Design and Evaluation of a Parallel K-Nearest Neighbor Algorithm on CUDA-enabled GPU

Shenshen Liang[a], Ying Liu[a], Cheng Wang[b]， Liheng Jian[a]
[a]*Graduate University of Chinese Academy of Sciences, Beijing, 100190, China*
[b]*Agilent Technologies, Beijing, 100102, China*
{liangshenshen08, jianlih06r}@mails.gucas.ac.cn, yingliu@gucas.ac.cn, zheng_wang@agilent.com

Abstract

Recent developments in Graphics Processing Units (GPUs) have enabled inexpensive high performance computing for general-purpose applications. Due to GPU's tremendous computing capability, it has emerged as the co-processor of CPU to achieve a high overall throughput. CUDA programming model provides the programmers adequate C language like APIs to better exploit the parallel power of the GPU. K-nearest neighbor (KNN) is a widely used classification technique and has significant applications in various domains, especially in text classification. The computational-intensive nature of KNN requires a high performance implementation. In this paper, we propose CUKNN, a CUDA-based parallel implementation of KNN. It launches two CUDA kernels, distance calculation kernel and selecting kernel. In the distance calculation kernel, a great number of concurrent CUDA threads are issued, where each thread performs the calculation between the query object and a reference object; in the selecting kernel, threads in a block find the local-k nearest neighbors of the query object concurrently, and then a thread is invoked to find the global-k nearest neighbors out of the queues of local-k neighbors. Various CUDA optimization techniques are applied to maximize the utilization of GPU. We evaluate our implementation by using synthetic datasets and a real physical simulation dataset. The experimental results demonstrate that CUKNN outperforms the serial KNN on an HP xw8600 workstation significantly, achieving up to 46.71X speedup on the synthetic datasets and 42.49X on the physical simulation dataset including I/O cost. It also shows good scalability when varying the number of dimensions of the reference dataset, the number of objects in the reference dataset, and the number of objects in the query dataset.

978-1-4244-6359-6/10/$26.00 ©2010 IEEE

1. Introduction

The size of various datasets has increased tremendously in recent years as speedups in processing and communication have greatly improved the capability for data generation and collection in areas such as scientific experimentation, business and government transactions, as well as the Internet. Traditional knowledge discovery systems have been found lacking in their ability to handle current datasets, due to characteristics such as their large sizes and high dimensionality. Consequently, new techniques that can automatically transform these large datasets into useful information are in strong demand. Data mining, which discovers interesting, meaningful and understandable patterns hidden in massive datasets, has become a hot research domain. Important application areas include business intelligence, customer relationship management, WWW, scientific simulation, e-commerce, bioinformatics and many more.

Classification is one of the important data mining techniques whereby a model is trained on a dataset with class labels and then used to predict the class label of unknown objects. K-nearest neighbor (KNN) [1] is one of the most widely used classification algorithms. For each query (unknown) object, KNN has to scan all the objects in the reference dataset. Assume the number of reference objects is n and the number of query objects is m, the complexity required to classify the query objects is O(*mn*), which is computational intensive [1]. Therefore, a conventional serial program handing large datasets would potentially be unable to run in-core or take a tremendous amount of time.

Therefore, parallel computing is an essential component of the solution to speed up KNN. Modern supercomputers include shared memory parallel

computers (SMPs) and massively parallel systems (clusters). However, taking into consideration the high cost and the power consumption, parallel KNN is not appealing to the medium-sized business and individuals.

The rapid increase in the performance of graphics hardware has made GPU a strong candidate for high performance computing. GPUs now include fully programmable processing units that follow a stream programming model. High level languages have emerged to support easy programming. NVIDIA's GPU with CUDA (Compute Unified Device Architecture) environment provides standard C like interface to manipulate the GPUs [2]. GPUs with CUDA provide tremendous memory bandwidth and computing power. For example, NVIDIA's GeForce 8800 GTX can achieve a sustained memory bandwidth of 86.4GB/s and a peak of 346 GFLOPS/s; NVIDIA's Tesla C1060 can achieve a bandwidth of 102 GB/s and a peak of 933 GFLOPS/s [2]. In addition, low cost is the highlight of GPU. For example, GeForce 8-series products cost only $90, and the high-end product, Tesla C1060, costs only $1,500. One million US dollars can buy 8.3 TFLOPS/s CPU computing capability but 439.1 TFLOPS/s GPU capability [2]. The computing power of GPU is equivalent to a medium-sized supercomputer which is orders of magnitude more costly. Such a low cost provides the medium-sized business and individuals great opportunities to afford supercomputing facilities.

Therefore, recently, a great number of applications have turned to perform parallel computing on GPU+CPU heterogeneous system where the GPU acts as the computation accelerator, including Finite Difference Time Domain (FDTD) [3], Computational Fluid Dynamics (CFD) [4], Magnetic Resonance Imaging (MRI) [5], neural network [6], support vector machine [7], intrusion detection [8], etc.

In this paper, we present an efficient parallel implementation of KNN algorithm on CUDA-enabled GPU, called CUKNN. Two CUDA kernels are constructed: *distance calculation* kernel and *selecting* kernel. The distance calculation kernel is parallelized in a data-parallel fashion, where threads in a block share the reference objects with others which are loaded into the shared memory from the global memory. Threads in block search for *local-k* nearest neighbors of each query object concurrently, and use one thread to find *global-k* nearest neighbors of each query object. We evaluate the performance by comparing the execution time of CUKNN on a Tesla C1060 card with an efficient serial KNN program on an Intel Xeon CPU on both synthetic data and real physical simulation data. It not only performs very efficiently in terms of speedup, but also shows good scalability. Experimental results present up to 46.71X in the synthetic data and 24.57X in the real data.

The rest of this paper is organized as follows. Section 2 presents the background knowledge of this paper. Section 3 overviews the related work. In section 4, we present CUKNN. Experimental results are presented in section 5 and we summarize our work in section 6.

2. CUDA parallel computing architecture

2.1. NVIDIA's GPU architecture

GPUs have a parallel architecture with massively parallel processors. The graphics pipeline is well suited to rendering process because it allows the GPU to function as a stream processor. NVIDIA's GPU with the CUDA programming model provides an adequate API for non-graphics applications. The CPU treats a CUDA device as a many-core co-processor.

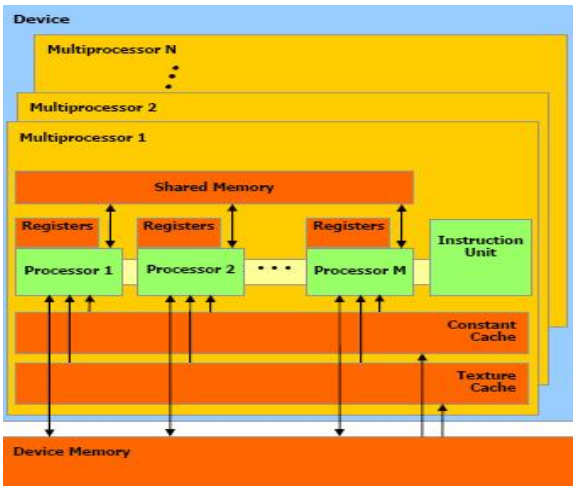

Figure 1. A set of SIMD stream multiprocessors with on-chip shared memory.

At the hardware level, CUDA-enabled GPU is a set of SIMD stream multiprocessors (SMs) with 8 stream processors (SPs) each. GeForce 8800GTX has 128 SPs and Tesla C1060 has 240 SPs. Each SM contains a fast shared memory, which is shared by all its processors as shown in Figure 1. A set of local 32-bit registers is available for each SP. The SMs communicate through the global/device memory. The global memory can be read from or written to by the host, and are persistent across the kernel launches of the same application,

however, it is much slower than shared memory. Shared memory is managed explicitly by the programmers. Compared to the CPU, the peak floating-point capability of the GPU is an order of magnitude higher, as well as the memory bandwidth.

2.2. CUDA programming model

At the software level, CUDA model is a collection of threads running in parallel. The unit of work issued by the host computer to the GPU is called a kernel. CUDA program is running in a data-parallel fashion. Computation is organized as a grid of thread blocks as shown in Figure 2. Each SM executes one or more thread blocks concurrently. A block is a batch of SIMD-parallel threads that runs on the same SM at a given moment. For a given thread, its index determines the portion of data to be processed. Threads in a common block communicate through the shared memory.

CUDA consists of a set of C language extensions and a runtime library that provides APIs to control the GPU. Thus, CUDA programming model allows the programmers to better exploit the parallel power of the GPU for general-purpose computing.

3. Related work

Various parallel data mining algorithms have been explored, including parallel decision tree (e.g. SPRINT [9], BASIC [10], ScalParC [11]), parallel ARM (e.g. CCPD [12], CD [13]), and parallel clustering (e.g. MAFIA [14]), etc. By applying various optimization techniques, CCPD (Common Count Partitioned Database) obtained about 6.8X speedup on a 12-node SGI Power Challenge machine on synthetic data by IBM Quest Data Generator. CD (Count Distribution) performs better on clusters, showing up to 13X speedup on 16 nodes of a 32-node IBM SP2. Distributed ARM has also been developed [15].

CUDA-based general-purpose parallel computing has become popular in the past two years. Che et al. [16] demonstrated the feasibility of applying CUDA to general-purpose applications. Pros and cons were discussed and suggestions were presented to NVIDIA's GPU and CUDA. Catanzaro et al. [7] implemented a fast support vector machine using GPU with CUDA, and it showed over 100X speedup in the best case. Neural network is another good application of CUDA since the data dependence between neurons is small. Billconan et al. [6] implemented a CUDA-based Neural Network for handwriting recognition applications, running 10 times faster than the serial NN on the CPU. Vasiliadis et al. [8] presented a CUDA-based intrusion detection system which achieved a maximum traffic processing throughput of 2.3 GB/s. K-means is partially paralleled with CUDA in [16], where each thread performs a portion of the distance computation.

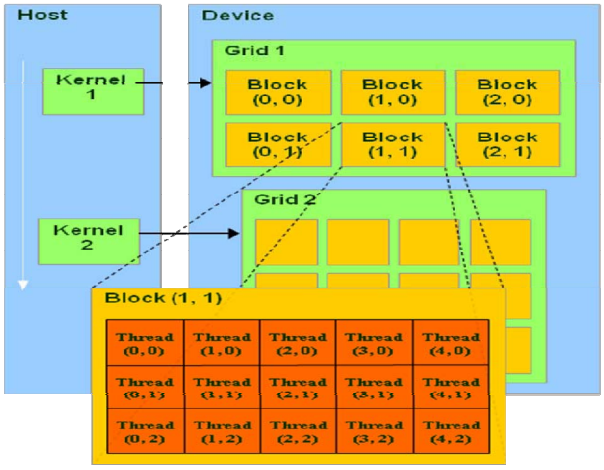

Figure 2. CUDA programming model.

Garcia et al. [17] introduced a fast k-nearest-neighbor algorithm. It is implemented in C and MATLAB using GPU with CUDA. In this paper, multiple blocks of threads are launched, where each thread computes the distance between the query object and a reference object. Different query objects are sorted in parallel. But the sorting method in [17] is different with that in our approach. In addition, [17] performs the classification for one query per iteration, which is not as efficient as our approach.

4. CUKNN: KNN parallelization on CUDA-enabled GPU

4.1. K-nearest neighbor classifier

KNN classifier predicts the class label of the query object by the majority of the class labels of its k nearest neighbors in the reference objects. Given a query (unknown) object p, a KNN classifier scans the reference dataset for k objects closest to p, and then, p is assigned to the majority class label of the k nearest neighbors. "Closeness" is usually defined as a distance metric, such as Euclidean distance, Cosine distance, etc. The most time-consuming part of KNN includes the distance calculation component and the k nearest neighbor selecting component. GPU's many-core architecture is exactly suitable for such computational

intensive applications. Therefore, our work focuses on accelerating these two components on GPU.

4.2. Distance calculation kernel

The goal of this kernel is to maximize the concurrency of the distance calculation invoked by different threads. In addition, since the latency of global memory access (400-600 clock cycles per access) is high, global memory access should be minimized.

Distance calculation can be fully parallelized since it is independent between pairs of objects. This property makes KNN perfectly suitable for a GPU parallel implementation. The distance calculation kernel is parallelized in a data-parallel fashion. In our algorithm, after transferring the data from CPU to GPU, each thread performs the distance calculation between the query object and a reference object. A unique segment of the reference dataset is loaded into the shared memory from the global memory. Threads in a common block share the reference objects with others. Since the number of reference objects is huge, a large number of threads and blocks will be launched in this kernel. The process is illustrated in Figure 3.

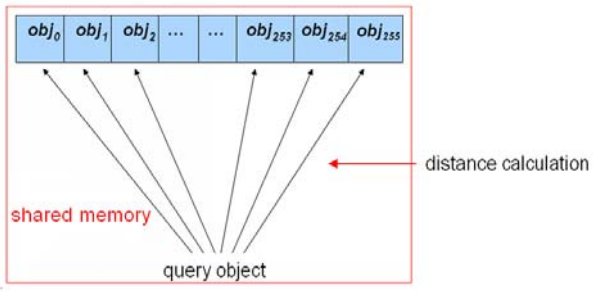

Figure 3. Distance calculation is parallelized in a data-parallel fashion. Each thread performs the distance calculation between the query object and a reference object.

4.3. Selecting kernel

Once the distances between the query object p and all the reference objects are obtained, selecting kernel is performed to find the k nearest neighbors of p.

First, let's focus on the shared memory of each CUDA block. The distances calculated by the threads in a block are stored in the shared memory. Thread t_i takes care of one distance, d_i. By looking through the distances calculated by other threads, t_i obtains the rank of d_i. All the threads in a block obtain such ranks simultaneously. The shortest k distances are called the *local-k* nearest neighbors of p (Note that the k nearest neighbors are sorted in ascending order). Also, all the CUDA blocks generate the ranks concurrently.

Second, one thread t is used to find the *global-k* nearest neighbors of p from all the *local-k* neighbors obtained from all the CUDA blocks. Assume m blocks are launched, and each block stores a queue for k shortest distances in ascending order. In the first iteration, thread t selects the global shortest one from the m queues of *local-k* neighbors. Notice that since the *local-k* is a queue in ascending order, the top one in each queue is the local closest neighbor of p. The smallest one of the m top ones is the global nearest neighbor of p. Once it is selected, it will be popped out of the queue and the second one will be pushed to the top. This step will be repeated k times until the *global-k* nearest neighbors of p are selected. This process is shown in Figure 4.

Once the *global-k* nearest neighbors are obtained, the class label of p is determined by the majority of the class labels of the *global-k* neighbors. Since k is not large (usually less than 20), it's not necessary to use GPU to perform this task of voting. So, we transfer the *global-k* nearest neighbors to the CPU, and let the CPU determine the class label of the query object p.

4.4. Multiple query objects

Usually, multiple query objects are to be classified in real applications. Our method is as follows:

Distance calculation kernel. In addition to a segment of reference objects, all the query objects are also divided into segments and loaded into the shared memory of every block in the GPU kernel. Then, the kernel is launched where each thread performs the distance calculation between the r query objects and a reference object. Thus, r distances are maintained in a thread.

Selecting kernel. For any given query object, the selecting process is exactly the same as that in Section 4.3. Assume r query objects to be classified, r threads are used to find the *global-k* nearest neighbors with one query per thread. Thus, the selecting process for the r query objects is performed in parallel.

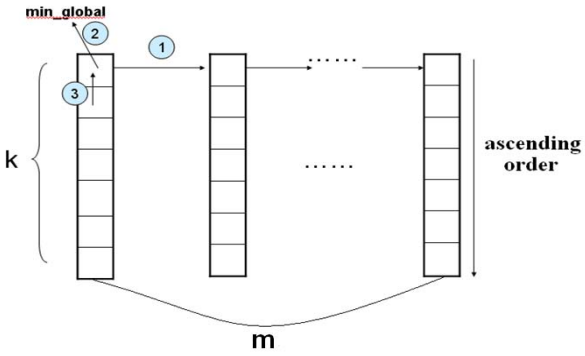

Figure 4. Global-k nearest neighbors from m queues of local-k nearest neighbors.

4.5. Optimization

Various optimization techniques are applied in our algorithm to better utilize the computing power of GPU.

4.5.1. CUDA Stream. A stream is a sequence of operations that execute in order. Unlike CPU, GPU has two sets of pipelines: memory pipeline and processor pipeline. So, data loading can be performed while other threads performing arithmetic operations simultaneously. In our algorithm, we divide the reference dataset into several streams, which are transferred to the global memory of GPU in random order. The distance calculation performed on SPs overlaps the transferring of another stream from the CPU to the GPU. In this way, part of the I/O time is "hidden" by the distance calculation. Thus, the overall execution time will be significantly reduced.

4.5.2. Memory coalescing. Memory coalescing is a technique in which threads access the neighboring and sequential address in a coalesced manner. In the distance calculation kernel, in order to reduce the global memory access cost, our algorithm loads the reference objects and the query objects in a memory coalesced manner. The objects for a half-warp are coalesced into a single memory read transaction, rather than 16 transactions. Thus, the bandwidth between the global memory and the shared memory is utilized efficiently.

5. Experimental evaluation

The device we used in our experiments is NVIDIA's Tesla C1060 card with 240 1.30 GHz SPs and 4GB global/device memory. NVIDIA driver 180.22 and CUDA 2.1 are installed. All the experiments were performed on an HP xw8600 workstation with a quad-core 2.66 GHz Intel Xeon CPU and 4 GB main memory, running the Red Hat Enterprise Linux WS 4.7. In order to give a fair comparison, we implemented a serial KNN algorithm in C/C++. We use synthetic data generated by MATLAB and a real physical simulation dataset for our evaluation. Details of these datasets are described in the following subsections. The outputs of CUKNN are guaranteed to be exactly the same as that of the serial KNN. The execution time depends on the size of the reference set and the query set, and the number of dimensions of the objects. In this paper, quick sort is applied in the serial KNN since it is one of the most efficient sorting algorithms. We also compare CUKNN with another typical KNN version that uses insertion sort.

5.1. Synthetic data

We performed CUKNN on a set of synthetic datasets generated by MATLAB, where the number of reference objects is 262,144, the number of neighbors k is 7.

Figure 5 shows the execution time of CUKNN when varying the number of dimensions of the objects. Observed from Figure 5, CUKNN is scalable in terms of the number of dimensions. Actually, our algorithm shows a linear scalability.

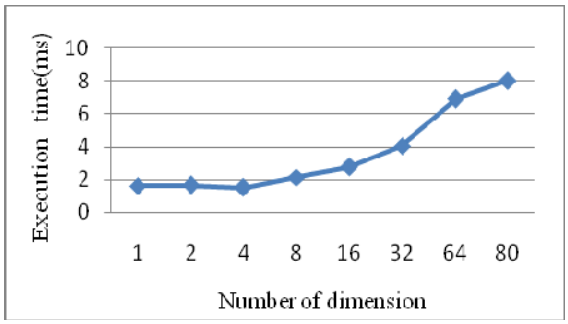

Figure 5. Execution time with varying number of dimensions of the objects.

Figure 6 shows the scalability of CUKNN when varying the number of the objects from 8K to 1048K and the number of dimensions of the objects is set at 8. From the experimental results, we can see that the execution time of CUKNN rises linearly when varying the number of the objects in the reference dataset.

Due to the fact that he cost of finding the k nearest neighbors is trivial in comparison with the distance computation workload, we do not vary k in our

experiments.

Figure 7 shows the speedups when running CUKNN on a set of datasets (k is set at 7). We compare the execution time of two serial KNN methods with that of CUKNN. Note that the I/O time is included in the total execution time. The speedup rises as the number of reference objects does. The best case of either serial KNN happens when the dataset is 524 K, where our CUKNN achieves 21.81X speedup over quick sort based KNN and 46.71X speedup over insertion sort based KNN. Figure 7 also indicates that CUKNN performs better when data size getting larger.

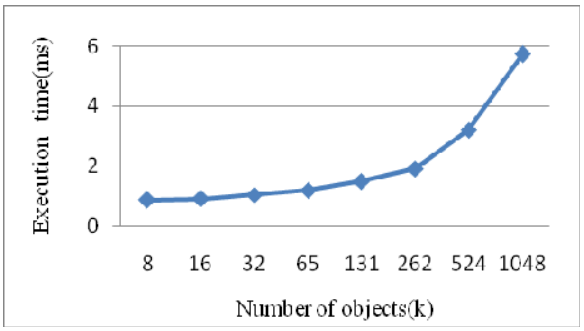

Figure 6. Execution time with varying number of reference objects.

Also, we vary the number of objects in the query dataset from 16K to 1048K. The number of reference objects is 32,768, the number of dimensions is 8 and k is 7. Figure 8 shows the execution time of CUKNN and two serial KNN versions with insertion and quick sort. We can see that the time of CUKNN rises slowly while the other two rises significantly. It indicates that CUKNN performs even better when the number of query objects getting larger.

5.2. Physical simulation data

We also evaluate CUKNN on a real-world data set. This dataset was used to predict the classification of the particle in high energy collider experiments of quantum physics [18]. The source database stores the features and corresponding class of each sample. The samples are converted into textual records, where unique IDs are assigned. The attributes denote physical features and the label indicates the class. The number of reference objects is 32,768 and the number of attributes (dimensions) of each record is 65.

We evaluate the scalability of CUKNN by varying the number of query objects. As shown in Table 1, the execution time rises relatively slow comparing with the number of query objects, while the time of the serial KNN rises significantly. Observed from the experiments, we can see that as the number of objects increases, more and more CUDA blocks and threads are launched, more concurrency is incurred, hence the stream processors (SPs) are able to run at full speed without interruption.

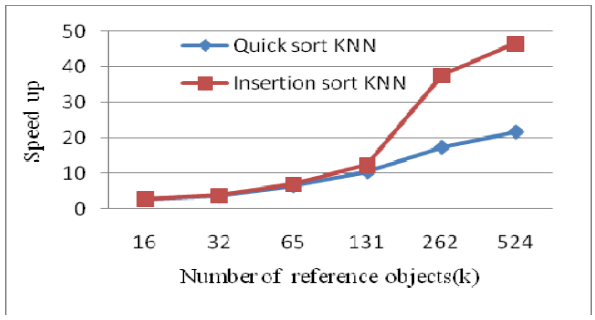

Figure 7. Speedups with varying number of reference objects.

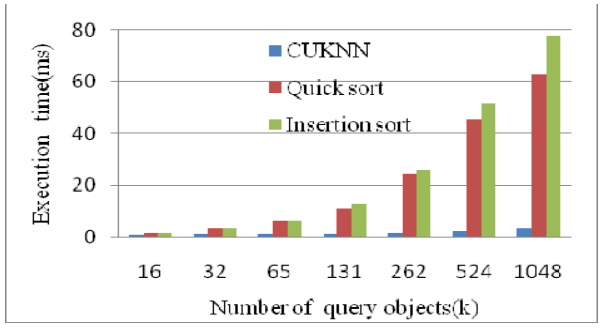

Figure 8. Execution time with varying number of query objects.

Table 1. Execution time (s) and speedups when varying the number of query objects.

# of query records	2000	4000	6000	8000	10000	12000
Serial Code	59.63	116.79	174.39	231.58	289.27	347.15
CUKNN	2.29	3.47	4.64	5.84	7.02	8.17
Speedup	26.04	33.66	37.58	39.65	41.21	42.49

The best case happens at 12,000 with 42.49X. It indicates that our CUKNN is suitable to solve large-scale applications.

5.3. Optimization

Table 2 shows the data transferring cost for CUKNN without using CUDA *stream*, where we can see that a large portion of the overall execution time devotes to data transferring from the CPU to the GPU.

Table 2. Distribution of overall execution time in CUKNN on synthetic datasets

# Dimension	Computation time(ms)	Overall execution time(ms)	Transfer proportion
4	0.241	19.994	98.79%
8	0.284	28.581	99.01%
16	0.289	50.508	99.43%
32	0.336	91.662	99.63%
64	0.337	172.970	99.81%

In order to reduce the data transferring cost, CUDA stream technique is applied in our algorithm. Figure 9 shows execution time of CUKNN using and not using CUDA streams, respectively. The number of the reference objects is 262,144, and k is set at 7. We vary the number of dimensions of the objects. The experimental results verified that stream optimization can significantly reduce the overall execution time of CUKNN.

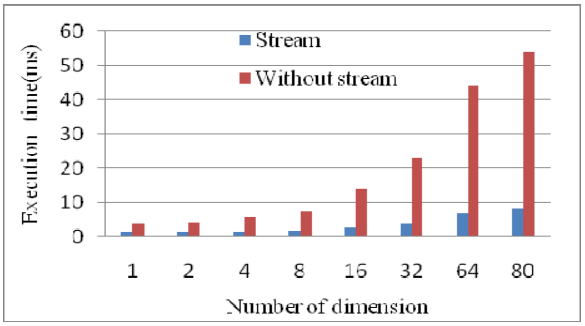

Figure 9. Execution time with stream and without stream.

6. Conclusion

In this paper, we presented a CUDA-based parallel implementation of KNN. CUKNN is actually a hybrid implementation of the CPU and the GPU. The GPU performs two CUDA kernels: distance calculation and selecting . Data elements in these two kernels are processed in a data-parallel fashion. Memory coalescing technique is used in distance calculation kernel to reduce the number of memory read transactions from the GPU global memory to the on-chip shared memory. Stream is also applied in this kernel so that the next memory read latency is hidden by the current computation. Experiments showed good scalability on synthetic data and real physical simulation data. CUKNN presented up to 46.71X speedup compared to the serial KNN with quick sort in the total execution time in synthetic data and 42.49X speedup in real data. The results showed that our proposed CUKNN is a promising solution for classification for large large-scale applications.

As the emergence of the CUDA programming model, GPU has become a promising platform for supercomputing. Its superior floating-point computation capability and low cost appeal to the medium-sized business and individuals. For example, the computing power of a Tesla C1060 card (933 GFLOPS/s) is equivalent to a medium-sized SMP. Problems that used to require a computer cluster to process can now be solved on a desktop. We believe the GPU-based parallel computing will provide compelling benefits for data mining applications.

Acknowledgments

This project is partially supported by the NVIDIA's Professor Partnership, 2009, and National Natural Science Foundation of China, #70621001, #90718042.

References

[1] M. Kamber, J. Han, Data Mining: Concepts and Techniques, 2nd Edition, Morgan Kaufmann, 2005.

[2] NVIDIA CUDA Programming Guide 2.1, 2008, http://www.nvidia.com/object/cuda_develop.html

[3] Balevic, L. Rockstroh, W. Li, et al. "Acceleration of a Finite-Difference Time-Domain Method with General Purpose GPUs (GPGPUs)", Proc. of International Conference on Computer and Information Technology, 2008, vol. 1-2, pp. 291-294.

[4] A Fast Double Precision CFD Code using CUDA,http://www.nvidia.com/object/cuda_home.html#state=detailsOpen;aid=65d33580-ff45 -439e-9f5c-5fdc1814f542

[5] W.K. Jeong, P.T. Fletcher, R. Tao, et al. "Interactive Visualization of Volumetric White Matter Connectivity", IEEE Transaction on Visualization and Computer Graphics, 2007, Vol 3, Issue. 6, pp. 1480-1487.

[6] Kavinguy, "A Neural Network on GPU", http://www.codeproject.com/KB/graphics/GPUNN.aspx.

[7] Catanzaro, N. Sundaram, K. Keutzer, “Fast Support Vector Machine Training and Classification on Graphics Processors”, Proc. of International Conference on Machine Learning, 2008, pp. 104-111.
[8] G. Vasiliadis, S. Antonatos, M. Polychronakis, et al, “Gnort: High Performance Network Intrusion Detection Using Graphics Processors”, Recent Advances in Intrusion Detection (RAID), 2008, Vol. 5230, pp. 116-134.
[9] J.C. Shafer, R. Agrawal, M. Mehta, “SPRINT: A Scalable Parallel Classifier for Data Mining”, Proc. of Inernationall Conference on Very Large Data Bases, Mumbai (Bombay), India, Sept. 1996.
[10] M. J. Zaki, C.T. Ho, R. Agrawal, “Scalable Parallel Classification for Data Mining on Shared-Memory Multiprocessors”, IEEE International Conference on Data Engineering, March 1999.
[11] M.V. Joshi, G. Karypis and V. Kumar, “ScalParC: A New Scalable and Efficient Parallel Classification Algorithm for Mining Large Datasets”, Proc. of International Parallel Processing Symposium, April 1998.
[12] M.J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li, “Parallel Data Mining for Association Rules on Shared-memory Multi-processors”, Proc. of Supercomputing, Pittsburg, PA, 1996.
[13] R. Agrawal, C. Shafer, “Parallel Mining of Association Rules”, IEEE Transactions on Knowledge and Data Engineering, 1996.
[14] H.S. Nagesh, A. Choudhary, S. Goil, "A Scalable Parallel Subspace Clustering Algorithm for Massive Data Sets", Proc. of International Conference on Parallel Processing, 2000.
[15] M.E. Otey, S. Parthasarathy, C. Wang, A. Veloso, W. Meira Jr., “Parallel and Distributed Methods for Incremental Frequent Itemset Mining”. IEEE Transactions on Systems, Man, and Cybernetics, Part B 34(6): 2439-2450, 2004.
[16] S. Che, M. Boyer, J.Y. Meng, et al, “A Performance Study of General Purpose Applications on Graphics Processors”, Journal of Parallel and Distributed Computing, 2008, Vol. 68, Issue. 10, pp. 137-1380.
[17] V. Garcia, E. Debreuve, M. Barlaud, “Fast K Nearest Neighbor Search using GPU”, IEEE Conference on Computer Vision and Patter Recognition Workshops, 2008, Vol. 1-3, pp. 1107-1112.
[18] http://kodiak.cs.cornell.edu/kddcup/datasets.html