CUKNN: A PARALLEL IMPLEMENTATION OF K-NEAREST NEIGHBOR ON CUDA-ENABLED GPU

Shenshen Liang[1], Cheng Wang[2], Ying Liu[1,3], Liheng Jian[1]

[1]Graduate University of Chinese Academy of Sciences, Beijing, China 100190
[2]Agilent Technologies Co. Ltd., Beijing, China 100102
[3]Fictitious Economy and Data Science Research Center, Chinese Academy of Sciences
Beijing, China 100190
{liangshenshen08, jianlih06r}@mails.gucas.ac.cn, zheng_wang@agilent.com
yingliu@gucas.ac.cn

ABSTRACT

Recent development in Graphics Processing Units (GPUs) has enabled inexpensive high performance computing for general-purpose applications. Due to GPU's tremendous computing capability, it has emerged as the co-processor of the 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 is a widely used classification technique and has significant applications in various domains. The computational-intensive nature of KNN requires a high performance implementation. In this paper, we present a CUDA-based parallel implementation of KNN, CUKNN, using CUDA multi-thread model. Various CUDA optimization techniques are applied to maximize the utilization of the GPU. CUKNN outperforms significantly and achieve up to 15.2X speedup. It also shows good scalability when varying the dimension of the training dataset and the number of records in training dataset.

Index Terms — KNN, classification, CUDA, parallel computing

1. INTRODUCTION

Data mining is to discover interesting, meaningful and understandable patterns hidden in massive data sets [1]. *K*-nearest neighbor (KNN) [1] is one of the most widely used techniques in classification applications and has tremendous applications in business, science, information retrieval, WWW, etc. Especially, in text classification, it has been experimentally demonstrated that KNN outperforms other classification techniques in terms of computational complexity or accuracy, such as neural network, decision tree, naïve Bayesian classifier, etc.

As the rapid development of hardware and software, the size of databases easily goes to hundreds of Gigabytes. For each testing object, KNN has to scan all the objects in the training set, and it is computational intensive [1]. The conventional serial programs may take a considerable long time or probably be unable to run in-core at all.

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. In addition, low cost is the highlight of GPU. 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, lots of applications have turned to perform parallel computing on GPU+CPU heterogeneous system where the GPU acts as the computation accelerator, including neural network [3], support vector machine [4], intrusion detection [5], k-nearest neighbor [6], Finite Difference Time Domain (FDTD), Magnetic Resonance Imaging (MRI), Computational Fluid Dynamics, data mining, etc.

In this paper, we present an efficient parallel implementation of KNN using CUDA-enabled GPU, called CUKNN. CUKNN constructs two multi-thread kernels: distance calculation kernel and sorting kernel. 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. It not only performs very efficiently in terms of speedup, but also shows good scalability. Experimental results show up to 15.2X in dataset.

978-1-4244-5076-3/09/$26.00 ©2009 IEEE

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.

At the hardware level, CUDA-enabled GPU is a set of SIMD stream multiprocessors (SMs) with 8 stream processors (SPs) each. Each SM contains a fast shared memory, which is shared by all its processors. 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. Shared memory is managed explicitly by the programmers. It's much slower than shared memory. 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*. 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. CUKNN: A CUDA-BASED PARALLEL IMPLEMENTATION OF KNN

3.1. *K*-nearest neighbor

KNN algorithm is a method for classifying objects based on the closest training objects. KNN is a lazy learner where a testing object is classified by a majority vote of its *k* nearest neighbors in the training objects. Given an testing (unknown) object *p*, a KNN classifier searches the training data set for the *k* objects that are closest to *p*. "Closeness" is defined by a distance metric, such as Euclidean distance, etc. *p* is assigned to the most common class among its *k* nearest neighbors. The most time-consuming part of KNN includes distance calculation and sorting. Therefore, our work focuses on accelerating these two phases.

3.2. Distance calculation kernel

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

The computation of the distances can be fully parallelized since the distances between pairs of tuples are independent. This property makes KNN perfectly suitable for a GPU parallel implementation. In our algorithm, after transferring the data from CPU to GPU, each thread performs the distance calculation between the unknown object and a training object. The training set is loaded into the shared memory of each SM from the global memory. Each SM manages a unique portion the training set. Threads in a common block share the training objects with others. Thus, a large number of threads and blocks are launched. In this way, the distance calculation is parallelized in a data-parallel fashion. The process is illustrated in Figure 1.

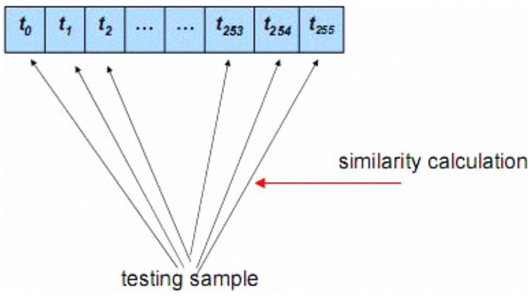

Figure 1. Illustration of the distance calculation in parallel

Optimization methods are applied to improve the performance.

1) Stream is used to improve concurrency. A stream is a sequence of operation 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 this paper, we divide the training dataset into several streams, where the streams can be transferred to the global memory of GPU in random order. The calculation between the unknown object *p* with objects in stream *i* is parallelized with the transferring of stream *j* from CPU to GPU. Thus, some of the I/O time is "hided" by the distance calculation.

2) Since the objects in the global memory are stored consecutively, all the threads in a half-warp access the data in sequence. Data for a half-warp is coalesced into a single memory read transaction, rather than 16 transactions. In this

way, memory coalescing is achieved. Thus, the bandwidth between the global memory and the shared memory is utilized efficiently.

3.3. Sorting kernel

After calculating the distances between the unknown object, *p*, and the training objects, sorting is performed to find the *k* nearest neighbors to *p*.

First, the distances between *p* and the objects in a block are stored in the shared memory. In our paper, each thread takes care one distance. For a given thread, it obtains the rank of its distance by comparing it with other distances in the shared memory. Such ranks are generated simultaneously so that the nearest *k* neighbors of *p* in the block are obtained, called *local k* nearest neighbors. Also, the *local k* neighbors are sorted according to the rankings.

Second, we need to generate the global k nearest neighbors across all the blocks from the local k ones of an unknown tuples. Provided each unknown tuple has m local ks, we use one thread to deal with them. The algorithm is: the thread scans the smallest ones of each local k. Notice that the local k is a queue in ascending order, and the first one in the queue is the local smallest. The smallest one of m local ones is selected and it's the global smallest. Then the second smallest one of the selected queue is compared with the rest (m-1) local smallest ones. In this way, each scan generates one nearest neighbor. With only k*m scans, the global k nearest neighbors of p are obtained. Figure 2 illustrates this process.

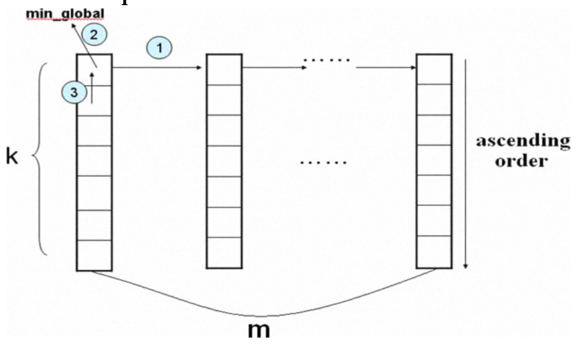

Figure 2. Global k generated from m queues of local k

Once the *global k* nearest neighbors are obtained, it's easy to figure out the class label of *p*. Since *k* is usually not large (less than 20 on average), it's not necessary to use GPU to perform this task. So, we transfer the *k* nearest neighbors to the CPU, and let the CPU find out the majority labels of the *k* neighbors and determine the label of the unknown tuple.

3.4. Multiple unknown objects to be classified

When the number of unknown objects rises, the algorithm will be different. In distance calculation kernel, the testing data is also loaded into the shared memory of each SM from the global memory. Then we launch threads and each unknown tuples has n threads to compute its distance to the training tuples in shared memory. (n is the number of training tuples in shared memory)In this way, multiple distances are computed.

In the sorting kernel, distances of different unknown tuples are loaded into shared memory. Then we launch m blocks of thread (m is the number of testing tuples in shared memory), with each block has n threads. The "local k " of m unknown tuples can be calculated in parallel.

Once the local k of present m unknown have all been generated, they are stored into global memory and the next m unknown ones are loaded into shared memory. The distance calculation and sorting kernels are recursive operated until local ks of all unknown tuples are obtained. Thus, distances between n training tuples in shared memory and all the testing tuples are computed. These n training tuples will not used anymore and can be exchanged with other training tuples. The above operation is repeated until all the local k of all the testing tuples are obtained. In this way, multiple unknown objects can be classified.

4. EXPERIMENTAL EVALUATION

The device we used in our experiment 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 used synthetic data generated by MATLAB for the purpose of evaluation. Details of these data sets are described in the following subsections. The outputs of CUKNN are guaranteed to be exactly the same as that of the serial KNN.

4.1. Scalability

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

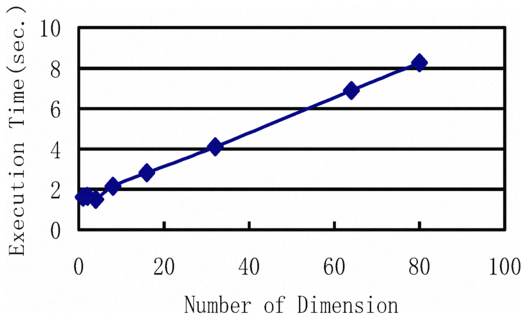

Figure 3. Execution time with varying dimension of data objects

Figure 3 shows the execution time of CUKNN when varying the dimension of data objects. Figure 4 shows the scalability of CUKNN by varying the number of data objects from 8K to 1048K. The dimension of data objects is set as 8. From the experimental results, we can see that the execution time of CUKNN increases near linearly with the number of transactions as soon as the varying factor is not too small.

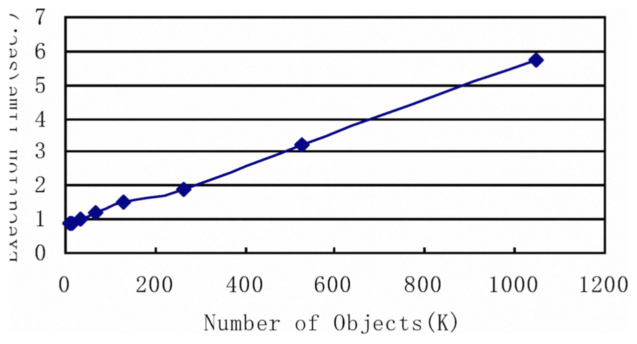

Figure 4. Execution time with varying number of data objects

From Figure 3 and 4, we can see that CUKNN is scalable in terms of the dimension of data objects, number of data objects. The execution time is approximately linear to the number of candidates in a transaction database.

Since the expense of picking out the k nearest neighbors is trivial in comparison with the distance computation both in CUKNN and the serial KNN, it is meaningless to check the change of k.

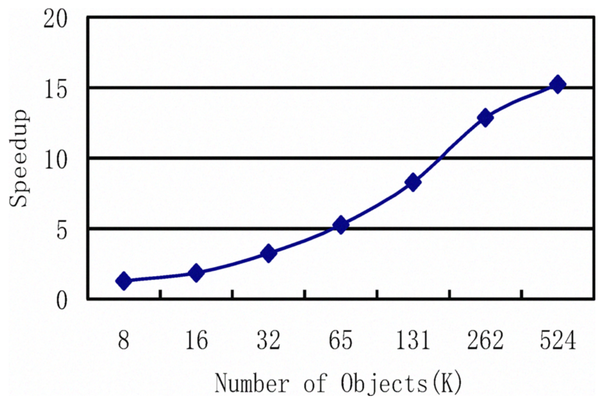

Figure 5. Speedups with varying the quantity of data objects

4.2. Speedup.

Figure 5 shows the speedups when running CUKNN on a set of database with respect to 7. We compare the execution time of the serial KNN with that of CUKNN. The I/O time is also included. The speedup increases as the number of data object increases. The best case (15.2X) happens when the dataset is 524 K. [6] achieved 400X speedup in the best case by comparing with the running time of an un-optimized serial code running on an out-of-date machine.

5. CONCLUSION

In this paper, we presented a 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 nearest-neighbors sorting. 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 data objects. CUKNN presented up to 15.2X speedup in overall execution time. The result shows that CUKNN is suitable for large-scale dataset.

ACKNOWLEDGEMENT

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

[4] B. 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.

[5] 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.

[6] 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.