J Supercomput (2013) 64:942–967
DOI 10.1007/s11227-011-0672-7

Parallel data mining techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA)

Liheng Jian · Cheng Wang · Ying Liu · Shenshen Liang · Weidong Yi · Yong Shi

Published online: 26 August 2011

Abstract Recent development in Graphics Processing Units (GPUs) has enabled inexpensive high performance computing for general-purpose applications. Compute Unified Device Architecture (CUDA) programming model provides the programmers adequate C language like APIs to better exploit the parallel power of the GPU. Data mining is widely used and has significant applications in various domains. However, current data mining toolkits cannot meet the requirement of applications with large-scale databases in terms of speed. In this paper, we propose three techniques to speedup fundamental problems in data mining algorithms on the CUDA platform: s*calable thread scheduling scheme for irregular pattern*, *parallel distributed top-k scheme*, and *parallel high dimension reduction scheme*. They play a key role in our CUDA-based implementation of three representative data mining algorithms, *CU-Apriori*, *CU-KNN*, and *CU-K-means*. These parallel implementations outperform the other state-of-the-art implementations significantly on a HP xw8600 workstation with a Tesla C1060 GPU and a *Core-quad* Intel Xeon CPU. Our results have shown that GPU + CUDA parallel architecture is feasible and promising for data mining applications.

L. Jian · Y. Liu (✉) · S. Liang · W. Yi
School of Information Science and Engineering, Graduate University of Chinese Academy of Sciences, Beijing, China
e-mail: yingliu@gucas.ac.cn

C. Wang
Agilent Technologies Co. Ltd., Beijing, China

Y. Liu · Y. Shi
Research Center on Fictitious Economy and Data Science, Chinese Academy of Sciences, Beijing, China

Y. Shi
University of Nebraska at Omaha, Omaha, USA

Keywords Parallel computing · CUDA · Data mining · Classification · Clustering · Association rules mining

1 Introduction

Data mining is to discover interesting, meaningful, and understandable patterns hidden in massive data sets [1]. It has become a hot research domain in recent years [2–10]. Important application areas are business intelligence, customer relationship management, WWW, scientific simulation, e-commerce, bioinformatics, and many more.

The size of various data sets has increased tremendously in recent years as speedups in processing and communication have greatly improved the capability for data generation and collection in all areas. It is quite common to see databases on the order of gigabytes or terabytes. A sequential data mining algorithm handling these large data sets would potentially be unable to run in-core or would take a tremendous amount of time. Therefore, users have to turn to rely on parallel and distributed computing techniques to accelerate the computation. The use of multiple processors enables the use of more memory so that a larger dataset can be handled in the main memory attached to the processors. Hence, parallel data mining are in strong demand in various domains.

In academia, many parallel data mining algorithms have been proposed on distributed-memory parallel machines and shared-memory parallel machines [6, 7, 9, 11] in the past decades. However, none of the current available successful commercial mining systems applies these parallel data mining techniques, including SAS Enterprise Miner, IBM Intelligent Miner, and SPSS Clementine. Actually, real time analysis cannot be achieved on these commercial systems either. One of the reasons that hinder the development of parallel data miners is the high cost of parallel computing systems, such as clusters, which usually are beyond the capability of small or medium business to afford.

Low cost and low power consumption requirements are the motivating powers of recent development of parallel architectures. One recent solution is the multi-core system. Examples are Intel *Core-duo* and *Core-quad* products, the Sony/Toshiba/IBM's *Cell*, etc. Although the multi-core system is low in cost and power consumption compared with traditional supercomputers, its scalability is poor since multiple cores have to be integrated onto a single chip.

Another promising solution is Graphics Processing Unit (GPU). GPU is originally a highly specialized architecture designed for graphics rendering, driven by the computer gaming industry. Nowadays, high level languages have emerged to support easy programming on GPUs. NVIDIA's GPU with CUDA (Compute Unified Device Architecture) environment provides a standard C-like interface to manipulate the GPUs [12]. GPUs with CUDA provide tremendous memory bandwidth and computing power. For example, NVIDIA's GeForce 8800 GTX can achieve a sustained memory bandwidth of 86.4 GB/s and a single-precision peak performance of 346 GFLOPS/s; NVIDIA's Tesla C1060 can achieve a bandwidth of 102 GB/s and a single-precision peak performance of 933 GFLOPS/s [13]. Low cost is another

highlight of GPU. For example, GeForce 8-series products cost only $90, and the high-end product, Tesla C1060, costs only around $1,000. One million US dollars can buy 8.3 TFLOPS/s CPU computing capability but 439.1 TFLOPS/s GPU capability [13]. Although it is low in cost, 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, there has been a trend to accelerate computational intensive applications on a GPU + CPU heterogeneous system where the GPU acts as the computation accelerator, including scientific, communication, military, business, medical and other domains. Successful examples are Finite Difference Time Domain (FDTD) [14], Computational Fluid Dynamics (CFD) [15], Magnetic Resonance Imaging (MRI) [16], Neural Network [17], Support Vector Machine [18], intrusion detection [19], etc.

Due to the great advantages of GPUs, we would like to explore the performance of parallel data mining algorithms on CUDA-enabled GPU. After careful analysis, we find that three problems are fundamental to many data mining algorithms and their performance on CUDA-enabled GPU is critical to the overall performance. Thus, in order to obtain the best performance, we propose three CUDA-based techniques for data mining parallelization: (1) *scalable threads scheduling scheme for irregular pattern*; (2) *parallel distributed top-k scheme*; and (3) *parallel high dimension reduction scheme*. We selected *Apriori* [20], *K-Nearest-Neighbor* (KNN) [21], *K-means* [22], the three widely used data mining algorithms as representatives based on how commonly they are used in industry and their potential parallelism. We implement *Apriori, KNN, K-means* by applying the above schemes, called *CU-Apriori, CU-KNN, CU-K-means*, respectively. Experiments are on an HP xw8600 workstation with a NVIDIA Tesla C1060 card and an Intel Xeon *Core-quad* 2.66 GHz CPU. By comparing the execution time of *CU-Apriori* with an efficient serial Apriori program on the *Core-quad* 2.66 GHz Intel Xeon CPU, it shows up to 6 times speedup on a real-world data set and 13.5 times on synthetic data sets. Compared with Fast-KNN [23], *CU-KNN* shows up to 8.31 times speedup on KDD-CUP 2004 quantum physics data set. Significant improvement is observed on our *CU-K-means* when comparing it with another CUDA-based implementation of *K-means* in GPUMiner.

The rest of this paper is organized as follows. Section 2 presents the background knowledge of GPU architecture and CUDA programming model. Section 3 overviews the related work. Section 4 describes the three typical algorithms briefly. In Sect. 5, we present our CUDA-based techniques for data mining parallelization on GPU with CUDA architecture. Then CUDA implementations of *Apriori, KNN, K-means* using our proposed techniques are presented in Sect. 6. Experimental results are presented in Sect. 7 and we summarize our work in Sect. 8.

2 NVIDIA's CUDA-enabled parallel computing

2.1 GPU architecture

Nowadays GPUs has evolved into a highly parallel, multithreaded, many-core processor with tremendous computational horsepower and very high memory bandwidth.

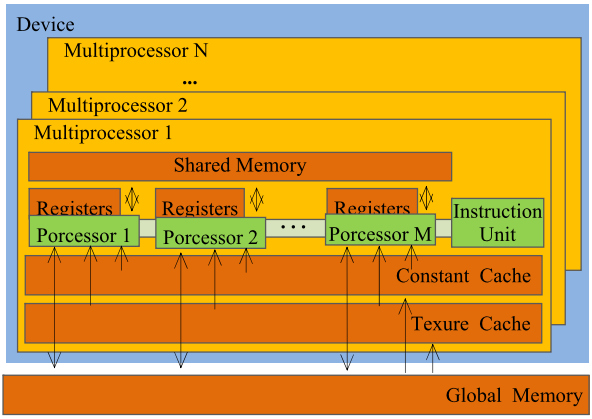

Fig. 1 A set of SIMD stream multiprocessors with memory hierarchy

In addition to rendering process, they are also suitable to general compute-intensive, highly parallel computation. NVIDIA's GPU with the CUDA programming model provides an adequate API for non-graphics applications. CPU sees 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. GeForce 8800GTX has 128 SPs and Tesla C1060 has 240 SPs. Each SM contains a fast *shared memory*, which is shared by all of its SPs as shown in Fig. 1. It also has a read-only *constant cache* and *texture cache* which is shared by all the SPs on the GPU. 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 or written by the host, and is persistent across kernel launches by the same application. Shared memory is managed explicitly by the programmers. Compared to the CPU, more transistors on the GPU are devoted to computing, so the peak floating-point capability of the GPU is an order of magnitude higher than that of the CPU, as well as the memory bandwidth due to NVIDIA's efforts on optimization.

2.2 CUDA programming model

At the software level, the 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 thread-parallel fashion. Computation is organized as a *grid* of *thread blocks* which consists of a set of threads as shown in Fig. 2. At instruction level, 32 consecutive threads in a *thread block* make up of a minimum unit of execution, which is called a *thread warp*. 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 single 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.

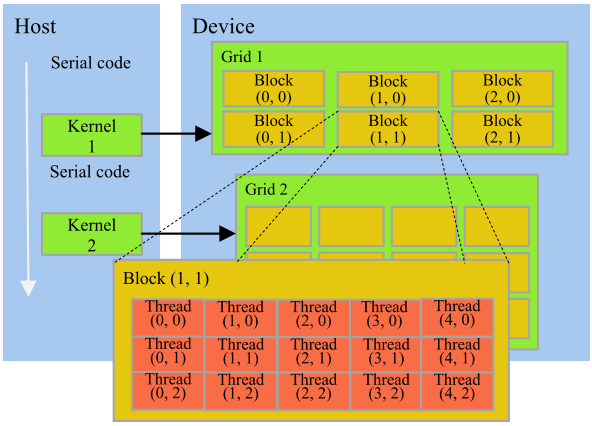

Fig. 2 Serial execution on the host and parallel execution on the device

3 Related work

3.1 Data mining

Data mining is the process to detect patterns and extract knowledge from large amount of data stored in databases, data warehouses, or other information repositories. Data mining techniques can be classified into several categories based on the nature of the problems being solved, such as association rules mining (ARM), classification, clustering, sequence mining, etc. Many algorithms have been developed in the past two decades, such as Apriori, FP-Growth [24], decision tree, Naïve Bayesian classifier, K-Nearest-Neighbor classifier, Neural Network, K-means [1–5, 8], etc.

3.2 Parallel data mining

Parallel associations rules mining has been developed on distributed-memory parallel machines and shared-memory parallel machines. Examples are Common Candidate Partitioned Database (CCPD) [25], Count Distribution (CD) [26], Intelligent Data Distribution (IDD) [27], FPM [28], PMIHP [29], etc. CCPD achieved 6.8 times speedup on a 12-node SGI machine, and CD achieved 13 times speedup on 16 nodes of a 32-node IBM SP2 distributed-memory parallel computer. However, taking into consideration the high cost and the power consumption, the performance of parallel ARM on supercomputers is not as good as expected.

Several parallel formulations of classification decision tree [30–32] have been proposed on conventional parallel machines. Most of these algorithms can be explained by the synchronous tree construction approach, partitioned tree construction approach, or Hybrid approach. SPRINT [30] parallelizes the split determining phase; however, it is not scalable in memory requirement. ScalParC [32] is scalable in both running time and memory requirement.

Examples of parallel clustering methods are MAFIA [33], parallel K-means [34], SLINK algorithm [35], Ward's minimum variance method [36], etc.

NU-MineBench [9] is the first benchmarking suite containing representative data mining algorithms. It consists of two association mining algorithms, two classifica-

tion algorithms, and four clustering algorithms. Five out of the 8 algorithms are parallelized on symmetric multi-processing computer (SMPs). Important performance characteristics of NU-MineBench are evaluated on an 8-way shared memory parallel machine using Intel VTune Performance Analyzer.

3.3 Parallel data mining on GPUs

Recently, Fang et al. [37] proposed GPUMiner, a system consisting of three components: (1) a CPU-based storage and buffer manager to handle I/O and data transfer between the CPU and the GPU; (2) a GPU-CPU co-processing parallel mining module; (3) a GPU-based mining visualization module. GPUMiner utilizes *bitmap* as the data structure since *bitmap* can enhance the efficiency of SIMD execution. Although Apriori in GPUMiner achieved significant speedups, it has the following weaknesses: (1) bitmap is a specialized data format for GPUMiner, requiring tedious preprocessing work to convert transactional database (the most widely used data format) to *bitmap*. The data preprocessing time is excluded from the measured execution time in GPUMiner. (2) Since each item takes one bit in bitmap, no matter it is present or not, the size of the database would be huge for sparse data, which may exceed the global memory on the GPU. In contrast, our *CU-Apriori* uses transactional database as the input data, avoiding the cost of preprocessing. The performance of our *CU-K-means* outperforms that of the K-means in GPUMiner.

Che et al. [38] implemented K-means on GPU using CUDA. K-means is partially paralleled, where each thread on the GPU only performs distance computation; the CPU takes over the reduction step that produces the new centroid for each cluster. In contrast, in our parallel implementation, *CU-K-means* and *CU-KNN*, the GPU not only performs the distance calculation, but also the reduction. A fast K-Nearest-Neighbor algorithm was introduced in [23]. A fast CUDA-based K-means is proposed dedicatedly for very large data sets which cannot be fit into the global memory of the GPU [39].

4 Typical data mining algorithms

In this section, we give a brief description of three widely used data mining algorithms: Apriori, K-Nearest Neighbor, K-means, which have been well parallelized on shared-memory CPU architecture, and now attract the interests of GPU researchers.

4.1 Apriori

Association rules mining (ARM) [1] is one of the most widely used techniques in data mining and has tremendous applications in business, science, and other domains. The objective of ARM is to identify frequently co-occurring itemsets in a database. It first finds all the itemsets whose occurrence are beyond a minimum support threshold, and then generates rules from the frequent itemsets based on a minimum confidence threshold.

Apriori is the most influential ARM algorithm due to its easy implementation and has been included in all the existing commercial and non-commercial data mining

Table 1a Pseudo-code of main function

Input: Database, D, minimum support threshold, *min_sup*
Output: Frequent itemsets, L

(1) $L_1 = \text{find\_frequent\_1-itemsets}(D)$;
(2) **for**$(k = 2; \mathrm{L}_{k-1} \neq \varnothing; k++)\{$
(3) $C_k = \textbf{apriori\_gen}(L_{k-1})$;
(4) **for each** transaction $t \in D\{$ //support counting
(5) $C_t = \text{subset}(C_k, t)$;
(6) **for each** candidate $c \in C_t$
(7) c.count ++;
(8) }
(9) $L_k = \{c \in C_k | c.\text{count} \geq \textit{min\_sup}\}$;
(10) }
(11) **return** $L = \mathrm{U}_k L_k$;

Table 1b Pseudo-code of sub-functions of Apriori

Procedure apriori_gen(L_{k-1}) //candidate generation
(1) **for each** itemset $l_1 \in L_{k-1}$
(2) **for each** itemset $l_2 \in L_{k-1}$
(3) if$((l_1[1] = l_2[1]) \wedge \cdots \wedge (l_1[k-1] < l_2[k-1]))$ then{
(4) $c = l_1 \triangleright\!\triangleleft l_2$;
(5) **if has_infrequent_subset**(c, L_{k-1}) then
(6) **delete** c;
(7) **else add** c **to** C_k;
(8) }
(9) **return** C_k;

Procedure has_infrequent_subset$(c; L_{k-1})$
(1) **for each** $(k-1)$-subset s **of** c
(2) **if** $s \notin L_{k-1}$ **then**
(3) **return** TURE;
(4) **return** FALSE;

software systems. Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of m distinct attributes, or called *items*. Each transaction T in the database D has a unique identifier, and contains a set of items, in the form of $\langle \text{TID}, i_1, i_2, \ldots, i_p \rangle$. An itemset with k items is called a *k-itemset*. The support of a *k-itemset* X is the fraction of the transactions in D containing X. X is a frequent itemset if X's support is greater than the user-specified minimum support threshold ε. The goal of frequent itemset mining is to find the complete set of frequent itemsets in a database. The *downward closure property* (any subset of a frequent itemset must also be frequent) is successfully introduced to cut down the number of candidates generated in each iteration. The detail of Apriori is presented in Tables 1a, 1b [1]. It performs a level-wise search. In each iteration (line 2 to 10), firstly, a set of k-candidates is generated by joining two frequent $(k-1)$-itemsets if they share a common $(k-2)$-prefix. A pruning procedure is invoked to

Table 2 Pseudo-code of K-Nearest-Neighbor classifier

```
Input: References, R, number of nearest neighbors, k
       Classes, C, Query objects, T
Output: L, Class labels of T
(1)   for each query object t_i ∈ T
(2)           for each reference r_j ∈ R
(3)               Dist_ij = calculate_distance(t_i, r_j);
(4)           D = select_k_shortest_distances();
(5)           C_i = the majority class of D;
(6)           L = U C_i;
(7)   return L
```

eliminate any candidate which contains an infrequent subset; secondly, the support of every candidate itemset is counted by scanning the database. The loop terminates when no more frequent itemsets are discovered.

The generation of candidate itemsets is computational intensive. The support counting to filter out frequent itemsets is memory bound. Both of them are suitable for parallelization since the operations on different itemsets are independent. The most challenging part is that the number of candidates or frequent itemsets is unpredictable in any iteration.

4.2 K-Nearest-Neighbor classifier (KNN)

Classification is one of the important data mining techniques whereby a model is trained on a reference data set with class labels and then used to predict the class label of unknown objects. Classification has many applications, such as credit scoring, fraud detection, network intrusion detection, churn management, etc. KNN is one of the classic classification algorithms, and has been widely used in the area of pattern recognition and text classification [1].

KNN is a method for classifying object based on its closest reference objects. Given a query (unknown) object p, a KNN classifier searches the reference data set for k objects that are closest to p, and then, p is assigned to the majority class among the k nearest neighbors. "Closeness" is usually defined as a distance metric, such as Euclidean distance, Cosine distance, etc. The pseudo-code of KNN is shown in Table 2.

The most time-consuming parts are the distance calculation component and the neighbor-selecting component. Distance calculation between a pair of objects is independent, suitable for parallelization.

4.3 K-means

Clustering is the process of grouping the data into classes or clusters so that objects within a cluster have high similarity in comparison to one another but are very dissimilar to objects in other clusters. Dissimilarities are accessed based on the attribute values describing the objects. Distance is often used as the similarity measure. Clustering has wide applications including market or customer segmentation, pattern

Table 3 Pseudo-code of K-means

```
Input: Objects, T, number of clusters, k
Output: Clusters, C
(1)   initialize k objects as the initial centroids;
(2)   repeat
(3)     for each t_i ∈ T                          // class label update
(4)       for each cluster centroid c_j ∈ C
(5)         Dist_ij = calculate_distance (t_i, c_j);
(6)         If (Dist_ij is min) then
(7)           t_i ∈ C_j;
(8)     for each cluster centroid c_j ∈ C          // centroid update
(9)       for each t_i ∈ C_j
(10)        sum up t_i
(11)      c_j = average of sum of t_i
(12)  until each c_j = c_previous
(13)  return C
```

recognition, biological studies, spatial data analysis, GIS, Web document classification, city planning, etc. [1].

K-means is the most popular clustering algorithm with no doubt. Given a set of n objects and the input parameter, k, it partitions objects into k clusters so that the intra-cluster similarity is high but the inter-cluster similarity is low. Firstly, it randomly selects k objects as centroids to represent k cluster. For each of the remaining objects, an object is assigned to the cluster to which it is most similar. Then K-means computes the new centroids for each cluster by averaging all the objects belonging to the same cluster. This process repeats until the criterion function converges. "Similarity" is also defined as a distance metric, similar to KNN, such as Euclidean distance, Cosine distance, etc. The detail of K-means is presented as Table 3 shows.

In this algorithm, class label determining of each object and centroid update are computational intensive. Class label determining can be parallelized efficiently since the pair-wise distance calculation is an independent operation. Centroid update has to wait for all the class labels of a cluster's members ready, so a reduction operation is required.

5 Parallelization techniques on CUDA-enabled platform

In order to make the best use of the architecture of the GPU and CUDA platform, we propose three schemes for data mining parallelization on CUDA-based platform.

5.1 Scalable threads scheduling scheme for irregular pattern

5.1.1 Overview of the irregular pattern problem on CUDA platform

What is regular pattern? Regular pattern refers to a problem whose size is predictable or predefined. A typical example is matrix multiplication where the size of the computation is usually determined before the computation starts. In such a problem, in

order to achieve an efficient solution, whether a task is assigned to the CPU or the GPU or the number of thread blocks is usually determined by the size of the problem before the GPU kernel starts. However, the size of a problem may change dynamically during a process, and we call such type of problems as irregular pattern problem. An irregular pattern problem is common in data mining algorithms. Unfortunately, CUDA computing model is announced to be not suitable for irregular pattern problem [12]. The method for a regular pattern problem has to be adopted to solve an irregular pattern problem, where the performance may not be optimized since some resources will be wasted during the computation on the GPU. How to minimize the waste of the resources and optimize the performance of irregular pattern problem is critical for data mining parallelization on GPU with CUDA.

5.1.2 Our solution

Scalable thread scheduling Since the problem size may change dynamically during a process, some resources on the GPU may become idle. But comparing to re-starting a GPU kernel with an updated size, launching, or exiting a thread block (including allocation or deallocation of shared memory and registers [12]) incurs trivial cost. So, we propose a scalable thread scheduling that makes a balance between the GPU resource cost and the overall performance.

First of all, estimate the upper bound of the number of threads/thread blocks, and allocate the GPU resources (including threads, blocks, registers, shared memory, etc.); secondly, if the results on some thread blocks have been determined useless, let the corresponding thread blocks quit immediately. In this way, the overall performance can be improved by sacrificing part of the computation resource.

Parallel compressing Since some of the thread blocks quit, or some threads within an active block fail to produce desired results, the output of this kernel will be sparse. Thus, a compressing method is required to save the memory cost. We propose a 2-step method as follows:

Step 1. An in-block reduction-like operation is performed, where only a fraction of the threads in a thread block participate in each iteration. For example, in the first iteration, every other thread (stride width $= 1$) in the block moves the useful element forward. The stride width doubles in the next iterations. When stride width $= 2$, every other 2 consecutive threads, such as 0 and 1, 4, and 5, etc. move the useful elements forward. This procedure, shown in Fig. 3, is similar to the *interleaved addressing reduction* in CUDA SDK [40], which is highly optimized. Finally, the elements in a given thread block are moved to a consecutive data segment.

Step 2. After step 1, data segments compressed by different thread blocks are still separated from each other. Our goal is to compress them into a single global consecutive data segment. We use the CPU to sum up the number of useful elements in each data segment and mark up its corresponding starting point in a global buffer. Next, each thread block copies its local data segment into the global buffer concurrently by the starting point provided by the CPU. Eventually, a global consecutive data segment is obtained.

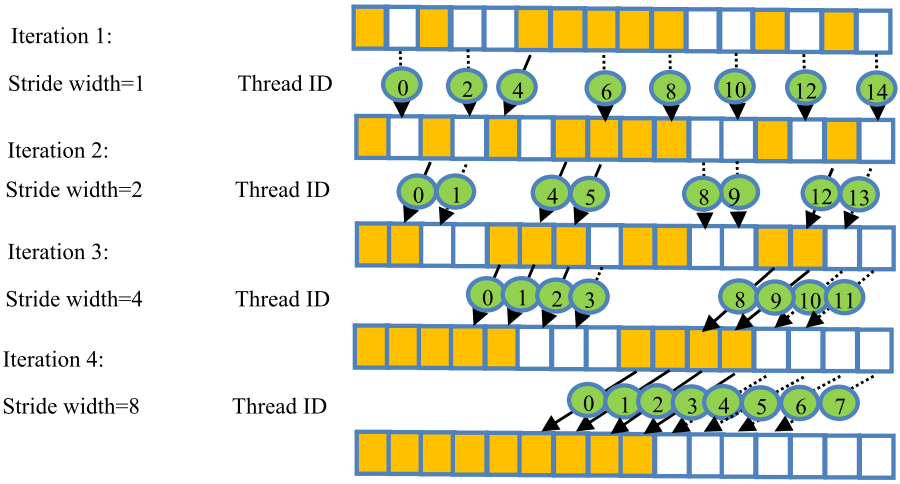

Fig. 3 Interleaved addressing compressing

Since step 1 and step 2 makes good use of both the power of the CPU and the GPU, it is an optimal solution for parallel compressing.

Any irregular pattern computation problem can be processed by applying the proposed scheme, where the scheduling method balances between the GPU resource cost and the overall performance, and the compressing method saves the space cost.

5.2 Parallel distributed top-k scheme

5.2.1 Overview of the top-k problem on CUDA platform

Top-k problem is to select the k minimum or maximum elements from a data collection. Although a global sort is not necessary, comparison between the k elements and all the other elements is necessary. It is not only common in data mining algorithms, but also fundamental in many other applications.

Insertion sort [23] has been proved to be efficient when k *is* small. Without loss of generality, let's assume the problem is to select the k minimum elements in a data collection. It sorts the first k elements in ascending order and maintains them in a queue, and then updates the queue when scanning the rest $n - k$ elements. Finally, the k minimum elements remain in the queue. The complexity of insertion sort is $O(kn)$. A CUDA-based insertion sort [23] is not efficient. Its weakness is that the on-chip shared memory assigned to a thread block may be too small to hold the queue. So, the queue with k elements has to be maintained in the global memory where huge cost of global memory access will be incurred.

5.2.2 Our solution

We propose to reduce the computation and tackle the weakness of the CUDA-based insertion sort in [23] by using multiple local sorts rather than a global sort. Our solution works as follows.

Step 1: Local sort We divide the data collection into multiple data partitions with equal size, which is small enough to fit in the shared memory and hold by some threads in a thread block. Each partition is sorted by those threads within a thread block concurrently. Each thread within those threads holds one element, compares it with every element in the data partition in the shared memory, and obtains its rank in the given partition. Then each element is written to a buffer in the shared memory according to its rank to make a sorted queue. This step is highly efficient because the sorting is in linear complexity, performed by the threads concurrently and the access to the shared memory incurs no memory conflict. The complexity of step 1 is $O(\lceil n/m \rceil)$, where n is the total number of elements and m is the number of partitions.

Step 2: Approximate top-k queue Based on the sorted queues generated in step 1, we use the head (minimum element) of each sorted queue to form a smaller data collection with m elements. Then select k minimum elements from this data collection, and sort the k elements. This sorted queue is called as an approximate *top-k* queue. In this step, if part of the shared memory of a thread block is large enough to hold a sorted *top-k* queue, some threads sorts the m elements by insertion sort, which is similar to [23]; otherwise, divide the sorted *top-k* queue into multiple sub-queues, each of which can fit in the shared memory, assign some threads in a block with a sub-queue, and iteratively produce each sub-queue by insertion sort [23] on m elements. Eventually, a sorted queue with k elements is stored in the global memory of the GPU. The complexity of step 2 is the same as that of insertion sort for m elements, but is highly efficient since huge cost of global memory access in [23] is avoided.

Step 3: Global top-k queue This step is the core of this scheme. First of all, let us introduce *exclusive property*: If element a is less than element b which belongs to a sorted queue q, any element greater than b in q cannot be less than a. Selecting the global top k minimum elements on the GPU applies this property. Let us use the example in Fig. 4 to illustrate how to select the global top k minimum elements. Up to now, the approximate top-k queue, A, maintained in the global memory (by step 2), and k local sorted queues, whose heads are in A and sorted not only within a queue but also among the queues according to their heads (by step 1), are hold by a thread. The head of $A, 2$, is the global minimum element, so pop 2 out of A and move it to G, the global top-k queue. Next, the thread compares the elements in its local queue with the current head of $A, 4$. Only one queue, q_1, has element less than 4, which is headed by 2. Exam the elements in q_1, where any one less than 4 will be popped and moved to G. Then pop 4 out of A and move it to G. So far, 8 turns to be the head of A. What we need to do is to make the thread compare the elements in the current and subsequent local queues with the current head of A and pop all the elements less than 8 until on more smaller elements are found. So, q_1 and q_2 are examined, and 5, 6, 7 are popped and moved to G. This process repeats until k global minimum elements have been identified. In the worst case, only $k-1$ local sorted queue are compared, the complexity of step 3 is $O(k * \lceil n/m \rceil)$, where k or $\lceil n/m \rceil$ is usually much smaller than n and m.

By using *exclusive property*, the computation is significantly reduced. Since only one queue and the head of the approximate *top-k* queue are held in the shared memory

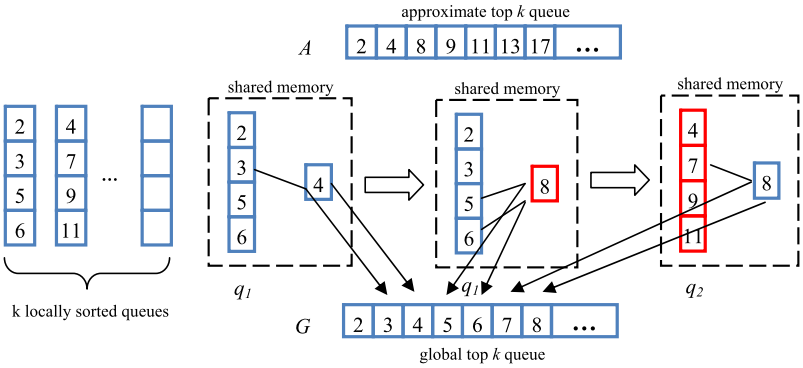

Fig. 4 Selecting the global top k minimum elements

at any moment, the memory utilization is efficient. In addition, loading a queue into the shared memory of a thread is managed in a coalesced manner. Thus, our selection of global top-k queue is optimized well.

5.3 Parallel high dimension reduction scheme

5.3.1 Overview of high dimension reduction problem on CUDA platform

In data mining applications, data often have very high dimensionality. For example, a record in text mining may consist of hundreds of attributes, exceeding the size of the shared memory allocated to each thread block on the GPU. In such case, the record has to be broken into multiple sub-records to fit in the shared memory. Imagine that we would like to perform a sum operation on a huge number of high-dimensional records, which is a typical reduction problem, is breaking them down into many sub-records a feasible solution? The answer is definitely "no", because the cost for manipulating the records and the temporal results will be high. How to perform reduction on high-dimensional data on CUDA-enabled platform is a challenging problem. As far as our knowledge goes, this problem has not been well processed yet.

5.3.2 Our solution

By observing that different attributes (dimensions) in a record are independent, we see the same attribute (dimension) on all records as a vector. Each thread block only takes care of one distinct attribute of all the records. Rather than perform reduction on the high-dimensional data, we perform one-dimensional reduction on each attribute, as shown in Fig. 5. Since such reduction operation has been well exploited in CUDA SDK [40], we choose *sequential addressing reduction* for the one-dimensional reduction.

This scheme aims at maximizing the thread parallelism, and cutting down the cost for manipulating the shared memory.

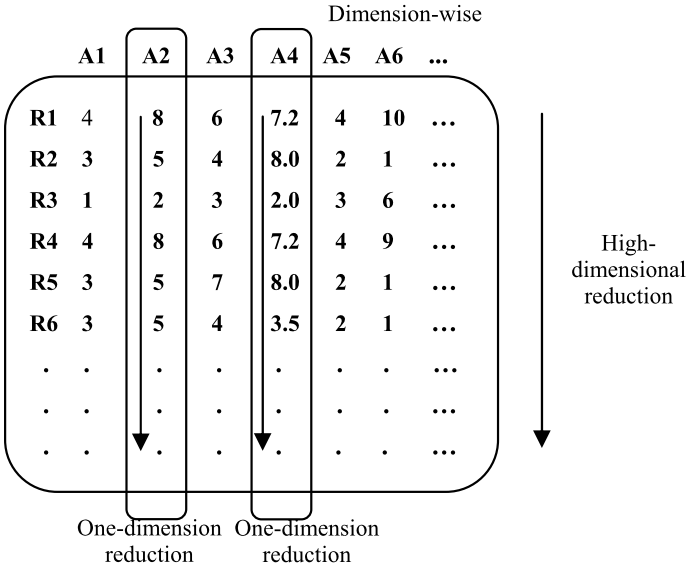

Fig. 5 High-dimensional reduction vs. one-dimensional reduction

6 Algorithm implementation

In this section, we present the detailed CUDA implementations of three typical data mining algorithms, *CU-Apriori*, *CU-KNN*, *CU-K-means*, respectively; especially, we present how our proposed schemes are applied to these algorithms.

6.1 CU-Apriori

According to the performance analysis result from Intel VTune, we observed that two cores, *candidate generation* and *support counting,* take most of the computation of Apriori. So, we parallelize the two kernels in the following subsections.

6.1.1 Candidate generation

Candidate generation procedure joins two frequent $(k-1)$-itemsets and prunes the unpromising k-candidates. Since the task of joining two itemsets is independent between different threads, it is suitable for parallelization. Since a 2D thread schedule may incur half of the threads idle, so we adopt our *scalable threads scheduling scheme for irregular pattern*. As illustrated in Fig. 6, in this kernel, the 2D thread grid is divided into multiple 2D-blocks, and each block consists of multiple threads. Each thread is assigned with two frequent $(k-1)$-itemsets, compares the two $(k-1)$-itemsets to make sure that they share the common $(k-2)$-prefix, and then generate a k-candidate itemset. Note, in Fig. 6, the thread blocks under the red separator quit from the execution immediately after the grid is launched through control of indices of thread blocks, so duplicate work is avoided. The cost of incurring such blocks is negligible.

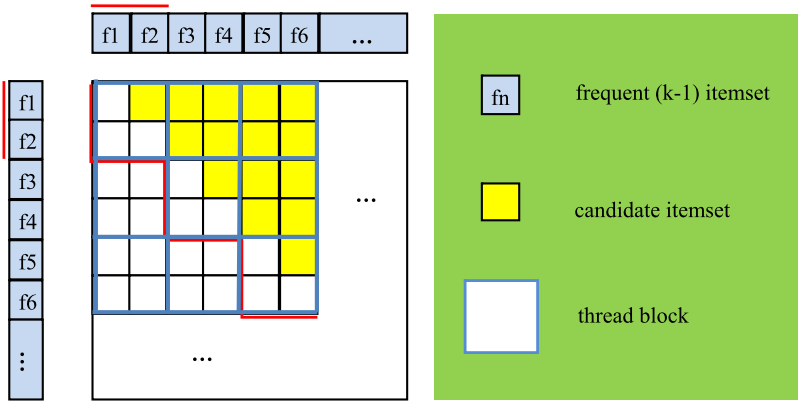

Fig. 6 Thread blocks scheduling for candidate generation

Table 4a Serial CPU-based code

```
(1)  unsigned int done = 0;
(2)  for (k = 0; k < itemset_len − 1; k++) {
(3)      if(x_itemset[k] == y_itemset[k]) contimue;
(4)      else
(5)          break;
(6)  }
(7)  if(k == itemset_len − 1) done = 1;
```

Table 4b CUDA Code with no branch divergence

```
(1)  unsigned int mach = 0;
(2)  unsigned int done = 0;
(3)  for (k = 0; k < itemset_len − 1; k++) {
(4)      if(x_itemset[k] == y_itemset[k]) mach++;
(5)  }
(6)  if(mach == itemset_len − 1) done = 1;
```

When a thread is checking if two frequent $(k-1)$-itemsets are eligible to be joined or not, branch divergence may happen, because some itemsets share a common prefix, but the others may not. In order to maximize the runtime concurrency of the threads, we adjust the operation as shown in Table 4b, where the *else* path is removed so that all the threads must follow the same path at any given moment. Comparing with operation shown in Table 4a, although some threads may be idle for one clock cycle when $x\_itemset[k] \neq y\_itemset[k]$ for a given k, branch divergence is eliminated, so our implementation is more efficient. In this stage, itemsets are read in a coalescing manner and the shared memory is used to store them, making the comparing operation work efficiently.

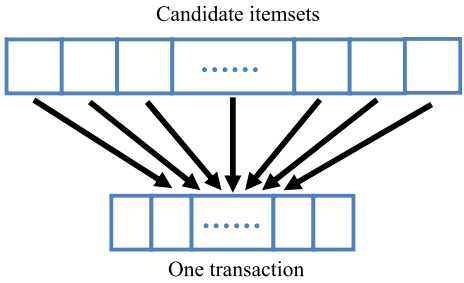

Fig. 7 Data parallelization of support counting

6.1.2 Support counting

Support counting procedure records the number of occurrence of a candidate itemset by scanning the transaction database. Since the counting for each candidate is independent with others, it is suitable for parallelization. The problem decomposition of this kernel is presented in Fig. 7, where each thread is assigned with one candidate itemset and counts its support by scanning the transactions simultaneously.

To take advantage of the shared memory, transactions are loaded into the shared memory and shared by all threads within a thread block. To load data in a memory coalescing manner, transactions are divided into fixed-sized segments and fetched one by one through the cooperation of threads. To utilize the concurrency of memory pipeline and computing pipeline of the GPU, prefetching is applied so that the next memory read latency is hidden by the current computation. In addition, similar to the candidate generation kernel, the divergence problem is also avoided by adopting the same code in Table 4b.

6.1.3 Output compressing

Since the output of candidate generation procedure is sparse, and that of the support counting as well, a compressing method is needed to save up the space. Our *parallel compressing* method is adopted here. In candidate generation procedure, indices are used to denote the frequent $(k-1)$-itemsets, and we perform the compressing on the indices rather than on the k-candidates. Eventually, the candidate itemsets are moved into a consecutive segment according to their indices. In support counting procedure, if the support of an itemset is over the specified threshold, ε, it is determined to be frequent. So, we only compress itemsets whose support is no less than the threshold.

6.2 CU-KNN

According to the performance analysis result from Intel VTune, we observed that two cores, *distance calculation* and *selection of k nearest neighbors,* take most of the computation of KNN. We present the CUDA implementation in the following subsections. In this algorithm, GPU takes care of all the computation, with no data transfer between host memory and device memory during the classification.

6.2.1 Distance calculation

Distance calculation can be fully parallelized since pair-wise distance calculation is independent. This property makes KNN perfectly suitable for a GPU parallel implementation. The goal of this kernel is to maximize the concurrency of the distance calculation invoked by different threads and minimize the global memory access.

This kernel is parallelized in a 2D data-parallel fashion. We implement it by referring the *matrix multiplication* provided in the CUDA SDK [40]. To address the multi-dimension characteristic of objects in this application, we divide each object into several partitions in terms of dimension (attribute) with a predefined size, and perform the distance calculation in an iterative way. One attribute partition of query objects and the corresponding attribute partition of reference objects are loaded from the global memory to the shared memory per time through a coalesced read by the cooperation of the threads, and then a local computation is made to produce a local sum. The final value of distance is obtained by accumulating these local sums.

6.2.2 Selection of k nearest neighbors

The selection of k nearest neighbors of a query object is essentially to find the k shortest distances, which is a typical *top-k* problem. So, we implement it by our *parallel distributed top-k scheme*.

Since selecting the k nearest neighbors for different query objects is independent, we parallelize it by assigning some threads in step 1&2 and each thread in step 3 with a task of selecting the k shortest distances for a single query object.

6.3 CU-K-means

According to the performance analysis result from Intel VTune, we observed that three cores, *cluster label update, centroid update,* and *centroid movement detection*, take most of the computation of K-means. We present the parallelization of the three kernels in the following subsections. Similar to *CU-KNN*, almost all the calculations are performed on the GPU. The flow of *CU-K-means* is presented in Table 5.

Table 5 Pseudo-code of CU-K-means

```
(1)  Transfer input data and initialized centroids to the GPU
     memory;
(2)  while (centroid_moving) {
(3)       label-updating ( );
(4)       global-reduction ( );
(5)       centroids-movement-detection ( );
(6)       Calculate global_squared_error on the host;
(7)       if (global_squared_error < threshold) centroid_moving
          = false;
(8)  }
(9)  Transfer the clustering results to the main memory.
```

6.3.1 Cluster label update

In this kernel, each thread performs the distance calculation of an object to all the centroids, and selects the nearest centroid. Each object is assigned to the cluster whose centroid is closest to it. Attribute partitions of the objects are loaded into the shared memory in a coalesced manner, and the centroids as well, so the bandwidth between the global memory and the shared memory is utilized efficiently.

6.3.2 Centroid update

In this kernel, each new centroid is obtained by averaging the attribute values of all the records belonging to the common cluster. This can be seen as a high-dimensional reduction problem. Thus, we implement it using our *parallel high dimension reduction scheme*.

6.3.3 Centroid movement detection

In this kernel, we need to detect if the new centroids move far away from the centroids in the last iteration. That is, if the *square error* between the new centroids and the old ones is small enough (below a user specified threshold), we can claim that there is no movement happening. In order to adopt *parallel high dimension reduction scheme*, preprocessing the new and old centroids is required. Firstly, we calculate the square of the difference between every attribute of the new and old centroids, called centroid difference matrix; secondly, perform the *parallel high dimension reduction scheme* on the centroid difference matrix, and the output is a record whose size is $\lceil k * dimension\_of\_object/threads\_per\_block \rceil$; thirdly, since the attributes of the record is small, this record is transferred to the main memory, and summed up to get *global_squared_error*. The program is terminated when the criterion is met. The cost of the data transfer between the main memory and the global memory is negligible.

7 Experimental results

In this section, we present the performance evaluation of *CU-Apriori*, *CU-KNN* and *CU-K-means*.

7.1 Experimental setup

The device we used in our experiment is a NVIDIA's Tesla C1060 card, which is a dedicated general-purpose computing GPU with 240 1.30 GHz SPs and 4 GB 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 *Core-quad* 2.66 GHz Intel Xeon CPU and 4 GB main memory, running the Red Hat Enterprise Linux WS 4.7 operating system. The traditional program of CPU architecture is compiled by *gcc 3.4.6*, which is also invoked by the CUDA compiler *nvcc* to compile the CPU part code in the CUDA programs. To obtain a reliable measure, execution time of each experiment was measured as the average of 10 runs.

7.2 CU-Apriori results

In order to give a fair comparison, we used the serial Apriori in NU-MineBench, which strictly follows the idea in [20]. We used both synthetic data and a real-world data set for the purpose of evaluation.

We compared the total execution time of *CU-Apriori* with that of the serial *Apriori*. The I/O time is also included. The execution time of *CU-Apriori* includes both the CPU part and the GPU part. The outputs of *CU-Apriori* are guaranteed to be exactly the same as that of the serial Apriori.

7.2.1 Synthetic data

IBM Quest Market-Basket Synthetic Data Generator [41] is well known in association rule mining area, which has been adopted by many researchers. We used two transaction data sets generated by it. The name of a data set gives a brief description of what is contained, where 'D' indicates the number of transactions, 'T' indicates average items in transactions, 'I' indicates the number of different items (per 1,000), 'P' indicates average length of maximal pattern.

Figures 8a, 8b show the execution times when the two programs run on D50K.T10.I5.P8 and D100K.T10.I5.P8, respectively. The speedup increases as the support threshold decreases. The lower the threshold, the more candidates, that is, the more threads, which makes all the stream processors (SPs) of the GPU fully utilized. The best case (13.5 times) is observed with D50K.T10.I5.P8 when the support threshold is 0.85%.

7.2.2 Real-world data

We also evaluated *CU-Apriori* on a real-world data set, Linux source code (version 2.6.18) [42]. This set of data was used to find the frequent programming rules in programming pattern mining. The source code is converted into a transactional database, where unique IDs are assigned to program elements, including functions, variables, arithmetic operators, etc. Each transaction corresponds to a function, consisting of a set of program elements, which are viewed as items. There are totally 61,792 transactions and 7,598 distinct items in the data set after data cleaning. The maximum transaction length is 160.

The corresponding results are reported in Fig. 9. When the support threshold is lower than 2%, *CU-Apriori* consumes less time than the serial Apriori. The best case (5.86 times speedup) is observed at support 0.8% since the workload is the largest. The worst case (0.52 times speedup) happens at support 6%, when only three SMs are used, thus a great deal of global memory access latency is not hidden. Therefore, as the computation workload increases, more speedups are observed.

7.2.3 Efficiency snapshot of CUDA programs

In the following experiments, we investigated the relationship between the GPU's workload and speedup. GPU achieves the best efficiency when the number of concurrent threads is much more than the number of SPs, so that all the SPs are driven to

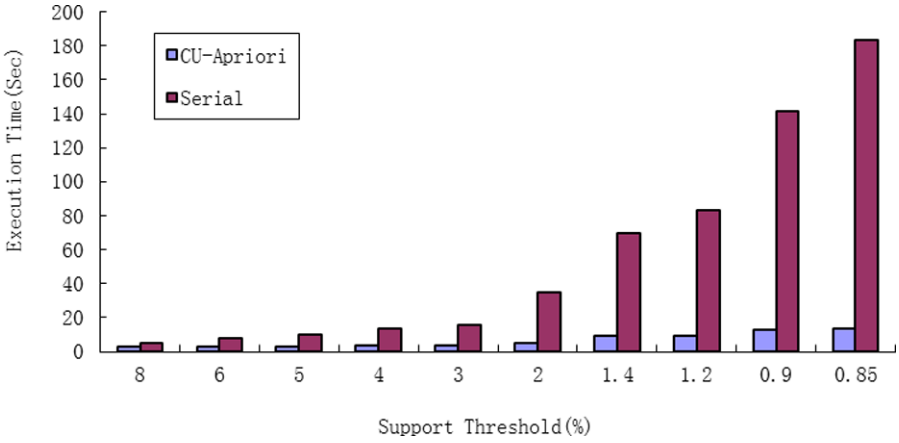

Fig. 8a Execution time with varying minimum support threshold on D50K.T10.I5.P8

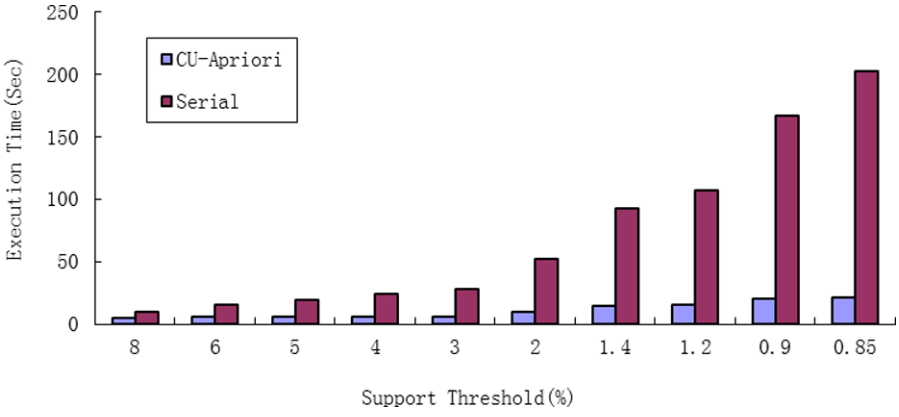

Fig. 8b Execution time with varying minimum support threshold on D100K.T10.I5.P8

run at full speed without idling and the global memory access latency can be hidden. Since one pair of frequent itemsets or a candidate itemset is assigned to a thread, the more itemsets, the more concurrent threads are launched.

Firstly, we varied the support threshold on real-world data. As shown in Table 6, the execution time of *CU-Apriori* increases relatively slow comparing with the number of frequent itemsets, while the time of the serial Apriori increases significantly. When the threshold is high, the number of candidate itemsets is small, so the stream processors (SPs) of the GPU are not fully utilized. As the number of candidate itemsets increases, more and more CUDA blocks and threads are launched, so the GPU has more workload to make the stream processors run at full speed.

Secondly, we show the detail of one execution of *CU-Apriori*. Table 7 reports the speedups in all iterations when running on synthetic data D50K.T10.I5.P8 with support 0.85%. Small speedups are observed at the beginning when the volume of

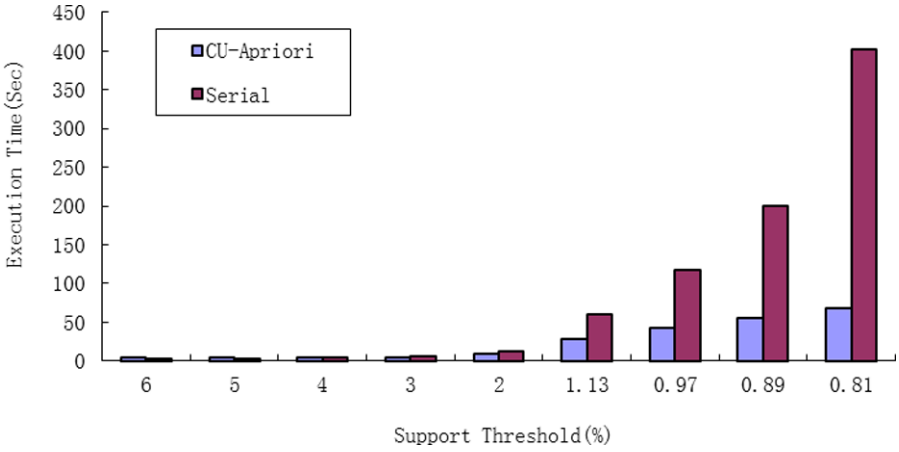

Fig. 9 Execution time with varying minimum support threshold

Table 6 Experimental results when varying the support threshold on the real-world data

Minimum support (%)	CU-Apriori (sec.)	Serial Apriori (sec.)	# of frequent itemsets
6.00	4.10	2.13	753
5.00	4.10	2.55	1,107
4.00	4.97	3.75	2,032
3.00	4.98	5.36	4,101
2.00	9.01	12.02	12,360
1.45	16.69	26.54	27,884
1.29	20.95	38.01	36,897
1.13	28.50	59.78	51,945
0.97	42.06	117.52	80,490
0.89	55.17	199.49	102,146
0.80	68.69	402.84	127,421

frequent itemsets is not big enough. As the number of frequent itemsets increases, the best speedup (22.66 times) is obtained when 8-frequent itemsets are being joined.

We did not compare the execution time of *CU-Apriori* with that of GPUMiner. Although Apriori in GPUMiner achieved excellent efficiency, it utilizes *bitmap* as the data structure whose size would be huge for sparse data, and may exceed the global memory on the GPU. What we use is the most widely used data structure, transactional database. Tedious pre-processing work is needed in order to convert transactional database to bitmap. However, the pre-processing time is excluded from the measured execution time in GPUMiner.

7.3 CU-KNN results

Since Garcia et al. [23] claimed that their Fast-KNN using GPU was up to 400 times faster compared to a brute force CPU-based implementation, we would like to make

Table 7 Comparison of execution time in different iterations

Pattern length (# Iteration)	CU-Apriori (sec.)	Serial Apriori (sec.)	# of frequent itemsets
1	0.86	0.02	26
2	0.19	0.29	325
3	0.24	1.05	2,159
4	0.57	3.21	7,871
5	0.97	10.30	17,858
6	1.87	26.34	26,949
7	2.29	47.66	30,254
8	2.29	51.98	26,511
9	1.48	26.64	18,602
10	0.82	9.35	10,697
11	0.31	3.13	5,057
12	0.16	0.71	1,804
13	0.08	0.12	405
14	0.07	0.03	47

a comparison between our CU-KNN with their Fast-KNN. The source code of the Fast-KNN using GPU is downloadable at http://www.i3s.unice.fr/~creative/KNN/.

We use KDD-CUP 2004 quantum physics data set [43] for our evaluation. This data set is used to predict the classification of the particles in high energy collider experiments of quantum physics. It stores the physical features and the class label of each particle. The particles are converted into textual records, where unique IDs are assigned. In order to make full use of the thread warps (16 threads per warp), we use 32,768 (2^{15}) records out of 50,000 records. The number of attributes (dimensions) of each record is 65. Because query objects are transferred and processed in batches both in *CU-KNN* and in Fast-KNN, I/O time for query objects is included in the comparison.

We evaluate the performance of these two CUDA-based KNN by varying the number of query objects and k. As shown in Fig. 10a, 10b *CU-KNN* outperforms Fast-KNN in any case. As the number of query objects increases, the time of *CU-KNN* is increasing linearly. In contrast, Fast-KNN consumes much more time. Observed from the experiments, as k increases, *CU-KNN* runs faster than Fast-KNN. The best case is observed at 2,000 query objects with 8.31 times when k is 114. These results indicate that our *CU-KNN* is more suitable to solve large-scale real applications than Fast-KNN.

7.4 CU-K-means results

Since Che et al. [38] showed that their CUDA-based K-means obtained up to 35 times speedup over a four-threaded CPU based counterpart, and Fang et al. [37] claimed that K-means in GPUMiner was up to 5 times faster than [38], we would like to make a comparison between our *CU-K-means* with K-meansBitmap in GPUMiner. The source code of GPUMiner is downloadable at http://code.google.com/p/gpuminer/.

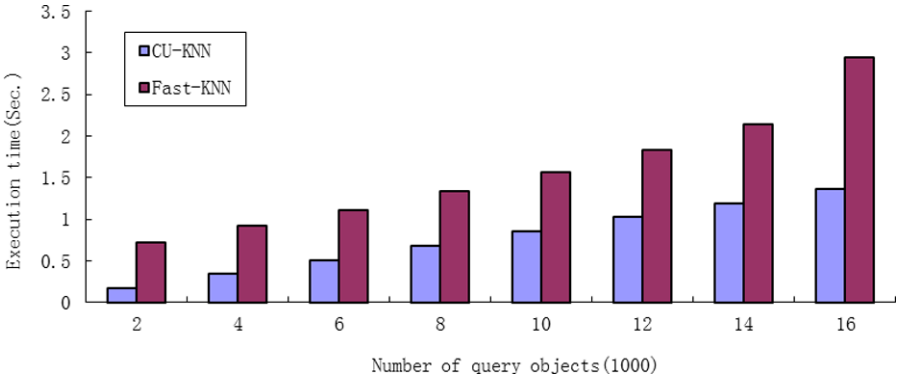

Fig. 10a Execution time with varying number of query objects when k is 64

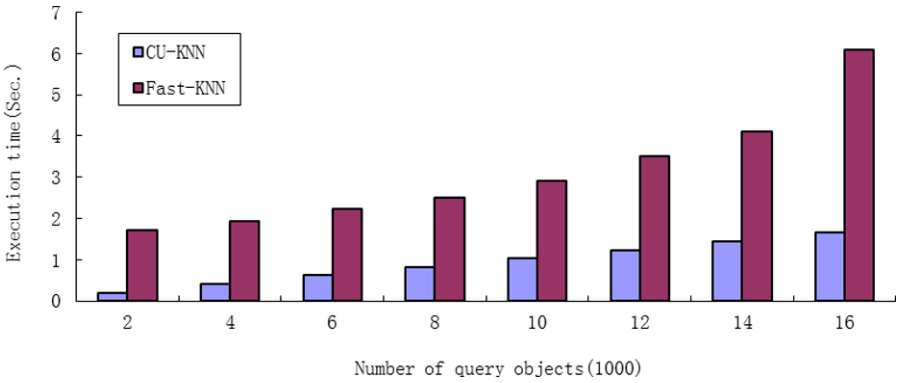

Fig. 10b Execution time with varying number of query objects when k is 114

We use KDD-CUP 1999 intrusion detection dataset [44], where each object contains 41 attributes. Each attribute is a floating-point number, and quite a lot of attributes have a value of 0.0, making the dataset a sparse matrix. To be consistent with the studies in [37], the number of clusters in our experiments was set at 24, and the convergence threshold was set at 0.00001.

When the number of dimension in an object increases, our proposed *thread scheduling scheme* has a chance to maximize the parallelism of the execution to take advantage of the GPU architecture. Figure 11 shows that *CU-K-means* behaves better on multi-dimensional data objects, which takes about 2/3 time of K-meansBitmap on the two larger datasets, and obtains 5 times speedup on the small dataset with 50 K objects. It also shows that *CU-K-means* is scalable in terms of the number of objects, whereas K-meansBitmap does not have such a characteristic.

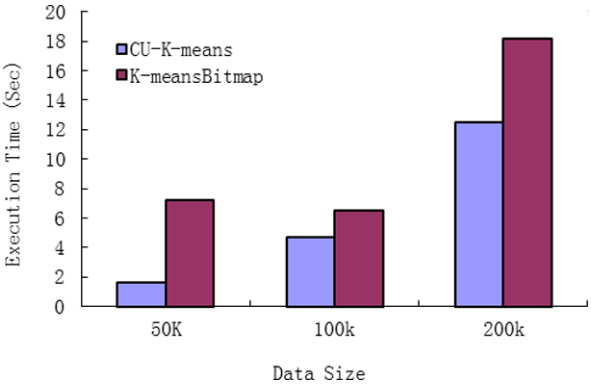

Fig. 11 Execution time with varying number of objects for multi-dimensional data

8 Conclusion

In this paper, we have focused on parallelization of data mining applications on the GPU with CUDA architecture. To exploit the new parallel platform for data mining, we firstly proposed three CUDA-based parallel techniques: (1) *scalable threads scheduling scheme for irregular pattern*, addressing irregular pattern computing problem; (2) *parallel distributed top-k scheme*, selecting top-*k* values in a parallel fashion; (3) *parallel high dimension reduction scheme,* performing high dimensional data reduction. Then we implemented three typical data mining algorithms using the above techniques on CUDA platform, *CU-Apriori*, *CU-KNN*, and *CU-K-means*. With the support of our proposed techniques, CUDA implementations of these algorithms work efficiently. Our experiments showed that *CU-Apriori* shows up to 6 times speedup over an efficient serial Apriori program on a real-world data and 13.5 times on synthetic data. Compared with Fast-KNN, *CU-KNN* shows up to 8.31 times on KDD-CUP 2004 quantum physics data set. Significant improvement of *CU-K-means* is also observed when comparing with GPUMiner.

From the above exploration, we believe that the GPU with CUDA parallel computing architecture will provide compelling benefits for data mining applications. In addition, its superior floating-point computation capability and low cost will definitely appeal to medium-sized business and individuals. Applications that used to rely on a cluster or a supercomputer to process will be solved on a desktop.

Acknowledgements This project is partially supported by grants from the Natural Science Foundation of China #70621001/70921061 and #70531040, 2009 NVIDIA's Professor Partnership, the President Grant of Graduate University of Chinese Academy of Sciences #085102 GNOO, #085102 HNOO, and the Chinese Academy of Sciences' grant for the Overseas Collaboration Group.

References

1. Kamber M, Han J (2005) Data mining: concepts and techniques, 2nd edn. Morgan Kaufmann, San Mateo
2. Peng Y, Kou G, Shi Y, Chen ZX (2008) A descriptive framework for the field of data mining and knowledge discovery. Int J Inf Technol Decis Mak 7(4):639–682
3. Olson D, Shi Y (2007) Introduction to business data mining. McGraw-Hill/Irwin, New York

4. Zhou L, Lai KK, Yen J (2009) Credit scoring models with AUC maximization based on the weighted SVM. Int J Inf Technol Decis Mak 8(4):677–696
5. Zhang Q, Segal RS (2008) Web mining: a survey of current research, techniques, and software. Int J Inf Technol Decis Mak 7(4):683–720
6. Zaki MJ (1999) Parallel and distributed association mining: a survey. IEEE Concurr 7(4):4–25, Special issue on Parallel Mechanisms for Data Mining
7. Srivastava A, Han E, Kumar V, Singh V (1999) Parallel formulation of decision-tree classification algorithms. Data Min Knowl Discov 3(3):237–261
8. Gaber MM, Yu PS (2006) Detection and classification of changes in evolving data streams. Int J Inf Technol Decis Mak 5(4):659–670
9. Liu Y, Pisharath J, Liao WK, Memik G, Choudhary A, Dubey P (2004) Performance evaluation and characterization of scalable data mining algorithms. In: 16th IASTED international conference on parallel and distributed computing and systems (PDCS). MIT, Cambridge, pp 620–625
10. Dehuri S, Mall R (2009) Parallel processing of olap queries using a cluster of workstations. Int J Inf Technol Decis Mak 6(2):279–299
11. Ergu D, Kou G, Peng Y, Shi Y, Shi Y (2011) The analytic hierarchy process: task scheduling and resource allocation in cloud computing environment. J Supercomput. doi:10.1007/s11227-011-0625-1
12. NVIDIA (2008) CUDA programming guide 2.1. http://www.nvidia.com/object/cuda_develop.html
13. Tesla (2009) C1060 computing processor. http://www.nvidia.com/object/product_tesla_c1060_us.html
14. Balevic A, Rockstroh L, Li W et al (2008) Acceleration of a Finite-Difference Time-Domain method with general purpose GPUs (GPGPUs). In: Proc of international conference on computer and information technology, vol 1–2, pp 291–294
15. Cohen JM, Molemaker MJ (2009) A fast double precision CFD code using CUDA. In: 21st International conference on parallel computational fluid dynamics
16. Jeong WK, Fletcher PT, Tao R et al (2007) Interactive visualization of volumetric white matter connectivity. IEEE Trans Vis Comput Graph 3(6):1480–1487
17. Kavinguy B (2008) A neural network on GPU. http://www.codeproject.com/KB/graphics/GPUNN.aspx
18. Catanzaro B, Sundaram N, Keutzer K (2008) Fast support vector machine training and classification on graphics processors. In: ICML '08: proceedings of the 25th international conference on machine learning, pp 104–111
19. Vasiliadis G, Antonatos S, Polychronakis M et al (2008) Gnort: high performance network intrusion detection using graphics processors. Recent Adv Intrusion Detect 5230:116–134
20. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules. In: Proc of international conference on very large data bases, pp 487–499
21. Fix E, Hodges JL (1951) Discriminatory analysis, non-parametric discrimination: consistency properties. Technical Report 21-49-004(4), USAF School of Aviation Medicine, Randolph Field, Texas
22. Lloyd SP (1982) Least squares quantization in PCM. IEEE Trans Inf Theory 28(2):129–137 (Original version: Technical Report, Bell Labs, 1957)
23. Garcia V, Debreuve E, Barlaud M (2008) Fast k nearest neighbor search using GPU. In: IEEE conference on computer vision and patter recognition workshops, vols 1–3, pp 1107–1112
24. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Proc of international conference on management of data, pp 1–12
25. Zaki MJ, Ogihara M, Parthasarathy S, Li W (1996) Parallel data mining for association rules on shared-memory multi-processors. In: Proc of supercomputing, p 43
26. Agrawal R, Shafer C (1996) Parallel mining of association rules. IEEE Trans Knowl Data Eng 8(6):962–969
27. Han EH, Karypis G, Kumar V (2000) Scalable parallel data mining for association rules. IEEE Trans Knowl Data Eng 12(3):337–352
28. Cheung DW, Xiao YQ (1999) Effect of data distribution in parallel mining of associations. Data Min Knowl Discov 3(3):291–314
29. Holt JD, Chung SM (2007) Parallel mining of association rules from text databases. J Supercomput 39(3):273–299
30. Shafer J, Agrawal R, Mehta M (1996) SPRINT: a scalable parallel classifier for data mining. In: Proc of international conference on very large data bases, pp 544–555
31. Zaki MJ, Ho CT, Agrawal R (1999) Scalable parallel classification for data mining on shared-memory multiprocessors. In: IEEE international conference on data engineering, pp 198–205

32. Joshi MV, Karypis G, Kumar V (1998) ScalParC: a new scalable and efficient parallel classification algorithm for mining large datasets. In: Proc of international parallel processing symposium, pp 573–579
33. Nagesh HS, Choudhary A, Goil S (2000) A scalable parallel subspace clustering algorithm for massive data sets. In: Proc of international conference on parallel processing, pp 477–484
34. Forman G, Zhang B (2000) Linear speed-up for a parallel non-approximate recasting of center-based clustering algorithms, including K-Means, K-Harmonic Means, and EM. In: Proc ACM SIGKDD workshop on distributed and parallel knowledge discovery (KDD'00), Boston, MA
35. Sibson R (1973) SLINK: An optimally efficient algorithm for the single link cluster method. Comput J 16(1):30–34
36. Ward JH (1963) Hierarchical grouping to optimize an objective function. J Am Stat Assoc 58(301):236–244
37. Fang WB, Lau KK, Lu M, Xiao XY et al (2008) Parallel data mining on graphics processors. Technical Report HKUST-CS08-07. http://code.google.com/p/gpuminer/
38. Che S, Boyer M, Meng JY et al (2008) A performance study of general purpose applications on graphics processors using CUDA. J Parallel Distrib Comput 68(10):1370–1380
39. Wu R, Zhang B, Hsu MC (2009) Clustering billions of data points using GPUs. In: UCHPC-MAW'09, pp 1–5
40. CUDA SDK 3.2 (2010) http://developer.nvidia.com/object/cuda_3_2_downloads.html
41. IBM synthetic data generator (2011) http://www.cs.loyola.edu/~cgiannel/assoc_gen.html
42. The Linux Kernel Archives (2007) http://www.kernel.org/1480-1487
43. KDD Cup 2004 Data (2011) http://kodiak.cs.cornell.edu/kddcup/datasets.html
44. KDD Cup 1999 Data (2011) http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html