ISBN 978-952-5726-07-7 (Print), 978-952-5726-08-4 (CD-ROM)
Proceedings of the Second Symposium International Computer Science and Computational Technology(ISCSCT '09)
Huangshan, P. R. China, 26-28,Dec. 2009, pp. 151-155

A Practical GPU Based KNN Algorithm

Quansheng Kuang, and Lei Zhao*
School of Computer Science and Technology, Soochow University, Suzhou 215006, China
Email: kqs.net@163.com, zhaol@suda.edu.cn

Abstract—**The KNN algorithm is a widely applied method for classification in machine learning and pattern recognition. However, we can't be able to get a satisfactory performance in many applications, as the KNN algorithm has a high computational complexity. Recent developments in programmable, highly paralleled Graphics Processing Units (GPU) have opened a new era of parallel computing which deliver tremendous computational horsepower in a single chip. In this paper, we describe a practical GPU based K Nearest Neighbor (KNN) algorithm implemented by CUDA. In our algorithm, a data segmentation method has introduced in the distances computation step to adapt to the CUDA thread model and memory hierarchy. We obtain highly increase in performance compared to ordinary CPU version.**

Index Terms—**K Nearest Neighbor, Data Segmentation, GPU, CUDA**

I. INTRODUCTION

For the past decade, the programmable Graphic Processing Units (GPU) has evolved into a kind of many-core processor with highly paralleled and multithreaded features. Compared with generic x86 based CPU, the current GPU provide tremendous computational horsepower and higher memory bandwidth. Nowadays, the GPU has been at the leading edge of chip-level parallelism and expanded the scope of application from 3D rendering to general purpose computing.

The KNN algorithm is a widely applied method for classification or regression in pattern recognition and machine learning. As a lazy learning, KNN algorithm is instance-based and used in many applications in the field of statistical pattern recognition, data mining, image processing and many others. The KNN algorithm is simple but computationally intensive. When the size of train data set and test data set are both very large, the execution time may be the bottleneck of the application.

In this paper, we propose a novel parallel KNN algorithm based on GPU. Our algorithm is specially designed for NVIDIA Compute Unified Device Architecture (CUDA), adopting the thread model and memory hierarchy of NVIDIA's GPU. A data segmentation method and a parallel Radix Sort are proposed to make full use of the computational horsepower of the GPU. As the results, on an inexpensive graphics card we can archive over 30X speedup than an ordinary CPU version. Therefore, KNN algorithm under huge number dataset and high dimension dataset are now practical and feasible.

The organization of the paper is as follows. Section 2 describes related work, including KNN algorithm and the programming architecture of the GPU. Section 3 presents the details of implementation of KNN algorithm based on GPU. In Section 4, experimental data are given and we conduct the analysis of the results. Finally, we conclude the paper in Section 5.

II. RELETED WORK

A. Principle of KNN algorithm

KNN algorithm is widely applied in pattern recognition and data mining for classification, which is famous for its simplicity and low error rate.

The principle of the algorithm is that, if majority of the k most similar samples to a query point q_i in the feature space belong to a certain category, then a verdict can be made that the query point q_i fall in this category. Similarity can be measured by the distance in the feature space, so this algorithm is called K Nearest Neighbor algorithm. A train data set with accurate classification labels should be known at the beginning of the algorithm. Then for a query data q_i, whose label is not known and which is presented by a vector in the feature space, calculate the distances between it and every point in the train data set. After sorting the results of distances calculation, decision of the class label of the test point q_i can be made according to the label of the k nearest points in the train data set.

Each point in d-dimensional space can be expressed as a d-vector of coordinates, such as:

$$p = (p_1, p_2, ..., p_n) . \tag{1}$$

The distance between two points in the multi-dimensional feature space can be defined in many ways. Using Euclidean distance is usually to be the most ordinary method, that is:

$$dist(p,q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2} . \tag{2}$$

Alternatively, Manhattan distance can also be used as:

$$dist'(p,q) = \sum_{i=1}^{n} | p_i - q_i | . \tag{3}$$

The quality of the train data set directly affects the classification results. At the same time, the choice of parameter K is also very important, for different K could result in different classification labels.

* Corresponding Author: zhaol@suda.edu.cn.

© 2009 ACADEMY PUBLISHER
AP-PROC-CS-09CN005

The KNN algorithm is simple in calculation and can be applied to high-dimensional data sets. Nevertheless, when the test set, train set, and data dimension are larger than expected, the computational complexity will be huge and the operation time will be very long. When test set and train sets contain *m* and *n* vectors in *d*-dimensional feature space respectively, the time complexity of this algorithm is $O(m \bullet n \bullet d)$. At present, there are also some optimizations to improve the efficiency of algorithm, such as using KD-Tree to improve storage efficiency, or to lower precision for improve efficiency such as Approximate Nearest Neighbor Searching (ANN). There are also some papers present that some points in the train set take little or no effects to the final result, which could be cut to reduce the computational scale. In some cases, these methods can reduce the executing time by half.

B. GPGPU and CUDA

Nowadays, the theoretically performance of GPU is far more than that of GPU. The reason behind the discrepancy in floating-point computation capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly paralleled computation, which exactly what graphics rendering does. Therefore, the GPU is designed to be more transistors in it are devoted to data processing rather than data caching and flow control.

Considering the huge computational horsepower delivered by GPU, methods were taken to make GPU play an active role in non-graphics purpose, which called General Purpose GPU (GPGPU). Nevertheless, applications specially designed for graphics hardware abstraction using graphics languages is difficult before CUDA appears. CUDA (Compute Unified Device Architecture), parallel programming model is designed to overcome this challenge by providing standard programming languages such as C to the programmers instead of imposing them to map non-graphics application through the graphics application programming interfaces.

Figure 1 shows the threads abstraction of CUDA. The host means the CPU while the device refers to the GPU.

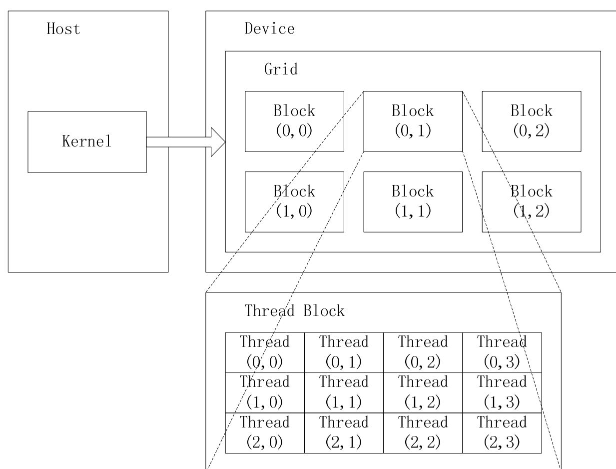

Figure 1. Threads Organization in CUDA

The beginning and the end of the application executed by the CPU must be serial code, in which one or more steps could be organized parallelism. The parallel code, which called "Kernel", is assigned to the device as a grid of Thread Blocks. The Thread Block containing hundreds of threads is dispensed to a Streaming Multi-processor (SM) for execution, which is composed by 8 Streaming Processors (SP). Each 32 threads in a Thread Block are organized into a Warp during executing. This is also referred as SIMT (Single Instruction, Multiple Threads) model.

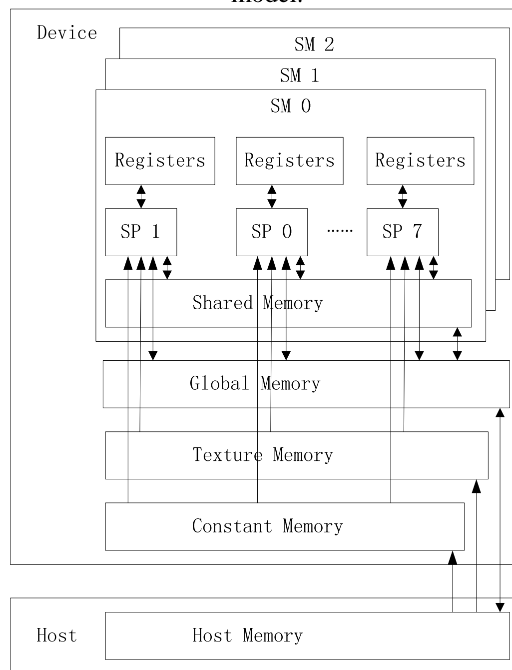

Figure 2. Memory Hierarchy in CUDA

Another important key point in CUDA architecture is the memory hierarchy. The register is the fastest but could only be accessible by a thread. Each SM contains 16KB of shared memory, which is shared by a Thread Block. The Global Memory is the video memory in the graphics card, which is usually have wide bandwidth and high frequency and much more faster than the host memory. The Texture Memory and the Constant Memory have the same speed as Global Memory but read-only and cached in the SM, as illustrated in Figure 2.

III. KNN ALGORITHM BASED ON CUDA

A. Overview

The basic process of KNN algorithm is as follows. First, the data pre-processing phase is to initialize the labeled *d*-dimensional train data set as well as the test data set to be classified. Second, select one test point in the test data set and calculate the distances between it and each point in the train data set. The next phase is to sort the results of distances computation, and find out *K* smallest results according to the parameter *K*. The fourth step is to determine the class label of the test point by the election result of *K* points. Finally, select another point in test data

set and go to step two repeatedly until the test data set is empty.

Euclidean distance is used in this paper. For the purposes of distances comparing, it is not necessary to compute the final square root in the Euclidean distance expression. So the squared distance will be used in phase two to reduce the computation.

According to the results of the analysis, the distances calculation phase can be highly paralleled and can reach a high speedup ratio in GPU implementation. The sorting step can also obtain benefits by using GPU acceleration. The remaining step, such as the determination of class labels are simple and consume little time that will be implemented on the CPU.

B. Segmentation method in distances computation

In the distances computation phase, distances between every point in the test set and each point in the train set should be calculated.

For the consideration to reduce program branching and to streamline operations, this paper adopts the way in matrices to restore multi-dimensional data set. The train data set A and the test data set B are both d-dimensional sets. That is, the number of features or columns to describe each vector in data set is d. The number of vectors or instances or samples in the train data set is n, while m for the test data set. Consequently, we restore the train data set as a $n \times d$ matrix in the memory, and a $m \times d$ matrix for the test data set. The result set C, which containing all the distances between each pair of points in A and B, is described in a $m \times n$ matrix. So the element in data set C which located in column x and row y, presents the distance between the vector in A whose row number is x, and the vector in B whose row number is y. As discussed earlier, the distances restored in C are squared Euclidean distances as the computation of square root does not affect the sorting results. The overall computational complexity of this phase is $O(m \bullet n \bullet d)$.

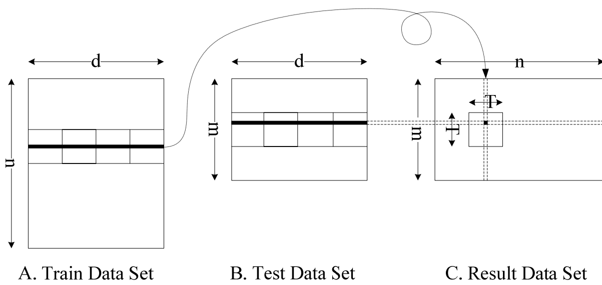

Figure 3. Data Segmentation Method in Distances Computation

The result data set C is divided into a large number of the tiles with the width of T. Each thread in the GPU takes charge of one element in C, i.e. computes one distance between a pair of vectors in A and B. Each Thread Block containing $T \times T$ threads that calculates one tile in C. Consequently, there are $(m/T) \times (n/T)$ Thread Blocks in all. In order to take full advantage of the high-speed Shared Memory in the GPU, we introduce a batch loading strategy when reading data from Global Memory. That is, each tile in A and the corresponding tile in B is loaded from Global Memory to Shared Memory for one step of calculation, when before each thread computes the square of difference between tow corresponding elements in A and B, and adds the result to the exact position in C. The batch loading strategy should be repeated d/T times to obtain a tile of squared Euclidean distances between T vectors in A and another T vectors in B. The pseudo code of the kernel function is shown as follows:

Algorithm 1: distances_computation(train, test, result)

```
{Each block is given the 2-Dimensional identifier bx, by, and tx,
ty for each thread. }
sub_result = 0;
temp = 0;
for each sub-tile in dimension/T do
    loads shared_train in train set to shared memory ;
    loads shared_test in test set to shared memory;
    syncthreads;
    for k=0 to T do
        temp = shared_test[ty][k] - shared_prob[tx][k];
        sub_result += temp * temp;
    end for
    syncthreads;
    add sub_result to the corresponding position in result set;
end for
```

Some limitations of CUDA specification are as follows: the maximum number of threads per block is 512; the maximum number of active threads per multiprocessor is 768; the maximum number of active blocks per multiprocessor is 8; the amount of shared memory available per multiprocessor is 16 KB organized into 16 banks, etc. According to the consideration of various constraints, it is appropriate to set T to the number of 16. Consequently, each Thread Block contains 256 threads, every Stream Multiprocessor would execute 3 Threads Blocks at the same time, and the number of Thread Blocks being parallel execution should be 3 times of the number of Stream Multiprocessor on the GPU.

This data segmentation strategy could make full use of the Shared Memory and could reduce reading and writing to the Global Memory. We can achieve 90X speedup in a low-end GPU than a CPU in the experimental result. Please refer to Section 4 for details.

C. Parallel Sort based on CUDA

Generally speaking, it is difficult to use GPU to accelerate sorting algorithms to a wondrous speedup ratio as in the distance computation phase, for there are too many branches in the thread and it's not fit for the GPU execution. Another reason is that the computational complexities of the current sorting algorithms are already very low. Among the CPU serial sorting algorithms, Quick Sort being the fastest one, is dominating the performance evaluation even in the same time complexity of $O(n \log n)$ algorithms when applying large amounts of

data. An implementation of a GPU-based parallel Bitonic Sort for huge data set introduced by us could bring a good performance of 10X~20X speedup compared to the CPU serial ordinary version of Bitonic Sort. However, it is not so significant compared to CPU Quick Sort.

Finally, the sorting algorithm we applied in this paper is a CUDA-based Radix Sort proposed in reference [7]. In a Radix Sort, it assumes that the keys are d-digital numbers and sorts one digit from least to most significant of the keys at a time. The implementation of the Radix Sort is divided into four steps:

1) Each block loads and sorts its tile in Shared Memory using b iterations of 1-bit split. Empirically, we can reach best overall performance by choosing b = 4.

2) Each block writes back the results to Global Memory, including its 2b-entry digit histogram and the sorted data tile

3) Conduct a prefix sum over the p×2b histogram table, which stored in column-major order, to compute global digit offsets.

4) Using prefix sum results, each block copies its elements to their corresponding output position.

This sorting algorithm can reach many times in performance compared with CPU Quick Sort. In the final performance test, the sorting phase occupied the largest proportion of the overall computing time. The sorting phase becomes the bottleneck in performance of the whole application.

D. Label decision

This step is to decide the classification label of the query point in test data set, according to the *K* nearest points in train data set. In this paper, a simple statistical election is made to complete the target among the labels of *K* points. As the result of the previous phase, we can get *K* nearest neighbor of a query point in the train data set, then we statistic the occurrences of each classification label. The most frequently occurred label would be chosen as the forecasting label of this query point. Weighted statistical methods can also be use in this step. The principle of this method is that the nearer neighbor to the query point should have a higher weight. We also have to define the weight values in advance in this method.

However, the computational complexity of this phase is low and it consumes little time. In our experimental results, no more than 20ms was spent during execution of this step. Meanwhile, the program branches are very high, and it is difficult to optimize for the GPU execution. Therefore, the CPU is adapted to accomplish this work.

IV. Experimental Results

A. Environments

The computer used to do this comparison is a Pentium D 2.8GHz dual core CPU with 1.5GB of DDR2 memory. The graphic card used is a G92 based NVIDIA GeForce 9600GSO with 96 streaming processors and 192bit 384MB of DDR3 memory interfaced with a PCI-Express 1.1 port.

The test data, Adult dataset, is from the UCI Machine Learning Repository. The number of classification label is 2, with 123 numbers of features. The values had been normalized into [0, 1] of real numbers. The a1a data set including a train set with the size of 30956 and the test data set is 1605, while the a2a data set including a train set of 30296 and the test set is 2265.

B. Performance and analysis

The CUDA implementation of GPU-based algorithm introduces by this paper is identified as “GPU”. For comparison, the original CPU serial algorithm is identified as “CPU” and Approximate Nearest Neighbor Searching using brute force method with KD-Tree optimization is identified as “ANN-Brute”. The initialization and input-output part of the program using in the methods are almost the same. The execution times of core part in each algorithm are shown in Figure 4.

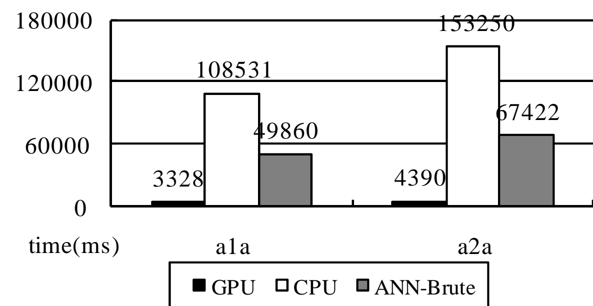

Figure 4. Execution Time-Overall

As is shown in the figure, the advantage of GPU algorithm in overall execution time is obvious. In a1a data set, we achieved the speedup of 32.61X compared with CPU algorithm and 14.98X with ANN-Brute method. In a2a data set, we reached the speedup of 34.91X compared with CPU algorithm and 15.36X with ANN-Brute method.

Since a kind of data segmentation method is presented in this paper and the tile size is 16, so we obtained appreciative performance in high-dimensional data sets. But if the data dimension is relatively small or even lower than 16, the performance will be reduced, when we should reconsider the tile size as parameter. In this condition, we can use rectangle tile instead of squared tile. For example, use an 8-width and 32-height tile, to adapt some small dimension data set, while the number of threads per block is still kept 256.

It could also be found during the experiments that, the decision of classification label phase in the algorithm is about 16ms. This phase is relatively simple and the execution time is very short and negligible, when Distances calculation and sorting phase occupying most of the time. Because the ANN-Brute algorithm as a whole process is completely different from GPU and CPU algorithms, we use Table 1 to illustrate the execution time in each phase in GPU-based algorithm and CPU comparison algorithm. The distances calculation phase is presented in T1, the sorting phase is T2 and the label

decision time is T3. The proportion of distances computation phase is shown in the last column.

TABLE I. EXECUTION TIME IN EACH PHASE

	T1 (ms)	T2 (ms)	T3 (ms)	T1/(T1+T2+T3) (%)
a1a GPU	828	2484	16	24.88%
a1a CPU	77871	30645	15	71.75%
a2a GPU	1141	3233	16	25.99%
a2a CPU	110492	42752	16	72.10%

As is shown in the table, the distances calculation phase is occupying the major proportion of the execution time in CPU implementation. In the CUDA implementation proposed in this paper, we had already achieved 94X and 96X speedup separately in this phase and made it occupying a smaller proportion of the overall execution time, transforming the bottleneck to the sorting phase. In fact, the complexity of sorting algorithm had already relatively low and most of the time was spent in reading and writing operation to the Graphics Memory, resulting that in the sorting phase we could only obtain 12X~13X speedup than that in CPU implementation.

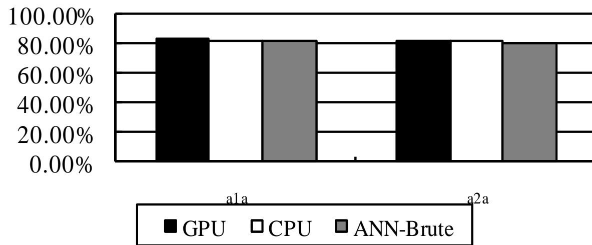

Figure 5. The Accuracy of Classification

Being a classification algorithm, the accuracy of the method should be presented as routine. Figure 5 gives us the accuracy of the algorithm. The accuracy of the three methods shows little difference, for the principle of them is just the same. In KNN Brute Search algorithms, the final result depends entirely on the quality of the data set. The slight difference of the results is due to sorting step-for the same distances between the query point and that from train set with different category label was cut by the parameter K differently because of unstable sorting algorithms.

V. CONCLUSION

This paper presented a CUDA based KNN algorithm, which could take full advantage of the computational horsepower of GPU and its multi-leveled memory architecture, making the performance of the method obtain greatly enhancement compared with CPU implementation. The tremendous increase in performance reached a cluster of computers, on which is only a PC with a 500RMB ($73) cost graphics card. This method is valuable for the KNN method in high dimensions, large amounts of data for applications.

ACKNOWLEDGMENT

The paper is supported by National Natural Science Foundation of China (No. 60873047) and Natural Science Foundation of Jiangsu Province of China (No. BK2008154).

The authors are grateful to all the people for helpful suggestions. The authors would like to thank all the reviewers for their helpful comments on earlier drafts of this paper.

REFERENCES

[1] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. "An optimal algorithm for approximate nearest neighbor searching fixed dimensions", Journal of the ACM, 45(6):891–923, 1998.

[2] D. M. Mount, S Arya. "ANN: A library for approximate nearest neighbor searching", http://www.cs.umd.edu/~mount/ANN/

[3] Enhua Wu, Youquan Liu, "Emerging technology about GPGPU", APCCAS. IEEE Asia Pacific Conference on Circuits and Systems, 2008.

[4] "NVIDIA CUDA Compute Unified Device Architecture: Programming Guide", Version 2.3, July 2009.

[5] Feng Cao, Anthony K. H. Tung, and Aoying Zhou, "Scalable clustering using graphics processors", Lecture Notes in Computer Science, Advances in Web-Age Information Management - 7th International Conference, WAIM 2006.

[6] Daniel Cederman and Philippas Tsigas, "A Practical Quicksort Algorithm for Graphics Processors", In the Proceedings of the 16th Annual European Symposium on Algorithms (ESA 2008), Lecture Notes in Computer Science Vol.: 5193, pages 246 - 258, Springer-Verlag 2008.

[7] Nadathur Satish, Mark Harris, Michael Garland, "Designing efficient sorting algorithms for manycore GPUs", Proc. 23rd IEEE Int'l Parallel & Distributed Processing Symposium, May 2009.

[8] Wenbin Fang, "Parallel Data Mining on Graphics Processors", Technical Report HKUST-CS08-07, Oct 2008.

[9] V. Garcia, E. Debreuve, M. Barlaud.Fast, "K nearest neighbor search using GPU", In Proceedings of the CVPR Workshop on Computer Vision on GPU, June 2008.

[10] Buck, Ian, "GPU Computing: Programming a Massively Parallel Processor", CGO '07.

[11] Mark Harris, "Parallel Prefix Sum (Scan) with CUDA", http://www.nvidia.com/object/cuda_home.html, 2008-1.

[12] Mark Harris, "Optimizing Parallel Reduction in CUDA", http://www.nvidia.com/object/cuda_home.html, 2007-11.

[13] Xiaowen Chu, Kaiyong Zhao, Mea Wang, "Massively Parallel Network Coding on GPUs", IPCCC 08.

[14] X.-W. Chu, K.-Y Zhao, and M. Wang, "Practical Random Linear Network Coding on GPUs", Technical Report, Dec 2008.

[15] M. Suhail Rehman, Kishore Kothapalli, P. J. Narayanan, "Fast and Scalable List Ranking on the GPU", 23rd International Conference on Supercomputing (ICS), June 2009.

[16] John Stratton, Sam Stone, Wen-mei Hwu, "MCUDA: An Efficient Implementation of CUDA Kernels on Multi-cores", Technical report, IMPACT-08-01, March, 2008.