Parallel Frequent Pattern Mining without Candidate Generation on GPUs

Fei Wang and Bo Yuan
Intelligent Computing Lab, Division of Informatics
Graduate School at Shenzhen, Tsinghua University
Shenzhen 518055, P.R. China
wangfeifast@gmail.com, yuanb@sz.tsinghua.edu.cn

Abstract—The graphics processing unit (GPU) has evolved into a key part of today’s heterogeneous parallel computing architecture. A number of influential data mining algorithms have been parallelized on GPUs including frequent pattern mining algorithms, such as Apriori. Unfortunately, due to two major challenges, the more effective method for mining frequent patterns without candidate generation named FP-Growth has not been implemented on GPUs. Firstly, it is very hard to efficiently build the FP-Tree in parallel on GPUs as it is an inherently sequential process. Secondly, mining the FP-Tree in parallel is also a difficult task. In this paper, we propose a fully parallel method to build the FP-Tree on CUDA-enabled GPUs and implement a novel parallel algorithm for mining all frequent patterns using the latest CUDA Dynamic Parallelism techniques. We show that, on a range of representative benchmark datasets, the proposed GPU-based FP-Growth algorithm can achieve significant speedups compared to the original algorithm.

Keywords—FP-Tree; FP-Growth; GPU; Dynamic Parallelism

I. INTRODUCTION

Association rule mining is an important problem in data mining. The valuable association information can be further used as the foundation in many higher level data mining tasks such as motif discovery [18], anomaly detection [15], and recommendation [25]. It consists of discovering frequent itemsets and forming conditional rules among which frequent pattern mining is the most essential and complex subtask.

The task of mining frequent itemsets is formally stated as follows: let I = \{i_1, i_2, \ldots, i_n\} be a set of items and D be a set of transactions, where each transaction T is a subset of I. We call a subset X ⊆ I an itemset and X is a(k) itemset if X consists of k items. The support of itemset X, represented as \(\text{sup}(X)\), is the number of transactions in D that contain X. If \(\text{sup}(X)\) is not less than \(\text{min}_\text{sup}\) (the minimum threshold of the support), X is regarded as a frequent itemset. Frequent mining algorithms aim to find the universal set of frequent itemsets efficiently.

Frequent mining has received significant attention since the publication of the AIS [1] and Apriori algorithms [2] as well as their extensions [17, 21, 29]. The core idea is to iteratively generate a set of candidate patterns of length k + 1 from the set of frequent patterns of length k. We call these algorithms Apriori-like algorithms. Despite of the promising performance achieved, when the minimum support threshold is low, these algorithms tend to produce very poor performance in terms of running time and memory cost due to the huge number of candidate sets and the need of multiple scans of the database.

Unlike Apriori, the FP-Growth method [9] compresses the original database into a prefix tree called FP-Tree. One of its major advantages is that it only needs to scan the database twice: one for finding frequent 1-itemsets and the other for building the FP-Tree. It employs a recursive divide-and-conquer method to discover all the frequent itemsets. It can often improve the performance of Apriori-like algorithms by an order of magnitude, especially on compact datasets. After that, Wang et al. [26] proposed a top-down processing method to mine FP-Tree without generating conditional pattern bases. Grahne and Zhu [8] introduced a FP-Array technique to reduce the traversal time of FP-Tree. In [19, 28], parallel FP-Growth algorithms were proposed but they suffered from high communication cost. Li et al. [16] implemented a parallel FP-Growth based on MapReduce and achieved linear speedup.

The GPU is an inexpensive, energy efficient and highly effective SIMT (Single Instruction, Multiple Thread) parallel computing device, which can be found in many mainstream desktop computers and workstations. Fang et al. [3] first used CUDA-enabled GPU [32] for mining frequent itemsets. The GPU was used to accelerate the process of generating frequent itemsets of length k+1 from frequent k-itemsets in Apriori. The GPU-based Apriori was parallelized within each hierarchy level and was two orders of magnitude faster than the CPU-based Apriori. However, it is still slower than the CPU-based FP-Growth [3] due to its relatively low degree of parallelism. As to the GPU-based FP-Growth, researchers have been frustrated by issues including irregular data structures and complex algorithmic control of FP-Growth [3, 29, 30].

There are two major challenges in implementing parallel FP-Growth on GPUs. The first one is how to build the FP-Tree in parallel on GPUs as building the FP-Tree is an inherently sequential process and there is no effective method to build an arbitrary tree in parallel on GPUs. The second one is how to traverse the FP-Tree in parallel to discovery all the frequent itemsets. In this paper, we propose to map the FP-Tree into a binary radix tree. Then, we use a fully parallel method [13] to build each node of the tree in parallel and use the atomic operation to build the header table and the side-link of the FP-Tree at the same time. After that, we redesign the TD-FP-Growth [26] into a parallel algorithm as it can avoid the need to generate conditional pattern bases and physical projections of the sub-FP-Tree. In this way, it can avoid the dynamic memory allocation on GPUs, which is a very costly operation. We also employ bitmap to represent itemsets to achieve better performance on GPUs.
In the experimental studies, we implemented the proposed GPU-based FP-Tree construction procedure and the parallel TD-FP-Growth. We compared its performance to three other existing algorithms on four datasets from FIMI’ 04 [33] of different sizes and levels of density to demonstrate the speedup of our algorithm.

This paper is organized as follows. In Section II, we give an overview of existing work on GPU and TD-FP-Growth. We also introduce an optimized parallel method for the construction of radix tree and analyze the relationship between FP-Tree and radix tree. The details of algorithm implementations are described in Section III. Comprehensive experimental results are presented in Section IV and this paper is concluded in Section V.

II. BACKGROUND AND RELATED WORK

A. General Purpose GPU Computing

Recently, the GPU has been widely used not only as a traditional graphics co-processor for advanced video processing tasks but also a general purpose many-core computing device. Compared to CPU based systems, GPU systems can dramatically reduce the cost of computing as well as the energy consumption. Nowadays, it is possible to achieve one GFLOPS for less than $0.2 and the latest NVIDIA GPU cards can produce around 20 GFLOPS for each watt consumed. Typical GPU applications include matrix multiplication [14], databases [5, 7, 12], data stream mining [6], FIMI mining [3], subsequence search [20] and GPU-based primitives for database applications [11, 22].

CUDA (Compute Unified Device Architecture) [32], introduced by NVIDIA in 2007, is a hierarchical and heterogeneous programming architecture for NVIDIA GPUs in which a large number of threads are organized into warp, block and grid hierarchically. It allows developers to write GPU codes in a way similar to C/C++, significantly reducing the complexity of GPU computing. Some new features in CUDA, such as atomic operation [32] and dynamic parallelism [32], can significantly improve the programmability of GPU. The atomic operation, which ensures serial access to a section of memory by multiple threads, helps to maintain the header table of a FP-Tree. Dynamic parallelism, which allows one GPU thread to launch additional GPU threads without the intervention of CPU, can help to traverse the FP-Tree in parallel. Nevertheless, as a co-processor, the GPU still relies on the CPU for memory allocation and the dynamic memory allocation on GPU is known to be expensive. This feature makes the implementation of irregular data structures in the FP-Tree a very challenging task.

B. TD-FP-Growth

TD-FP-Growth [26] is a top-down variation of FP-Growth, which employs a top-down method to explore the FP-Tree. The major advantage of this algorithm is that it does not construct conditional pattern bases and sub-trees. Instead, it only needs to create conditional sub-header tables, which all refer to the global FP-Tree. Algorithm 1 gives the pseudocode of TD-FP-Growth. An example of the data structure is given in Fig. 1.

Algorithm 1: TD-FP-Growth Algorithm

```
Procedure buildsubtable (l)
(1) for each node u on the side-link of l do
(2) Walk up the path from u once do
(3) if encounter a J_node v then
(4) Link v into the side-link of l in H;
(5) count(v) = count(v) + count(u);
(6) H(l) = H(l) + count(u);

Procedure mine-tree(IX, HI)
(1) for each entry I (top-down order) in H do
(2) if H(i) >= minsup, then
(3) Output IX;
(4) buildsubtable(I); // create header table H;
(5) mine-tree(IX, HI);
```

Figure 1. Transaction table, FP-Tree and header table [26]

C. Maximizing Parallelism in the Construction of Radix Tree

How to efficiently build a tree on GPUs has been a research focus in the field of GPU computation [10]. In 2008, Zhou et al. [30] proposed an efficient parallel BFS (breadth-first search) method to build a k-d tree using CUDA. Similarly, in 2011, Garanmga et al. [4] also used the BFS method to implement a simpler and faster algorithm to build BVHs (Bounding Volume Hierarchy). Although these methods are parallelized within each hierarchy level, there is no parallelism between two levels. As a result, there is a lack of parallelism especially on top levels (with less number of nodes).

In 2012, Karras et al. [13] proposed a fully parallel algorithm to build the radix tree, octree and k-d tree. This algorithm takes advantage of the structure features of ordered binary radix trees to build each node of a tree in parallel. It stores the internal nodes and leaf nodes in two separate arrays, I and L. For an ordered binary radix tree, the locations of leaf nodes are fixed. So, only the indices of the children of each internal node need to be calculated.
For an ordered binary radix tree without duplicate keys as shown in Fig. 2 (top), it has some desired properties. Firstly, because the leaf nodes are in lexicographical order, we can use a linear range \([i, j]\) to represent the keys covered by an internal node. Let \(\delta(i, j)\) denote the length of the longest common prefix between keys \(k_i\) and \(k_j\). Secondly, each internal node partitions its keys according to their first different bit, the one following \(\delta(i, j)\). This bit will be zero for a certain number of keys starting from \(k_i\) and one for the remaining ones until \(k_j\). We call the index of the last key where the bit is zero a split position, denoted by \(\gamma\). Since the bits are zero for \(k_\gamma\) and one for \(k_\gamma+1\), the subranges are given by \([i, \gamma]\) and \([\gamma+1, j]\). Thirdly, for this data layout, the index of each internal node is either the first or the last key. So, the left child of any internal node is located at the end of its range \([i, \gamma]\) and the right child is located at the beginning of its range \([\gamma+1, j]\).

Given these properties, in order to build the children of an internal node, we need to determine its range and find the split. This procedure is fully parallel because each internal node has no dependency on each other. The outline of this algorithm is shown in Algorithm 2.

\begin{algorithm}
\begin{enumerate}
\item for each internal node with index \(i \in [0, n-2]\) in parallel
\item Determine direction of the range (+1 or -1)
\item Compute upper bound for the length of the range
\item Find the other end using binary search
\item Find the split position using binary search
\item Output child pointer and parent pointer
\item end for
\end{enumerate}
\end{algorithm}

Algorithm 2: Fully Parallel Construction of Binary Radix Tree

D. Relationship Between Binary Radix Tree and FP-Tree

Since the construction of a FP-Tree is an inherently sequential process while we can construct a binary radix tree in parallel using Algorithm 2, it is interesting to think whether we can map a FP-Tree into a binary radix tree, as they are both prefix trees. Note that a FP-Tree without the header table is equivalent to a trie. Furthermore, we can use a binary radix tree to represent a trie, as illustrated in Fig. 2.

Property. For a database with unique keys, there is an important property of the relationship between binary radix tree and trie. Given a node in trie, if its count \(\geq 2\), we can find it among the right nodes of the binary radix tree. If the count of a node is 1, we can find the node as a leaf node in the binary radix tree. With this property, we can build a binary radix tree on GPUs rather than a trie in order to implement the parallel FP-Growth.

III. IMPLEMENTATION

In this section, we present the design and implementation details of the proposed GPU-based FP-Growth. We exploit the bitmap representation of transactions and accelerate support counting. Our implementation follows the general workflow of the original algorithm [26], which consists of finding frequent 1-items, building FP-Tree and mining FP-Tree. We also discuss how to exploit some advanced GPU features to optimize the memory access operations.

A. Finding Frequent 1-item

There are two options to represent the transactions, namely, horizontal and vertical data layouts. Fig. 3 shows an example of these data layouts.

![Example of Horizontal and Vertical Data Layouts](image)

Property. For a database with unique keys, there is an important property of the relationship between binary radix tree and trie. Given a node in trie, if its count \(\geq 2\), we can find it among the right nodes of the binary radix tree. If the count of a node is 1, we can find the node as a leaf node in the binary radix tree. With this property, we can build a binary radix tree on GPUs rather than a trie in order to implement the parallel FP-Growth.

III. IMPLEMENTATION

In this section, we present the design and implementation details of the proposed GPU-based FP-Growth. We exploit the bitmap representation of transactions and accelerate support counting. Our implementation follows the general workflow of the original algorithm [26], which consists of finding frequent 1-items, building FP-Tree and mining FP-Tree. We also discuss how to exploit some advanced GPU features to optimize the memory access operations.

Figure 2. Binary Radix Tree (top) and Trie (bottom)

Figure 3. Horizontal data layout (left), vertical data layout (top right) and bitmap representation (bottom right) [3]

Horizontal layout requires scanning all transactions to perform support counting for each item, which limits the data parallelism of GPUs. Therefore, we adopt the vertical layout in which a row represents whether an item is in a transaction and a column implies which items are contained in the transaction. So, the number of 1s in a row is the support of an item.

Fig. 4 shows how support counting is performed within a single thread block. Each thread uses \(\_\_\text{popc}()\) intrinsic function, which implements the \(\text{popcount}\) function in one instruction cycle on NVIDIA Fermi and Kepler GPUs [32], to count the number of 1s in a 32-bit integer and store the result in an integer array in shared (on-chip) memory. Then, a parallel summation reduction algorithm [34] is used to add all the support values recursively into its first element. The resulting support value of each item is written back to device memory and the infrequent items will be removed.

![Horizontal Data Layout Support Counting](image)
designed based on horizontal data layout, we need to transpose parallel bitmap matrix transposition, which is very efficient on bitmap matrix to fit into Algorithm 2. As Algorithm 2 is Finally, we get a sorted bitmap matrix with unique keys and an integer array to record the count of each transaction.

B. FP-Tree Construction

Before constructing the FP-Tree, we have to standardize the bitmap matrix to fit into Algorithm 2. As Algorithm 2 is designed based on horizontal data layout, we need to transpose the bitmap matrix. We use the method in [27] to implement a parallel bitmap matrix transposition, which is very efficient on GPU. Then, we sort the transactions and remove duplicates. Finally, we get a sorted bitmap matrix with unique keys and an integer array to record the count of each transaction.

Algorithm 3: Constructing a binary radix tree with header table

<table>
<thead>
<tr>
<th>Item</th>
<th>Count</th>
<th>Index-array</th>
<th>Flag-array</th>
<th>NodeCount</th>
</tr>
</thead>
</table>

To construct a binary radix tree with header table, we need to improve Algorithm 2. For example, right internal nodes and some leaf nodes must be linked into header table as they are the map of the nodes in the original FP-Tree.

Algorithm 3 describes the details of constructing a binary radix tree with header table. TransCount(i, j) returns the total count of the transactions from i to j. δ(i, j) is the length of the prefix of two transactions, which can be evaluated efficiently by logical XOR between two transactions and counting the leading 0s in the result. This can be done by using the _clz() compiler intrinsic in CUDA.

The algorithm simply allocates an array of N-1 internal nodes, and then processes all of them in parallel. Each thread starts by determining the range of objects that the node corresponds to, using a binary search, and then proceeds to split the range as usual. Finally, it selects children for the node according to their respective sub-ranges. If a sub-range has only one transaction, the child must be a leaf. So, we reference the corresponding leaf node and link it to header table. Otherwise, we reference another internal node from the array and if it is a right child, we link it to header table.

C. Redesign of the Header Table

The header table is the key data structure of Algorithm 1 and Algorithm 3. In Algorithm 1, we need to traverse the header table top-down serially. In both algorithms, the header table is the only data structure that requires maintenance. So, our objective is to redesign the header table to improve linking and reading efficiency on GPU and remove the constraint of the top-down order.

![Figure 5. The design of header table](image)

The header table is redesigned as shown in Fig. 5. As the access of linked list is very costly on GPU, we store the index of leaf or internal node in index-array rather than a head pointer of a linked list. We use a flag-array to label whether the index is a leaf index or internal index. NodeCount records the count of the “conditional pattern base” [9], which can remove the constraint of the top-down order. An atomic operation is used to maintain the header table.

D. Parallel Mining of Frequent Items

TD-FP-Growth processes nodes at upper levels before those at lower levels. It is important to ensure that any modification made at upper levels will not affect lower levels. To obtain the conditional pattern base of a pattern, this algorithm simply walks up the paths above the nodes on the current side-link and updates the counts on the paths. In this way, Algorithm 1 can update the count information on the paths in place without creating a copy of each path.

To parallelize Algorithm 1, we need to remove the data dependency of the original FP-Tree: the count of conditional pattern base should not be read directly from the FP-Tree. Instead, the count information is maintained in each header table and the FP-Tree only offers the path information.
Algorithm 4 describes the parallel FP-Growth. As the count information is stored in respective header table, the constraint of the top-down order is removed. Thus, we can mine each item in a header table in parallel.

The implementation of Algorithm 4 on GPU employs the latest CUDA dynamic parallelism technique, which refers to the capability of a GPU to generate new work for itself. In this way, each thread block is in charge of the computation for each header table and each parallel-mine-tree procedure runs as a kernel function in a thread block.

In Algorithm 4, the header table requires frequent reading and writing while a created header table also needs to be transferred from parent function to child function. Thus, the memory access pattern of the header table has a great effect on the performance of Algorithm 4. So, we design a unique memory mechanism for it. A new header table is created and edited dynamically in the shared memory and the header table is copied to device memory to be accessed by child function. For child function, we use the Read-Only Data Cache, which is a new feature of Kepler GPU, to improve the access efficiency of header table. Both the shared memory and Read-Only Data Cache are on the streaming multiprocessor and achieve as low as one cycle memory access latency.

### Procedure parallel-mine-tree $(X, H)$

1. for each entry $I$ in $H$ in parallel
2. if $H(I) >= \text{minsup}$, then
3. Output $IX$;
4. buildsubtable($I$);
5. parallel-mine-tree($IX$, $H$);

### Procedure buildsubtable $(I)$

1. for each node $u$ in the Index-array of $I$ in parallel
2. Walk up the path from $u$ once do
3. if encounter a $J$-node $v$ then
4. Link $v$ into the Index-array of $J$ in $H_J$;
5. $\text{NodeCount}(v) = \text{NodeCount}(v) + \text{NodeCount}(u)$;
6. $H_J(J) = H_J(J) + \text{NodeCount}(u)$;

Algorithm 4: Parallel FP-Growth

### IV. EXPERIMENTS AND PERFORMANCE STUDY

#### A. Experiment Setup

The proposed GPU-based FP-Growth implementation was compiled using Visual Studio 2012 with CUDA 6.0. All experiments were performed on a PC with a NVIDIA Tesla K20 GPU and an Intel Core i5-2320 running Microsoft Windows 8. The Tesla K20 GPU features 13 streaming multiprocessors, 2496 CUDA cores and 5 GB GDDR5 memory with peak bandwidth of 208 GB/s. The CPU has four cores running at 3.2 GHz. The GPU uses a PCI-E bus to transfer data between the GPU memory and the main memory with a theoretical bandwidth of 16 GB/s.

#### Comparison

We compared our GPU-based FP-Growth with CPU-based Apriori, CPU-based FP-Growth and GPU-based Apriori. The single-thread CPU-based implementations were retrieved from the repository of Workshop on Frequent Itemset Mining Implementations (FIMI’04) [33]. To make a fair comparison, the CPU-based Apriori implementation was the Borglet version, which is the best Apriori implementation. The CPU-based FP-Growth implementation was the famous FP-Growth* [8], which is the best FP-Growth implementation. For the GPU-based Apriori, we chose the bitmap version used in [3], which runs entirely on the GPU.

#### Datasets

We used four representative datasets from FIMI’04 repository [33] to evaluate the four implementations, including T401I0D100K, Chess, Accidents and Retail. These datasets have distinct characteristics, which are summarized in Table I. The density of a dataset is defined as the average length of transactions divided by the number of items. T401I0D100K is a dummy dataset generated by IBM Almaden Quest research group while Accidents, Chess, Retail are real datasets and they can represent datasets with different levels of density. T401I0D100K and Accidents are examples of large datasets and Chess contains dense data with density 49%. Retail contains sparse data with density lower than 0.1%. As we focused on the in-memory performance, the sizes of all the datasets were smaller than the device memory or main memory.

#### Metric

The metric adopted in our experimental study was the total elapsed time to evaluate the efficiency of various implementations. The initial file reading and final result output steps were excluded from the measured time. Also, the time for converting the transaction database from horizontal data layout into bitmap representation was not counted. We executed each experiment for three times and recorded the mean values.

#### B. Performance Study

A preliminary experiment was conducted to evaluate the performance of the tree building procedure. Fig. 6 shows the running time (seconds) of this procedure with regard to each dataset on the GPU. We can see that constructing the binary radix tree with header table can be executed very efficiently on the GPU because these operations have a high degree of parallelism. Note that, in the sparse case, the procedure took clearly more time to run, since the size of the bitmap matrix is much larger than others.
Table 1. Four benchmark datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Items</th>
<th>Avg. Length</th>
<th>#Transactions</th>
<th>Density</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>T40I10D100K</td>
<td>1,000</td>
<td>40</td>
<td>100,000</td>
<td>4%</td>
<td>Synthetic</td>
</tr>
<tr>
<td>Chess</td>
<td>75</td>
<td>37</td>
<td>3,196</td>
<td>49%</td>
<td>Dense</td>
</tr>
<tr>
<td>Accidents</td>
<td>468</td>
<td>33.8</td>
<td>340,184</td>
<td>7.2%</td>
<td>Large</td>
</tr>
<tr>
<td>Retail</td>
<td>16,470</td>
<td>10.3</td>
<td>88,163</td>
<td>0.06%</td>
<td>Sparse</td>
</tr>
</tbody>
</table>

Fig. 7 to Fig. 10 present the running times (seconds) of the four implementations on the four benchmark datasets, respectively. Fig. 11 and Fig. 12 show the maximal speedup compared to CPU-based FP-Growth and GPU-based Apriori.

As Apriori performed very poorly with 10% threshold, it is excluded from the experiments. Furthermore, when the threshold is high (60%), the FP-Tree is compact and the CPU-FP-Growth can achieve very high efficiency while the GPU is not sufficiently utilized.
In general, on large or dense datasets, our implementation performed well and achieved over 20× speedup as the search space of these datasets is deep and wide, which can improve the occupancy rate on GPU. On Retail, CPU-FP-Growth had a poor performance because in this sparse case the FP-Tree was large, resulting in long traversal time.

V. CONCLUSION

In this paper, we proposed a novel parallel bitmap method to construct a FP-Tree on the GPU and redesigned the popular TD-FP-Growth into a parallel algorithm with improved header table. With careful implementations of tree building, parallel tree mining and optimized access of GPU memory, our method achieved up to over 20 times speedup compared to the optimal implementation of FP-Growth. In summary, for the first time, we designed a GPU accelerated FP-Growth, which can greatly improve the in-memory performance of the classical FP-Growth.

ACKNOWLEDGMENT

This work was supported by the NVIDIA CUDA Teaching Center awarded to Tsinghua University.

REFERENCES

[34] NVIDIA, Data Parallel Algorithm in CUDA SDK, Retrieved from: http://developer.download.nvidia.com