2011 IEEE International Conference on Cluster Computing

GPApriori: GPU-Accelerated Frequent Itemset Mining

Fan Zhang
Department of Computer Science
University of South Carolina
Columbia, SC, U.S.
zhangf@email.sc.edu

Yan Zhang
Department of Computer Science
University of South Carolina
Columbia, SC, U.S.
zhangy@email.sc.edu

Jason Bakos
Department of Computer Science
University of South Carolina
Columbia, SC, U.S.
jbakos@cse.sc.edu

***Abstract*—In this paper we describe GPApriori, a GPU-accelerated implementation of Frequent Itemset Mining (FIM). We tested our implementation with an Nvidia Tesla T10 graphic processor and demonstrate up to 100X speedup as compared with several state-of-the-art FIM algorithms on a CPU. In order to map the Apriori algorithm onto the SIMD execution model, we have designed a "static bitset" memory structure to represent the input database. This data structure improves upon the traditional approach of the vertical data layout in state-of-the art Apriori implementations. In our implementation, we perform a parallelized version of the support counting step on the GPU. Experimental results show that GPApriori consistently outperforms CPU-based Apriori implementations. Our results demonstrate the potential for GPGPUs in speeding up data mining algorithms.**

Keywords: Association rule mining, Frequent itemset mining, CUDA GPU computing, Parallel Computing.

I. INTRODUCTION

Frequent Itemset Mining (FIM) algorithms are used for finding common and potentially interesting patterns in large-scale databases. In FIM algorithms, the data in the database are called transactions, each of which is a set of items labelled by a unique ID. The purpose of FIM is to find the most frequently-occurring subsets from the transactions. The frequency of the subset is measured by support ratio, which is the number of transactions containing the subset divided by the total number of transactions in the database. FIM algorithms are given a minimum support ratio threshold, and returns all the frequent item sets with support ratio meeting the threshold.

FIM is common in many research and commercial applications. An example can be shown in the sales data analysis of supermarkets. Customers usually purchase goods in a pattern (e.g. people who buy vegetables often also buy salad dressing), and those common shopping patterns can be discovered by mining receipts. Analysis of those patterns can be useful for designing the layout of the supermarket: products usually sold together can be placed near each other. FIM is also useful in database management systems, information retrieval, bioinformatics, data stream analysis, and computer vision.

Our GPU implementation includes a set of fine-grained parallel data structures and algorithms design to achieve premising degree of speed up on modern GPU compared with state-of-the-art serial implementation. Experiment results show that our GPU implementation is more effective (over 100x speed up) on large and dense datasets.

II. BACKGROUND AND RELATED WORK

Three of the best-known FIM algorithms are Apriori [1, 2], Eclat [3], and FP-Growth [4]. Apriori and Eclat iteratively generate k+1-sized frequent item sets by joining frequent k-sized item sets. This step is called *candidate generation*. After generating each new set of candidates, the algorithm scans the transaction database to count the number of occurrences of each candidate. This step is called *support counting*. The primary difference between Apriori and Eclat is the way they represent candidate and transaction data and the order that they scan the tree structure that stores the candidates. FP-Growth is the most recently-developed algorithm and operates much differently. It executes two complete scans over the transaction database to build up a frequent pattern tree, and then generates frequent item sets by bottom-up traversal and identifying sections of the tree that represent frequent subsets. The main difference from the previous two approaches is that FP-Growth doesn't generate candidate sets iteratively.

In general, much of the work in FIM algorithm development were focused on serial algorithms, which is likely due to the high degree of data dependence that is fundamental to FIM methods. Single-threaded performance comparisons generally show that the FP-Growth method is generally faster than Apriori and Eclat, however, when minimum support is high, Apriori outperforms FP-Growth [5]. More importantly, however, is that Apriori contains more easily exploitable task- and data-level parallelization than GP-Growth, giving it potentially more scalability than GP-Growth for parallel execution. In other words, while GP-Growth may outperform Apriori on a single processor, Apriori has more performance potential for multi- and many-core platforms.

There has been much recent interest in implementing FIM algorithms. Ferec Bodon implemented Apriori using trie-based data structure and candidate hashing [6], Christian Borgelt implemented Apriori in his work [7] using recursion pruning, Bart Goethals implemented Apriori based on Agrawal's algorithm [2]. Comparison

978-0-7695-4516-5 2011

U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/CLUSTER.2011.61
IEEE computer society

between those implementations can be found in Bodon's work [6].

III. INTRODUCTION TO APRIORI ALGORITHM

Apriori-based frequent item mining algorithms are based on the property that the subset of a frequent itemset must be frequent, this property allows the algorithm to incrementally build longer candidates from shorter ones. In the algorithm, first we generate all 1-item candidates, test their frequencies by scanning the transaction database, keep those whose support ratio is larger than given threshold, then join the 1-item candidates to generate 2-item candidates and test the frequency of 2-items candidates. This procedure will continue until there's no frequent candidate left in the new generation.

Apriori algorithm requires all current candidate sets to be stored in memory, which can be expensive when the candidate set is large. The *trie* data structure has been developed to overcome the fast expanding of candidates. The key idea is that the candidates from the k^{th} generation and k+1th generation share the same k-length prefix, thus those candidates from different generations can be stored in a hierarchical tree structure. New candidate generation can be done by merging the leaf nodes and their siblings and appending new leaves to the current leaf layer. Figure 1 shows an example of a candidate trie.

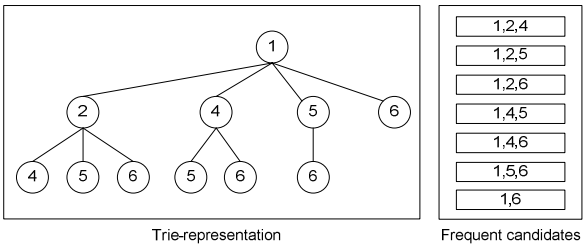


Figure 1 Example of trie representation

On the other hand, the method used to represent the transactions is also an important aspect of FIM implementation. The most straightforward way to store transactions is to store a list of items that comprise each transaction. This is called the horizontal representation. An alternative approach is the vertical representation, which instead stores a list of transaction ids that correspond to each item. This approach is referred to as a "tidset". Each list can also be represented as a bitmask, which is referred to as a "bitset".Figure 2 shows a comparison of vertical representation (tidset and bitset).

The vertical representation has been utilized by most of the state-of-art Apriori algorithms. Experimental results show that the vertical representation usually can speed up the algorithm by one order of magnitude on most of the test dataset.

When the candidates are represented as bitsets, new candidates can be generated by joining leaf nodes with siblings in the trie, and the support of the new candidate can be computed by counting number of elements in its vertical list. This method of candidate generation is called Equivalent-Class Clustering, which is first devised by Zaki et.al [8]. It speeds up candidate generation by avoiding the slow $O(n^2)$ complete join.

Transactions	ID
1,2,3,4,5	1
2,3,4,5,6	2
3,4,6,7	3
1,3,4,5,6	4

(A)

Candidate	tidset	bitset
1	1,4	1001
2	1,2	1100
3	1,2,3,4	1111
4	1,2,3,4	1111
5	1,2,4	1101
6	2,3,4	0111
7	3	0010
1,2	1	1000
1,3	1,4	1001
1,4	1,4	1001

(B)

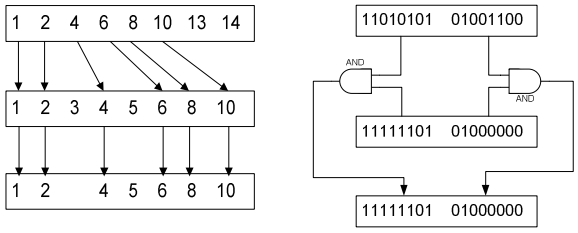
Figure 2 Comparison of horizontal representation (A) and vertical representation of transactions (B), the differences of tidset and bitset are also shown in (B)

IV. GPAPRIORI IMPLEMENTATION

In this section we describe GPApriori. The novelty of our approach is includes a new trie and vertical transaction list data structures and fine-grain parallelization of the support counting algorithm.

1) Data structure orientation

Accelerating Apriori with a GPU involves careful consideration of the vertical transaction list representation. Tidsets are stored as linear ordered arrays, and when traversing them during the support counting operation, the resultant memory access pattern and instruction stream branching behavior is unpredictable and leads to poor performance on the GPU.



(a) Tidset join is uncoalesced (b) Bitset join is coalesced

Figure 3 A comparison between tidset join and bitset join, tidset join is not continuous in memory access and may cause uncoalesced read on GPU

As shown in Figure 3, the tidset representation is compact but join operations on tidsets are highly data dependent and difficult to parallelize. On the other hand, the bitset representation requires more memory space but it is more suitable for designing a parallel set join operation, which is better suited for GPU. Joining two bit-represented

transaction lists can be performed by a "bitwise and" operation between the two bit vectors.

2) Support counting

In Apriori, Support ratio is computed by scanning transaction database to count the occurrences of the candidates. This mainly involves considerable binary searches and trie traversal, both of which will cause irregular memory access when placing on GPU.

Our GPU Support counting is based on *complete intersection.* In complete intersection, candidates are copied from main memory to graphic memory by host code, the GPU calculates their support ratio value by executing bitwise intersections on their vertical transaction lists, and the support value results are copied back to main memory.

Figure 4 shows how complete intersections are computed. Only the vertical lists of first generation will be saved in graphics memory, As shown in the example, the fourth generation is {(1,2,4,5), (1,2,4,6), (1,2,5,6),…}, and the supports are computed by intersecting (V_1,V_2,V_4,V_5), (V_1,V_2,V_4,V_6) and (V_1,V_2,V_5,V_6). Compared to the equivalent class clustering method, complete intersection adds computational complexity in order to reduce memory usage and memory operations. On a GPU, the cost of these additional logic operations is lower than performing the additional memory references required to transfer the candidates from the host.

3) Support counting on CUDA

CUDA computation is organized into threads and threads are organized into blocks. Each list intersection will be computed by one block.

Figure 5 shows how support counting is computed on one thread block. Threads within the same block will process a word-length subset, the size of vertical lists are aligned on the 64 byte boundary to ensure coalesced memory access. The intersection result of each thread is stored in a 32-bit integer, and the number of "1" bit in the integer is counted by CUDA build-in popcount function and stored in an integer array in shared (on-chip) memory. A parallel summation reduction algorithm [9] is used to add all the support values recursively into its first element . The resultant support number for the candidate is written back to graphic memory and then transferred back to main memory.

Several optimization techniques which make the algorithm be faster are (1) *candidate preloading* that is executed at the beginning of the kernel execution in which candidates will be preloaded to shared memory to prevent repeating global memory read, *manual, hand-tuned loop unrolling* to further improve the kernel speed; and (3) *hand-tuned block size*.

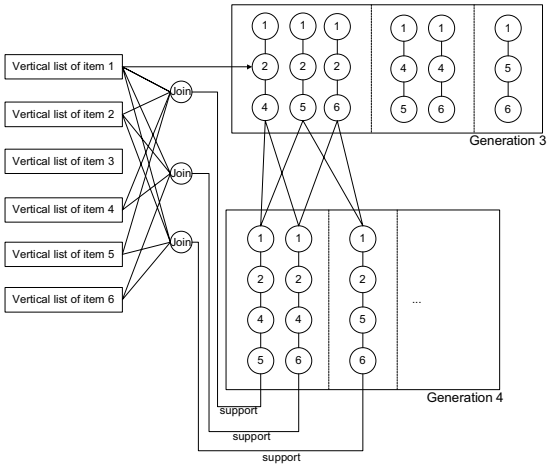


Figure 4 Complete intersection in support counting

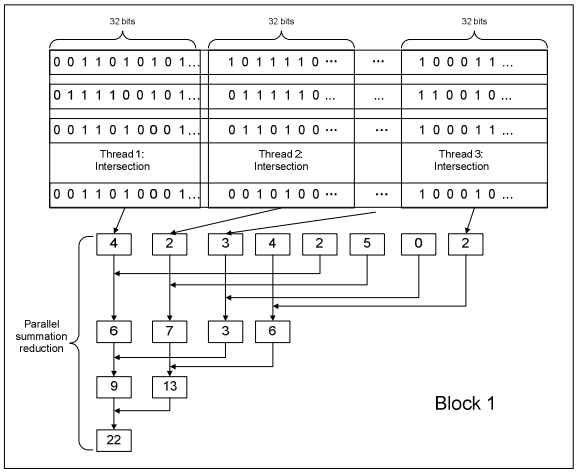


Figure 5 Thread dispatching across computation block

V. EXPERIMENTAL RESULTS

In this section we will compare our GPU implementation with various types of CPU Apriori algorithms.

A. Experimental environment

We performed our experiments using a Dell PowerEdge R710 sever connected to a Tesla S1070 GPU server with four Tesla T10 GPUs, although we currently use only one GPU. Detailed information of tested implementation is listed in Table 1.

Table 2 lists the detailed information about the datasets used in the experiments. Our benchmark datasets are from the Frequent Itemset Mining Repository [10] and include one synthetic dataset file from IBM Almaden Quest research group, T40I10D100K, two dataset files from UCI dataset and PUMSB dataset: chess and pumsb, and one dataset file from Karolien Geurts containing anonymized traffic accident data: accidents.

TABLE 1 TESTED FREQUENT ITEM MINING ALGORITHMS

Algorithm	Platform
GPApriori	Single thread GPU+ single thread CPU
CPU_TEST	Single thread CPU
Borgelt Apriori	Single thread CPU
Bodon Apriori	Single thread CPU
Gothel Apriori	Single thread CPU

TABLE 2 EXPERIMENTAL DATASETS

Dataset	#Item	Avg.length	#Trans	Type
T40I10D100K	942	40	92,113	Synthetic
pumsb	2,113	74	49,046	Real
chess	75	37	3196	Real
accidents	468	34	340,183	Real

Figure 6 shows the detailed performance comparison. Each of our performance results are listed as a speed up relative to performance given by the Borgelt implementation.Borgelt Apriori is one of the most recently developed and state-of-the-art implementations of the Apriori algorithm.

The comparison of GPApriori and CPU_TEST shows the degree of acceleration our GPU implementation achieves compared with equivalent CPU code. On the smaller dataset chess, the GPU version can achieve a 10X speed up, while on a the larger dataset accident, the speed up ranges from 50X to 80X. In general, the performance scales with the size of the dataset.

Figure 6 also shows the comparison between GPApriori and the other three Apriori algorithms (Borgelt Apriori, Bodon Apriori and Gothel Apriori). Both Bodon and Borgelt utilize the vertical tidset while Gothel uses the horizontal representation. We show only in 6(a) the performance of Gothel algorithm because it performs very slowly on the other three datasets.

Experimental results show that GPApriori outperforms Borgelt Apriori on most of the moderate sized datasets with 4X-10X speed up and on a large dataset accident, the speed up ratio can reach up to 80X.

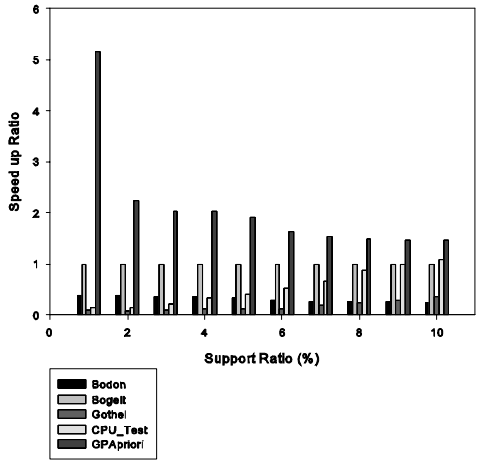


Figure 6(a)

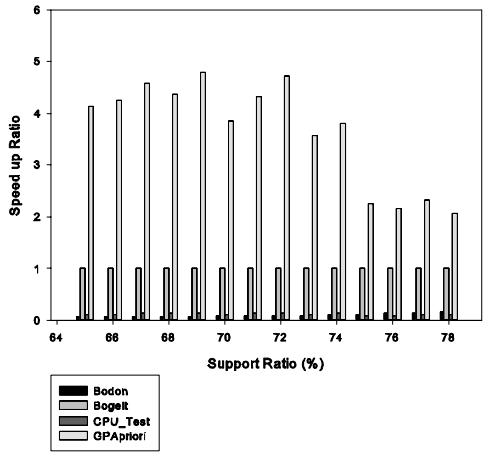


Figure 6(b)

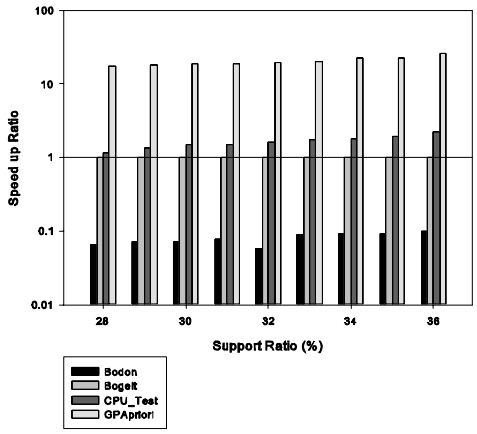


Figure 6(c)

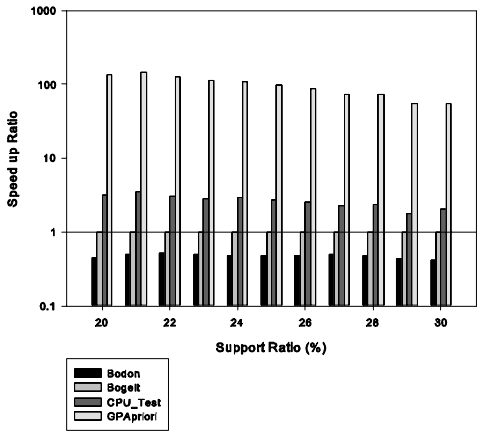


Figure 6(d)

Figure 6 Time performance comparison of Apriori algorithms on selected datasets: T40I10D100K 6(a), pumsb 6(b), chess 6(c) and accident 6(d)

VI. Conclusion and Future Work

We present a GPU parallel Apriori algorithm (GPApriori), GPApriori utilizes trie-based candidate set, vertical data layout and bit set representation of vertical transaction lists. The support counting procedure is optimized for GPU execution. The experimental results show that compared with state-of-the-art single threaded Apriori algorithm, the GPU version achieves one to two order magnitude in speed up.

Future work on the research includes how to parallelize other FIM algorithm such as FPGrowth and Eclat on GPU, as well as devise a load-balanced computation model across CPU/GPU platform and GPU cluster.

VII. References

[1] R. Agrawal and H. Mannila, *Fast Discovery of Association Rules*, in *Advances in Knowledge Discovery and Data Mining*. 1996. p. 307-328.

[2] R. Agrawal and R. Srikant. *Fast algorithm for mining association rules*. in *VLDB* 1994. p. 487-499

[3] M. J. Zaki and K. Gouda. *Fast Vertical Mining Using Diffsets*. in *Proc. SIGKDD*. 2003. p. 326-335

[4] J. Han, H. Pei, and Y. Yin. *Mining Frequent Patterns without Candidate Generation*. in *SIGMOD*. 2000. p. 1-12

[5] N. Govindaraju and M. Zaki, *Advances in Frequent Itemset Mining Implementations*, in *FIMI*. 2003.

[6] F. Bodon, *A Trie-based APRIORI Implementation for Mining Frequent Item Sequences*, in *OSDM*. 2005. p. 56-65.

[7] C. Borgelt. *Efficient Implementations of Apriori and Eclat*. in *Proc. FIMI*. 2003.

[8] M. Zaki and S. Parthasarathy, *New Algorithms for Fast Discovery of Association Rules*, in *KDD*. 1997. p. 283-296.

[9] NVidia. *Data Parallel Algorithm in CUDA SDK* Available from: http://developer.download.nvidia.com.

[10] *Frequent Itemset Mining Dataset Repository* Available from: http://fimi.ua.ac.be/data.