Frequent Itemset Mining on Graphics Processors

Wenbin Fang, Mian Lu, Xiangye Xiao, Bingsheng He[1], Qiong Luo
Hong Kong University of Science and Technology
{wenbin, lumian, xiaoxy, luo}@cse.ust.hk
Microsoft Research Asia[1]
savenhe@microsoft.com

ABSTRACT

We present two efficient *Apriori* implementations of Frequent Itemset Mining (FIM) that utilize new-generation graphics processing units (GPUs). Our implementations take advantage of the GPU's massively multi-threaded SIMD (Single Instruction, Multiple Data) architecture. Both implementations employ a bitmap data structure to exploit the GPU's SIMD parallelism and to accelerate the frequency counting operation. One implementation runs entirely on the GPU and eliminates intermediate data transfer between the GPU memory and the CPU memory. The other implementation employs both the GPU and the CPU for processing. It represents itemsets in a trie, and uses the CPU for trie traversing and incremental maintenance. Our preliminary results show that both implementations achieve a speedup of up to two orders of magnitude over optimized CPU *Apriori* implementations on a PC with an NVIDIA GTX 280 GPU and a quad-core CPU.

1. INTRODUCTION

Frequent itemset mining (FIM) aims at finding interesting patterns from databases, or called *transaction databases*. Each database transaction contains a set of *items*, such as grocery items purchased in a basket. A FIM algorithm scans the database, possibly multiple times, and finds itemsets that occur in transactions more frequently than a given threshold. The number of occurrences is called *support*, and the threshold *the minimum support*.

Two representative FIM algorithms are *Apriori* [3] and FP-growth [16]. *Apriori* iteratively generates candidate itemsets of *K+1* items, or *(K+1)-itemsets*, from *K*-itemsets, and scans all transactions to check whether the candidate itemsets are frequent. In comparison, FP-growth recursively builds pattern trees to represent frequent itemsets, without candidate generation. According to a report from the first Workshop on Frequent Itemset Mining Implementations (FIMI'03) [12], FP-growth implementations were generally an order of magnitude faster than *Apriori*; however, on several datasets, an *Apriori* implementation, apriori_borgelt, was slightly faster when the support was high.

Utilizing parallel architectures has been a viable means for improving data mining performance [4, 7, 9, 32]. In this paper, we study whether we can adapt the existing CPU-based FIM algorithms to new-generation graphics processing units (GPUs). GPUs can be regarded as massively multi-threaded many-core processors. Different from multi-core CPUs, the cores on the GPU are virtualized, and GPU threads are executed in SIMD (Single Instruction, Multiple Data) and are managed by the hardware. Such a design simplifies GPU programming and improves program scalability and portability, since programs are oblivious about physical cores and rely on hardware for thread management. Nevertheless, it also makes the implementation of algorithms with complex control flows a challenging task on the GPU, even though the GPU has an order of magnitude higher computation capability as well as memory bandwidth than a multi-core CPU.

Taking advantage of the massive computation power and the high memory bandwidth of the GPU, previous work has accelerated database operations [13, 14, 19], approximate stream mining of quantiles and frequencies [15], MapReduce [17] and *k-means* clustering [8]. To the best of our knowledge, there has been no prior work that focuses on studying the GPU acceleration for FIM algorithms, even though parallel FIM has been studied on simultaneous multithreading (SMT) processors [11], shared-memory systems [28], and most recently multi-core CPUs [25].

As a first step, we consider the GPU implementation of *Apriori*, with intention to extend to FP-growth. The *Apriori* algorithm is not only applied in frequent itemset mining or association mining, but also in other data mining tasks, such as clustering [27], and functional dependency [22]. Existing *Apriori* FIM algorithms are optimized for data locality; however, the data structures in use, e.g., tries, are non-aligned and the access patterns are largely irregular, e.g., pointer-chasing. These characteristics may hurt the efficiency on the GPU since SIMD operations favor aligned and sequential data accesses [34].

Addressing the challenge in implementing *Apriori* on the GPU, we adopt a bitmap data structure to represent transactions in our two GPU-based FIM implementations. Specifically, the bitmap stores the occurrences of items in transactions, and is efficient to be partitioned to SIMD processors. Furthermore, we utilize a lookup table to facilitate support counting, which is usually the most time-consuming component in the *Apriori* algorithm. One implementation of ours

Proceedings of the Fifth International Workshop on Data Management on New Hardware (DaMoN 2009) June 28, 2009, Providence, Rhode-Island

uses another bitmap to represent itemsets, which enables the entire algorithm to run on the GPU. We denote this implementation as PBI (Pure Bitmap-based Implementation). PBI features regular data access patterns, which are best fit to the GPU; however, it may cause redundant computation and data access between frequent itemsets of different sizes. To reduce the redundancy, the other implementation of ours adopts a trie structure to represent itemsets, and utilizes the CPU for trie traversal and incremental maintenance. We denote this Trie-based Implementation as TBI. We have evaluated our implementations using both synthetic and real-world datasets. Both of our implementations are up to two orders of magnitude faster than optimized CPU-based *Apriori* implementations on three experimental datasets.

Organization: The remainder of the paper is organized as follows. We give a brief overview of prior work on GPGPU and frequent itemset mining in Section 2. We present the details of our two implementations in Section 3. In Section 4, we present our experimental results. Finally, we conclude in Section 5.

2. BACKGROUND AND RELATED WORK

In this section, we briefly review related work on GPGPU (General-Purpose Computation on GPUs), and frequent itemset mining algorithms.

2.1 General Purpose GPU Computing

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 graphics applications. Recently, the GPU has been used as a hardware accelerator for various non-graphics applications, such as matrix multiplication [23], databases [13, 14, 19], 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. [26].

Recently, GPGPU programming frameworks such as NVIDIA CUDA allow the developer to write the code for the GPU with familiar interfaces similar to C/C++. Such frameworks model the GPU as a many-core architecture (as shown in Figure 1) exposing hardware features for general-purpose computation. In particular, CUDA exposes a hierarchical multi-threaded model for NVIDIA's latest GPUs, with hardware features including the fast on-chip *local memory* (NVIDIA terms it as *shared memory*). CUDA groups lightweight GPU threads into *thread blocks*. Threads within the same *thread block* are divided into SIMD groups, called *warps*, each of which contains 32 threads. The GPU has an on-board device memory, which is of a high bandwidth and a high access latency. A *warp* of threads can combine accesses to consecutive data items in one device memory segment into a single memory access transaction, or called *coalesced access*.

While GPGPU programming frameworks greatly reduce the complexity of GPGPU computing, developers must carefully design and implement their algorithms in order to fully utilize the GPU architectural features. In particular, GPUs are originally designed for graphics rendering, instead of general purpose computing. Therefore, GPUs are specialized for compute-intensive and highly parallel applications, especially in the SIMD style parallelism. Furthermore, as a co-processor, the GPU relies on the CPU for memory allocation. As such, the common practice for efficiency is to allocate the GPU memory statically before initiating the GPU computation kernel and to avoid dynamic allocation or reallocation during the GPU kernel execution. Additionally, due to the limited bus bandwidth between the GPU memory and the CPU memory, it is best to eliminate frequent, small-sized data transfers between the CPU and the GPU.

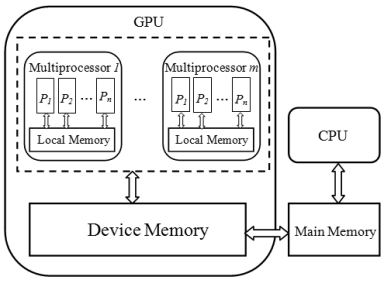


Figure 1: The many-core architecture model of the GPU

Recently, GPU-based primitives as the building blocks for higher-level applications [18, 19, 30] have been proposed to further reduce the complexity of GPU programming. The parallel primitives [19] are a small set of common operations exploiting the architectural features of GPUs. We utilize *map*, *reduce*, and *prefix sum* primitives in our two FIM implementations. Following the previous studies [19], we improve our implementation using memory optimizations, including the local memory optimization for temporal locality, the coalesced access optimization of device memory for spatial locality, and the built-in vector data type to reduce the number of memory accesses. Different from the previous work, we study the GPU acceleration of *Apriori* for FIM, which incurs much more complex control flows and memory accesses than performing database joins [19] or maintaining quantiles from data streams [15].

2.2 Frequent Itemset Mining

The Frequent Itemset Mining (FIM) problem was introduced by Agrawal et al. [2], as the first step to mine association rules in market basket data. Let $I = \{I_1, I_2, ..., I_m\}$ be a set of m items, and $T = \{T_1, T_2, ..., T_n\}$ the transaction database, where T_i is a transaction containing a set of items from I. An k-itemset that consists of k items from I, is frequent if it occurs in T not less than s times, where s is a user-specified minimum support threshold, and $s \leq n$. We denote s/n as *minsup*. The FIM problem is to find all itemsets in a given transaction database that occur more frequently than *minsup*.

There are two representative algorithms for mining frequent itemsets, namely, *Apriori* [3] and FP-growth [16]. *Apriori* iteratively mines frequent 1-itemsets, 2-itemsets, ..., until K-itemsets, where K is the maximum number of items of an frequent itemset. In each iteration, the algorithm generates candidate itemsets, or candidates, and counts the support for each candidate by scanning all transactions. In comparison, FP-growth works through divide-and-conquer. It recursively constructs a *conditional database* and a *conditional FP-tree*, and mines the FP-tree in a pattern growth method, by the concatenation of the suffix pattern to the

frequent patterns generated from the precedent conditional FP-tree. The advantage of FP-growth is that it avoids generating a number of candidates as well as repeated scanning of the transaction database.

FIM has been widely studied in distributed systems [4, 7, 10, 24]. Aouad et al. [4] designed a distributed *Apriori* in heterogeneous computer cluster and grid environments using dynamic workload management to tackle memory constraints, achieve balanced workloads, and reduce communication costs. Buehrer [7] and El-Hajj [10] proposed variants of FP-growth on computer clusters, lowering communication costs and improving cache, memory, and I/O utilization. Most recently, Li et al. [24] demonstrated a linear speedup of FP-growth on thousands of distributed machines using Google's MapReduce infrastructure.

Researchers have also studied FIM problems on modern CPUs. The key issue is how to fully exploit the instruction-level parallelism (ILP) and thread-level parallelism (TLP) on the multi-core CPU. Ghoting [11] et al. improved FP-growth [16] through a cache-conscious prefix tree for spatial locality and ILP, and a tiling strategy for temporal locality. Liu et al. [25] proposed a cache-conscious FP-array from compacting the FP-tree [16] and a lock-free, dataset-tiling tree construction algorithm for TLP. Ye et al. [31] explored the parallelization of Bodon's trie-based *Apriori* algorithm [6] with a database partitioning method. Recently, two benchmarks for mining on multi-core processors, including the PARSEC Benchmark Suite [5] and NU-MineBench [29], have been proposed to facilitate architectural studies.

In comparison to previous parallel CPU-based FIM algorithms, our algorithms are designed for the GPU with massive SIMD parallelism, instead of distributed systems and multi-core CPUs. In the literature, other parallel *Apriori* algorithms focus on I/O performance, while our GPU-based algorithms are in-memory, exploiting the SIMD architectural feature provided by GPUs.

3. IMPLEMENTATION

In this section, we present the design and implementation of our two GPU-based *Apriori* algorithms: the Pure Bitmap-based Implementation (PBI) and the Trie-based Implementation (TBI). Both implementations exploit the bitmap representation of transactions, which facilitates fast set intersection to obtain transactions containing a particular itemset. Furthermore, together with a lookup table, the bitmap representation also accelerates support counting, which is a time-consuming component in *Apriori*. PBI uses bitmap data structure to represent itemsets, while the TBI uses a trie. In particular, we put the trie on the CPU to perform trie traversal and incremental maintenance for efficiency. We implemented PBI and TBI on NVIDIA CUDA.

3.1 Overview

Both of our PBI and TBI implementations follow the workflow of the original *Apriori* algorithm, as shown in Algorithm 1. In the algorithm, we first generate all frequent items, or 1-itemsets. Next, we iteratively invoke **Candidate_Generation** to generate candidate K-itemsets, and then perform support counting in **Freq_Itemset_Generation** to generate frequent K-itemsets, where $K > 1$. K increments after each iteration. Both **Candidate_Generation** and **Freq_Itemset_Generation** can have different implementations.

Algorithm 1 *Apriori*

```
1: //C_K: Candidate K-itemsets.
2: //L_K: Frequent K-itemsets.
3: //T: Transaction database
4: Generate all frequent items L_1
5: K = 2
6: while L_{K-1} ≠ ∅ do
7:    //Generate candidate K-itemsets
8:    C_K = Candidate_Generation(L_{K-1})
9:    //Count supports and generate frequent K-itemsets
10:   L_K = Freq_Itemset_Generation(C_K, T, minsup)
11:   K = K + 1
12: end while
```

In the *Apriori* algorithm, there are two major data structures. One represents transactions, and the other represents itemsets. Both of our GPU-based implementations adopt a bitmap data structure to represent transactions, and both invoke the **Freq_Itemset_Generation** procedure in Algorithm 1 entirely on the GPU. The PBI implementation represents itemsets in another bitmap, and executes **Candidate_Generation** on the GPU. In comparison, the TBI implementation represents itemsets in a trie, and utilizes the CPU to help traverse and build the trie.

3.2 Bitmap and Support Counting

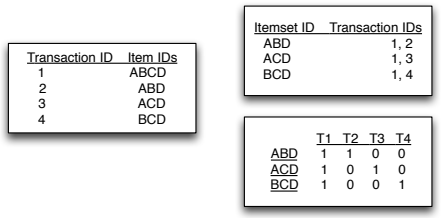


Figure 2: Horizontal data layout (left), vertical data layout (top right), and bitmap representation (bottom right).

There are two choices to represent the transactions, namely, horizontal and vertical data layouts [33]. In the horizontal layout, each transaction has a transaction identifier, followed by a list of items in a predefined order. In the vertical layout, each itemset has an itemset identifier, followed by a list of transactions containing that itemset. We denote the transaction list of a K-itemset as a K-tranlist. Figure 2 shows an example of the horizontal and vertical data layouts, together with the corresponding bitmap structure.

Traditionally, a CPU-based *Apriori* implementation adopts horizontal data layout. However, such layout requires scanning all transactions to perform support counting, which limits the data parallelism of GPUs. Therefore, we adopt the vertical data layout instead. We intersect two $(K-1)$-tranlists to obtain a K-tranlist *TID_LIST* for a particular K-itemset IK. Next, we count the number of transactions in the *TID_LIST* as the support of IK. Such intersection-and-counting process for generating a K-itemset is independent from one another, so that we can easily parallelize the procedure for generating different frequent itemsets.

To further improve the intersection operation and sup-

port counting, we store the vertical data layout in a bitmap, which is an array of bits. We refer it as a *transaction-bitmap*. In an $m \times n$ *transaction-bitmap*, where m is the number of items and n is the number of transactions, bit (i, j) is set to 1 if item i occurs in transaction j. We store a *transaction-bitmap* in the built-in vector data type int4 (a structure containing four 32-bit integers), which is of size 16 bytes, because the GPU can read up to 16 bytes of data from the device memory to registers in one instruction. This way, we can reduce the number of device memory accesses by a factor of four, compared with reading data in the granularity of 32-bit integers. Each row of a *transaction-bitmap* is rounded in 16 bytes, with the last 128 bits padded with 0, if it is less than 128 bits. Thus, the row vector in a *transaction-bitmap* is of size $\lceil n/128 \rceil \times 16$ bytes. We transform the support counting into intersection of row vectors of the *transaction-bitmap*, followed by counting of the number of 1's in the intersection result.

We construct a lookup table that stores the mapping of an integer and the number of 1's in its binary representation. For example, the number of 1's in 00000001 11000000 (448 in the decimal form) is 3. This lookup table is read-only. Since accessed frequently, we put it in the read-only, cacheable constant memory on the GPU. The constant memory can achieve as low as one cycle memory access latency. In comparison, accessing device memory incurs hundreds of cycles. The size of constant memory, 64KB, constrains the size of our lookup table to be (2^{16} entries $\times 1$ byte/entry) $= 65536$ bytes. For each lookup, we can obtain the number of 1's of a 16-bit integer.

Algorithm 2 shows **Freq_Itemset_Generation**, which runs entirely on the GPU, and is invoked by both PBI and TBI. Each GPU thread block processes one candidate K-itemset in parallel. Threads within the same thread block intersect two $(K-1)$-tranlists, count the number of 1's for every 16 bits in K-tranlist, and add up all counts using parallel reduce.

Figure 3 illustrates an example for support counting within a particular thread block. In this example, there are two threads in the thread block. For ease of representation, we assume that the data type *int* is of size 8 bits, so the vector *int4* is of size 32 bits. At the beginning, each thread reads two int4 vectors from two $(K-1)$-tranlists respectively, and performs bitwise AND operation on these two int4 vectors. Next, each thread queries the lookup table to obtain the counts of 1's for every 16 bits of the intersection result. Finally, we synchronize all threads in the same thread block, and perform parallel reduce to add up the counts as the support for the K-itemset.

Algorithm 2 Freq_Itemset_Generation

```
1: for each candidate K-itemset in parallel do
2:     Intersect two (K − 1)-tranlists in parallel
3:     Query the lookup table to count the number of 1's for
       every 16 bits in K-tranlist in parallel
4:     Perform parallel reduce to add up the counts of ev-
       ery 16 bits and obtain the support for K-itemset.
5:     if support of K-itemset ≥ minimum support then
6:         Output K-itemset
7:     end if
8: end for
```

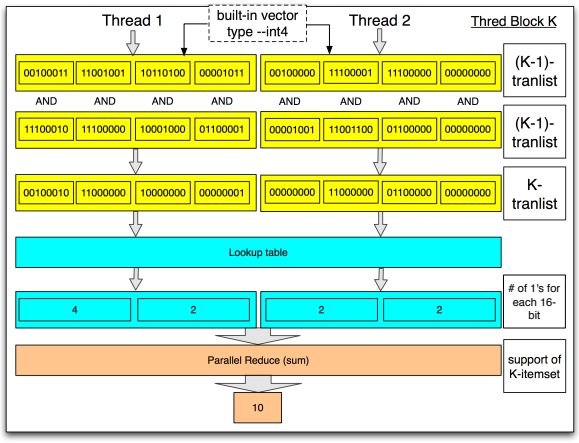


Figure 3: Support counting within a thread block.

3.3 Pure Bitmap Implementation

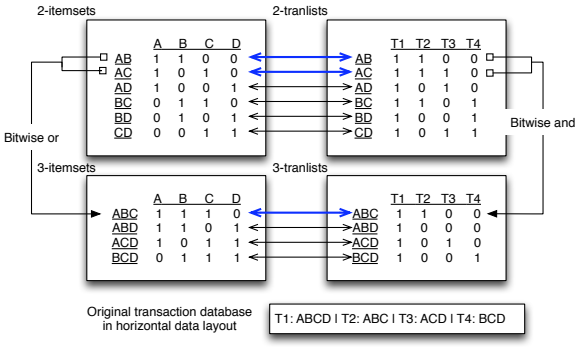


Figure 4: Generating candidate 3-itemsets from frequent 2-itemsets in PBI.

In the Pure Bitmap Implementation (PBI), we represent itemsets in a bitmap. In an $m \times n$ bitmap representing K-itemsets, where m is the number of K-itemsets and n is the number of all items, bit (i, j) is set to 1 if itemset i contains item j. Each row is also rounded in 16 bytes. We impose a lexicographical order among all K-itemsets.

The **Candidate_Generation** procedure consists of two steps, namely, a join to generate a candidate K-itemset from two $(K-1)$-itemsets, and a pruning to select the candidate K-itemset whose $(K-1)$-subsets are all frequent. Algorithm 3 shows the **Candidate_Generation** procedure for PBI. We denote the i-th $(K-1)$-itemset as L_i, and j-th $(K-1)$-itemset as L_j, where $i < j$. The k-th item in L_i is denoted as $L_i[k]$. Each GPU thread handles an L_i, and joins it with L_j. The join predicate is $(L_i[0] = L_j[0]) \wedge (L_i[1] = L_j[1]) \wedge \ldots \wedge (L_i[K-2] = L_j[K-2]) \wedge (L_i[K-1] < L_j[K-1])$.

In pruning, we check whether all $(K-1)$-subsets of a generated candidate K-itemset are frequent. We perform a binary search on a $(K-1)$-itemsets to determine if a $(K-1)$-subset of the candidate K-itemset is frequent. Figure 4 depicts an example for generating candidate 3-itemsets from frequent 2-itemsets in PBI. For example, in order to generate the candidate itemset ABC, we join two 2-itemsets AB and AC by performing a bitwise OR operation on the cor-

responding vectors in the bitmap of 2-itemsets. In the following **Freq_Itemset_Generation** procedure, we perform a bitwise AND operation to obtain the transaction list for candidate itemset ABC.

Candidate_Generation uses a bitmap to represent itemsets, which allows uniform and efficient bitwise operations to perform joins on the GPU, and avoids the overhead of frequent data transfer between GPU memory and CPU memory. However, when the number of items is large, it also incurs excessive non-coalesced device memory accesses. Given m frequent $(K-1)$-itemsets, and n items. In order to check whether one $(K-1)$-itemset is frequent, we need to access $(\log m \times \lceil n/128 \rceil \times 16)$ bytes of data, where $\log m$ is the cost of performing a binary search, and $\lceil n/128 \rceil \times 16$ is the size of a row (in bytes) in the bitmap of $(K-1)$-itemsets. Typically, if $m = 10000$ and $n = 10000$, we need to access about 16 KB for checking only one $(K-1)$-subset. This problem in our pure bitmap-based solution triggers us to consider adopting another data structure in the **Candidate_Generation** procedure in the presence of a large number of items.

Algorithm 3 PBI Candidate_Generation

```
1: //L_x represents the x-th (K − 1)-itemset, that is, the
   x-th row vector in the bitmap for (K − 1)-itemsets.
2: for each L_i in parallel do
3:   for each L_j where j = i + 1 to m do
4:     if L_i and L_j are joinable then
5:       //Join
6:       Union on L_i and L_j to obtain a candidate K-
         itemeset by performing a bitwise OR operation
7:       //Pruning
8:       (K − 1)-subset test on the candidate K-itemset
         by a binary search in the (K−1)-itemset bitmap.
9:     else
10:      break
11:    end if
12:  end for
13: end for
```

3.4 Trie-Based Implementation

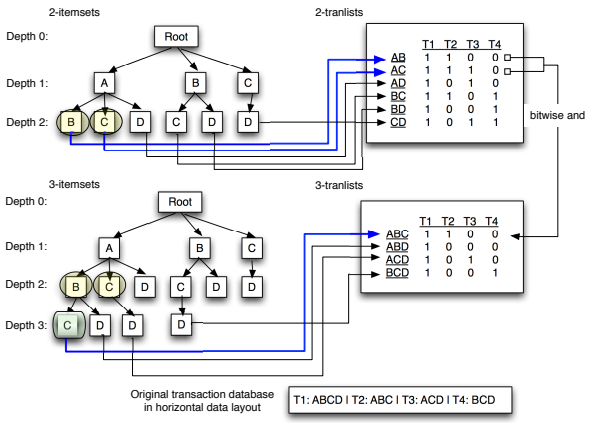


Figure 5: Generating candidate 3-itemsets from frequent 2-itemsets in TBI.

Instead of using bitmap to represent itemsets, we adopt the trie data structure, which is also used in the state-of-the-art *Apriori* implementation [6]. A trie is a rooted, directed prefix tree. The root is defined to be at depth 0. If a node is at depth K, then its children are at depth $K+1$. Each node stores an item id. A node at depth K concatenating all its ancestors represents an K-itemset. The trie-based *Apriori* implementation on the CPU [6] stores the support in each node, and counts support by scanning the transactions in the horizontal data layout. For each transaction, it finds paths from the root to the leaves in the trie corresponding to candidate itemsets contained in the transaction, and the support values of these leaves are all increased by one. Different from the CPU trie-based implementation, we represent transactions in a bitmap, and perform support counting on the GPU, as described in Section 3.2.

The candidate generation based on trie traversal is implemented on the CPU. This decision is based on the fact that, the trie is an irregular structure and difficult to share among SIMD threads. Thus, we store the trie representing itemsets in the CPU memory, and the bitmap representation of transactions in the GPU device memory.

We incrementally construct the trie level by level, which matches the iterative process of *Apriori*. By growing the trie to depth K, we generate all the frequent K-itemsets. Algorithm 4 shows the **Candidate_Generation** procedure in TBI. We perform join for every node at depth $K-1$ with each of its right siblings. We keep all children of a node sorted in lexicographical order on item id, so that we can efficiently check whether a $(K-1)$-subset is frequent by performing a series of binary searches to follow a path with the same prefix as the $(K-1)$-subset. In each iteration, after generating all candidate K-itemsets on the CPU, we transfer the bookkeeping data to the GPU memory for support counting. The bookkeeping data include a set of triples, in the form of $(I^i_{K-1}, I^j_{K-1}, I_K)$, where I^i_{K-1} and I^j_{K-1} are two $(K-1)$-itemsets generating the candidate K-itemset I_K. After the support counting on the GPU, we need to transfer the set of bookkeeping data to eliminate candidate K-itemsets. If the K-itemset I_K is a false candidate, then the triple is set to $(null, null, I_K)$. At this moment, we obtain all the frequent K-itemset in the trie, and start the next iteration of generating $(K+1)$-itemset, if any.

Figure 5 shows the same example as Figure 4, generating candidate 3-itemsets from 2-itemsets. In order to generate the candidate ABC, we need to join AB and AC, which are represented by the leftmost node B and the second leftmost node C at depth 2 respectively. Next, we test all 2-subsets other than AB and AC for ABC, which include only BC. We follow the path with prefix BC, and find this 2-itemset. Finally, we keep the candidate ABC for further support counting in the **Freq_Itemset_Generation** procedure. Note that, each leaf node in the trie is associated with a row vector in the bitmap representing transactions, so that we can easily perform a bitwise AND operation on two 2-tranlists to obtain a 3-tranlist, and then count the support.

4. EVALUATION

In this section, we present experimental results on evaluating our two GPU-based *Apriori* implementations.

4.1 Experimental Setup

Our experiments were performed on a PC with an NVIDIA GTX 280 GPU and an Intel Core2 quad-core CPU, running

Algorithm 4 TBI Candidate_Generation

```
1: //u represents a node at depth K − 1 in the trie.
2: for each u at depth K − 1 do
3:    for each w that is a right sibling of u do
4:       //Join
5:       Union on the two (K − 1)-itemsets represented by
         u and w to obtain a candidate K-itemeset
6:       //Pruning
7:       (K − 1)-subset test on the candidate K-itemset by
         following the path of the trie with the same prefix
8:    end for
9: end for
```

on Microsoft Windows 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 of size 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 version of CUDA is 2.0.

Comparison. We compared our GPU-based algorithms with three CPU-based *Apriori* and one CPU-based FP-growth, since there is no any GPU-based *Apriori* or FP-growth implementation in the public domain. A single-threaded CPU-based implementation is from the repository of Workshop on Frequent Itemset Mining Implementations (FIMI'03) [20], which is the best *Apriori* implementation, denoted as *BORGELT*. There is not any multi-threaded *Apriori* implementation publicly available, so we decided to parallelize one by ourselves. *BORGELT* uses a trie to represent transactions, and performs support counting recursively. Thus, it is quite tricky to parallelize *BORGELT*. Instead, we parallelized another famous single-threaded CPU-based *Apriori* implementation from Bart Goethals [21], which stores transactions in horizontal data layout. For this implementation, we parallelized the support counting step using OpenMP, and the parallelized version running on a quad-core CPU is more than three times faster than the serial version. We denote the parallelized implementation from Goethals as *GOETHALS*. Furthermore, we ported our TBI implementation to the CPU, and parallelized the for-loop in the support counting part (Algorithm 2) using OpenMP, and we denote it as *TBI-CPU*. Our two GPU-based *Apriori* implementations are denoted as *PBI-GPU* and *TBI-GPU* respectively. Table 1 summarizes the characteristics of the two GPU-based, and the three CPU-based *Apriori* implementations. The CPU-based FP-growth implementation is from PARSEC benchmark [5], which is implemented in OpenMP, denoted as *FP-GROWTH*. All multi-threaded CPU-based algorithms ran on four CPU threads.

Datasets. We used three representative datasets from FIMI'03 repository [20] to evaluate the five *Apriori* implementations, including *T40I10D100K*, *Chess*, and *Retail*. These three datasets have distinct characteristics from one another, which are summarized in Table 2. *T40I10D100K* is a synthetic dataset simulating market basket data. *Chess* and *Retail* are real-world datasets. The density of a dataset is defined to be the average length of transactions divided by the number of items. *Chess* is the representative of dense data, whose density is 49%, the highest among all datasets in FIMI'03 repository. *Retail* represents sparse data, whose density is lower than 1%. The implementations PBI-GPU, TBI-GPU, and TBI-CPU require transaction data to be in bitmap data structure. For sparse data, the bitmap representation of transactions in vertical layout is larger than the original one in horizontal data layout (180 MB vs 4 MB for *Retail*). However, for dense data, the bitmap representation compresses transaction data (30 KB vs 335 KB for *Chess*). We refer the reader to the FIMI'03 report [12] for the complete experimental results of various FIM implementations on all datasets.

Metric. We measured the total elapsed time for evaluating the efficiency of all the implementations. Since we are focusing on in-memory performance, we excluded the initial file input and final result output from the total time measurement. In addition, we exclude the time for converting the transaction database from horizontal data layout into bitmap representation, since the conversion can be performed offline or we can collect the source data and store them in bitmap initially. 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%.

4.2 Results

4.2.1 *Comparison to CPU-based* Apriori

Figure 6(a) depicts the running time for the five *Apriori* implementations on the synthetic dataset *T40I10D100K*, Figure 7(a) on the dense dataset *Chess*, and Figure 8(a) on the sparse dataset *Retail*. On these three datasets, our GPU-based implementations outperform the parallelized *GOETHALS* by a factor of 2.7 to 130, and the best *Apriori* implementation *BORGELT* by a factor of 1.2 to 24, when *minsup* varies. The GPU-based implementations have larger speedup over the CPU-based ones on the dense dataset than on the sparse dataset.

The time for data transfer between the GPU memory and the CPU memory (TRANSFER), candidate generation (CANDIDATE), and support counting (COUNTING) dominates the total running time. Thus, we break the total running time into three parts - TRANSFER, CANDIDATE, and COUNTING, and present the time breakdown result in Figure 6(b) on the synthetic dataset *T40I10D100K*, Figure 7(b) on the dense dataset *Chess*, and Figure 8(b) on the sparse dataset *Retail*.

Let us analyze the performance divergence of the five *Apriori* implementations, according to the running time and time breakdown results on the three datasets in Figure 6, Figure 7, and Figure 8.

TBI-CPU vs GOETHALS. This comparison shows the impact of bitmap representation for the transaction database. Both implementations adopt a trie to represent itemsets, thus they have roughly the same performance on candidate generation step. However, the vertical layout for the transaction database allows *TBI-CPU* to perform independent support counting on different candidate itemsets, which extracts the multi-threaded parallelism to maximum. On the other hand, *GOETHALS* uses horizontal layout to store transactions, so that it should repeatedly scan the whole transaction database to do support counting. From the

Implementation	Platform	Candidate Generation	Support Counting	Itemsets	Transactions
PBI-GPU	GPU	Multi-threaded on the GPU	Multi-threaded on the GPU	Bitmap	Bitmap
TBI-GPU	GPU+CPU	Single-threaded on the CPU	Multi-threaded on the GPU	Trie	Bitmap
TBI-CPU	CPU	Single-threaded on the CPU	Multi-threaded on the CPU	Trie	Bitmap
GOETHALS	CPU	Single-threaded on the CPU	Multi-threaded on the CPU	Trie	Horizontal layout
BORGELT	CPU	Single-threaded on the CPU	Single-threaded on the CPU	Trie	Trie

Table 1: Five *Apriori* Implementations

Dataset	#Item	Avg. Length	#Transactions	Density	Characteristics	Data size	Bitmap size
T40I10D100K	1,000	40	100,000	4%	Synthetic	$\sim$ 15 MB	$\sim$ 12 MB
Retail	16,469	10.3	88,162	0.6%	Sparse/Real	$\sim$ 4 MB	$\sim$ 180 MB
Chess	75	37	3,196	49%	Dense/Real	$\sim$ 335 KB	$\sim$ 30 KB

Table 2: Three experimental datasets

time breakdown result, we can see that, *GOETHALS* always has a larger ratio of support counting time. However, on the sparse dataset *Retail*, TBI-CPU only outperforms *GOETHALS* by a factor of 1.28, due to the large size of bitmap representation of the sparse transaction database. Even though TBI-CPU does not need to scan the whole bitmap for support counting, accessing a part of large bitmap (150 MB for *Retail*) may be as costly as scanning the whole transaction database with small size (4 MB for *Retail*).

TBI-GPU vs TBI-CPU. This comparison investigates the impact of the GPU acceleration for support counting. *TBI-GPU* differs from *TBI-CPU* only in the support counting step. Although *TBI-GPU* suffers from intermediate data transfer between the GPU memory and the CPU memory, it gains significant performance from the massive SIMD parallelism provided by the GPU. Especially for the sparse dataset *Retail*, *TBI-GPU* has 7.8x speedup over *TBI-CPU*. Huge bitmap representation for the sparse dataset requires more memory accesses than that of dense dataset. In this case, *TBI-GPU* is able to hide large memory latency by well utilizing massive SIMD parallelism on the GPU. Since our study focuses on the GPU-based implementation, we haven't exploited data locality in CPU cache for *TBI-CPU*.

PBI-GPU vs TBI-GPU. This comparison shows the effect of different itemset representations - bitmap-based and trie-based. PBI-GPU and TBI-GPU invokes exactly the same support counting procedure on the GPU. The performance difference only comes from the candidate generation. The dense dataset *Chess* has very few items (75 in total), hence the bitmap representation of itemsets for PBI-GPU is of small size. On the other hand, the sparse dataset *Retail* has many items (16469 in total), so PBI-GPU should process large bitmap of itemsets. Thus, the number of items determines the performance of PBI-GPU's candidate generation. Therefore, PBI-GPU outperforms TBI-GPU on the dense dataset, due to smaller size of bitmap representation for itemsets, while TBI-GPU is better on the sparse dataset.

PBI-GPU/TBI-GPU vs BORGELT. On all datasets with different *minsup*, both GPU-based implementations win over the best CPU-based *Apriori* implementation of FIMI'03, except that *PBI-GPU* is 20% slower than *BOGELT* on the sparse dataset *Retail* with *minsup* 0.01%. The bitmap structure for representing transactions helps the GPU SIMD parallelism, and boosts the performance of both GPU-based implementations.

To sum up, our both GPU-based implementations outperform CPU-based implementations in up to an order of magnitude on the sparse dataset, and up to two orders of magnitude on the dense dataset. The two GPU-based implementations gain performance from the bitmap representation of transactions. *PBI-GPU* outperforms *TBI-GPU* on the dense dataset, while *TBI-GPU* is better on the sparse dataset.

4.2.2 Comparison to CPU-based FP-growth

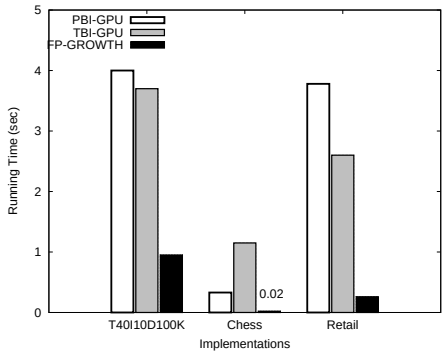
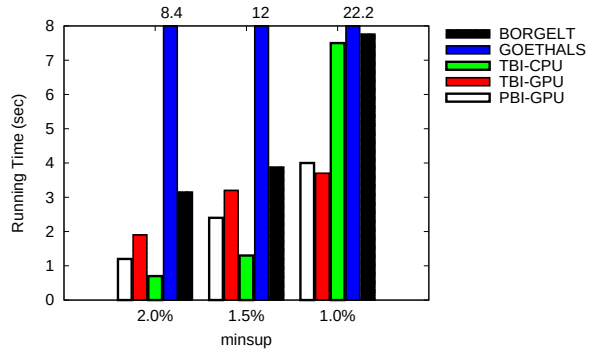


Figure 9: Execution time of PBI-GPU, TBI-GPU, and CPU-based FP-growth on T40I10D100K, Chess, and Retail with *minsup* 1%, 60%, and 0.01%.

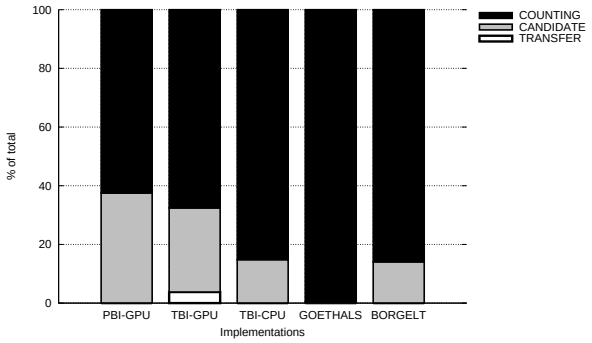
Figure 9 illustrates the running time of FP-GROWTH, PBI-GPU, and TBI-GPU on *T40I10D100K*, *Chess*, and *Retail* with *minsup* 1%, 60%, and 0.01% respectively. We can see that CPU-based FP-growth is faster than our both GPU-based implementations by a factor of 4 to 16. This leaves us enough room to explore more efficient GPU-based FIM algorithms.

5. CONCLUSION AND FUTURE WORK

We have presented two GPU-based implementations of *Apriori* algorithm for frequent itemset mining. Both implementations employ a bitmap data structure to encode the transaction database on the GPU and utilize the GPU's SIMD parallelism for support counting. One implementation stores the itemsets in a bitmap, and runs entirely on the GPU. The other one utilizes a trie to store the itemsets, and adopts a GPU-CPU co-processing scheme. The preliminary evaluation results show that both of our GPU-

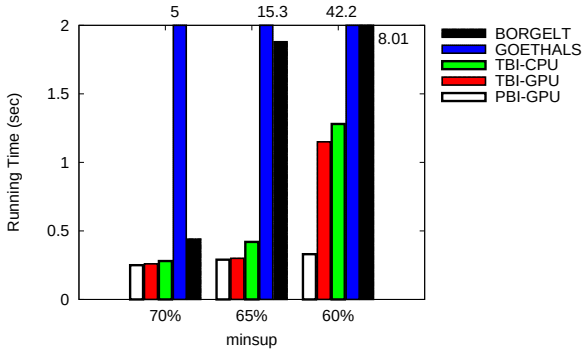


(a) Running time with various minsup

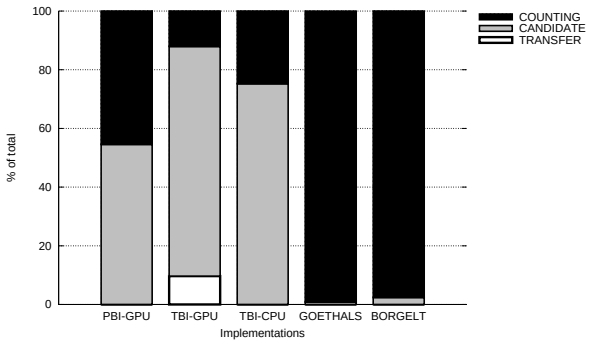


(b) Time breakdown with minsup 1%

Figure 6: Experiments on the synthetic dataset *T40I10D100K*



(a) Running time with various minsup



(b) Time breakdown with minsup 60%

Figure 7: Experiments on the dense dataset *Chess*

implementations are up to two orders of magnitude faster than optimized CPU-based implementations.

We are considering improvements of our current implementations. For example, our bitmap representation of transactions is space inefficient for sparse datasets. We are investigating data compression techniques [1]. Moreover, we are developing a buffering mechanism between the GPU memory and the CPU memory for memory ping-pong.

We also plan to explore other mining algorithms with GPU acceleration, for instance, FP-growth and classification. 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 support thresholds during the progress. Such interaction can greatly improve the mining quality.

Acknowledgments

The authors thank the anonymous reviewers for their insightful suggestions. This work was supported by grant 617208 from the Hong Kong Research Grants Council.

6. REFERENCES

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. *SIGMOD*, 2006.

[2] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD*, 1993.

[3] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. *VLDB*, 1994.

[4] Lamine M. Aouad, Nhien-An Le-Khac, and Tahar M. Kechadi. Distributed frequent itemsets mining in heterogeneous platforms. *Journal of Engineering, Computing and Architecture*, 2007.

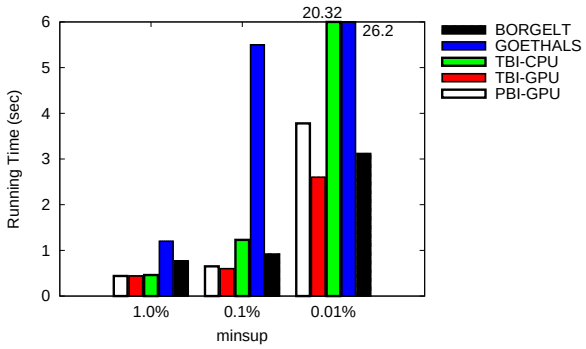
[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. *PACT*, 2008.

[6] Ferenc Bodon. A fast apriori implementation. *FIMI*, 2003.

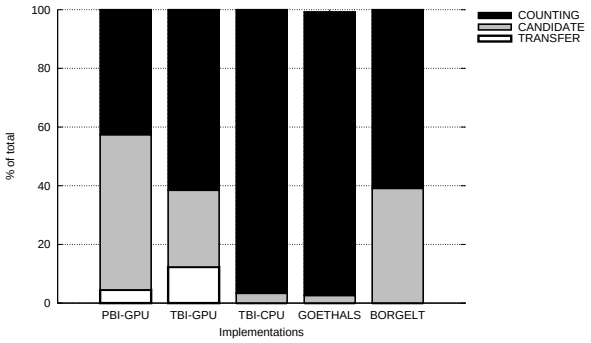
[7] Gregory Buehrer, Srinivasan Parthasarathy, Shirish Tatikonda, Tahsin Kurc, and Joel Saltz. Toward terabyte pattern mining: an architecture-conscious solution. *PPoPP*, 2007.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of parallel and Distributed Computing*, 2008.

[9] Shengnan Cong, Jiawei Han, Jay Hoeflinger, and David Padua. A sampling-based framework for



(a) Running time with various minsup



(b) Time breakdown with minsup 0.01%

Figure 8: Experiments on the sparse dataset *Retail*

parallel data mining. *PPoPP*, 2005.

[10] Mohammad El-Hajj and Osmar R. Zaiane. Parallel leap: Large-scale maximal pattern mining in a distributed environment. *ICPADS*, 2006.

[11] Amol Ghoting, Gregory Buehrer, Srinivasan Parthasarathy, Daehyun Kim, Anthony Nguyen, Yen-Kuang Chen, and Pradeep Dubey. Cache-conscious frequent pattern mining on a modern processor. *VLDB*, 2005.

[12] Bart Goethals and Mohammed Javeed Zaki. Advances in frequent itemset mining implementations: Introduction to fimi'03. *FIMI*, 2003.

[13] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. *SIGMOD*, 2006.

[14] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. *SIGMOD*, 2004.

[15] Naga K. Govindaraju, Nikunj Raghuvanshi, and Dinesh Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. *SIGMOD*, 2005.

[16] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 2004.

[17] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. *PACT*, 2008.

[18] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient gather and scatter operations on graphics processors. *Supercomputing*, 2007.

[19] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational joins on graphics processors. *SIGMOD*, 2008.

[20] http://fimi.cs.helsinki.fi/. *FIMI repository.*

[21] http://www.adrem.ua.ac.be/goethals/software/files/apriori.tgz. *Apriori implementation from Bart Goethals.*

[22] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 1999.

[23] E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. *Supercomputing*, 2001.

[24] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. Pfp: Parallel fp-growth for query recommendation. *ACM Recommender Systems*, 2008.

[25] Li Liu, Eric Li, Yimin Zhang, and Zhizhong Tang. Optimization of frequent itemset mining on multiple-core processor. *VLDB*, 2007.

[26] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krĺźger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer Graphics Forum*, 2007.

[27] Lance Parsons, Ehtesham Haque, and Huan Liu. Evaluating subspace clustering algorithms. *SDM*, 2004.

[28] S. Parthasarathy, M. J. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared memory systems. In *Knowledge and Information Systems*, 2001.

[29] Jayaprakash Pisharath, Ying Liu, Wei keng Liao, Alok Choudhary, Gokhan Memik, and Janaki Parhi. Nu-minebench 2.0. Technical report, Northwestern University, 2005.

[30] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *Graphics Hardware*, 2007.

[31] Yanbin Ye and Chia-Chu Chiang. A parallel apriori algorithm for frequent itemsets mining. *SERA*, 2006.

[32] Mohammed J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 1999.

[33] Mohammed J Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. *KDD*, 1997.

[34] Jingren Zhou and Kenneth A. Ross. Implementing database operations using simd instructions. *SIGMOD*, 2002.