2012

Parallel Trie-based Frequent Itemset Mining on Graphics Processors

Jay Junjie Yao
CUNY City College

Parallel Trie-based Frequent Itemset Mining on Graphics Processors

Thesis

Submitted In Partial Fulfillment of the Requirements for the Degree of

Master of Science

(Computer Science)

At

The City College of New York

of the

City University of New York

By

Jay Junjie Yao

May 2012

Approved:

____

Professor Jianting Zhang, Thesis Advisor

____

Professor Douglas Troeger,

Chairman of Department of Computer Science

ABSTRACT

The purpose of this thesis is to design and implement new data structures and algorithms for Frequent Itemset Mining (FIM). Leveraging the trie-based bitmap representation of a Frequent Itemset originally presented in Fang et al 2009 [3] and the latest General Purpose Computing on GPU technology (GPGPU), we improve the efficiency of the trie-based FIM implementation of Apriori [1]. We analyze two strategies in generating a Frequent Itemset Trie, including Depth-First Search (DFS) and Breadth-First Search (BFS) of trie traversals. These techniques induce different trade-offs in terms of the memory I/O accesses and runtimes. We use a hierarchical structure of Itemset Trie to project the Frequent K-Itemsets at each of the paths on the trie. In the comparison to previous CPU-based FIM algorithms, our algorithms are designed for the GPUs with massive Single Instruction, Multiple Data (SIMD) parallelism instead of distributed systems and multi-core CPUs. This research is unique in the sense that, while a few existing parallel Apriori algorithms focus on improving I/O performance, our GPU-based algorithms are in-memory and utilize the SIMD architectural feature provided by modern GPUs. Our preliminary results show that we are able to achieve an overall speed improvement up to 2 times over an optimized parallel Apriori FIM algorithm, up to 9 times faster in Frequent 2-Itemsets Generation and 50 times faster in Candidate 1-Itemsets Support Counting on a large dataset with 100,000 transactions and a minimum support of 1%.

Table of Content

Table of Figures

List of Algorithms

List of Tables

1. Introduction

Frequent Itemset Mining (FIM) is a classical and important problem in data mining. FIM was first introduced in 1998 by Rakesh Agrawal and Ramakrishnan Srikant [1]. FIM finds all itemsets with transaction supports that are larger or equal to a user-specified minimum threshold. The support of a Frequent Itemset is the percentage of data transactions that contain the itemset. A typical example of such itemset is that customers typically rent “Star Wars”, “Empire Strike Back” and “Return of the Jedi” together. Denote K to be the length of an itemset, a K-Itemset is an itemset that has K items. If the support of a K-Itemset is above the user-defined threshold, the itemset is a Frequent K-Itemset. A Frequent K-Itemset is called a maximal Frequent K-Itemset if it is not a subsequent of Frequent (K+N)-Itemsets, where N is larger than zero. Frequent Itemsets get larger when the given minimum support gets lower. The condensed representations of Frequent Itemsets include Closed Itemset [6], Approximate K-Set [7] and Weighted Itemset [8]. Additionally, the work reported in [9] focused on discovering a minimal set of unexpected itemsets. In these methods, transaction dataset is stored in a secondary storage so that multiple scans over the dataset can be performed. Traditional FIM approaches have mainly considered the problem of mining static transaction databases. Three kinds of FIM approaches over static databases have been proposed: reading-based [2], writing-based [4], and pointer-based [5].

Based on the Apriori [1] principle, we design and implement an algorithm called Parallel Trie-based Frequent Itemset Mining (PTFIM) for FIM. Leveraging the trie-based bitmap representation of Frequent Itemset presented in [3] and the latest GPGPU technology, we improve the efficiency of the trie-based FIM. We analyze DFS and BFS of trie traversals in generating Frequent Itemset Tries. We use a hierarchical structure of Itemset Trie to project the Frequent Items on each node of the trie. Each path with K node, which starts from the nodes at the level 1 of the Frequent Itemset Trie, is mapped to a Frequent K-Itemset. We have tested our algorithm on various sizes of synthetic datasets and different minimum supports of each dataset. Our preliminary results show that PTFIM is almost 2 times faster than an optimized parallel Apriori FIM algorithm in overall performance, up to 9 times faster in Frequent 2-Itemset Generation and 50 times faster in Candidate 1-Itemset Support Counting on a large dataset with 100,000 transactions and a minimum support of 1%.

The rest of the thesis is organized as follows. In Chapter 2, we introduce the background of General Purpose computing on GPU (GPGPU), FIM preliminaries, the related work and discuss the possible improvements. In Chapter 3, we present our Parallel Trie-based Frequent Itemset Mining (PTFIM) by illustrating the processes of building transaction bitmap representation using GPUs and by demonstrating the data structures and implementations. In Chapter 4, we evaluate the performance of the proposed PTFIM approach with various synthetic datasets and compare the results with the state-of-the-art. In Chapter 5, we conclude the thesis and discuss the future work.

2. GPGPU Background and Related Work

2.1 General Purpose Computing on GPU (GPGPU)

A GPU can be viewed as a special parallel hardware device with a massively multi-threaded processor and high bandwidth between the processor and its device memory. Reasonably current GPUs that have been equipped with commodity personal computers are capable of performing general computing. These GPUs have hundreds of cores and can launch a large number of threads simultaneously. The concurrent bandwidth of transferring data between GPU global memory and CPU main memory is up to 8 GB/s. Furthermore, the memory bandwidth between GPU cores and GPU memories can be more than 100GB/s [14]. With these memory bandwidths, we can further explore efficient algorithms for FIM. There are mainly two kinds of GPU programming languages: graphics API such as OpenGL and GPGPU languages such as Compute Unified Device Architecture (CUDA) [10].

The latest NVIDIA CUDA programming framework has a graphics-free API and allows developers to write code in C/C++. A basic CUDA programming model is shown in Figure 1 [11]. CUDA exposes hardware features including the fast inter-processor communication via the local shared memory. CUDA introduces lightweight thread block. Threads within a same thread block are divided into SIMD groups, called warps.

Each warp typically contains 32 threads. The global memory of GPUs has a high bandwidth but high access latency. CUDA allows a thread block to copy required data from global memory to its local shared memory that can only be accessed by the threads within the thread block. A warp of threads can combine accesses to consecutive data items in the global memory into a single memory access transaction, called coalesced memory access.

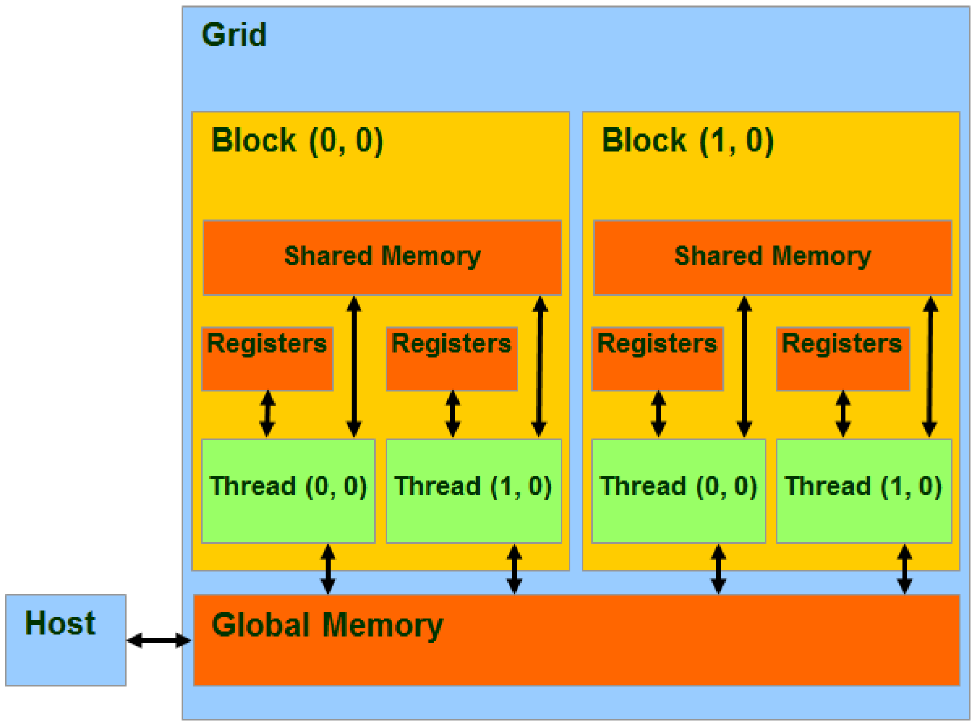

Figure 1. CUDA Programming Model [11]

While GPGPU programming framework is designed to reduce the complexity of parallel computing on GPGPUs, developers should be careful to design data structures and implement their algorithms in a way to fully utilize the GPGPU parallel architecture. It is better to utilize coalesced memory access to reduce the access latency between threads and the GPU global memory. Unlike memory caching on CPUs, coalesced memory access in CUDA can only be shared by threads in a same execution group, not by the next execution of the same thread [10]. When local variables are organized in consecutive 32-bit words, within a warp, threads with consecutive IDs can achieve coalesced memory accesses of consecutive memory addresses. For other common practice of efficiency, developers can pre-calculate the data in one-dimensional arrays and perform a one-time memory allocation and host-device memory transfer to avoid dynamic memory allocation or memory reallocation while the kernel functions are being executed on GPUs.

2.2 Related Work

2.2.1 Frequent Itemset Mining Algorithm and the Apriori Algorithm

Several algorithms have been proposed to find sequential Frequent Itemsets. Two representative FIM algorithms are Apriori [1] and FP-Growth [4]. Apriori iteratively mines Frequent 1-Itemset, 2-Itemset, …, until K-Itemset. In each iteration, Apriori generates Candidate K-Itemset and counts the support for each candidate by scanning all transactions. In comparison, FP-Growth mines the complete set of Frequent Itemsets without Candidate Itemsets Generation. FP-Growth transforms the problem of finding long frequent patterns into searching for shorter ones recursively and concatenating the suffix of frequent patterns. FP-Growth uses the least frequent items [4] as a suffix, which offers good selectivity. Performance study [12] demonstrates that FP-Growth implementation is generally an order of magnitude faster than Apriori. However, when the support is high on the dataset, an Apriori implementation might be faster. On the other hand, Apriori is easier to adapt for parallel architectures. The work reported in [3] is a typical parallel Apriori FIM implementation. It demonstrates that we can take the advantage of the massive computation power and high memory bandwidth of GPU to develop an efficient FIM algorithm base on Apriori. In the Apriori algorithm, the database is transformed by replacing the items in each transaction with a set of itemset IDs. Apriori can be demonstrated by splitting the problem of FIM into the following phases with a sample dataset in Figure 2.

A. Sort Phase

   In this phase, the given transaction database is sorted by the transaction IDs as a major key as shown in Figure 2. In the example, a transaction is defined by combining all the items bought by the same customer.

B. Frequent 1-Itemset Phase

   Apriori finds all the distinct items as the Candidate 1-Itemsets and starts support counting of Candidate 1-Itemsets to determine Frequent 1-Itemsets. Assume the user-defined support is 2, Figure 2 shows the Frequent 1-Itemsets, which are <1>, <2>, <3>, <4>, and <5>.

Transaction Id	Transaction Itemset	C1	F1 | Support
1	<1, 5>	<1>	<1> | 4
2	<1, 2, 3, 4>	<2>	<2> | 2
3	<1, 3>	<3>	<3> | 3
4	<1, 2, 3, 4, 5>	<4>	<4> | 2
5	<5>	<5>	<5> | 3

Figure 2. Transaction Database 1

C. Frequent Itemset Phase

Denote Frequent (K-1)-Itemsets to be F_{k-1} and Candidate K-Itemsets to be C_k. In this phase, Apriori uses F_{k-1} as the seeds to generate C_k. After sorting the item IDs in all F_{k-1}, Apriori selects a Frequent (K-1)-itemset $F_{k-1}(1)$ and looks for another Frequent (K-1)-Itemset $F_{k-1}(2)$ with the same first (K-2) items and takes the first (K-2) items into C_k. Then Apriori selects the last item in $F_{k-1}(1)$ and the last item in $F_{k-1}(2)$ into C_k. Apriori repeats the same procedures for all $F_{k-1.}$ By removing the duplicated itemsets in C_k, Apriori performs support counting and derives Frequent K-Itemsets. Figure 3 illustrates the Apriori Candidate 3-Itemsets Generation when the minimum support is 2.

C2	F2 | Support	C3
<1, 2> <1, 3> <1, 4> <1, 5>	<1 ,2> | 2 ; <1, 3> | 2; <1, 4> | 2; <1, 5> | 2	<1, 2, 3> <1, 2, 4> <1, 2, 5> <1, 3, 4> <1, 3, 5> <1, 4, 5>
<2, 3> <2, 4> <2, 5>	<2, 3> | 2; <2, 4> | 2;	<2 , 3, 4>
<3, 4> <3, 5>	<3, 4> | 2;	<3, 4, 5>
<4, 5>		

F3 | Support	C4	F4	C5
<1, 2, 3> | 2; <1, 2, 4> | 2; <1, 3, 4> | 2;	<1, 2, 3, 4>	<1, 2, 3, 4> | 2	Empty
<2, 3, 4> | 2;			

Figure 3. Frequent Itemset Phase Illustration

Apriori collects the Frequent K-Itemsets and passes them as the seeds for next Frequent Itemset Phase at the (K+1) level. If there are no Frequent K-Itemsets or Candidate (K+1)-Itemsets, Apriori goes to the next Maximal Phase.

D. Maximal Phase

In this phase, Apriori finds the maximal itemsets among the set of Frequent K-Itemsets (F). Let the length of the longest itemset in F to be k, the process is as follows:

```
For (k=0; k> 1; k--) do:
    For each $F_k$ in Frequent K-Itemsets do:
        From the table of all possible sub-itemsets of F,
            Delete $F_k$ that is in the table.
```

After the procedure terminates, the final maximal large itemsets of the sample dataset in Figure 2 are these three itemsets <1, 2, 3, 4>, <1, 3, 5>, and <4, 5>.

2.2.2 GPUMiner

Researchers have been studying FIM problems on modern CPU and GPU architectures. GPUMiner [3] proposed by Fang et al is one of the GPU-based implementations of the Apriori algorithm. GPUMiner uses an Itemset Trie to store the itemsets and adopts a bitmap data structure to represent transactions. GPUMiner builds and traverses the Candidate Itemset Trie and Frequent Itemset Trie on CPUs. By utilizing a pre-generated support count lookup table, GPUMiner generates Frequent 1-Itemsets and Frequent 2-Itemsets on CPUs and Frequent 3-Itemsets and up on GPUs.

There are two bitmap representations of transactions, i.e., the horizontal data layout and the vertical data layout. Denote the number of items as N and the number of transactions as M. In horizontal layout, a row maps to a transaction. Horizontal layout has N columns and each column maps to an item in an increasing order of item IDs. If an item j is in a transaction i, the corresponding bit (i, j) is 1, otherwise is 0. In vertical layout, a row maps to an item. Vertical layout has M columns with each column maps to a transaction in an increasing order of transaction IDs. If a transaction j contains an item i, the corresponding bit (i, j) is 1, otherwise is 0. Figure 4 shows the horizontal data layout and the vertical data layout of bitmap representations with an example dataset that has two transactions, <0, 2>, <0, 1>.

Item Id Trans Id	0	1	2	3
0	1	0	1	0
1	1	1	0	0

Horizontal Bitmap Layout

Trans Id Item Id	0	1
0	1	1
1	0	1
2	1	0
3	0	0

Vertical Bitmap Layout

Figure 4. Horizontal Layout vs. Vertical Layout

Using the horizontal data layout, FIM requires multiple scans of all transactions to perform Candidate Itemset Support Counting, which limits the data parallelism of GPUs. On the other hand, GPU-based Apriori algorithm adopts the vertical data layout in order to parallelize bit-wise AND operations on two item rows and support counting on an item row.

GPUMiner pre-generates a binary file with the transaction bitmap and all the necessary metadata information of the dataset: the number of transactions that is the width of the bitmap, the number of items that is the height of the bitmap, maximum length of items of all transactions and the bitmap representing the transaction dataset. GPUMiner uses CPU to generate Frequent 1-Itemsets and Frequent 2-Itemsets. Firstly, GPUMiner extracts all the metadata information of the dataset and the dataset bitmap from the pre-generated binary file, which have been discussed above. If 16 does not divides the number of the transactions in the dataset, GPUMiner pads each bitmap item row with zeros in order to have the number of bits of each row to be divided by 16.

GPUMiner constructs a lookup table for support counting of all Candidate K-Itemsets. The lookup table (LT) is an integer array whose indices are from 0 to 65,535, which is the largest 2-btye unsigned integer. The values of elements in the lookup table are the number of bits of their unsigned integer indices. For computing the support count on a candidate itemset row, GPUMiner obtains the number of supports of 16 transactions at a time by querying the lookup table. The lookup table is stored in CPU main memory and in GPU constant memory for support counting lookups. Figure 5 illustrates an example of computing the support count of a candidate itemset row (Cand) on a dataset that has 16 transactions. The user defined minimum support is 4. The Cand is the result of bitwise AND operation between a frequent itemset row (Parent) and a candidate item row (Child). The Cand is split into an array of 16-bits integers and each of them is mapped to an unsigned integer. In this case, the binary representation of Cand is 01011010 and thus the unsigned integer index of LT is 90. GPUMiner goes to LT[90] to retrieve the value as the support count for Cand which is 4. Since the support count satisfies the minimum support, GPUMiner stores the Cand to frequent itemset row (Freq).

Transaction Bitmap

TransIdx	**0**	**1**	**2**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Parent	0	1	1	0	1	0	0	0	1	0	1	1	0	1	0
Child	1	0	0	1	0	1	0	0	1	1	1	1	1	1	1
Cand	0	0	0	0	0	0	0	0	1	0	1	1	0	1	0
Freq	0	0	0	0	0	0	0	0	1	0	1	1	0	1	0

Lookup Table:

Cand in 8-bit Representation	**000000000**	**0000001**	**00000010**	**…**	**01011010**	**…**
LT_Idx	0	1	2	…	90	…
LT_Val	0	1	1	…	4	…

Figure 5. Support Counting using Lookup Table

GPUMiner adapts Itemset Tries to store the Candidate K-Itemsets and Frequent K-Itemsets. The root is defined to be at level 0. If a node is at the level K, its children are at the level (K+1). GPUMiner performs Candidate K-Itemset Generation on CPUs. At the level K on the Frequent K-Itemset Trie, each leaf node adds all its right siblings as children nodes with the itemset IDs at the level (K+1). The result Itemset Trie is the Candidate (K+1)-Itemset Trie.

GPUMiner does not store the Frequent (K-1)-Itemset Bitmap in GPUs. For each Candidate K-Itemset Support Counting on GPUs, GPUMiner performs a DFS trie traversal on the Candidate K-Itemset Trie from the left most candidate leaf item node to the right most candidate leaf item node. GPUMiner defines a boundary number (B). GPUMiner visits up to B nodes at each level from level 1 to level (K-1) at a time. While GPUMiner traverses on the Candidate K-Itemset Trie, GPUMiner only generates a set of Frequent (K-1)-Itemsets with B nodes to be transferred to GPUs. GPUMiner performs bitwise AND operation on the Frequent (K-1)-Itemsets and candidate items to generate Candidate K-Itemsets until it reaches the level of candidate items. The support counting of Candidate K-Itemsets with B nodes is calculated by using the lookup table on GPUs as discussed above. When GPUMiner detects that the current set of Frequent (K-1)-Itemsets with B nodes is no longer satisfied the needs of support counting for the next B candidate item nodes, GPUMiner recursively returns to the upper levels to seek another set of Frequent (K-1)-Itemsets with B nodes to cover the next B candidate item nodes

2.2.3 Issues and possible improvements of GPUMiner

For Candidate K-Itemset Support Counting on GPUs, GPUMiner uses a lookup table that stores the mappings among 16-bit integers and their numbers of 1-bits. This lookup table is read-only and is stored in the constant memory on GPUs. We can avoid accessing constant memory in support counting by using GPU integer arithmetic function __popc(), which return the number of 1bits in an 32-bit unsigned integer. The average throughput of integer arithmetic function __popc() is 32 operations per GPU clock cycle in CUDA 3.x and up [26]. Thus we can have significant speedup in support counting of candidate itemsets without using the lookup table mechanism.

On the other side, GPUMiner uses a DFS trie traversal on Candidate K-Itemset Trie for support counting. GPUMiner generates sets of Frequent (K-1)-Itemsets with B nodes on Candidate K-Itemset Trie to satisfy the need of the candidate items. For Frequent (K-1)-Itemset Generation, GPUMiner performs multiple GPU memory allocations and host-device memory transfers. Thus GPUMiner has a significant data transfer overhead by transferring Frequent (K-1)-Itemsets and candidate item rows back and forth between CPUs and GPUs. To improve this, we can pre-calculate all the necessary data arrays for the Candidate K-Itemset Support Counting including the data bitmaps of the Candidate K-Itemset and Frequent (K-1)-Itemset. Then we can perform a one-time memory allocation and host-device data transfer. After we finish support counting on GPUs, we only need to transfer the candidate items and support list arrays to CPUs to reduce data transfer overheads. We can also truncate the candidate data bitmap on GPUs for the next level support counting to further reduce data transfer overheads.

3. PTFIM Implementation

In this chapter, we will develop an algorithm with new design and implementation based on the Apriori principle and we term our algorithm as Parallel Trie-based Frequent Itemset Mining (PTFIM). PTFIM is a very computationally intensive application that makes it suitable for parallel execution on GPUs. PTFIM follows the workflow of Apriori discussed in Chapter 2. By taking advantage of the shared memory and large numbers of processors in GPU architecture, PTFIM can perform the FIM in a more efficient way. We use BFS in trie generation and traversal on the Frequent Itemset Trie. Using a hierarchical structure of item-based trie, we project Frequent K-itemsets to the paths from the root to the nodes at the level K on the trie. We save the trie structure in the CPU memory in order to perform trie traversal, nodes pruning and incremental maintenance. The Candidate K-Itemset Support Counting of PTFIM is implemented on GPUs for efficiency purposes.

This chapter is organized as the following. Section 3.1 presents the procedure of building the transaction bitmap from a raw dataset. Section 3.2 presents the overall data structure and algorithm designs. Section 3.3 and 3.4 provide details on GPU-based implementations, i.e., Candidate Itemset Support Counting and Candidate Bitmap Truncation.

3.1 Build Transaction Bitmap of PTFIM

Given a plaintext dataset with lines of transactions and each item ID separated by space or tab in the transaction, PTFIM first needs to transform the dataset to a bitmap representation and build a trie. As shown in Algorithm 1 listed in Figure 6, PTFIM uses a linked-list structure and scans the dataset line by line to transform transactions to the linked-list. While PTFIM is scanning the dataset file, PTFIM retrieves the additional information of the dataset, such as the minimum item ID, the maximum item ID, the number of transactions, the maximum number of items of all transactions, and the distinct item IDs. PTFIM computes the total number of distinct items and the total number of transactions in the dataset. PTFIM uses the information to build a bitmap of the dataset on GPUs.

The next step is to determine the width and height of the dataset bitmap. As we know the minimum and maximum ID of items, the height of the dataset bitmap is the result of (Max Item ID – Min Item ID + 1). The width of the bitmap is the number of transactions. In order to fully utilize GPU integer arithmetic function __popc() for compute the numbers of 1-bits of 32-bit unsigned integers in Candidate K-Itemset Support Counting, PTFIM appends with 0-bits at the last 32 bits if 32 does not divide the number of bits of width of transaction bitmap. As an example In Figure 7, we transfer the item IDs in the transaction linked list to two integer arrays, i.e., transaction index array (trans_index_array) and transaction array (trans_array). The trans_array is a one-dimension integer array with the entire item Ids in the dataset. The trans_index_array is an array of start index of each transaction of the trans_array.

```
typedef struct intarray_linkedlist
{
        struct intarray_linkedlist * next;
        int* data;
        int len;
        intarray_linkedlist()
        {
                next = NULL;
                data = NULL;
                len = 0;
        }
} intarray_linkedlist;
```

```
typedef struct trans_linkedlist
{
        struct trans_linkedlist * next;
        struct intarray_linkedlist* data;
        int index;
        trans_linkedlist()
        {
                next = NULL;
                data = NULL;
                index = -1;
        }
} trans_linkedlist;
```

Figure 6. Structures of Transaction Linked List

Algorithm 1. Transform a Dataset File to a Linked List Structure

----

```
Initialize a transaction linked list;
Open the file of Dataset;
For each line of transaction:
    Parse the line for Item IDs;
    Find the min/max IDs of items;
    Store Item IDs in a Hash-Map structure;
    Store Item IDs into an integer linked list;
End For
Extract the distinct items from the Hash-Map structure to form an integer
array of distinct item IDs;
```

Algorithm 2. Building Dataset Bitmap on GPUs

```
Define and initialize global memory bitmap_data[];
For each Thread:
    For each item ID, i, in j^th transaction:
        Set 1-bit on j^th bit of bitmap_data [i];
    End For
End For
Copy the bitmap_data to CPU;
```

PTFIM assigns 32 threads in each GPU block. Each GPU block processes 32 transactions. For example, if the dataset has 100,000 transactions, PTFIM needs 3125 blocks. In Algorithm 2, PTFIM defines an integer array (bitmap_data) in global memory as a transaction bitmap. In each thread of a GPU block, PTFIM scans each item IDs in the jth transaction and uses the item ID as the row index of transaction bitmap. PTFIM sets 1-bit on jth bit of bitmap_data[item_ID]. After PTFIM finishes building the bitmap on GPUs, it transfers the transaction bitmap to CPUs. The transaction bitmap and its associated information are saved into a disk file so that they can be used across multiple runs of experiments.

Using the example in Figure 7, Figure 8 illustrates how a single thread sets a 1-bit on jth transaction position of an item row in the transaction bitmap. Assume the dataset has 16 transactions. In Block_0, Thread_0 scans trans_0 whose first item ID is 1. PTFIM sets 1-bit on the first bit in bitmap_data[1]. Since the second item ID in trans_0 is 2, PTFIM sets 1-bit on the first bit in bitmap_data[2].

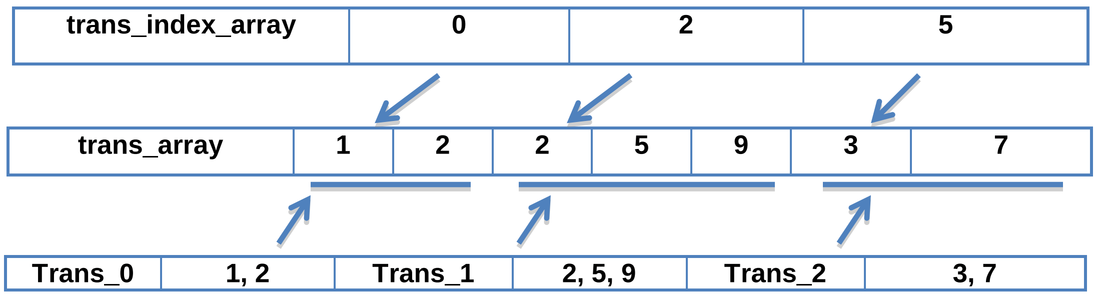

Figure 7. Transaction Index Array and Transaction Array

trans_index_array	0		2			5		
trans_array	1	2	2	5	9	3	7	…

bitmap_data:

(trans_ID) (item_ID)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8. Set 1-Bit in the Transaction Position of Item Rows

3.2 Data Structure and Algorithm Design of PTFIM

As shown in Algorithm 3, for each level of Frequent K-Itemset generation, PTFIM first generates Candidate K-Itemset Tire and constructs candidate data arrays listed in Table 1. Using the procedures described in Section 3.3, PTFIM transfers the candidate data arrays to GPUs for calculating the support count of Candidate K-Itemsets. PTFIM transfers the support count array of Candidate K-Itemsets back to CPUs and uses it to generate the Frequent K-Itemsets and Frequent K-Itemset Trie. After PTFIM obtains the Frequent K-Itemsets, using the procedures described in Section 3.4, PTFIM truncates the Candidate K-Itemset Bitmap on GPUs to be the Frequent K-Itemset Bitmap for the use of support counting of Candidate (K+1)-Itemsets. PTFIM repeats these procedures until there are no items in the Frequent K-Itemsets or in Candidate (K + 1)-Itemsets.

Algorithm 3. Frequent K-Itemset Generation of PTFIM

```
Build Candidate 1-Itemset Trie;
Construct candidate data arrays and transfer to GPU for support counting
of Candidate 1-Itemsets;
Return support array to CPUs to generate Frequent 1-Itemsets;
Build Frequent 1-Itemset Trie;
If (Frequent 1-Itemset Trie is empty)
    Return
K = 2;
While (Frequent (K-1)-Itemset is not empty)
    Build Candidate K-Itemset Trie;
    If (Candidate K-Itemset Trie is empty)
        Break;
    Construct candidate data arrays;
    Transfer candidate data arrays to GPUs for support counting of
    Candidate K-Itemsets;
    Return support array to CPUs to generate Frequent K-Itemsets;
    Build Frequent K-Itemset Trie;
    If (Frequent K-Itemset Trie is empty)
        Break;
    K++;
```

Itemset Trie

PTFIM employs an Itemset Trie (Figure 11 to Figure 17) as the core data structure on CPUs. **At the level 1, PTFIM builds a Candidate 1-Itemset Trie by using each item as a Candidate 1-itemset and all the itemsets become the leaf nodes of the Candidate 1-Itemset Trie.** After PTFIM performs Candidate K-Itemset Support Counting on GPUs, PTFIM sets the support counts to the leaf nodes at the level K on Candidate K-Itemset Trie. PTFIM prunes the item leaf nodes at the level K and their corresponding branches, which are not contained in Frequent K-Itemsets, to generate the Frequent K-Itemset Trie. Based on the Frequent K-Itemset Trie, for each node at the level K, PTFIM adds all right siblings of the node as children nodes. The result is the Candidate (K+1)-Itemset Trie. PTFIM performs DFS trie traversal on the Frequent K-Itemset Trie to check if all the K-subsets of Candidate (K+1)-Itemset are contained in Frequent K-Itemset Trie in order to find the maximal Frequent K-Itemset. If a K-subset is not in the Frequent K-Itemset trie, the corresponding Candidate (K+1)-Itemset branch is pruned on the Candidate (K+1)-Itemset Trie. PTFIM combines all the candidate item leaf nodes of Candidate (K+1)-Itemset Trie to be Candidate (K+1)-Itemset.

For GPU-based Candidate K-Itemset Support Counting, PTFIM needs to transfer the information about the Candidate Itemset Trie to GPUs. Since it is efficient to use one-dimension arrays on GPUs, PTFIM defines a group of candidate data arrays for Candidate K-Itemset Support Counting on GPUs. As discussed above, PTFIM first calculates the size of Candidate K-Itemset to perform GPU memory allocations for all candidate data arrays. PTFIM then stores the item IDs of candidate nodes in a candidate item ID array and the parent item IDs of the candidate nodes in a candidate parent item ID array. PTFIM also computes the numbers of children of nodes at the level (K-1) to be candidate children size array and computes the start indices of candidate children of nodes at the level (K-1) in a candidate children item ID array. Since PTFIM only transfers the necessary data arrays for each level for support counting, the data transfer overhead between CPUs and GPUs are significantly reduced.

A Running Example

Taking the dataset in Table 2 for example, itemsets, <0>, <1>, <2>, <3>, <4>, <5> are the Candidate 1-Itemsets, which are the leaf nodes of Candidate 1-Itemset Trie in Figure 11. After PTFIM performs support counting of Candidate 1-Itemsets, itemsets <0>, <1>, <2> have enough supports and become Frequent 1-Itemsets. PTFIM prunes nodes <3>, <4>, <5> at the level 1 of Candidate 1-Itemset Trie in Figure 12 to generate Frequent 1-Itemset Trie in Figure 13. Using the Frequent 1-Itemset Trie in Figure 13 as the base of Candidate 2-Itemset Tire, at the level 1, PTFIM knows the right sibling item nodes of frequent item node 0 are node 1 and node 2. PTFIM adds candidate node 1 and node 2 as children nodes for node 0. **PTFIM also adds item node 2 as a child node of frequent item node 1 at level 1**. At the level 1, since frequent item 2 has no right siblings to add as its children nodes, PTFIM prunes frequent item 2 node and generates the Candidate 2-Itemset Trie as shown in Figure 14.

Figure 10 illustrates the data arrays for Candidate 2-Itemset Support Counting of Candidate 2-Itemset Trie in Figure 14. The item row indices of Frequent 1-Itemset Bitmap are stored in cand_parent_idx that represent the nodes at the level 1 of Candidate 2-Itemset Trie. The cand_parent_idx has two frequent items 0 and 1. The item row indices of

Candidate 2-Items Bitmap are stored in cand_array that represent the nodes at the level 2 of Candidate 2-Itemset Trie. The cand_array has three candidate items 1, 2 and 2. For Candidate 2-Itemset Support Counting, the Frequent 1-Itemset Bitmap is the transaction bitmap (bitmap_gpu). The candidate item bitmap is also bitmap_gpu at the level 1. PTFIM performs bitwise AND operations between item rows of Frequent 1-Itemset Bitmap and item rows of the candidate item bitmap to obtain item rows of Candidate 2-Itemset Bitmap (cand_bitmap): cand_bitmap[0] = bitmap_gpu[0] & bitmap_gpu[1]; cand_bitmap[1] = bitmap_gpu[0] & bitmap_gpu[2]; cand_bitmap[2] = bitmap_gpu[1] & bitmap_gpu[2]. PTFIM computes the support counting on item rows of cand_bitmap and stores the support counts of candidate items to cand_sup_list. The cand_sup_list thus has values 2, 2 and 2. According to the support count of each candidate and the minimum support, PTFIM determines whether each candidate item is a frequent item in the Frequent 2-Itemset and stores the boolean results in the boolean array freq_array. The freq_array has values 1, 1 and 1 in this example. PTFIM stores the size of Frequent 2-Itemset, which is the number of 1's in freq_array, to make freq_size_array[3] to be 3 in this example. PTFIM truncates the cand_bitmap to be the Frequent 2-Itemset Bitmap (freq_bitmap) to be detailed in Section 3.4. After PTFIM transfers the data arrays from GPUs to CPUs, PTFIM builds the Frequent 2-Itemset Trie based on freq_array as illustrated in Figure 15. We next continue to explain how PTFIM constructs the Candidate 3-Itemset Trie.

PTFIM constructs the Candidate 3-Itemset Trie in Figure 16 with the same procedures as in generating Candidate 2-Itemset Trie in Figure 14. In this case, the Candidate 3-Itemset has one path <0, 1, 2>. The cand_parent_idx has one value 0, which is the first item row in freq_bitmap. The cand_array also has one value 2. The candidate item bitmap is bitmap_gpu. PTFIM performs bitwise AND operations between item rows of freq_bitmap and item rows of the bitmap_gpu to obtain the item row of Candidate 3-Itemset Bitmap stored in cand_bitmap: cand_bitmap[0] = freq_bitmap[0] & cand_array[2]. PTFIM computes the support counting on cand_bitmap[0] and stores the support counts of candidate items to cand_sup_list that has one value 2. Thus the candidate item is a frequent item of Frequent 3-Itemset. PTFIM stores 1 in freq_array[0]. PTFIM obtains one Frequent 3-Itemset and freq_size_array[2] is 1. Based on freq_array, PTFIM builds the Frequent 3-Itemset Trie in Figure 17. As PTFIM generates no Candidate 4-itemsets, the procedure terminates.

Table 1. Declarations of PTFIM Data Arrays

Data Type	**Name**	**Description**
uint *	bitmap_gpu	Transaction bitmap on GPUs.
uint*	cand_bitmap	Candidate bitmap on GPUs for & (Bitwise AND) operations.
uint*	cand_parent_idx	Last level frequent item IDs.
uint*	cand_index_array	Indices of Candidate Itemset under last level parents.
ushort *	cand_array	Candidate item IDs
uint*	cand_sup_list	Support count of Candidate Itemset.
uint*	freq_size_array	Frequent items size array at each level.

ushort*	freq_array	Boolean array to indicate whether a candidate item is a frequent item.
uint*	freq_bitmap	Frequent bitmap on GPUs for & (Bitwise AND) operations.

Note: unsigned int == uint; unsigned short == ushort;

Table 2. Example Dataset with a Minimum Support of 2

Trans_0	2
Trans_1	0, 1, 2
Trans_2	3, 4, 5
Trans_3	0, 1, 2
Trans_4	1, 2

Trans Id **Item Id**	**0**	**1**	**2**	**3**	**4**
0	0	1	0	1	0
1	0	1	0	1	1
2	1	1	0	1	1
3	0	0	1	0	0
4	0	0	1	0	0
5	0	0	1	0	0

Figure 9. Transaction Bitmap of the Dataset in Table 2

cand_parent_idx	**0**		**1**
cand_index_array	0		2
cand_array	1	2	2
cand_sup_list	2	2	2
freq_array	1	1	1
freq_size_array[1]	3		

Data arrays in Candidate 2-Itemset Support Counting

cand_parent_idx	**0**
cand_index_array	0
cand_array	2
cand_sup_list	2
freq_array	1
freq_size_array[2]	1

Data arrays in Candidate 3-Itemset Support Counting

Figure 10. Data Arrays in Support Counting of Candidate K-Itemsets

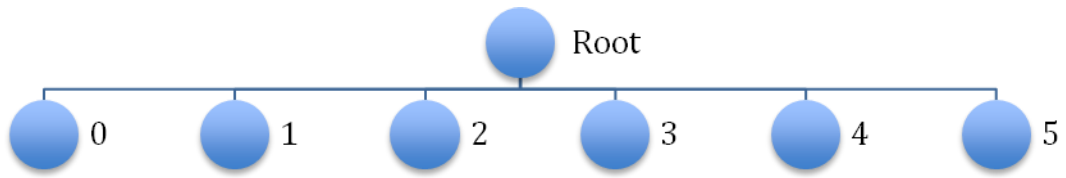

Figure 11. Candidate 1-Itemset Trie

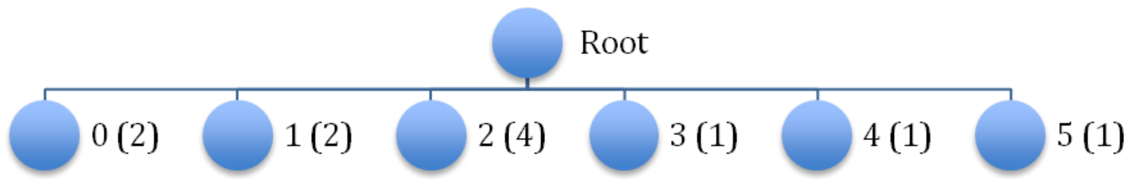

Figure 12. Support Counting on Candidate 1-Itemset Trie

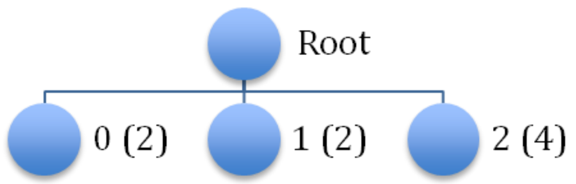

Figure 13. Frequent 1-Itemset Trie

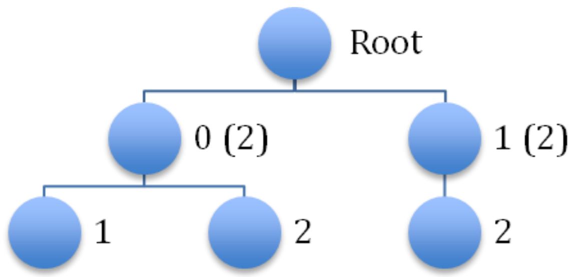

Figure 14. Candidate 2-Itemset Trie

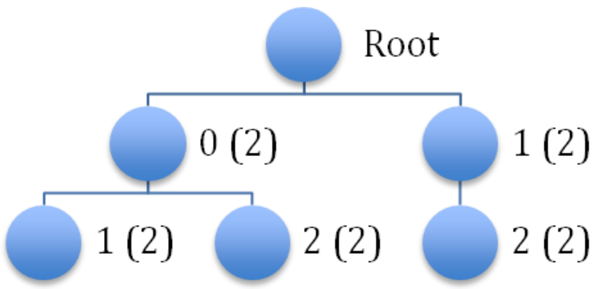

Figure 15. Frequent 2-Itemset Trie

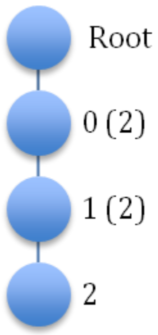

Figure 16. Candidate 3-Itemset Trie

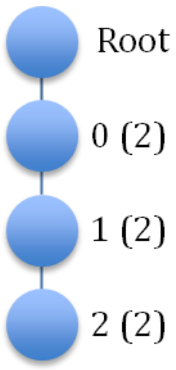

Figure 17. Frequent 3-Itemset Trie

3.3 Candidate Itemset Support Counting of PTFIM on GPUs

3.3.1 Candidate 1-Itemset Support Counting of PTFIM

As discussed in Section 3.2, once PTFIM transfers the transaction bitmap of dataset to GPU, PTFIM does not need to transfer the bitmap back to CPU. The transaction bitmap automatically becomes Candidate 1-Itemset Bitmap. PTFIM utilizes the transaction bitmap to perform Candidate 1-Itemset Support Counting on GPUs. Before PTFIM begins the Candidate 1-Itemset Support Counting on GPUs, we need to determine the dimension of the blocks and grids of GPUs threads as this could directly affect the performance of the entire process on GPUs. As discussed in Section 3.1, 32 divides the width of bitmap. PTFIM knows the number of 32-bit unsigned integer of each item row. As shown in Figure 18, using 256 GPU threads per block, PTFIM processes the support counting of 256 32-bit unsigned integers at a time. If 256 does not divide the number of unsigned integer, some threads will access the memory out of the bitmap item row in the last set of 32-bit unsigned integers. Algorithm 4 illustrates the procedures for the calculation of the number of minimum loops min_loop and number of maximum loops max_loop that individual threads should take. According to the indices of threads, PTFIM separates them into two groups. One group of threads processes min_loop, the other group of threads processes max_loop. PTFIM also computes the number of threads in each group in order to prevent some threads from accessing the memory out of the boundary of the bitmap item rows.

Algorithm 4. Preparation of Candidate Itemset Support Counting

```
Define Block_Size = total_candidate_item_count, Thread_Size = 256;
transaction_ints = bitmap_gpu_width /32;
min_loop  = transaction_ints / Thread_Size;
max_loop = min_loop + 1;
For each i in Thread_Size:
        If((transaction_ints - i*max_loop )%min_loop  == 0):
                threads_max_loop = i;
                threads_min_loop =
                        (transaction_ints - i*max_loop)/min_loop;
                If((threads_max_loop + threads_min_loop) == Thread_Size):
                        break;
End For
```

Index of Item Row	0	…	255	256	…	511	512	…	767	…
Thread_Id	0	…	255	0	…	255	0	…	255	…
Loop Count	1st			2nd			3rd			…

Figure 18. Memory Access of a Item Row in GPU

After PTFIM completes the preparations for GPUs support counting, PTFIM allocates a support array in shared memory for storing the support numbers of all threads. PTFIM utilizes the CUDA arithmetic function __popc() in each thread and each GPU block calculates the support counts of every 256 unsigned integers of candidate item row in an iteration. PTFIM updates the support counts in the block's shared memory. When a block finishes support counting of entire item row, threads in the block with even index perform a prefix sum [15] to add up the support counts in the shared memory. At last, only the first thread in the block is responsible for writing the accumulated result of the support of this item row to the support count array of Candidate 1-Itemsets in the GPU’s global memory. PTFIM transfers the support count array back to CPUs and uses it to generate the Frequent 1-itemsets with the procedures discussed in Section 3.2.

3.3.2 Candidate K-Itemset Support Counting of PTFIM

With the procedures discussed in Section 3.2, PTFIM builds Candidate K-Itemset Trie and generates the candidate data arrays for GPU support counting. Before PTFIM transfers all candidate arrays to GPUs for support counting, PTFIM determines the size of threads in each block and calculates the integer size of each item row of the candidate bitmap. PTFIM can use a size of threads up to 1024 if the integer size is larger than 1024 in order to fully utilize the computational power of GPUs. As shown in Algorithm 4, PTFIM calculates the min_loop, max_loop, ths_min_loop and ths_max_loop for GPUs Candidate K-Itemset Support Counting. Algorithm 5 illustrates the process of Candidate K-itemset Support Counting on GPUs. For each GPU block, PTFIM processes support counting of a set of candidate items under the same parent frequent item. PTFIM fetches the candidate item row from the transaction bitmap with candidate parent item ID and the frequent itemset row from the Frequent Itemset Bitmap with the same candidate parent item ID. PTFIM then performs bitwise AND operation between the candidate item row and the frequent itemset row to generate a Candidate Itemset. PTFIM stores the results to Candidate Itemset Bitmap. Appling the CUDA arithmetic function, __popc(), PTFIM calculates the support count of each candidate itemset row in GPU registers. PTFIM then adds up all the support counts of a block and updates the final support count to global memory. The results are subsequently used to determine if the candidate item is a frequent item on CPUs.

```
-----------------------------------------------------------------------------------
Algorithm 5. Candidate K-Itemset Support Counting
-----------------------------------------------------------------------------------
Declare shared memory for the frequent children Itemset size, support
count array (sup_cont) and frequent parent itemset row;
Find out the candidate children size of the parent frequent itemset.
Find out the start indices of the candidate K-Itemsets in the candidate
items array.
Find out the index of item row of frequent parent item node in the Frequent
Itemset Bitmap.
Load the frequent parent itemset row to shared memory parent_bitmap.
Foreach candidate item in candidate items array
   If (threadIdx < threads_max_loop)
      Foreach k in max_loop
           Apply bitwise AND operation between frequent itemset row
                and candidate item row to obtain candidate itemset
           Apply __popc() to calculate the support count of
                the candidate itemset, and store in sup_cont.
        End For
     Else
        Foreach k in min_loop
           Apply bitwise AND operation between frequent itemset row
                and candidate item row to obtain candidate itemset
           Apply __popc()  to calculate the support count of
                the candidate itemset, and store in sup_cont.
        End For
   End If
Prefix Sum on sup_count to obtain the support count of candidate itemset.
Only threadIdx == 0, do:
   Update the sup_count [0] to global memory.
   If sup_count [0] > min_sup, the candidate itemset is frequent.
End For
-----------------------------------------------------------------------------------
```

3.4 Candidate Itemset Bitmap Truncation

Since each itemset row in Candidate K-Itemset Bitmap corresponds to a candidate item. PTFIM uses a boolean array to indicate whether each of a candidate item is frequent or not. As shown in Figure 19, PTFIM copies the corresponding candidate itemset row from Candidate K-Itemset Bitmap to Frequent K-Itemset Bitmap on GPUs according to the boolean array.

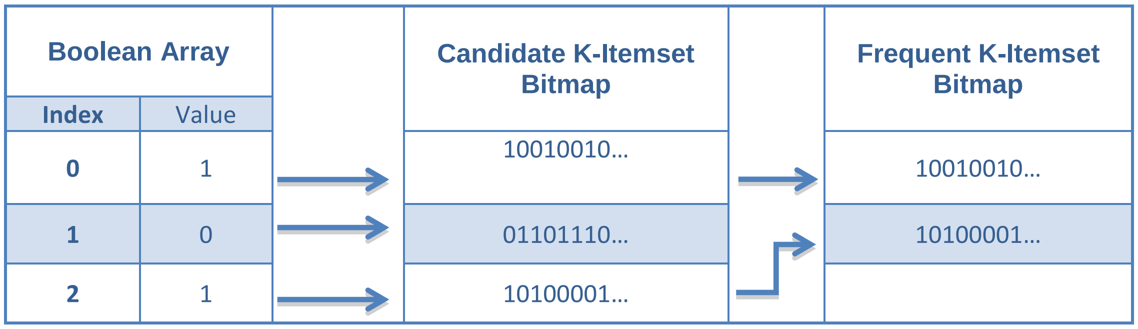

Figure 19. Candidate K-Itemset Bitmap Truncation

4. Evaluations

4.1 Experimental Setup

All the experiments were performed using a Dell Precision T7500 PC powered by Microsoft Windows 7. The machine has dual Intel Xeon 5650 CPUs running at 2.6GHz and 24GBytes of main memory. The machine is equipped with an NVidia Quadro 6000 GPU with GPU clock 1.15GHz and memory clock 1.494GHz [14]. The GPU global memory has a capacity of 6GBytes with a peak bandwidth up to 144GBytes/Sec. A PCI-E bus is used to transfer data between the CPU main memory and the GPU main memory with a bandwidth up to 8GBytes/Sec. All experimental data is stored in a 160GBytes SSD SCSI hard disk.

We use three synthetic datasets from FIMI ’03 repository [13], i.e., T40I10D100K, T10I4D100K, and T10I4D1K, to evaluate the implementation of PTFIM and compare the results with GPUMiner. These datasets have different numbers of transactions and different combinations of items in transactions, which have been summarized in Table 3. Both implementations of PTFTIM and GPUMiner are written and complied using Microsoft Visual Studio 2010. The version of CUDA is 4.0. We measure the total elapsed time and the elapsed times for Candidate K-Itemset Support Counting at the each level to evaluate the efficiency of both GPU Apriori implementations, i.e., PTFIM and GPUMiner.

Table 3. Descriptions of Experimental Datasets

	T40I10D100K	T10I4D100K	T10I4D1K
#Transactions	100,000	100,000	1000
#Items	1000	1000	1000
Characteristics	Synthetic	Synthetic	Synthetic
Data Size	15.5MBytes	4MBytes	45KBytes
Bitmap Size	12.8MBytes	12.8MBytes	131KBytes

4.2 Experimental Results

Since we are focusing on in-memory performance evaluations, we exclude the initial time for converting dataset from the plaint text to bitmap representation that is needed by both PTFIM and GPUMiner. We collect the transaction bitmaps data in binary files. We evaluate the performance using different minimum supports on the three synthetic datasets for both PTFIM and GPUMiner. Recall in Chapter 1 that the minimum support is the percentage of the total number of transactions in the dataset.

For Frequent 1-Itemset Generation and Frequent 2-Itemset Generation, GPUMiner generates the Candidate Itemset Bitmap and computes the support of the Candidate 1-Itemset and Candidate 2-Itemset using a lookup table on CPUs while PTFIM performs Candidate 1-Itemset and Candidates 2-Itemset Support Counting on GPUs. PTFIM gains significant speed improvement from massive SIMD parallelism on GPUs for support counting. Table 4 shows us that PTFIM is up to 50 times faster than GPUMiner on the dataset T40I10D100K in Candidate 1-Itemset

Support Counting. These comparisons clearly demonstrate the impact of GPU acceleration for support counting over CPUs. As shown in Figure 20, PTFIM has an almost 9x speedup over GPUMiner on the dataset T40I10D100K with minimum support 1% in Frequent 2-Itemset Generation on GPUs.

Table 4. Candidate 1-Itemset Support Counting Elapsed Times

	T40I10D100K	**T10I4D100K**	**T10I4D1K**
GPUMiner	47ms	4ms	0.5ms
PTFIM	0.91ms	0.85ms	0.31ms

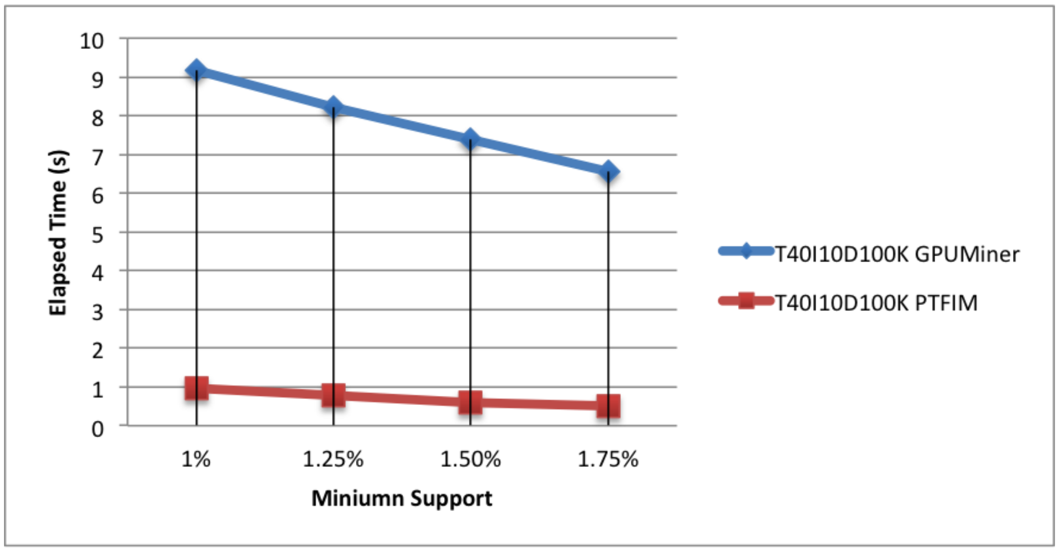

Figure 20. Frequent 2-Itemset Generation Elapsed Times on T40I10D100K

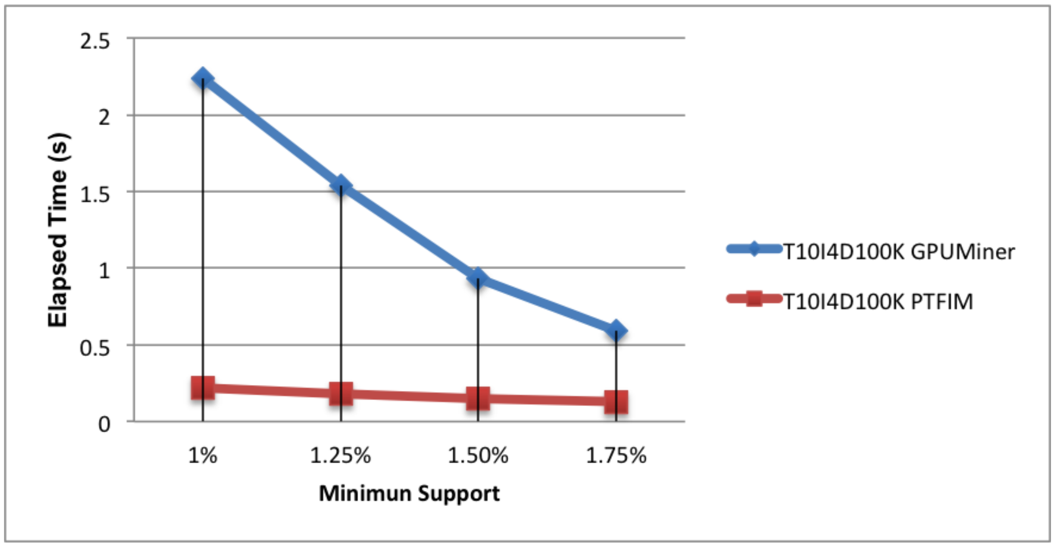

Figure 21. Frequent 2-Itemset Generation Elapsed Times on T10I4D100K

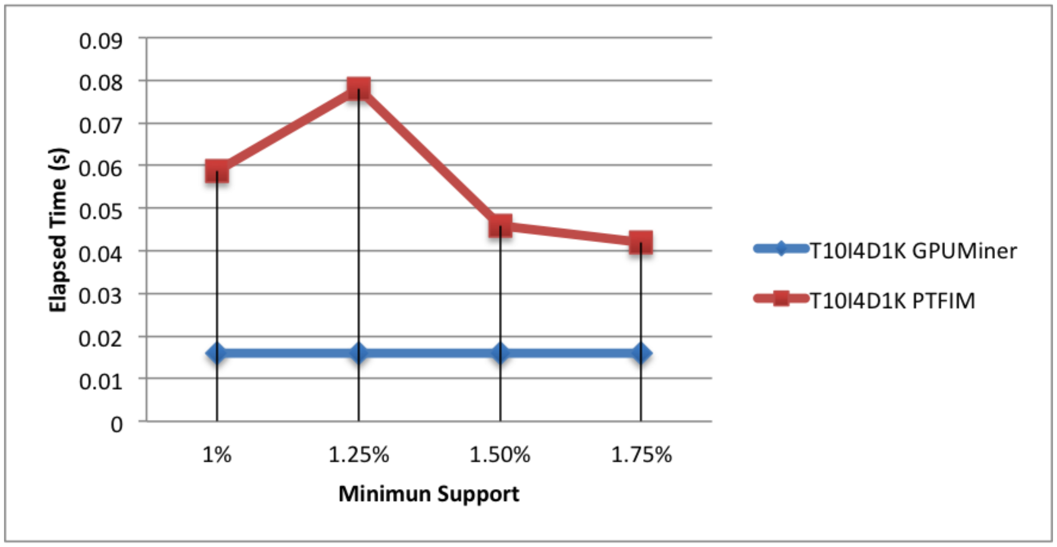

Figure 22. Frequent 2-Itemset Generation Elapsed Times on T10I4D1K

To compare the overall performance between PTFIM and GPUMiner we use 4 different minimum supports, i.e., 1%, 1.25%, 1.5% and 1.75%, on the three datasets. Figure 23 illustrates the overall runtimes of PTFIM and GPUMiner on dataset T40I10D100K with the four minimum supports. We can see that PTFIM is faster than GPUMiner by a factor of 2 to 5. These relatively low speedups are mainly due to the elapsed times for the generations of Candidate K-Itemset on CPUs in both algorithms. In Table 5, we list the elapsed times breakdown of both algorithms on dataset T40I10D100K with minimum support 1%. We can see that, PTFIM and GPUMiner have similar performance on Candidate K-Itemset Generation (on CPUs) and Candidate K-Itemset Support Counting (on GPUs) for K=3 and above. Although PTFIM is significantly faster (9-30X) than GPUMiner on support counting for K=1 and K=2, the overall speedup is reduced to 2 due to Amdahl's law [16]. As shown in Figure 25, there are only 1000 transactions in dataset T10I4D1K. In this case, GPUMiner outperforms PTFIM. The reason is that PTFIM incurs significant overhead of data transfers between GPU memory and CPU main memory relative to computation time.

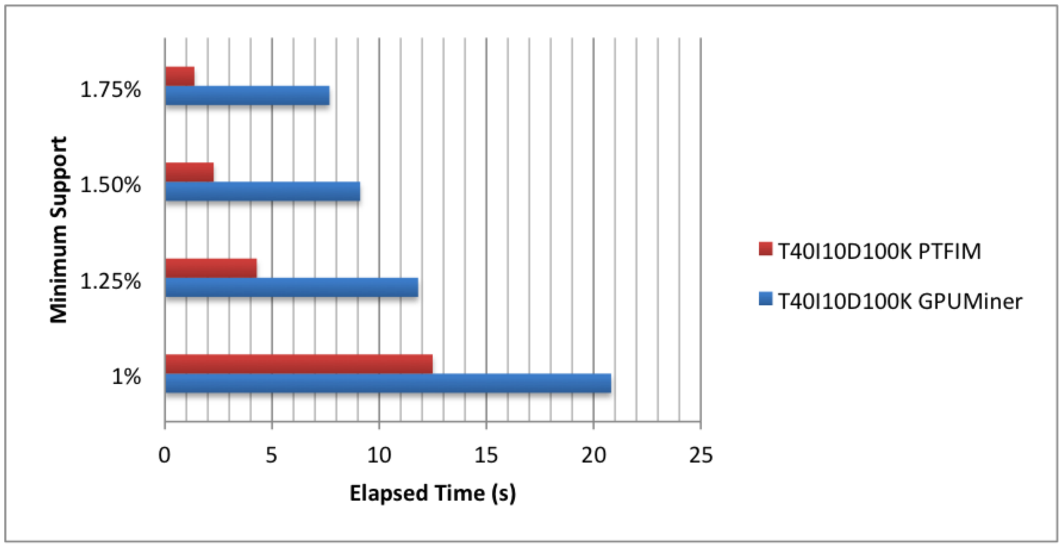

Figure 23. Program Elapsed Times on T40I10D100K

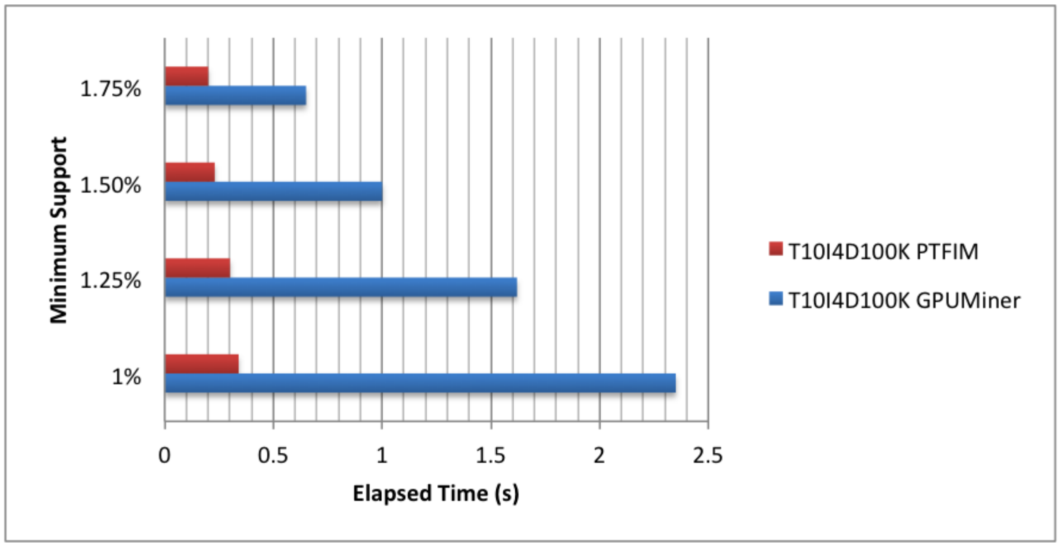

Figure 24. Program Elapsed Times on T10I4D100K

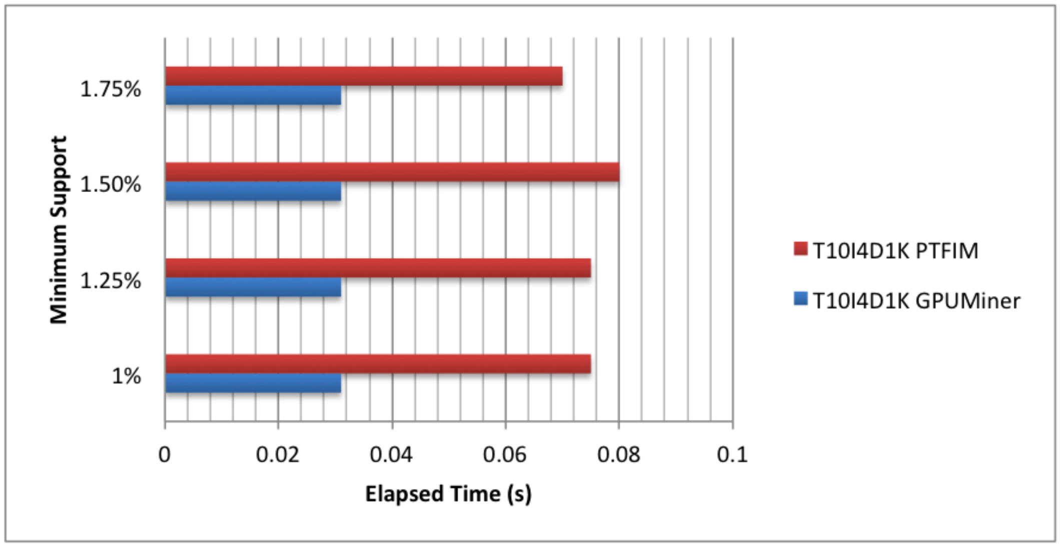

Figure 25. Program Elapsed Times on T10I4D1K

Table 5. Elapsed Times Breakdown of GPUMiner and PTFIM

	GPUMiner	PTFIM
Elapsed Time of Candidate 1-Itemset Support Counting	0.047s	0.00091s
Elapsed Time of Candidate 1-Itemset Memory Transfer	0s	0.01220s
Total Elapsed Time of Candidate 1-Itemset GPU Functions	0s	0.01310s
Elapsed Time of Frequent 1-Itemset Trie Generation	0.140s	0.01500s
Total Elapsed Time of Frequent 1-Itemset Generation	0.187s	0.028810s
Elapsed Time of Candidate 2-Itemset Generation	0s	0.03s
Elapsed Time of Candidate 2-Itemset Support Counting	9.18s	0.29s
Elapsed Time of Candidate 2-Itemset Memory Transfer	0s	0.35s
Elapsed Time of Candidate 2-Itemset Bitmap Truncation	0s	0.13s
Total Elapsed Time of Candidate 2-Itemset GPU Functions	0s	0.77s
Elapsed Time of Frequent 2-Itemset Trie Generation	0.12s	0.18s
Total Elapsed Time of Frequent 2-Itemset Generation	9.3s	0.98s
Elapsed Time of Candidate 3-Itemset Generation	4.79s	4.73s
Total Elapsed Time of Candidate 3-Itemset GPU Functions	1.83s	1.49s
Elapsed Time of Frequent 3-Itemset Trie Generation	0.28s	0.31s
Total Elapsed Time of Frequent 3-Itemset Generation	6.9s	6.53s
Total Elapsed Time of Frequent K-Itemset Generation (K>3)	4.5s	4.96s
Total Elapsed Time of Candidate K-Itemset Generation (K >= 1)	6.84s	6.4s
Total Program Elapsed Time	21.89s	11.28s

5. Conclusion and Future Work

In this thesis, we have presented an algorithm with a new design and implementation on GPGPU based on the Apriori FIM principle. The PTFIM algorithm leverages the trie-based representation and the latest GPGPU technology. PTFIM calculates all Candidate Itemset support counts on GPUs. PTFIM uses BFS trie traversal of Candidate Itemset Trie to perform one time data transfer to GPUs for Candidate Itemset Support Counting. This significantly reduces the data transfer overhead between GPU memory and CPU main memory. Experiments have demonstrated that PTFIM improves the efficiency of the trie-based FIM on various sizes of synthetic datasets and various minimum supports of each dataset. The results show that PTFIM is almost 2 times faster than GPUMiner on overall performance, up to 9 times faster in Frequent 2-Itemset Generation and more than 50 times faster in Candidate 1-Itemset Support Counting on a large dataset with 100,000 transactions with a minimum support of 1%.

GPU memory sizes are different among devices. It is possible that the available memory capacity is not enough to allow PTFIM to perform one-time data transfer between CPU main memory and GPU global memory. We are considering improvements for dynamic Candidate K-Itemset Support Counting. By dividing the Candidate K-Itemset Bitmap and Frequent (K-1)-Itemset Bitmap into parts, we can apply GPU support counting on each part of Candidate K-Itemset Bitmap. We then can combine the results of the parts to be the final Frequent K-Itemsets. In this way, PTFIM keeps the advantage of using the BFS trie traversal on the Candidate Itemset Trie while requires less GPU memory capacity.

6. References

[1] R. Agrawal and R. Srikant: Mining Sequential Patterns. Proceedings of the 11th IEEE International Conference on Data Engineering (ICDE), Taipei, Taiwan 1995: 3-14.

[2] R. Agrawal and R. Srikant: Fast Algorithm for Mining Association Rules. Proceedings of the 20th International Conference on Very Large Database (VLDB), Santiago, Chile, 1994: 487-499.

[3] Wenbin Fang, Mian Lu, Xiangyu Xiao, Bingsheng He, and Qiong Luo: Frequent Itemset Mining on Graphics Processors. Proceedings of the 5th International Workshop on Data Management on New Hardware (DAMON), 2009: 34-42.

[4] J. Han and J. Pei, Mining Frequent patterns by pattern-growth: methodology and implications. Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD), 2000: 14-20.

[5] S. Kevin, and R. Ramakrishnan: Bottom-up Computation of Sparse and Iceberg CUBEs. Proceedings of the ACM Conference on Management of Data (SIGMOD), 1999: 359-370.

[6] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal: Discovering Frequent Closed Itemsets for Association Rules. Proceedings of the 11th IEEE International Conference on Data Engineering (ICDE), 1999: 398-416.

[7] F. Afrati, A. Gionis, and H. Mannila: Approximating a Collection of Frequent Sets. Proceedings of ACM Conference on Knowledge Discovery and Data Mining (KDD), 2004: 12-19.

[8] F.Tao, F. Murtagh, and M. Farid: Weighted Association Rule Mining using Weighted Support and Significance Framework. Proceedings of ACM Conference on Knowledge Discovery and Data Mining (KDD), 2003: 661-666.

[9] B. Padmanabhan, and A. Tuzhilin: On Characterization and Discovery of Minimal Unexpected Patterns in Rule Discovery. IEEE Transactions on Knowledge and Data Engineering 18 (2), 2008: 202-216.

[10] Nvidia: CUDA Programming Guide.
http://developer.nvidia.com/object/gpucomputing.html.

[11] David Kirk and Wen-mei W. Hwu: ECE 498AL Spring 2010 Course website, University of Illinois, Urbana-Champaign.

[12] B. Goethals and M. J. Zaki: Advances in Frequent Itemset Mining Implementations: Introduction to FIMI03. Proceedings of the Frequent Itemset Mining Implementations Conference (FIMI), 2003.

[13] Frequent Itemset Mining Implementations Data Repository. http://fimi.ua.ac.be/data/

[14] Quadro 6000 Specifications. http://www.nvidia.com/object/product-quadro-6000-us.html

[15] Prefix Sum. http://en.wikipedia.org/wiki/Prefix_sum

[16] Amdahl’s law. http://en.wikipedia.org/wiki/Amdahl's_law