Parallel Data Mining on Graphics Processors

Wenbin Fang, Ka Keung Lau, Mian Lu, Xiangye Xiao, Chi Kit Lam, Philip Yang Yang, Bingsheng He[1], Qiong Luo, Pedro V. Sander, and Ke Yang[2]
Department of Computer Science and Engineering, Hong Kong University of Science and Technology
[1] Microsoft Research Asia, [2]Microsoft China Co., Ltd
wenbin@cse.ust.hk

ABSTRACT

We introduce GPUMiner, a novel parallel data mining system that utilizes new-generation graphics processing units (GPUs). Our system relies on the massively multi-threaded SIMD (Single Instruction, Multiple-Data) architecture provided by GPUs. As special-purpose co-processors, these processors are highly optimized for graphics rendering and rely on the CPU for data input/output as well as complex program control. Therefore, we design GPUMiner to consist of the following 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, and (3) a GPU-based mining visualization module. We design the GPU-CPU co-processing scheme in mining depending on the complexity and inherent parallelism of individual mining algorithms. We provide the visualization module to facilitate users to observe and interact with the mining process online. We have implemented the *k-means* clustering and the *Apriori* frequent pattern mining algorithms in GPUMiner. Our preliminary results have shown significant speedups over state-of-the-art CPU implementations on a PC with a G80 GPU and a quad-core CPU. We will demonstrate the mining process through our visualization module. Code and documentation of GPUMiner are available at http://code.google.com/p/gpuminer/.

1. INTRODUCTION

For the past decade, various data mining techniques have been developed to discover patterns, clusters, and classifications from various kinds of data [18]. While many algorithms focus on the effectiveness of mining, other work aims at performance improvement. Utilizing parallel architectures has been a viable means to improving data mining performance [39]. With the emergence of new-generation graphics processing units (GPUs) as high-performance commodity parallel hardware, we propose GPUMiner, a system that uses the GPU as a hardware accelerator for data mining.

GPUs can be regarded as massively multi-threaded manycore processors. Recent multicore CPUs, e.g., Sun's Niagra, are following a similar trend of exploiting parallelism in time and space. Different from multicore CPUs, the cores on the GPU are virtulized, and GPU threads are managed by the hardware. Such a design simplifies GPU programs and improves program scalability and portability, since programs are oblivious about physical cores and rely on hardware for thread creation and management. At present, the GPU possesses an order of magnitude higher computation capability as well as memory bandwidth than a multicore CPU.

Technical Report HKUST-CS08-07, Oct 2008.

Taking advantage of the massive computation power and the high memory bandwidth of the GPU, previous work has accelerated database operations [5, 15, 21, 35], approximate stream mining of quantiles and frequencies [16], MapReduce [22] and *k-means* clustering [10]. So far, there has been no prior work that focuses on systematically studying the GPU acceleration for general-purpose data mining algorithms.

In designing GPUMiner, we consider the following characteristics of the GPU. First, the GPU is a co-processor to the CPU. As such, we develop a CPU-based storage and buffer manager to handle disk I/O as well as data transfers between the GPU and the CPU memory. Second, the GPU processing is in SIMD (Single Instruction, Multiple-Data) and there is no language support for recursion. Therefore, we design "regular-shaped" data structures, e.g., bitmaps, and iterative algorithmic constructs, e.g., counting, that are suitable for the GPU, and perform GPU-CPU co-processing to complete a complex mining task. Third, the GPU is specialized for realtime graphics rendering. To allow users to observe and interact with the mining process online, we develop a GPU-based interactive visualization module. Since part of or the entire computation of mining is already done on the GPU, the efficiency of the visualization is further improved.

As a first step in GPU acceleration for mining, we selected two representative mining algorithms - the *k-means* clustering and the *Apriori* frequent itemset mining (FIM) algorithms to implement in GPUMiner. *k-means* is largely parallelizable and has been accelerated by 35 times on a most recent GPU over a four-threaded CPU [10]. We revisit the *k-means* to seek further improvement and to possibly gain insight for designing other mining algorithms on the GPU.

In comparison, there has been little work on accelerating FIM algorithms on the GPU, even though parallel FIM has been studied on simultaneous multithreading (SMT) processors [13], shared-memory systems [31], and most recently multicore CPUs [28]. Two representative FIM algorithms are *Apriori* [2] and FPGrowth [19]. FPGrowth grows multiple pattern trees recursively to count itemset frequency whereas *Apriori* scans transactions for the purpose. As a first step, we start with a state-of-the-art trie-based *Apri-*

ori [8] and design a GPU-CPU co-processing algorithm, with the trie residing on the CPU and itemset frequency counting done on the GPU.

In both *k-means* and *Apriori*, we design bitmaps to encode information on the GPU and to facilitate fast counting. In *k-means*, the bitmap represents the membership of data objects to clusters; in *Apriori*, the bitmap stores occurrences of items in transactions. Such data structures utilize the GPU efficiently and significantly speed up the computation. As a result, our *k-means* is up to five times faster than the prior GPU-based implementation [10], and our *Apriori* is an order of magnitude faster than the best CPU-based *Apriori* [8] implementation.

Additionally, we have developed GPU-based interactive visualization modules for *k-means* and *Apriori*. In the *k-means* visualization, data objects and cluster centroids are visualized on two dimensions by mapping the data space onto the display space. Users can select data attributes for visualization and can interact with the visualization online by pausing and resuming the process or changing view directions. In the *Apriori* visualization, we display itemsets in point sprites online as they are discovered, and organize the itemsets in a three-dimensional space by their sizes and item composition. We will demonstrate the online mining and visualization as well as various performance comparisons with algorithm and visualization frequency varied.

2. RELATED WORK

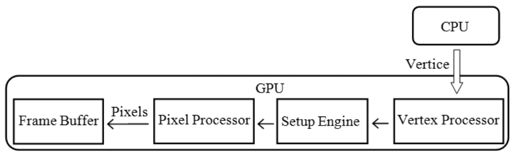
In this section, we briefly review related work on GPGPU (General-Purpose Computation on GPUs), parallel and distributed data mining algorithms and data mining visualization.

2.1 GPGPU

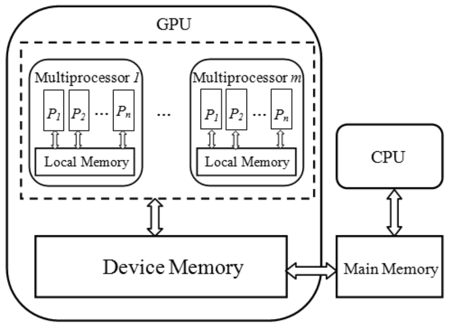
The GPU is an integral component in commodity machines. It was previously designed to be a co-processor to the CPU for games and other interactive applications. Recently, the GPU has been used as a hardware accelerator for various non-graphics applications, such as scientific computation, matrix multiplication [26], databases [14, 16, 22], and distributed computing projects including Folding@home and Seti@home. For additional information on the state-of-the-art GPGPU techniques, we refer the reader to a recent survey by Owens et al. [30].

There are mainly two kinds of GPU programming languages: graphics APIs such as DirectX and OpenGL, and GPGPU languages such as CUDA and CTM. The former kind processes the textures through the programmable hardware pipeline, as shown in Figure 1(a). Vertices and pixel processors are employed to drive the computation. Therefore, programming with the graphics APIs can directly utilize the hardware features related to rendering and visualization. Previously, GPGPU developers used graphics APIs to map applications to the graphics rendering mechanism [16, 26]. However, this kind of mapping may be inefficient and sometimes infeasible [30].

In contrast, GPGPU languages model the GPU as a manycore architecture (as shown in Figure 1(b)), provide C/C++-like interfaces, and expose hardware features for general-purpose computation. For example, CUDA exposes hardware features including the fast inter-processor communication via the *local memory*, as well as the massive thread parallelism. The GPU has a large amount of device memory, which has high bandwidth and high access latency. Recently, primitives as the building blocks for higher-level applica-



(a) The hardware pipeline



(b) The manycore architecture

Figure 1: The architecture model on the GPU

tions have been proposed and implemented [20, 22, 33]. These GPU-based primitives further reduce the complexity of GPU programming.

GPUMiner takes advantage of both kinds of GPU programming languages. It uses CUDA to implement mining algorithms and DirectX for visualization. Moreover, different from the previous studies [16, 20, 22, 33], GPUMiner focuses on the architectural issues of developing a GPU-based parallel system for general-purpose data mining.

2.2 Parallel and Distributed Mining

Parallel data mining has been widely studied in distributed systems [4, 9, 11, 27]. Aouad et al. [4] designed a distributed *Apriori* on heterogeneous computer clusters and grid environments using dynamic workload management to tackle memory constraint, achieve balanced workloads, and reduce communication costs. Buehrer [9] and El-Hajj [11] proposed variants of FPGrowth on computer clusters, lowering communication costs and improving cache, memory, and I/O utilization. Most recently, Li et al. [27] demonstrated a linear speedup of the FP-Growth algorithm over thousands of distributed machines using Google's MapReduce infrastructure.

Since SMT and multicore CPUs have emerged as the main-stream processor, researchers have studied representative mining algorithms, such as *Apriori* [2] and *k-means* [29] on multicore CPUs. The key issue is how to fully exploit the instruction-level parallelism (ILP) and thread-level parallelism (TLP) on the multicore CPU. Gothing [13] et al. improved FPGrowth [19] through a cache-conscious prefix tree for spatial locality and ILP, and a tiling strategy for temporal locality. Liu et al. [28] proposed a cache-conscious FP-array from compacting FP-tree [19] and a lock-free dataset tree construction algorithm for TLP. Ye et al. [38] explored parallelizing Bodon's trie-based *Apriori* algorithm [8] with a database partitioning method. Recently, two benchmarks for mining on multicore processors, including the PARSEC Benchmark Suite [7] and NU-MineBench [32], have been proposed to facilitate architectural studies. Our ongoing work in GPUMiner includes developing a GPU-based FPGrowth and comparing our algorithms with these benchmarks.

In comparison with previous serial or parallel CPU-based FIM algorithms, our algorithms are designed for the GPU with massive thread parallelism. Moreover, we attempt to identify the common techniques on implementing data mining algorithms on the GPU. In particular, our *Apriori* employs both the GPU and the CPU in the processing. The GPU handles frequency counting on transactions in a bitmap whereas the CPU manages the trie structure for result patterns. Such a design takes advantage of the GPU's SIMD massive parallelism as well as works well with the GPU's virtualized cores and hardware-managed threads.

There is little work done for parallel data mining algorithms on the GPU. Only recently, Che et al. [10] implemented *k-means* among five kinds of parallel applications on the GPU using CUDA. In their algorithm, the GPU is responsible for calculating the distances of each object to the k clusters in parallel, and the CPU takes over the reduction step that produces the new centroid for each sub-cluster. In comparison, our *k-means* algorithm utilizes the bitmap-based computation for the efficiency of SIMD execution, exploits the local memory optimization for temporal locality, and minimizes the data transfer between the main memory and the GPU memory.

2.3 Data Mining Visualization

Data mining visualization [25] uses interactive visualization techniques to facilitate fast and effective exploration into an abstract and often a large-scale dataset. Information visualization techniques [34], such as scatterplots, are usually employed to map the data into visual metaphors. We use scatterplots in our visualization due to its capability to express the spatial distribution and association of data items.

Ammoura et al. [3] proposed a system of visualizing OLAP data sources under the CAVE virtual reality environment. Yang [36] proposed to visualize large datasets in 3D scatterplots, also under a CAVE environment. In this paper, we also use scatterplots to visualize the data, but our scatterplot is 2D to avoid depth perception issues with 3D space [34], and our method uses commodity GPUs rather than specialized hardware.

Yang [37] proposed methods for visualizing frequent itemsets and association rules. They use parallel coordinates with line connections to represent association rules. This method efficiently visualizes the relationship between rules. However, parallel coordinates do not express well the spatial distribution of itemsets. We use scatterplots to efficiently express the spatial distributions.

Additionally, there are specialized visual mining techniques. For example, the SplatViz [6] tool in the SGI MineSet software uses a splatting technique to interactively visualize proteomics data. Although the data mining community has sent out the call for GPU-based visualization [17], little work has been done to utilize the GPU for both backend mining and frontend visualization. To the best of our knowledge, our work is the first to integrate mining computation and visualization on the new generation GPUs.

3. ARCHITECTURE OVERVIEW

Figure 2 illustrates the architectural design of GPUMiner. Our system consists of three major components, namely storage and buffer management, the mining, and the visualization.

As an integrated data mining system, GPUMiner has the following features.

High performance. The data mining algorithms in GPUMiner are designed and implemented as parallel ones exploiting the parallelism of the entire machine, including the co-processing parallelism between the CPU and the GPU, and the on-chip parallelism within each processor. In particular, these parallel algorithms are scalable to hundreds of processors on the GPU.

I/O handling infrastructure. GPUMiner provides a flexible and efficient I/O handling infrastructure for analyzing large amounts of data.

Online visualization. Data mining is often a long-running and interactive process. Visualization helps people mine large dataset more efficiently. GPUMiner provides online visualization for the user to observe and interact with the mining process.

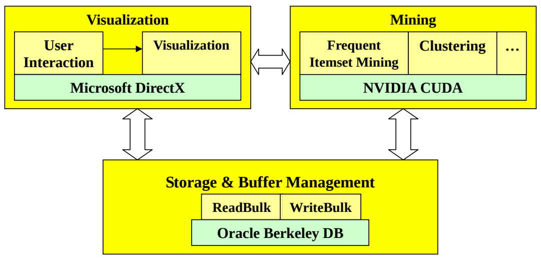
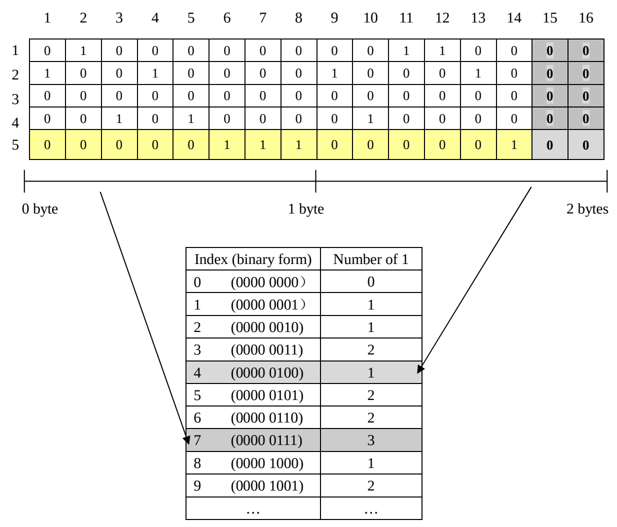


Figure 2: The System Architecture of GPUMiner

The storage and buffer management component is responsible for handling the data transfer between the disk, the main memory and the GPU memory. These three levels of memories form a memory hierarchy, where buffer management should be carefully designed between two adjacent levels. For simplicity and efficiency, this component supports bulk reads and bulk writes only, i.e., reading a chunk of data from the disk to the GPU memory, and writing a chunk of data from the GPU memory to the disk.

In our current implementation, GPUMiner utilizes Berkeley DB (Bdb) as the backend for storing the data persistently. Compared with the raw I/O APIs accessing data in plain text files or structured files, Bdb transparently provides the efficient buffer management between the disk and the main memory, together with convenient file I/O operations including in-place data update. Since current version of GPUMiner supports bulk reads and writes only, we store a bulk of data as a record in Berkeley DB with a unique key. Thus, a data chunk can be fetched or stored by the key. Based on the buffer management from Bdb, GPUMiner provides a lightweight I/O library consisting of two APIs, namely $ReadBulk$ and $WriteBulk$. $ReadBulk$ reads a chunk of data from the disk and transfers them to the GPU memory, whereas $WriteBulk$ outputs a chunk of data from the GPU memory to the disk. With these two APIs, developers can handle large data sets without considering explicit data allocation and data transfer among the GPU memory, the main memory and the disk.

The mining component consists of parallel data mining algorithms including clustering and frequent itemset mining. We choose GPGPU APIs to implement and optimize the mining algorithms due to their algorithmic complexity. With the infrastructure provided in the GPUMiner, we are adding more data mining algorithms such as FP-Growth and classification.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	1	0	0	0	0	0	0	0	0	1	1	0	0	**0**	**0**
2	1	0	0	1	0	0	0	0	1	0	0	0	1	0	**0**	**0**
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**0**
4	0	0	1	0	1	0	0	0	0	1	0	0	0	0	**0**	**0**
5	0	0	0	0	0	1	1	1	0	0	0	0	0	1	**0**	**0**

Index (binary form)	Number of 1
0 (0000 0000)	0
1 (0000 0001)	1
2 (0000 0010)	1
3 (0000 0011)	2
4 (0000 0100)	1
5 (0000 0101)	2
6 (0000 0110)	2
7 (0000 0111)	3
8 (0000 1000)	1
9 (0000 1001)	2
...	...

Figure 3: Counting 1s for the 5-th row of a 5x14 bitmap.

The visualization component visualizes the intermediate or final results as they are produced from the mining component online. It is implemented in DirectX to take advantage of the graphics pipeline for fast rendering. Since both the mining and the visualization components mostly run on the GPU, the data transfer between the two components is small. Moreover, the interactions and visualizations are designed to suit individual mining tasks, e.g., examining clustering results as they converge through iterations or observing itemsets of a specific number of items.

4. DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of the mining and the visualization components in GPUMiner. As a start, we implement two common algorithms in the clustering and frequent mining (FIM), namely *k-means* [29] and *Apriori* [2], respectively. Both algorithms are based on the bitmap technique exploiting data parallelism in the SIMD execution. Following the previous studies [22], we optimize our implementation using memory optimizations and thread parallelism. The memory optimizations include the local memory optimization for temporal locality and the coalesced access optimization for spatial locality.

In the visualization component, we visualize the intermediate clusters or patterns generated by the two algorithms for user interaction. Since both algorithms are iterative, our visualization naturally maps to multiple iterations/passes during the mining process.

4.1 GPU-Based Data Mining Algorithms

4.1.1 The Bitmap technique

Counting is a core operation in data mining algorithms. For example, *k-means* counts the number of data objects associated with a specific cluster, and *Apriori* counts the number of transactions containing the same item. We transform this association counting into counting the number of ones resulted from a set of boolean tests on the association. Since the association is usually a binary relation (e.g., <object, cluster> in *k-means* and <transaction, item> in *Apriori*), we use a bitmap, i.e., a 2D array of bits, to store the association.

The bitmap can be implemented in row-major or column-major. We take the row-major implementation as an example. Each row is rounded in bytes, padded with 0. Given Object i and Object j in an m x n bitmap, $0 < i \leq m$ and $0 < j \leq n$, if Object i and Object j are associated, the corresponding (i, j) bit is set to 1. Figure 3 shows an example of 5 x 14 bitmap. The row is rounded to 2 bytes, that is, 16 bits.

The bitmap supports efficient row-wise and column-wise operations exploiting the thread parallelism on the GPU. Given an m x n bitmap, we can use x threads to process m rows in parallel, or to process n columns in parallel. The tasks of processing rows or columns are uniformly assigned to the threads. Another common operation is counting on the number of ones in a set of rows/columns in the bitmap. This can also be implemented using row- or column-wise operations. Since this operation is common and important for the performance, we construct a summary vector for each row/column to accelerate counting. A summary vector stores the number of ones of each row. To count the number of ones in a set of rows, we first extract the corresponding counts from the summary vector, and then use a parallel reduce primitive [12] to calculate the final result.

Figure 3 illustrates an example for counting ones for the 5-th row of a 5 x 14 bitmap. Each row is rounded to 2 bytes, so that we need to look up the summary vector for 2 times. In this example, the first byte of the 5-th row is a bit-string of 0000 0111, which is the binary form of 7 in decimal. We use 7 as the index to look up the summary vector, getting 3, the number of ones for the first byte. We repeat the same process for the second byte, and then get 1. By summing up both results, we get the number of ones for the 5-th row.

4.1.2 K-Means

k-means is one of the most well-known and commonly used clustering algorithms. It takes an input parameter k, and partitions a set of n objects into k clusters according to a similarity measure. Each object has multiple attributes, or *features*. The mean values, or *centriods*, are a summary measure of the similarity of data objects within the same cluster.

The *k-means* algorithm works in iterations. At the beginning, the algorithm randomly chooses k of the objects as the initial *centroid* for each cluster. In each iteration, *k-means* associates each data object with its nearest *centroid*, according to the similarity metric. Next, it computes the new *centroids* by taking the mean of all the data objects in each cluster respectively. *K-means* terminates when the changes in the $centriods$ are less than some threshold.

The data transfer between the main memory and the GPU memory is small, once the input data is ready on the GPU memory. During the execution, only a 4-byte flag variable is transferred between GPU memory and CPU memory. Thus, our implementation avoids heavy memory ping-pong overhead described in Che et. al [10]. Our bitmap-based *k-means* algorithm outlines as follows:

Algorithm 1 Bitmap-Based K-Means

```
while $cpu\_flag$ do
    $makeBitmap\_kernel$
    transfer $cpu\_flat$ to $gpu\_flag$
    $computeCentriod\_kernel$
    transfer $gpu\_flag$ to $cpu\_flag$
$findCluster\_kernel$
```

Functions *makeBitmap_kernel*, *computeCentriod_kernel* and *findCluster_kernel* are three *kernel* functions executed on the GPU in parallel. In the *makeBitmap_kernel* function, each thread processes one data object, by finding out the nearest *centriod*. Next, the function sets the corresponding bit according to the association in a bitmap. We use a $k \times n$ bitmap to keep track of each data object and the corresponding centroid, in which, k is the number of clusters and n is the number of data objects. In the *computeCentriod_kernel* function, according to the bitmap, computes the sum of counts for data objects in one cluster according to the summary vector and then computes a new centroid. If the distance between the new centroid and the old one is bigger than a given threshold, then we consider the centroid changes. In this case, we set the gpu_flag variable to true, so that the iteration keeps going. If none of the centroid changes, the iteration terminates. Finally, the *findCluster_kernel* function makes up the association between each data object to its nearest centroid.

4.1.3 Apriori

Frequent itemset mining finds sets of items that appear in a percentage of transactions with the percentage, called *support*, larger than a given threshold. The *Apriori* algorithm finds all frequent itemsets in multiple passes, given a support threshold. At the first pass, it finds the frequent items. Generally at the l-th pass, the algorithm finds the frequent itemsets each consisting of l items (called l-itemsets). In each pass, e.g., the $(l+1)$-th pass, it first generates *candidate* $(l+1)$-itemsets from the frequent l-itemsets, then counts supports of these candidates and prunes those candidates whose supports are less than a given support threshold. The algorithm ends when no candidate is generated in a pass.

An l-itemset becomes a candidate if all of its subsets are frequent [2]. Candidate generation has two steps. The first step finds pairs of l-itemsets that have the same prefix $(l-1)$-itemset and differ only in the l-th item, and joins each pair of $< i_1, ..., i_{l-1}, i_l >$ and $< i_1, ..., i_{l-1}, i'_l >$ to generate a *potential candidate* $< i_1, ..., i_{l-1}, i_l, i'_l >$. The second step checks the l-subsets of the potential $(l+1)$-candidates. If all l-subsets of a potential $(l+1)$-candidate are frequent, it becomes a candidate.

After candidate generation, the algorithm scans the transaction set to count supports for the candidates. For each transaction, the algorithm decides which candidates are contained by it, and the supports of contained candidates are each increased by one. After the entire transaction set is scanned, the algorithm identifies the candidates with support no less than the support threshold as frequent itemsets.

The state-of-the-art *Apriori* algorithm is the trie-based implementation [8]. It uses a trie to store candidates and their supports. Each node in the trie has two attributes: item id and support. Trie-based *Apriori* counts support by reading the transaction set, just as the original *Apriori* [2] does. What is different is that, for each transaction, it finds paths from the root to the leaves in the trie corresponding to contained candidates in the transaction. The supports of the leaves in the found paths are each increased by one. We adopt the trie-based implementation on the GPU.

No matter there is a trie or not, *Apriori* scans the transaction set multiple times, one scan per pass, to count supports of candidates. The counting operations are time consuming and is the performance bottleneck. For example, in our experiments, the FIMI'03 best *Apriori* implementation, with a typical support threshold of 2% and a large data set $T10I4D100K$ [8], spent about 90% of total time on support counting. Thus, we implement the counting operations with the bitmap technique.

We use an $I \times T$ bitmap, denoted as IT, to store the occurrences of items in the transaction set, where T is the total number of transactions and I is the total number of unique items in all transactions. In the bitmap, bit $IT[i][t]$ is one if item i is contained in transaction t; otherwise, it is zero. We implement the bitmap as a two dimensional array with the first index being the item id and the second index being the transaction id. In this storage format, transaction sets of items correspond to equal-sized bitsets such that the overlap of transaction sets of items can be easily obtained through bitset intersection operations. One scan of the transaction set is sufficient to put 1s in the corresponding bits in the array.

The bitmap-based *Apriori* algorithm works as follows. In the first pass, we obtain the supports of items through counting 1s in their corresponding transaction bitsets in IT. In the second pass, for each 2-itemset, we intersect the two items' corresponding transaction bitsets and count 1s in the result as the support. From the third pass, support counting is a recursive procedure. We start from the root of the trie. For each node i in the trie (corresponding to an item), if it does not have leaf nodes, we return directly. Otherwise, we intersect i's transaction bitset $IT[i]$ with the intersection result R' of its ancestors' transaction bitsets, which is passed from the caller recursion. Let $R = IT[i]\&R'$ be the result. After that, we examine whether the level of i in the trie equals to l, the current number of passes. If it does, we count 1s in R and attach the count to node i as its support. Otherwise, we call our procedure recursively and pass R for bitset intersection to this procedure.

Our bitmap-based support counting is efficient for the following reasons. First, we use intersection and 1-bit counting on bitmaps to obtain support. Both operations are efficient on both the CPU and the GPU. In particular, 1-bit counting can be done even more efficiently on the GPU through the parallel map primitive [22]. Second, the recursive procedure passes the previous intersection result as a parameter, hence eliminates repeated intersection operations for itemsets with the same prefix. Third, we scan the transaction set only once (for bitmap construction) instead of scanning it once per pass, therefore eliminate subsequential I/O.

We employ GPU and CPU co-processing to implement support counting. The trie traversal is implemented on the CPU. This decision is based on the following considerations. (1) The trie is an irregular structure and difficult to share among SIMD threads; (2) recursion, which is used to visit nodes in the depth first order (DFS), is not supported on the GPU; (3) compared with bitset operations, the time for trie operations is insignificant. Both the bitmap IT and the intermediate intersection results are stored in the GPU global memory since the size may be large. The 1-bit counting and intersection operations are implemented as GPU programs called kernels.

A naive GPU-CPU co-processing scheme is single node per recursion (SNPR). In each recursion, we perform intersection and 1-bit counting for a single node. The inputs of SNPR are 1) an item i, 2) its parent item p in the trie, 3) its level $level$ in the trie, and 4) the current pass l. In a recursion, we call the GPU kernel to intersect i's transaction bitset $IT[i]$ with the intersection result R' of its ancestors' transaction bitsets. The position of R' in the GPU memory is identified by p. If $level$ of node i equals to l, we count 1s in R

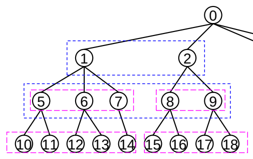


Figure 4: The merged nodes in MNPR of Apriori.

in the GPU kernel, pass the count back to the main memory, and assign the count to i's support. Otherwise, we call the recursion on each of i's children. We implement bitset intersection with a map primitive [22], and 1-bit counting is done on the map result.

This implementation is simple but not efficient because of two reasons: (1) for each single node, the data size of map primitive is small, thus the GPU computation resource is not fully utilized, (2) there are a large number of small GPU-CPU memory transfers incurred in GPU kernel calls, so the overhead is considerable.

Our optimized GPU implementation processes multiple nodes per recursion (MNPR). The main idea of the optimization is to group as many as possible intersection and 1-bit counting operations in one recursion on the GPU to reduce the kernel call overhead and to fully utilize the computation power of the GPU. We use a bound variable B to determine the number of parent and children pairs whose intersections and 1-bit counting are to be performed in the same GPU kernel call. The setting of B considers the size of transaction bitsets of each item and the size of the GPU memory. Suppose the available size of the GPU memory is G, the maximum number of items in one transaction is L, the total number of transactions is T, we have $B = \frac{G}{L \times T}$.

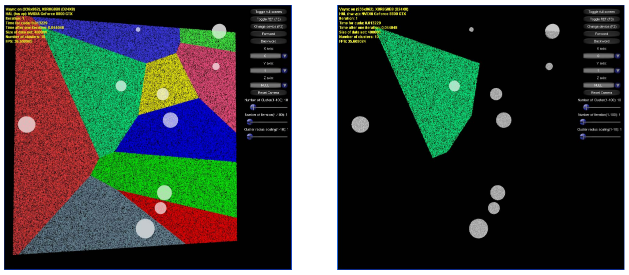
The inputs of MNPR are 1) a list of items IL, all in the same level of the trie, 2) the list of their parents PL in the trie, 3) the level $level$ of the items in IL, 4) the current pass l, and 5) bound B. In a recursion, we first call the GPU kernel to intersect transaction bitset of each item in IL with the intersection result of its ancestors' bitsets. The position of the ancestors' intersection result in the GPU memory is identified by PL. Then, we check whether $level$ equals to l. If so, we count 1s in the results in the kernel and pass the counts back to the CPU memory, and assign them to IL's supports. Otherwise, we divide the children of IL into serval segments. Each segment contains at most B items. We then call the recursion for each segment.

Figure 4 shows an example with $B = 5$. We pass the item ids of nodes IL ={*5, 6, 7, 8, 9*} and their corresponding parent nodes PL ={*1, 2*} together to the same GPU kernel to intersect {*1*} with {*5, 6, 7*} and {*2*} with {*8, 9*}. Compared to SNPR, MNPR reduces the number of kernel calls from 5 to 1. After that, we divide the children of {*5, 6, 7, 8, 9*} into segments {*10, 11, 12, 13, 14*} and {*15, 16, 17, 18*}. Then, we pass {*10, 11, 12, 13, 14*} and their corresponding parents {*5, 6, 7*} together to a GPU kernel in a recursion, and pass {*15, 16, 17, 18*} and their corresponding parents {*8,9*} together to a GPU kernel for another recursion. This optimization reduces the number of kernel calls from 5 to 1 and 4 to 1, respectively. In total, it reduces the number of kernel calls from 14 to 3 when visiting the branches of node 1 and 2 in the trie in DFS.

4.2 Visualization

The GUIs of *k-means* and *Apriori* modules of GPUMiner are displayed in Figures 5 and 6, respectively. We describe the visual analysis functionalities of each module.

4.2.1 Visualization of K-Means



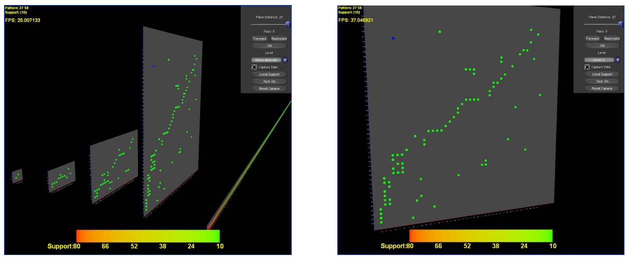
(a) All data objects (b) Data objects in a cluster

Figure 5: Visualization of K-Means clustering.

Figure 5(a) displays 400,000 data objects that are clustered into ten groups based on their 41 features. The user can select two features to be the two axes, and visualize the dataset as a scatterplot in the 2D space. Each point represents one data object; points within the same cluster share a color. The centroid of each cluster is drawn as a filled circle with the radius representing the number of objects of this cluster. We can see clearly the distribution, position and density of each cluster on the two displayed features. In this way, GPUMiner utilizes human visual system (HVS) to efficiently and effectively discover potential patterns or outliers.

The user could reassign which features are to be clustered on and which are to be displayed as his/her wish. The displayed features are not necessarily involved in the clustering. The user could use the keyboard to control the progress of clustering iteration to observe the intermediate results in animation. He/she can interactively change the number of clusters and the number of iterations to converge. The GPU performs interactive *k-means* clustering in the backend. The user could click on one cluster to observe it in isolation (5(b)), and zoom in to further explore this cluster.

4.2.2 Visualization of Apriori



(a) Parallel 2D scatterplots (b) One scatterplot

Figure 6: Visualization of Apriori.

Figure 6(a) displays the *Apriori* frequent itemset mining for transaction set. The window contains a series of parallel 2D scatterplots. The l-th scatterplot represents the mining result of pass l, i.e., frequent l-itemsets (l-patterns). In the l-th scatterplot, we map the combination of the first $l-1$ items of each l-pattern to the x-coordinate, and map the last item of the pattern to the y-coordinate of the scatterplot. As a result, each pattern is displayed as a 2D point in the scatterplot, and the 2D space of the l-th scatterplot is always warped into the x-axis of the adjacent (l+1)-th scatterplot. The first scatterplot is 1D, since the 1-patterns (frequent items) have no corresponding y-coordinate. The GPU performs interactive *Apriori* frequent item mining in the backend.

We use heat map color intensity of the points to indicate the support of the patterns. In each scatterplot, we can intuitively see the support of each pattern and the distribution of support of patterns with the same prefix. Across scatterplots, we can compare the associated patterns with subset relationships; we can also observe the change of frequencies increase in the size of patterns.

During the mining process, the user can use the keyboard to pause at a pass to observe the existing scatterplots. The user can perform 3D navigation into the world of parallel scatterplots. The navigation follows common 3D GUI conventions that use mouse movements to change the direction, and the arrow keys to move the position. The user can select a single scatterplot and zoom in to further explore it, as shown in Figure 6(b). When the user captures a point using a mouse, the support and item containment of the corresponding pattern are displayed.

Since both the backend computation and the frontend rendering are performed on the GPU, it would be ideal that the frontend directly fetches the results of the backend. Unfortunately, the current general-purpose APIs are compatible with only a few graphics resources. Therefore, when the computation results are not immediately visible to the graphics API, we have to do an extra buffer copy from the output of computing to the input of the rendering, even though both are already in the GPU memory. This extra copy does not affect the interactiveness of the demonstrations in this paper, but might be a performance issue for very large datasets. Nowadays, the inter-operability between general-purpose APIs and graphics APIs is being rapidly enhanced, and we expect a tighter integration of the two APIs to result in a higher performance in the future.

5. EXPERIMENTAL RESULTS

In this section, we present our experimental results on evaluating the mining and the visualization components of GPUMiner. We report results on both *k-means* and *Apriori*.

5.1 Experimental Setup

Our experiments were performed on a PC with an NVIDIA GTX280 GPU and an Intel Core2 Quad Core CPU, running on Microsoft XP SP3. The GPU consists of 30 SIMD multi-processors, each of which has eight processors running at 1.29 GHZ. The GPU memory is 1GB with the peak bandwidth of 141.7 GB/sec. The CPU has four cores running at 2.4 GHZ. The main memory is 2 GB with the peak bandwidth of 5.6 GB/sec. The GPU uses a PCI-E bus to transfer data between the GPU memory and the main memory with a theoretical bandwidth of 4 GB/sec. The PC has a 160 GB SATA magnetic hard disk.

All source code was written and compiled using Visual Studio 2005 with the optimization level /O2. The versions for CUDA, DirectX, and Berkeley DB are 2.0, 9.0 and 4.7.25, respectively.

Comparison. For *k-means*, we compare our algorithm with the CUDA-based implementation by Che et al. [10] (denoted as "UVirginia"). Che et al. showed that their CUDA based *k-means* implementation was up to 35x faster than the four-threaded CPU-based counterpart. We download their GPU-based implementations from their website [24] for comparison.

For *Apriori*, we compared our GPU-based algorithm with two CPU-based algorithms, since there is no GPU-based *Apriori* implementation in the public domain. One version is the CPU counterpart of our GPU-based implementation, denoted as *GPUMiner: Apriori-CPU*. The other version is the FIMI'03 best implementation [8] of *Apriori*, denoted as *FIMI: Apriori*.

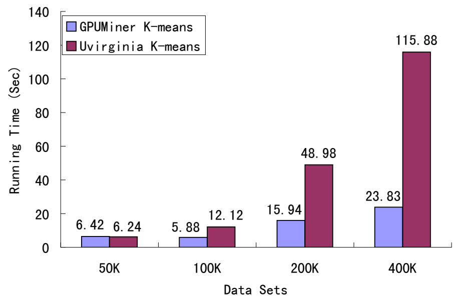


Figure 7: Execution time of the two implementations of K-Means on various number of data objects.

Data sets. Following previous studies [10], we use the KDD Cup 1999 [23] intrusion detection data set to evaluate *k-means*. Each object of the data set contains 41 features, and each feature is a floating point number. The number of clusters is 24, which represents 24 attack types.

We used the data sets from FIMI'03 repository to evaluate the three *Apriori* implementations. The three data sets we used were $T10I4D10K$, $T10I4D100K$, and $T40I10D100K$. The three data sets are representative small, medium, and large data sets used in FIMI.

Metric. We measured the total elapsed time for evaluating the efficiency of the mining component, and used FPS (Frame Per Second) to evaluate the visualization component. Since comparison experiments were conducted on the same input file and produce the same mining result, we excluded the initial file input and final result output from the total time measurement. We ran each experiment for three times, and calculated the mean value. The variance among different runs of the same experiment was smaller than 10%.

5.2 Results on Mining

Figure 7 shows the total running time of our *k-means* algorithm and UVirginia as the number of data objects increases. Our bitmap based *k-means* was up to 5x faster than UVirginia.

We further investigate the time breakdown in Table 1, of these two implementations processing 400 thousand data objects. We break the running time into three components, GPU-CPU data transfer, GPU computation, and CPU computation. The bitmap *k-means* well utilizes GPU computation resources and minimizes the GPU-CPU data transfer. The GPU computation time takes up more than 98% of total running time in our algorithm. In comparison, the data transfer time dominates the total time in UVirginia.

One interesting point is that on the sum of the computation time of GPU and CPU, UVirginia outperforms our bitmap implementation. The reason is that UVirginia performs the accumulation computation on the CPU, whereas ours on GPU. Essentially, we exploit the high computation capability on the GPU for less GPU-CPU data transfer. As a result, the overall performance is improved.

Figure 8 shows the execution time of the three *Apriori* implementations. The support threshold is set to 1% for all of the three data sets. Both the CPU and the GPU implementations of our bitmap-based algorithm are faster than *FIMI:Apriori*, achieving a speedup

Table 1: Time Breakdown of K-Means

	Total	Data Transfer	GPU	CPU
GPUMiner	23.8255s	0.1093s	23.5763s	0.1399s
UVirginia	115.8839s	110.8994s	0.0053s	4.9792s

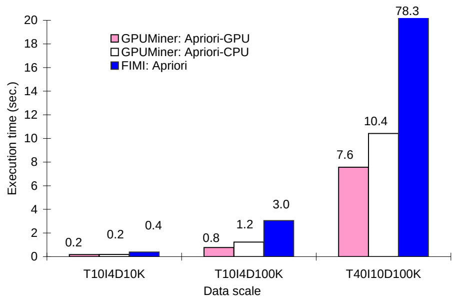


Figure 8: Execution time of the three implementations of Apriori on various data sets.

up to 10.4x and 7.5x, respectively. Furthermore, the speedup of our algorithm with the CPU or the GPU implementations, increases as the data size increases.

Figure 9 shows the execution time of the three implementations on $T40I10D100K$ with various support thresholds. When the support threshold changes from 0.5% to 2.0%, the GPU implementation of our bitmap-based *Apriori* achieves a speedup from 6.2x to 12.1x and the CPU implementation achieves a speedup from 3.8x to 9.5x. We can see that both implementations of our algorithm are much faster than the FIMI implementation throughout a range of support thresholds.

5.3 Performance of Visualization

In the following, we evaluate the visualization performance for *k-means* clustering and *Apriori* mining, respectively.

5.3.1 Performance of K-Means Visualization

We evaluate the visualization performance of *k-means* clustering (Figure 10). We vary the number of objects from 50K to 400K, and for each data size, measure (1) the elapsed time of *k-means* computation, labeled as "Compute", (2) the elapsed time after computation and before the first frame is drawn, labeled as "First Frame", and (3) the total time of the two steps, labeled as "Compute + First Frame". We measure the time for a single iteration of clustering since each iteration takes about the same time. The "First Frame" time mainly consists of the memory copying from general-purpose computing resources to graphics resources. We have also measured

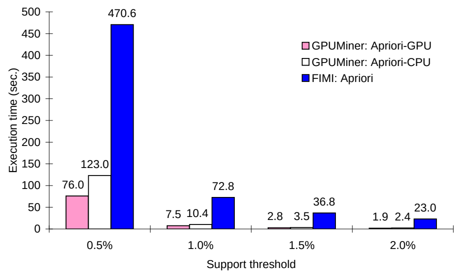


Figure 9: Execution time of the three implementations of Apriori varying support thresholds on $T40I10D100K$.

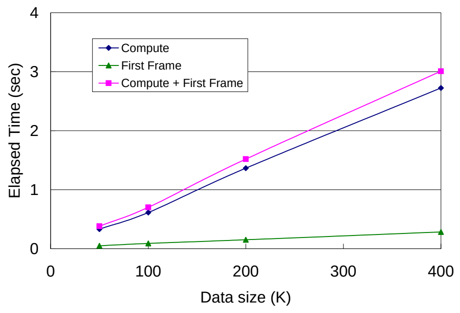


Figure 10: Elapsed time with varying data size.

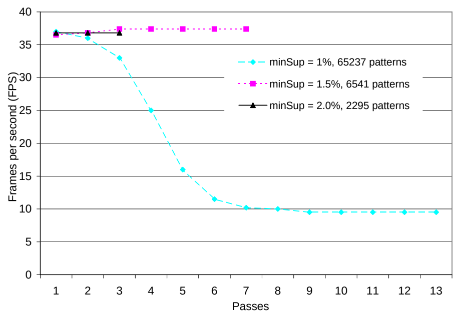


Figure 11: Visualization performance as number of passes progresses.

the rendering performance after the first frame is drawn. The FPS (frame-per-second) is around 36 for all the tested data sizes. This FPS indicates that the rendering overhead after the resource copying is negligible in comparison to the computation and the first frame.

The figure shows that both the computation and copying time increase linearly with the data size, and the copying overhead is only 10%-15% of the computing time. As the iteration number grows, the copying overhead becomes a tiny portion of the total elapsed time.

5.3.2 Performance of Apriori Visualization

We evaluate the visualization performance of *Apriori* with three different support thresholds 1%, 1.5% and 2%, on the T40I10D100K transaction set. In pass l, we tested the FPS of displaying the existing patterns discovered in the first l passes, i.e., the FPS of rendering the first l scatterplots. Figure 11 shows the FPS of each pass. In general, the FPS drops with the increase of pass with all support thresholds. The reason is that the number of scatterplots increases. However, when the support threshold is set to 1.5% or 2%, the total number of patterns (6541 or 2295) discovered is relative small. Therefore, the FPS is consistently above 35. When the support threshold is set to 1%, new patterns keep emerging, and the FPS finally drops to around 10 when all 65237 patterns are displayed. However, this visualization performance is still highly interactive. Note that the graphics resources to be copied (including the patterns and their support) from the CPU memory are relatively small in size, therefore the measured copy time is negligible in comparison to the mining and rendering time.

6. CONCLUSION AND FUTURE WORK

We have presented GPUMiner, a system that utilizes GPUs to accelerate general-purpose data mining. It consists of a CPU-based storage and buffer manager, a GPU-CPU coprocessing mining module, and a GPU-based interactive visualization module. Our storage and buffer manager handles file I/O and data transfers between the disk, main memory, and the GPU memory. Our mining module currently contains a GPU-based *k-means* and an *Aproiri* with GPU-CPU coprocessing. Both mining algorithms employ bitmaps to encode information on the GPU and utilize the GPU's SIMD massive thread parallelism for computation. Our visualization module takes intermediate results from the mining algorithms on the GPU and renders them on the screen efficiently. Additionally, it allows users to interact with the mining process online. We have presented preliminary performance results in comparison with state-of-the-art GPU or CPU implementations. We will demonstrate the system through our visualization module.

We are studying detailed performance of our algorithms. In particular, we will examine the time breakdown of our GPU-accelerated *Apriori* algorithm to identify the performance bottleneck - the GPU, the CPU, or the data transfer. Based on our findings, we may redesign the algorithm to balance the performance of the three components so as to improve the overall performance.

We are considering other improvements of our current implementations. For example, our bitmaps for counting are space inefficient for sparse datasets. We limit the data chunk size that can fit into the GPU memory. For efficiency, we are investigating data compression techniques [1]. Moreover, we are developing a buffering mechanism between the GPU memory and the main memory for memory ping-pong.

We also plan to add to GPUMiner other mining algorithms with GPU acceleration, for instance, FP-Growth and classfication. In particular, FP-Growth and its CPU-based variants have shown a superior performance; nevertheless, their irregular data structures and complex algorithmic control pose great challenges for GPU acceleration.

Finally, it could be desirable to enhance the interaction features of the mining process, for example, adjusting distance / confidence / support thresholds during the progress. Such interaction can greatly improve the mining quality. For example, *k-means* requires the knowledge of the number of clusters in advance. The interaction enables the developer to input the number of clusters observed in the visualization, and improves the convergence speed of the algorithm. Nevertheless, this kind of interaction requires more design and implementation effort in maintaining states of the mining process and in incremental computation by adjusting the mining parameters. Code and documentation, including a full version of this paper, are available at http://code.google.com/p/gpuminer/.

7. REFERENCES

[1] D. J. Abadi, S. R. Madden, and M. C. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, volume 1215, 1994.

[3] A. Ammoura, O. Zaiane, and R. Goebe. Towards a novel olap interface for distributed data warehouses. In *DaWaK*, 2001.

[4] L. Aouad, N. Le-Khac, and T. Kechadi. Distributed Frequent Itemsets Mining in Heterogeneous Platforms. 2007.

[5] N. Bandi, C. Sun, D. Agrawal, and A. E. Abbadi. Hardware acceleration in commercial databases: A case study of spatial operations. In *VLDB*, 2004.

[6] B. G. Becker. Volume rendering for relational data. In *InfoVis*, 1997.

[7] C. Bienia, S. Kumar, J. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[8] F. Bodon. A fast apriori implementation. In *FIMI*, volume 90, 2003.

[9] G. Buehrer, S. Parthasarathy, S. Tatikonda, T. Kurc, and J. Saltz. Toward terabyte pattern mining: an architecture-conscious solution. In *PPOPP*, 2007.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using cuda. In *JPDC*, 2008.

[11] M. El-Hajj and O. Zaiane. Parallel Leap: Large-Scale Maximal Pattern Mining in a Distributed Environment. In *ICPADS*, 2006.

[12] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Gpuqp: query co-processing using graphics processors. In *SIGMOD*, 2007.

[13] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In *VLDB*, 2005.

[14] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD*, 2006.

[15] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGMOD*, 2004.

[16] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *SIGMOD*, 2005.

[17] S. Guha, S. Krishnan, and S. Venkatasubramanian. Tutorial: Data visualization and mining using the gpu. In *KDD*, 2005.

[18] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2001.

[19] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.

[20] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In *PACT*, 2008.

[21] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *SC*, 2007.

[22] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. V. Sander. Relational joins on graphics processors. In *SIGMOD*, 2008.

[23] http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html. *KDD Cup 1999*.

[24] http://lava.cs.virginia.edu/wiki/rodinia/index.php/K Means. *CUDA based k-means implementation from Univ. of Virginia*.

[25] D. A. Keim. Information visualization and visual data mining. *TVCG*, 2002.

[26] E. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. 2001.

[27] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Chang. PFP: Parallel FP-Growth for Query Recommendation. *ACM Recommender Systems*, 2008.

[28] L. Liu, E. Li, Y. Zhang, and Z. Tang. Optimization of frequent itemset mining on multiple-core processor. In *VLDB*, 2007.

[29] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, 1967.

[30] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. 2007.

[31] S. Parthasarathy, M. J. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared memory systems. 2001.

[32] J. Pisharath, Y. Liu, W. Liao, A. Choudhary, G. Memik, and J. Parhi. NU-MineBench 2.0. Technical report, Technical Report, CUCIS of Northwestern Univ., 2005.

[33] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. 2007.

[34] R. Spence. *Information Visualization 2nd Edition*. Prentice-Hall (Pearson), 2007.

[35] C. Sun, D. Agrawal, and A. E. Abbadi. Hardware acceleration for spatial selections and joins. In *SIGMOD*, 2003.

[36] L. Yang. Interactive exploration of very large relational datasets through 3d dynamic projections. In *KDD*, 2000.

[37] L. Yang. Pruning and visualizing generalized association rules in parallel coordinates. In *TKDE*, 2005.

[38] Y. Ye and C.-C. Chiang. A parallel apriori algorithm for frequent itemsets mining. In *SERA*, 2006.

[39] M. J. Zaki. Data mining parallel and distributed association mining: A surray. *IEEE Concurrency*, 1999.

8. DEMONSTRATION

We will demonstrate both the visualization and the performance for the mining algorithms.

For the visualization of *k-means* using GPUMiner, we will combine the following interactions in the mining process: changing the number of clusters and iterations; step-by-step clustering; selecting one cluster; and zooming in. For the visualization of *Apriori* using GPUMiner, we will combine the 3D navigation, selecting one scatterplot and zooming in etc., to mine the data.

To demonstrate the performance, we use multi-threaded CPU-based implementations as the counterparts. We will compare the elapsed time of CPU- and GPU-based *k-means*, both in terms of one iteration (if the clustering is slow) and all iterations. We will compare the elapsed time of CPU- and GPU-based *Apriori* frequency mining, both in one pass and in all passes.