering and processing are accelerated by CUDA, from digital investigation [7] or physics simulation [8] to molecular dynamics [9,10].

3. K-Means algorithm on the GPU

3.1. Data objects assignment

Data objects assignment and k centroids recalculation are the most intensive arithmetic task load of k-means. There are two strategies in data objects assignment process suited to GPU-based k-means. The first is the centroids-oriented, in which distance from each centroid to all data objects are calculated and then, each data point will merge itself into the cluster represented by nearest centroid. This method has advantages when the number of processors of GPU is relatively small so that every processor can deal with data objects in series. Another is the data objects-oriented, namely, each data point calculates the distance from all centroids, then data object will be assigned to the cluster represented by the centroid with the shortest distance from it The latter strategy is adapted in this research because our GPU has more than one hundred processors [6] as illustrated by Fig.2.

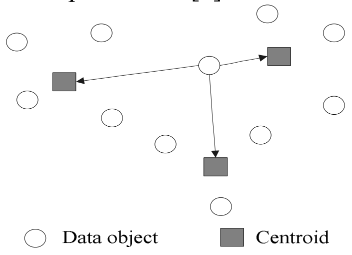

Fig.2 Oriented-data Objects Distances Computing

In k-means algorithm, every data point must choose the nearest centroid after calculating all the distances, this selecting process consists a series of comparison which could be carried out through Deep Buffer in early GPUs. [Because CUDA doesn't support Deep Buffer, instead we choose to use multiple threads to optimize the calculation] In this way, the latency of memory access could be avoided while one thread is waiting for memory access, and other threads will be optimized to use the arithmetic resources.

3.2. *K* centroids recalculation

The new centroid is the arithmetic means of all data objects. The positions of the *k* centroids are also parallel recalculated by GPU and every thread is responsible for a new centroid as Fig.3 illustrated.

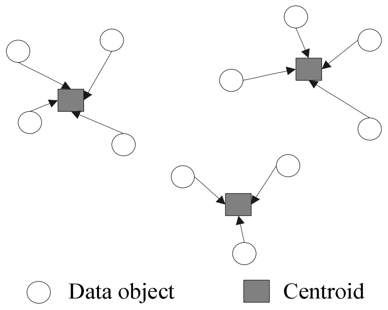

Fig.3 *K* centroids recalculation

After data objects assignment, we get the cluster label of every data point. A straightforward idea for recalculating the position of one centroid is to read all data objects and determine whether the data point belongs to this centroid or not. Unfortunately, massive condition statements are not suitable to the stream processor model of GPUs. We add another procedure that the cluster labels are downloaded from the device (GPU) to the host (CPU) and the host rearranges all data objects and counts the number of data objects contained by each cluster. And then, both structures are uploaded to the global memory of the device. In this way, every thread of CUDA kernel can complete its task by reading its own data objects continuously. Performance test in section 4.2 proved this strategy has high efficiency.

3.3. GPU based *K* Means

The main idea of GPU-based *k*-means is that data-parallel, compute-intensive portions of traditional *k*-means can be off-loaded from the host to the device to improve performance. More precisely, data objects assignment and *k* centroids recalculation executed many times, but independently on different data, can be isolated into two functions consisted of massive threads, parallel executing on the device. Actually, each function is compiled to the instruction set of the device and the target program, called a kernel, is downloaded to the device.

GPU-based *k*-means has three fundamental issues to be addressed, though the SIMD processors are accomplished in parallel computing. First, flow control and data caching of the device are weak for its more transistors are devoted to compute unit. Second, compared with the data transfer rate between the CPU and CPU's cache, the data transfer rate between GPU and GPU's memory (global memory) is much slower, then only appropriate size of block and grid is capable of winning device's power. In the end, the transfer time between the CPU's memory and GPU's memory is extra cost relative to traditional *k*-means on CPU. Performance enhancement can be obtained, as long as duty assignment for the host and the device, data storage, and parallel computing mode are reasonably designed and implemented.

To summarize, we give the whole GPU-based *k*-means in Fig.4.

In task assignment, the host is responsible for placing *k* objects into the space represented by the objects that are being clustered and rearranging all data objects and controlling iteration process, while the device for data-parallel intensive computing. In data storage, all data objects and centriods are stored as dynamic arrays on the device. The device has three kind memories shared by all threads of a kernel: *constant memory*, *texture memory* and *global memory*. We put all parameters in *global memory* as both other *constant memory* and *texture memory* are read-only and respective 64KB [6], which are insufficient to data. Another remarkable point is that the bandwidth between the device and the device memory is much higher than the bandwidth between the device memory and the host memory. In our approach, the cluster labels transfer between the host and the device is very small. In parallel computing mode, kernel is assigned enough computing routine and massive threads may reduce the global memory latency. This frame of GPU-based *k*-means is designed by the architectures of CPUs and GPUs, which is adapted to not only CUDA, but also other mainstream GPGPU environments, such as DPVM [11].

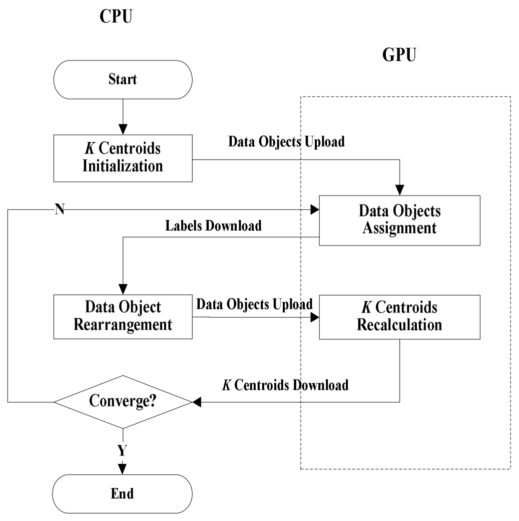

Fig.4 The frame of GPU-based *k*-means

4. Performance analysis

In this section, we compared the performance of GPU-based *k*-means with CPU-based *k*-means. All experiments were performed on a PC with an Intel Pentium D 965 CPU 3.7 GHz, 1GB main memory and Geforce 8800GTX graphic card, 1.35 GHz engine clock speed, 768 MB of device RAM, and 128 stream processors, organized into 16 multiprocessors.

The speedup measurement includes both clustering execution time and total execution time. Clustering execution time for one iteration of CPU-based *k*-means is denoted by *CKcc* and GPU-based *k*-means by *GKcc*. Total execution time of CPU-based *k*-means is denoted by *CKtc*=I/O+ α *CKcc and* GPU-based *k*-means by *GKtc*=I/O+ α *GKcc*. (α is a constant and used for controlling the influence of data size and convergence speed)

Up to now, all GPUs only stand for single-precision floating-point arithmetic. In order to validate the correctness of GPU-base *k*-means, we produce samples, consisted of random 32-bit floating point numbers from 100K to 1M between 0 and 1. And in the iteration, we simulate 64-bit floating point manipulations using Kahan's Summation Formula to achieve double-type precision, as CUDA follows IEEE-754 standard.

4.1. Clustering Cost and Total Cost Comparison

As illustrated by Fig.5, we compared *GKcc* with *CKcc*. It shows that the speed of GPU-based *k*-means could reach from 27 to 56 times of the CPU–based *k*-means. This performance improvement benefits from the high parallel computing ability of CUDA and the data objects rearrangement strategy. In CUDA and G80, 128 processors are all indistinctive, and not distinguished by pixel and vertex, so that they can run at same time without idle situation. In data objects rearrangement, invalid texture fetching and branch prediction which could cause low efficiency to GPUs are avoided.

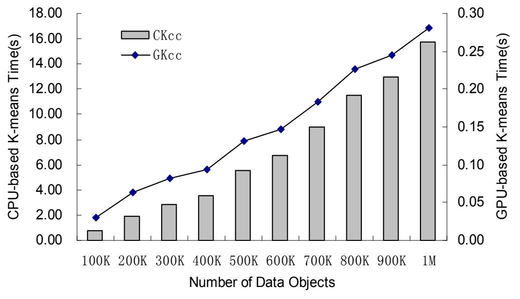

Fig.5 Clustering Execution Time Comparison

On the other hand, we compared *GKtc* with *CKtc* in Fig.6. The speed of GPU-based *k*-means could reach from 8 to 14 times of the CPU–based *k*-means. This is because the I/O cost is higher when α is set to 20 in this test.

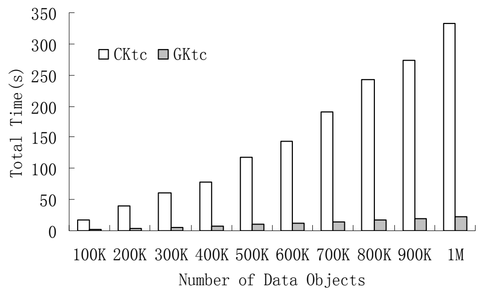

Fig.6 Total Execution Time Comparison

4.2. Data transfer cost

The data transfer is extra cost of GPU-based *k*-means. We also compared the data transfer cost with the iteration cost as illustrated by Fig.7. It shows that data transfer cost may nearly be ignored compared to the iteration cost. The data transfer rate between the host memory and the device memory, under PCI-E16x, is about 4 GB/s so that it hardly brings influence on the whole runtime of GPU-based *k*-means.

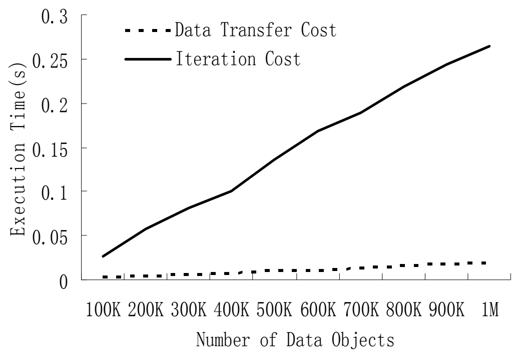

Fig. 7 Data Transfer Cost

4.1. *K* value analysis

To study the performance enhancement of GPU-based *k*-means under different values of *k*, the clustering speedup of *k*-means is described by Fig.8. We found that this ratio became larger as *k* increases. In data objects assignment, every thread continuously read *k* centroids from global memory and the larger *k* is, the higher memory utilization is. In data objects recalculation, *k* is the number of threads and all processors of G80 can work when k is none but more than 128.

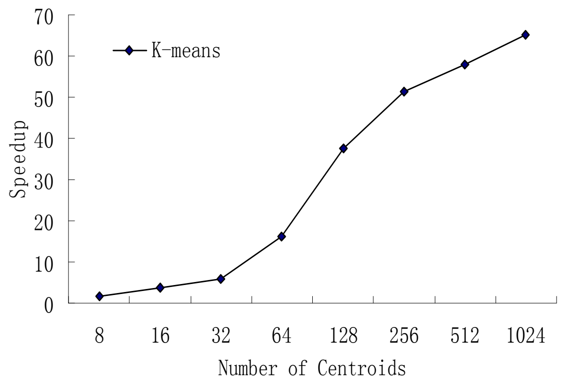

Fig. 8 Speedup under *k*

5. Conclusion

In this research, we illustrated that nearly all complex and time-cost computation of *k*-means can be sped up substantially by offloading work to GPU. The CUDA technology used in our experiments is modern GPGPU architecture, which is adopted by many NVIDIA GPUs. As current trends indicated, future GPU designs, also based on general purpose multiprocessors, will offer even more computational power. Our primary purpose in this research is to prove that developing GPU-aware data mining software is possible and useful. More GPU-based approach for clustering should be further explored. We plan to extend our existing implementation to improve the clustering accuracy. Another research direction is to employ GPU to other data mining algorithms.

Acknowledgements: This work is supported by the National Natural Science Foundation of China under grant No.60873148 and No.60773097.

6. References

[1] X.J. Wang, “K-means clustering for multispectral images using floating-point divide”. *Proceedings 2007 IEEE Symposium on Field-Program Custom Computing Machines (FCCM 2007)*, pp.151-59.
[2] H. Zhou, Y.H. Liu, “Accurate integration of multi-view range images using k-means clustering”. *Pattern Recognition* 2008, 41(1), pp.152-75.
[3] T. Kanungo, D. Mount, N. Netanyahu, et al. “An efficient k-means clustering algorithm: analysis and implementation”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2002, 24(7), pp. 881-892.
[4] J. Pisharath, Y. Liu, W. Liao, et al. “Nu-minebench 2.0. Technical report”, Northwestern University Department of Electrical and Computer Engineering, 2005. *http://cucis.ece.northwestern.edu/techreports/pdf/CUCIS-2004-08-001.pdf.*
[5] S. Che, M. Boyer, J. Meng, D. Tarjan, “A performance study of general-purpose applications on graphics processors using CUDA”, *Journal of Parallel and Distributed Computing*, 2008, 68(10), pp. 1370-1380
[6] Nvidia. “NVIDIA CUDA Compute Unified Device Architecture Programming Guide”, 2008, http://developer.download.nvidia.com/compute/cuda.
[7] L. Marziale, G.G. Richard, V. Roussev, “Massive threading: Using GPUs to increase the performance of digital forensics tools”, *Digital Investigation*, 2007(4), pp. 73-81.
[8] R.G. Belleman, J. Bédorf, S.F. Portegies Zwart, “High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA”, *New Astronomy*, 2008(13), pp.103-112.
[9] W.G. Liu, B. Schmidt, G. Voss, “Molecular Dynamics Simulations on Commodity GPUs with CUDA”. *Lecture Notes in Computer Science, High Performance Computing – HiPC 2007,* 2007, pp.185-196.
[10] J.A. Anderson, C.D. Lorenz, A. Travesset, “General purpose molecular dynamics simulations fully implemented on graphics processing units”, *Journal of Computational Physics*, 2008, pp. 5342-5359.
[11] M. Peercy, M. Segal, D.Gerstmann, “A Performance-Oriented Data Parallel Virtual Machine for GPUs”. *Proceedings of SIGGRAPH 2006*, 2006, pp.184-es.