J Supercomput (2013) 66:94–117
DOI 10.1007/s11227-013-0887-x

Accelerating frequent itemset mining on graphics processing units

Fan Zhang · Yan Zhang · Jason D. Bakos

Published online: 2 February 2013

Abstract In this paper we describe a new parallel Frequent Itemset Mining algorithm called "Frontier Expansion." This implementation is optimized to achieve high performance on a heterogeneous platform consisting of a shared memory multiprocessor and multiple Graphics Processing Unit (GPU) coprocessors. Frontier Expansion is an improved data-parallel algorithm derived from the Equivalent Class Clustering (Eclat) method, in which a partial breadth-first search is utilized to exploit maximum parallelism while being constrained by the available memory capacity. In our approach, the vertical transaction lists are represented using a "bitset" representation and operated using wide bitwise operations across multiple threads on a GPU. We evaluate our approach using four NVIDIA Tesla GPUs and observed a 6–30× speedup relative to state-of-the-art sequential Eclat and FPGrowth implementations executed on a multicore CPU.

Keywords Association rule mining · Frequent itemset mining · CUDA GPU computing · Parallel computing

1 Introduction

The goal of Frequent Itemset Mining (FIM) is to find frequently appearing subsets within a database of sets. Many scientific and industrial applications, including those in machine learning, computational biology, intrusion detection, web log mining, and e-business benefit from the use of frequent itemset mining. Much of the literature in frequent itemset mining emphasizes the development of algorithmic improvements

F. Zhang · Y. Zhang · J.D. Bakos (✉)
University of South Carolina, Columbia, USA
e-mail: jbakos@cse.sc.edu

F. Zhang
e-mail: zhangf@email.sc.edu

as opposed to parallelizing existing algorithms. As such, state-of-the-art FIM implementations are generally single-threaded and there is relatively little effort devoted to mapping these algorithms to high-performance platforms.

The objective of FIM is to analyze a database of itemsets and identify all item subsets that appear more frequently than a given, user-specified threshold. A naive, purely combinatorial solution to this problem is to generate all possible itemsets, cross-referencing each to the database to determine its occurrence frequency. This technique is generally not practical, which has motivated the development of more efficient algorithms. Perhaps the three oldest and best-known of these are Apriori, Eclat, and FPgrowth [2, 14, 24].

Apriori uses a method where it incrementally generates candidate subsets of increasing size in a breadth-first search (BFS) order. Since support counting for each candidates of the tree is independent, Apriori can be easily parallelized by parallel computation of each branch. However, this results in extremely high memory usage and slow support counting, making it impractical for processing large datasets. Eclat, on the other hand, also uses a candidate generation strategy but uses a depth-first (DFS) order which lowers its memory requirement but makes it impossible to parallelize.

Unlike Apriori and Eclat, FPGrowth does not rely on candidate generation, but instead constructs an “FPtree” that contains all the information from the input database. It then traverses this tree to infer the frequent itemsets. FPGrowth generally has better performance than Apriori and Eclat but its high memory requirement prevents it from being used on large datasets. Single-threaded performance comparisons show that the FPGrowth is faster than Apriori and Eclat. However, in certain situations, such as when the frequency threshold is high, Apriori will outperform FP-Growth [13]. On the other hand, Apriori and Eclat contain more easily exploitable task- and data-level parallelization. As such, while FPGrowth may outperform Apriori and Eclat on a single processor, Apriori and Eclat have more performance potential for multi- and many-core platforms. Aside from Apriori, Eclat, and FPGrowth, lesser-known and more exotic FIM algorithms have been proposed, such as MAFIA [9], k-DCI [20], and AIM [12].

Another challenge inherent in improving FIM performance is that most FIM algorithms scale poorly as the dataset density increases [21]. The dataset density is defined by the average transaction size divided by the total number of unique items. Figure 1 shows how the running time and memory requirement of Eclat and FPGrowth increases linearly with the size of dataset, and non-linearly with dataset density.

In this paper, we propose a new parallel FIM algorithm that is optimized for a heterogeneous platforms consisting of GPUs and CPUs. In our approach, software on the host CPU dynamically controls the search boundary and expansion rate on behalf of the GPU kernel. This allows for efficient utilization of the computational and memory resources of the GPU coprocessor. We have also redesigned and optimized the memory allocation strategy and added a data preprocessing procedure to improve the memory utilization of the algorithm. The experimental results demonstrate the performance benefit of our technique as compared to traditional FPgrowth and Eclat. We show that our algorithm is able to process large datasets with over 15 million transaction records (average size 4G). In most of the cases it can achieve up to a 10× speed up compared to state-of-the-art CPU implementations.

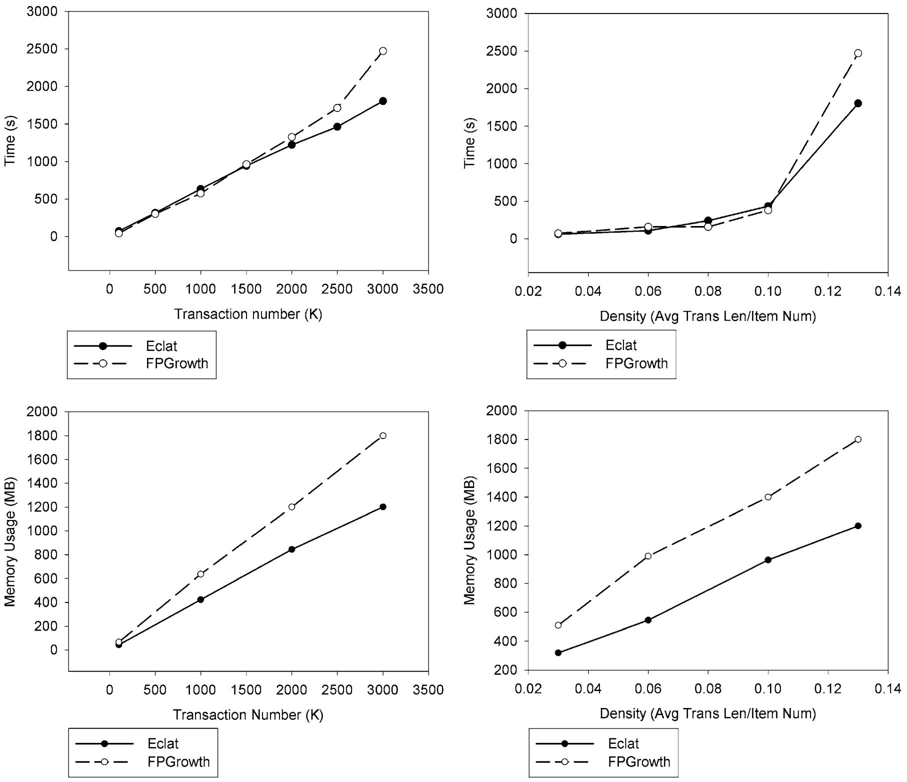


Fig. 1 A performance overview of FPGrowth and Eclat on database size and density variation, both of the algorithms perform badly on large and dense datasets

The key contributions of this work consist of the following: (1) the implementation of a finely parallelized GPU kernel for calculating the support of itemset candidates using the vertical bitvector representation, (2) a dynamic frontier expansion approach that maps candidates to blocks of GPU threads while enforcing a tight bound on memory usage, and (3) an approach for scaling to multiple GPUs.

2 Background

The Frequent Itemset Mining problem was first discovered and studied by Agrawal in 1994 [2] and has since seen intense research. Agrawal's group also developed the Apriori algorithm, a method to generate large frequent candidates from iteratively merging small frequent candidates.

Since its development, FIM methods have been improved by many research groups. Arguably, three of the most important improvements include: (1) changing the horizontal representation for transactions to a vertical representation, improving speed often the cost of additional memory requirement [23], (2) use of the trie data structure for candidate generation, which saves both time and space for joining candidates from previous generations and inserting new candidates [6], and (3) the Eclat method, which organizes frequent candidates in a lattice structure, utilizing both of the previous two methods and improving performance for many types of dataset [24].

A state-of-the-art implementation of Apriori is Borgelt Apriori [8], which carries out a BFS on the subset lattice and determines the support of itemsets using subset tests. Borgelt Apriori saves candidates in a trie and scans the transaction database to determine the support for each candidate after each generation. Borgelt also implemented Eclat using the "diffset" representation, which is a space-efficient method for representing candidates and transactions [7]. Experiments using datasets from the Frequent Itemset Mining Data Repository show that Eclat generally outperforms Apriori by an order of magnitude [5].

3 Related work

There has been recent interest in developing parallel algorithms for data mining targeting a diverse set of platforms and programming models. Current literature suggests that Apriori and Eclat have higher potential for parallelization than other methods [1, 19, 24]. Various parallel implementations of the Apriori algorithm have been developed for both shared memory and message-passing programming models. Significant sequential and parallel implementations of FIM algorithms are described below.

3.1 Sequential implementations

A well-known implementation of Apriori was developed by Bodon et al. [6]. Bodon Apriori utilizes the trie structure to improve performance. A DFS implementation of Apriori was developed by Walter et al. for the purpose of improving memory efficiency [15].

Later, Borgelt developed the well-known and highly optimized implementations of both trie-based Apriori and Eclat, as well as an FPGrowth implementation [7]. The Borgelt Eclat is capable of detecting dataset characteristics and automatically choosing the best corresponding data representation (including Tidset, Diffset, Bitset, and Rangelist). Most importantly, the Borgelt FIM implementations are publicly available and are regularly updated (the most up-to-date version of Borgelt Eclat and FPGrowth is dated Dec 2010).

3.2 Shared-memory parallel implementations

Parthasarathy et al. developed a frequent itemset mining algorithm using an SMP platform [18]. Their implementation used a hash tree to maintain the algorithm's core data. Their contribution was a load balancing method for subdividing the candidate trie amongst a group of threads. To do this, they developed a method called "candidate bitonic hashing" and corresponding data structures called the "common candidate partitioned database (CCPD)" and the "partitioned candidate common database (PCCD)". Their experiments on synthetic datasets showed an $8\times$ speedup on 12 processors. Liu et al. developed a parallel implementation of FPGrowth on multicore processors. Their implementation focused on improving cache performance for data exchange [16]. For this, they developed a cache-aware hash tree (CC-Tree) to replace the traditional FP-Tree. The resultant speedup of their implementation on eight processors was up to 6×.

3.3 Message-passing parallel implementations

A recent work by Ye et al. demonstrated a parallel Apriori algorithm based on a revised Bodon implementation that achieved a 2× speedup with eight processors [22]. Pramudiono et al. parallelized FP-Growth on a cluster and concluded that the complicated data structure of FPGrowth was a great obstacle for parallelization. Their resultant scalability was relatively low [19]. Craus et al. developed an MPI-based parallel Apriori algorithm that distributed the transactions among computing nodes prior to the start and had a fixed communication pattern [10]. However, their work focused on theoretical analysis and no real experimental results were collected. Another trie-based MPI implementation based on Bodon's algorithm was developed by Ansari et al. [3]. Their method generated the local data structure and synchronized data across nodes after candidate generation. The experimental results were collected using an eight node cluster connected with MPICH2 and they achieved a speedup of up to 6×. Aouad et al. introduced a parallel FIM algorithm on a cluster consisting of a diverse set of CPU architectures [4]. Their main contribution was to develop a load balancing strategy to manage nodes having different capabilities. The results showed that their method was able to gain 10× overall speedup under low support threshold.

3.4 GPU-accelerated implementations

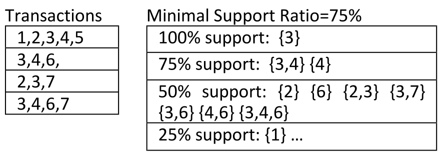
Fang et al. developed a GPU implementation of Apriori [11]. In this case, two versions of their GPU implementation, one based on the "pure bitmap" representation and another based on the "trie-based bitmap" representation were described. In their approach, the candidates and vertical transactions are coded into bitmaps and manipulated on the GPU. They used an NVIDIA GeForce GTX 280 GPU to test their algorithm. Their method achieved a speedup of 2×–10× as compared with a CPU-based serial Apriori implementation, but they were unable to outperform a CPU-based serial FPGrowth implementation.

Our previous work accelerated Apriori using a GPU and achieves up to 80× speedup over Apriori running on a CPU, though is not competitive when put against more advanced FIM algorithms such as Eclat and FPGrowth [25]. The objective of the work described in this paper is to develop a GPU implementation of Eclat that is competitive with the most advanced FIM algorithms.

4 Preliminaries

In this section, we introduce the basic concepts and algorithms in FIM. We begin by defining the problem that FIM is designed to solve and then discuss basic concepts in FIM algorithm design.

Fig. 2 A FIM example



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

Minimal Support Ratio=75%
100% support: {3}
75% support: {3,4} {4}
50% support: {2} {6} {2,3} {3,7} {3,6} {4,6} {3,4,6}
25% support: {1} …

4.1 Problem statement

FIM can be defined as follows: Given a transaction database and a minimum support, find all the item subsets with occurrence frequency higher than the given threshold. Using a supermarket metaphor, items represent merchandise–individual items for sale. A transaction (sometimes called a “basket”) represents a receipt–a combination of items that were purchased together. An itemset is a subset of the items that frequently appear in the transaction database. An itemset of size k is called a k-itemset. A FIM algorithm scans all the transactions and counts the appearance of k-itemsets within the dataset. The support of itemset X, or $support(X)$ is the number of the transactions that contain itemset X. An itemset is frequent iff its support is greater than a threshold value *min_sup*. The frequent itemset mining problem is to find all itemsets with support larger than *min_sup* in a given transaction database D. Sometimes it is more informative to represent the frequency of an itemset relative to the size of the transactions database (percentage rather than an absolute number). In this case, support ratio is used instead of support value.

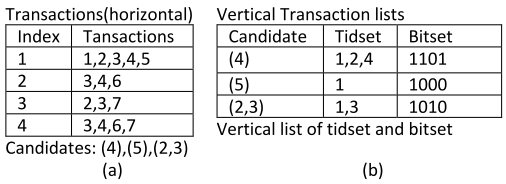
Figure 2 demonstrates a FIM example with minimal support ratio 75 %. In the figure, the itemset $\{3\}$ appears in all the transactions and is identified by support ratio 100 %, itemsets $\{3,4\}$ and $\{4\}$ appear in the first, second and fourth transactions (75 %). The other item sets from the dataset containing one through five items have 25 % support. All the itemsets with support ratio higher than or equal to 75 % comprise the output.

4.2 Current algorithms

A naive method of finding frequent itemsets would be to generate all the $k, k \in \{1..N\}$ subsets of the universe of m items, count their support by scanning the database, and output those meeting minimum support criterion. The naive method exhibits exponential complexity, since it requires the computation of the power set of m items, i.e. $\sum_{i=1}^{n} \binom{n}{i} = 2^n - 1$ and is impractical.

The earliest solution, as formulated by the Apriori algorithm, is based on the property that an item set is frequent iff all its sub-itemsets are frequent (support monotonicity). Using this property, the search space of frequent itemsets can be reduced by joining iteratively from smaller itemsets to larger ones and pruning candidates with infrequent subsets. The Apriori algorithm then went through a period of intensive study after first being published. Improvements are made in both the two critical steps of FIM, candidate generation and support counting.

Candidate generation is used to generate $k+1$ candidates from k frequent itemsets. Assume that the number of k itemsets are N, a complete join from the N itemsets expands candidate set size by $O(N^2)$. The cost of the complete join operation can


Transactions(horizontal)

Index	Tansactions
1	1,2,3,4,5
2	3,4,6
3	2,3,7
4	3,4,6,7

Candidates: (4),(5),(2,3)

(a)

Vertical Transaction lists

Candidate	Tidset	Bitset
(4)	1,2,4	1101
(5)	1	1000
(2,3)	1,3	1010

Vertical list of tidset and bitset

(b)

Fig. 3 Comparison of horizontal representation (a) and vertical representation of transactions (b)

be decreased by clustering itemsets using Equivalent Class Clustering (ECC), which prevents the creation of redundant candidates in each new generation. A detailed description of Equivalent Class Clustering can be found in the Eclat paper [24]. Applying ECC is able to reduce the quadratic time complexity to $O(\delta N)$, in which δ is the expectation of the equivalent class size. The Eclat algorithm is also the first FIM method to utilize Depth First Search order (DFS) candidate generation. DFS candidate generation reduces the memory requirement and is especially beneficial when a vertical data representation for support counting is used.

It is studied in Agrawal's work that a hybrid approach of BFS and DFS can be used to accelerate FIM on multiprocessors [1] for the candidate generation is independent on each branch of the tree. Their approach uses a "multiple candidate generation, single support counting" strategy. We will show in the next section that our approach uses an advanced "multiple candidate generation, multiple support counting" strategy that further parallelized the support couting.

Support counting is executed after the new candidates are generated; it counts the support value of the candidates, decides which of them are true frequent itemsets and removes infrequent candidates. Two ways to represent transactions in support counting are horizontal representation and vertical representation.

In the horizontal representation, each transaction ID is associated with a list of item IDs. The vertical representation is a transposition of this, where each item ID is associated with a list of transaction IDs. When using horizontal representation, support counting is performed by matching each candidate itemset against the sorted transaction database using a binary search (as used in Apriori). When using the vertical transaction representation, the support of new candidates are computed by intersecting the vertical list of the previous generation with the vertical list of the item that has been added to form the new candidate. The vertical representation speeds up support counting by saving the occurrence information for the counted candidates but in the other hand consumes more memory.

Figure 3 shows the horizontal and vertical transaction representation. Figure 3(a) shows the horizontal representation and Fig. 3(b) shows two forms of the corresponding vertical transaction lists: tidset and bitset. A tidset records itemset's occurrence information as an array of the transaction IDs, and a bitset represents the same information with a bit mask.

5 Algorithm and implementation

In this section, we describe the Frontier Expansion algorithm, which has two primary objectives. The first objective is to finely parallelize Eclat's computational kernel for

GPU acceleration. The second objective is to achieve a dynamic tradeoff between performance and memory requirement, allowing for a large dataset to be processed with limited memory.

The candidates are stored in the data structure called "Frontier Stack" maintained by the host (CPU). Each stack entry stores one candidate and a reference to its bitset vertical transaction list on GPU. The algorithm repeatedly expands the stack by consuming old candidates, generating new candidates, and deleting the infrequent candidates from the stack. The new candidates are generated by intersecting the old candidates with the common prefix on the top of the stack (this is also known as Tail Recursion). The frequency of the new candidates are computed by intersecting the bitset vertical transaction lists (support counting) on GPU.

During the frontier expansion procedure, vertical lists are frequently generated and discarded, calling for a large number of CUDA memory allocations and deallocations. To reduce the overhead of frequently invoking cudaMalloc, we developed a host-based runtime vertical list manager. It allocates space for the maximum number of vertical lists from GPU and stores the free list addresses in a stack. When the program needs a free vertical list, it pops and returns an address stored. When a vertical list is deallocated by the program, the manager revokes its address.

The vertical list manager uses a greedy strategy. It allocates the largest possible contiguous block of free memory on the GPU in order to process datasets as large as possible. More specifically, it determines the maximum GPU memory allocation size using a binary trial-and-error search approach. Once it determines the maximum memory pool, this value can be used to constrain the size of the dynamic frontier to place an upper bound on the GPU's memory usage.

5.1 Data preprocessing

Data Preprocessing is executed before the frontier stack is initialized. It takes three steps to preprocess the dataset for the purpose of reducing memory usage.

(1) Transactions, originally stored in horizontal format, are read from disk and converted to vertical format, using the $\langle item, bitvector\rangle$ representation.
(2) Because infrequent items will not appear in any frequent itemset, they can safely be removed from the dataset without altering the FIM results. In the second step, the frequency of each item is counted and the infrequent items and their corresponding vertical lists are deleted from the vertical list.
(3) The remaining items will be sorted by frequency from low to high and remapped to build a better balanced expansion search space.

Figure 4 shows an example of the steps in data preprocessing. After the preprocessing, the frontier stack is initialized by the frequent 1-item sets.

5.2 CPU candidate generation

In order to demonstrate the frontier expansion procedure, we introduce the concept of "Equivalent Class", which can be defined by a set of candidates with the same size, assumed k, shared the common $k-1$ prefix. For example, $(1, 2, 3)$, $(1, 2, 4)$ and

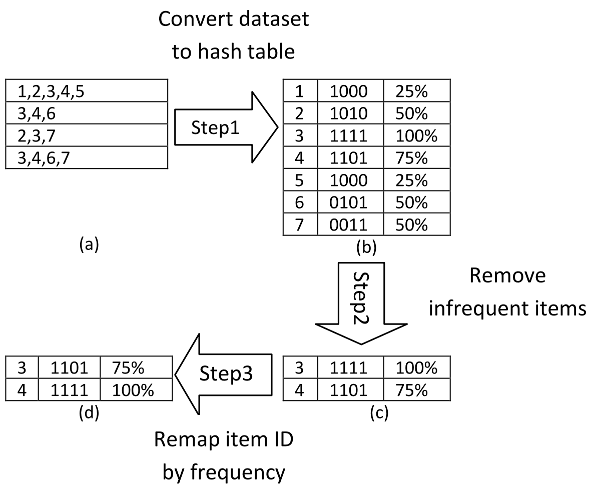


Fig. 4 The transition steps in data preprocessing

$(1, 2, 5)$ are in the same equivalent class $(1, 2, -)$. $(1, 2, 3)$ and $(1, 3, 4)$ are not in the same equivalent class because they have the different 2-prefix $(1, 2)$ and $(1, 3)$. $(1, 2, 3, 4)$ and $(1, 2, 3)$ are not in the same Equivalent Class because they have different size. All 1-item sets are in the same equivalent class.

The frontier stack is organized by grouping candidates into equivalent classes. The initial contents of the stack are 1-itemset, and these are all in the same equivalence class by definition. During the expansion, Equivalent classes are popped from the top of the stack and self-joined to generate new equivalent classes (the set of the new candidates). The support for each of these new candidates are then counted by GPU kernel. Those that meet the minimum support standard are pushed back into the stack. During the pushing-back and popping-out, the candidates in the stack are always organized in group of equivalent class and sorted descendently by their support values. The expansion procedure is repeated until the stack is empty. During each candidate generation, we keep popping equivalent classes from the stack and generate new candidates until the size of the new candidate list is larger than ε. We choose the value of ε equals the maximum parallelizable blocks number on the GPU.

Figure 5 shows an intermediate status of the frontier stack during the candidate generation of solving the FIM problem whose transaction database is given by Fig. 2 and the minimum support ratio equals 25 %.

In the pseudo code shown in Fig. 6, the if-statement returns false if the *frontier_stack* is empty (line 1 and 2). The while loop pops and generates new nodes(candidates) until *expansion_size* exceeds ε (line 6 to 8). The support of the new nodes will be counted on the GPUs by vertical list intersection (line 9). Infrequent nodes will be removed from the list and the frequent ones are pushed back to the stack (line 10 to 12). The loop continues until the stack is empty.

If the search space of frontier expansion is organized in a prefix tree, the expansion procedure can be modeled by tree-traversal—if the expansion strategy is pop-one-at-a-time, it is a depth first search. If the strategy is pop-all-at-a-time, it is breath first search. On a parallel computation architecture such as a multicore CPU or GPU, the breadth first search can process more candidates in the support counting phase and

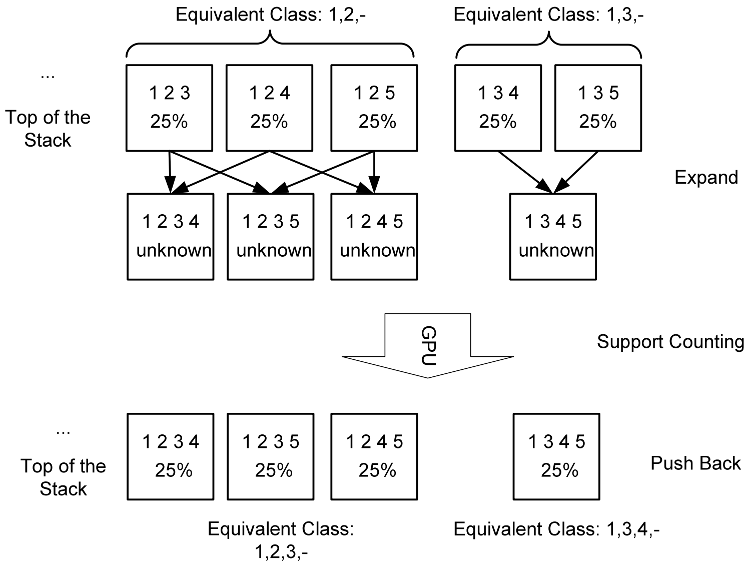


Fig. 5 An example of Frontier Expansion with $\varepsilon = 4$

Fig. 6 Pseudo code of Frontier Expansion

```
Algorithm: Frontier Expansion
Input: frontier_stack, ε, min_sup, frequent_itemset
1.  if frontier_stack is empty
2.      return false
3.  expansion_size=0
4.  frequent_itemset=∅
5.  while expansion_size < ε
6.      pop equivalent class s_eqv from stack
7.      t_eqv=expand(s_eqv)
8.      expansion_size=expansion_size+size(t_eqv)
9.      support_counting(t_eqv)
10.     remove infrequent nodes from t_eqv
11.     add t_eqv to frequent_itemset
12.     push t_eqv to frontier_stack
13. return true
```

can potentially benefit more from the data parallelism but the trade off is that the expansion requires a larger memory space. In the other hand, the expansion size (in our algorithm ε) can be chosen based on the parallelism capacity (on the GPU it is the maximum block number), to guarantee the maximum speed up with the minimal memory usage.

Figure 7 illustrates the advantage of Frontier Expansion compared with the breadth first search and depth first search. In the figure, each node is marked with the loop iteration in which each candidate is generated in the corresponding method. In other words, the edges of the tree show the loop dependencies. The numbers associated with each candidate show the size of the candidate set generated by each iteration, which determines the memory requirement of that iteration. As such, when dispatch-

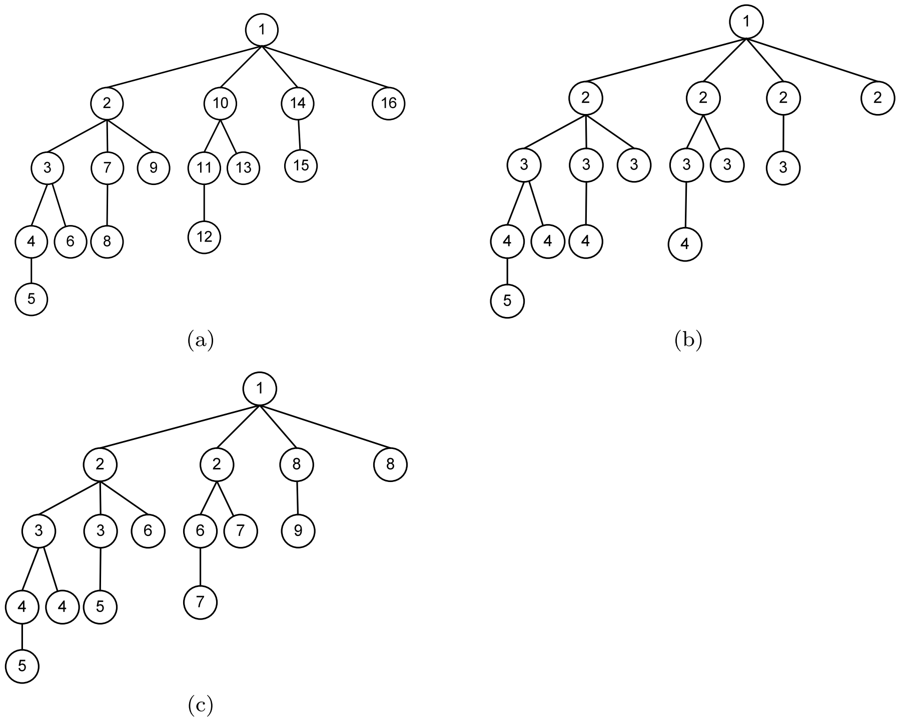


Fig. 7 Demonstration of the search tree of depth first search (**a**), breadth first search (**b**) and Frontier Expansion (**c**), assume that only two nodes can be processed concurrently. In this example the Frontier Expansion algorithm guarantees the maximum parallelism while keeping an efficient memory usage

ing a workload to a pool of processors, the set size determines the level of concurrency and thus the amount of exploitable parallelism.

Figure 7(a) demonstrates the depth first search method, it both minimizes the memory usage and parallelism. Figure 7(b) is breadth first search, which maximizes both memory usage and degree of parallelism. Figure 7(c) shows our scheduling strategy. The expansion size is adjusted according to the computation capacity of GPU devices to guarantee minimum memory usage under the maximum parallelism available of the GPUs.

Ideally, the size of each set of candidates sharing the same number will equal the number of processors while not exceeding the amount of available memory. Since the number of processors on a platform is limited, breadth first search is inefficient because it generates more candidates that can be processed in parallel and consumes more memory than required, while depth first search has no exploitable parallelism.

In other words, an optimal expansion factor ε is correlated with both the maximum parallelism available from GPU (maximum node number that can be processed simultaneously) and the size of GPU memory. Choosing a small ε does not fully utilize the CUDA computation units and choosing a large ε does not provide more parallelism but could cause memory exhaustion. A carefully selected ε from our experiments is the number of CUDA cores on Tesla T10 card which is the minimum

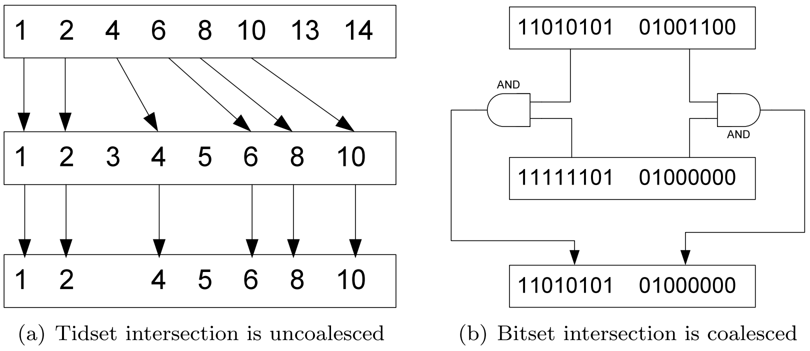


Fig. 8 A comparison between tidset intersection and bitset intersection. Tidset intersection may cause uncoalesced memory accesses on GPU. Bitset intersection is continuous in address and well coalesced

value to guarantee full parallelism on the GPU. This number is hardware dependent and should be modified if the implementation is compiled and executed on other GPU platforms.

5.3 GPU support counting

After candidate generation, the references of the vertical transaction lists of the candidates are passed to CUDA implemented GPU support counting kernel.

The CUDA thread structure consists of blocks and threads. In each thread, inherited parameter blockIdx and threadIdx can be used for self-identification. 32 threads within the same block (usually called warp) will be grouped on to the same SIMD stream processor, each stream processor is assigned to a set of warps, only one warp will be active at a time, and the stream multiprocessor rotates between warps [17].

Our GPU support counting kernel computes the vertical list intersection and sums the support value for each candidate (computed as the population count after the intersection). The GPU support counting kernel is passed a "work list" which is an $\varepsilon \times 3$ array. The array is a set of triple of (src_vlist1, src_vlist2, dst_ vlist) and stores the three operands for vertical list intersection (two source lists and one destination list). GPU blocks read in the source vertical lists (src_vlist1, src_vlist2) indexed by its block ID from the work list and execute an intersection operation. The intersection results will be stored into the destination vertical list (dst_vlist).

The bitset representation is better suited for GPU-based support counting than the tidset representation. As shown in Fig. 8, the tidset representation is compact but performing intersection operations on tidsets is highly data dependent and difficult to parallelize. On the other hand, the bitset representation is less data dependent. The length of the bitset equals the number of the transaction in the database. Each "0" or "1" represents whether the corresponding transaction contains the candidate or not. Intersection of two bit-represented transaction lists can be performed by a "bitwise and" operation between the two bitsets.

Figure 9 demonstrates how support counting is computed on the GPU. Each list intersection will be computed by one block. Threads in the block will process in-

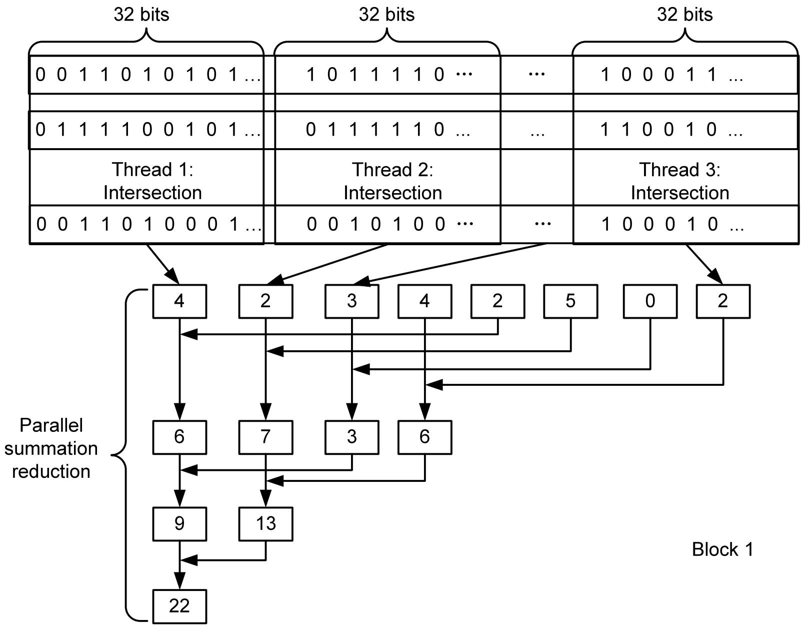


Fig. 9 Diagram of GPU support counting on one block

tersections of a word-length subset. The size of vertical lists are rounded to be the multiple of 64 bytes to ensure coalesced memory reads.

The intersection results of each thread are stored in a 32-bit integer, and the number of "1" bits in the integer is counted by CUDA popcount function and stored in an integer array sup[thread_num] in shared memory. The parallel summation reduction algorithm [17] is used to add all the values in sup[thread_ num] recursively into its first element sup[0]. The support number for the candidate is then written back to an output buffer on GPU memory and transferred back to host memory.

We applied several code optimization techniques in the kernel implementation, including: (1) work list preloading at the beginning of the kernel execution in which addresses of the source and destination vertical lists are preloaded to shared memory to prevent repeated global memory reads, (2) hot loop unrolling, and (3) determining the optimal number of threads per block to be 256 using trial-and-error-based tuning.

5.4 Producer consumer model on multi-GPUs

In order to scale our method to a multi-GPU platform, we used a producer-consumer model for splitting up the frontier stack into multithreads. The idea is that the equivalent classes can be expanded separately. After the frontier stack is initialized, the producer thread splits the stack into separate equivalent classes and stored them into the producer buffer. The consumer threads reads and process one equivalent class a time from the producer buffer.

Pseudo code of producer algorithm is shown in Fig. 10(a). The producer is assigned a single CPU thread and is created after transactions are loaded and the job

```
Algorithm: Producer
Input: Transaction file
Output: job FIFO

1.  Read transactions from file
2.  Data preprocessing
3.  for each candidate s
4.      if thread is sleeping
5.         Wait until wake up signal
6.  if job FIFO is not full
7.     Expand s to equivalent class c
8.     Push c in to the job FIFO
9.  else
10.    Set wake up timer, period=100ms
11.    Sleep
12. return
```

(a) producer

```
Algorithm: Consumer
Input: Job FIFO
Output: Frequent Item set result hash table

1.  do
2.     if thread is asleep
3.        Wait for wake up signal
4.     if job FIFO is empty
5.        Set wake up period = 100ms
6.        Sleep
7.     else
8.     Read equivalent s from job FIFO
9.     Frontier_Expansion(s, ε, min_sup)
10. while(FIFO is empty and producer
    thread has returned)
11. return
```

(b) consumer

Fig. 10 Pseudo code of producer and consumer processes

FIFO is initialized. Once the producer thread starts, it scans and preprocesses the transaction database using the method described in Sect. 5.2. The producer thread generates and inserts equivalent classes into the job FIFO until it is blocked by receiving a FULL signal from the job FIFO controller. Once it occurs, it will be put to sleep until awaken up by a timer (sent every 100 ms) or sent an EMPTY signal from the job FIFO. The producer thread returns once all the equivalent classes are generated and inserted into the job FIFO.

Figure 10(b) shows the pseudo code of consumer thread. The consumer threads read one equivalent class a time from the job FIFO and execute frontier expansion. After the data are read, the consumer threads search for an available GPU and invoke the device. The vertical transaction lists of the nodes will be copied to GPU. The consumer thread will keep reading data from the job FIFO until it receives the EMPTY signal. Once the EMPTY signal is received, the consumer thread queries the status of producer thread. When all the data has been processed the producer thread terminates. The GPU memory and GPU device will be released right before the consumer thread terminates.

The algorithm takes in four parameters, the input and output files, minimum support ratio (in percentage), and number of GPUs. If the GPU number argument is greater than the actual number of available GPU devices, it will be ignored and replaced by the actual device number. The output contains frequent itemsets, the total running time, and the time spent in support counting and candidate generation.

In general, most datasets will allow for full utilization of the GPU. However, there are certain dataset characteristics that can lead to runtime inefficiency, such as unbalanced and small datasets. Unbalanced datasets may potentially cause long branches in the search tree and adds data dependency to candidate generation. On the other hand, the vertical transaction list generated from small datasets are short. When processed by GPU support counting kernel, those short bitsets may not fully utilized the concurrent threads in stream multiprocessor and result in low throughput.

Table 1 Synthetic dataset selection and characteristic

Name	Trans#	Avg Trans Len	Density	Item number
T40I10D100 K	100 K	40	0.13	300
T40I10D500 K	500 K			
T40I10D1000 K	1000 K			
T40I10D1500 K	1500 K			
T40I10D2000 K	2000 K			
T40I10D2500 K	2500 K			
T10I5D3000 K	3000 K	10	0.03	300
T15I10D3000 K		15	0.05	
T20I10D3000 K		20	0.06	
T25I10D3000 K		25	0.08	
T30I10D3000 K		30	0.1	
T35I10D3000 K		35	0.12	
T40I10D3000 K		40	0.13	
T40I10D3000 K	3000 K	40	0.1	400
			0.08	500
			0.07	600
			0.05	700

6 Results and analysis

In this section, we describe the performance of Frontier Expansion algorithm, and the analysis on the experimental results.

Our results are tested and collected on a Dell PowerEdge R710 server. The server was connected to a Tesla S1070 GPU server with four Tesla T10 Processors. The code was written in C++ and compiled with CUDA 4.0.

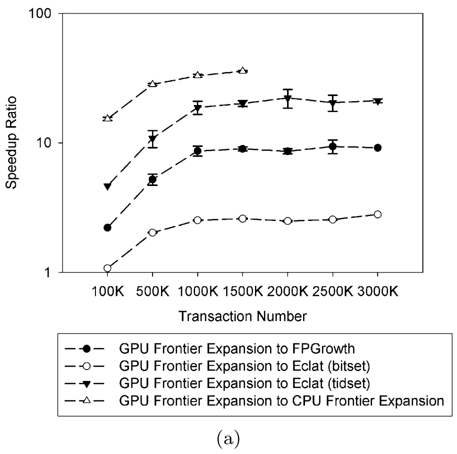
In this section when we use the term “GPU” to describe the device itself, which in our case contains 30 “streaming multiprocessor” cores, with each of these containing eight scalar cores, for a total of 240 GPU cores.

In our results we compare the performance of GPU Frontier Expansion with three algorithms, Borgelt Eclat 3.35 (which includes the tidset and bitset implementation), the original FPGrowth implemented by Goathal et al., and the proposed approach CPU Frontier Expansion (where the support counting is implemented on CPU instead of GPU).

The synthetic datasets are generated by IBM Market-Basket Synthetic Data Generator from Paolo Palmerini’s DCI website with minor changes to allow for compilation with g++ 4.4 [5]. The real datasets we use are collected from UCI Data Mining Repository. To make the datasets sufficiently large to demonstrate the speed up of GPU Frontier Expansion, we use randomize mutation algorithm to over-sample three of the most commonly used datasets: accident, connect and chess. The characteristics of the datasets can be found in Table 1 and Table 2.

Table 2 Real dataset selection and characteristic

Name	Trans#	Avg Trans Len	Item number
Accident	3400 K	34	468
Chess	1598 K	44	75
Connect	1351 K	37	127



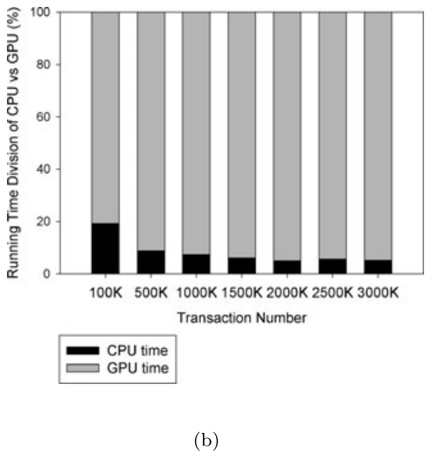


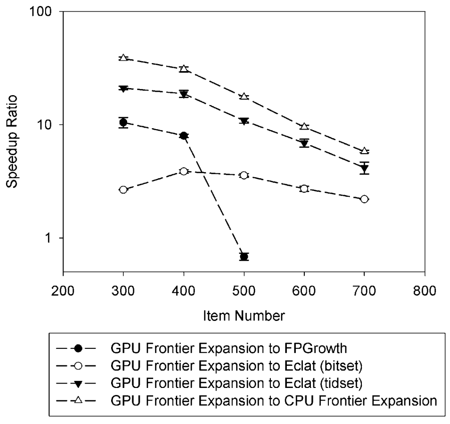
Fig. 11 Performance comparison between Eclat, FPGrowth and Frontier Expansion with variation on transaction number, the other parameter of the datasets are: Average transaction length is 40, unique item number is 300, minimum support ratio is 1 %, GPU number = 1

In each of the experimental subsections we demonstrate the overall performance and analysis of the CPU and GPU workload. We run each experiment five times, plotting the arithmetic mean with error bars.

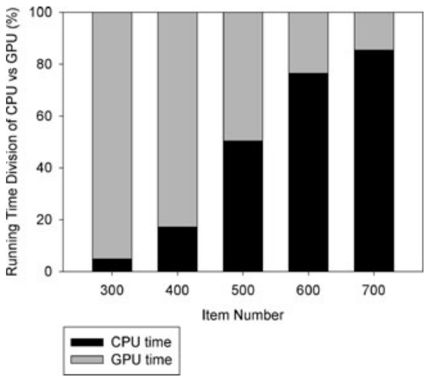
The CPU-GPU workload analysis shows the computation workloads distribution along X coordinate. The whole computation is comprised of three parts: initialization (on CPU and GPU), candidate generation (on CPU), and support counting (on GPU). We do not show the initialization part because in our approach we initialize the GPU memory with the input dataset and this data remains in the GPU memory throughout the entire execution. The resultant communication time is trivial. For example, a typical dataset size is 500 MB requires approximately 300 ms to transfer to the GPU, while typical execution times are on the order of minutes or (more commonly) hours, and sometimes even days. We do not show the synchronization overhead because it is also trivial due to the producer–consumer model used.

6.1 Performance on synthetic datasets

Figure 11 demonstrates the performance results with respect to the number of transactions in the dataset. To obtain a clear understanding on how Frontier Expansion is affected by database size, we fix the other parameters of the dataset generator to be the following: (1) average transaction length = 40, (2) unique item number = 300. The minimum support is set to 1 % and GPU number is set to 1. As shown, Frontier

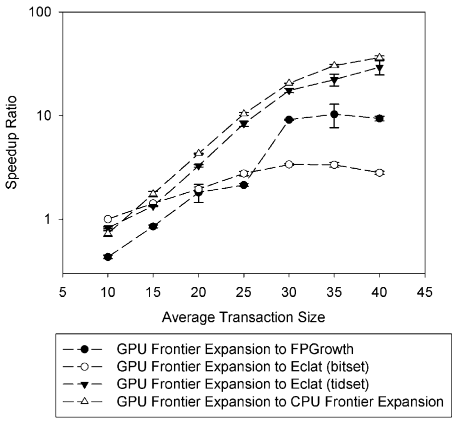


(a)

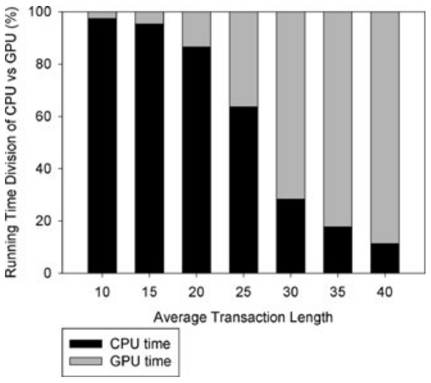


(b)

Fig. 12 Performance comparison between Eclat, FPGrowth and Frontier Expansion with variation on item number. The other parameters are set to: Transaction Number = 3000 K, minimum support ratio is 1 %, GPU number = 1



(a)



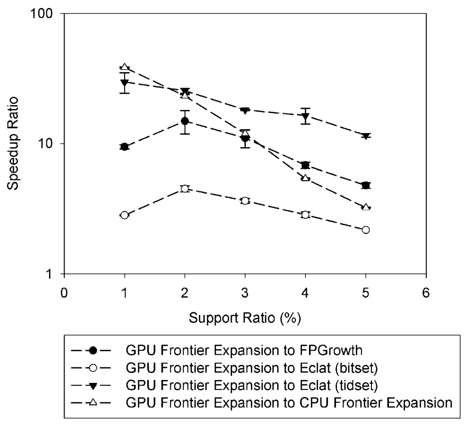
(b)

Fig. 13 Performance comparison between Eclat, FPGrowth and Frontier Expansion with variation on transaction length. The other parameters are set to: Transaction Number = 3000 K, minimum support ratio is 1 %, GPU number = 1

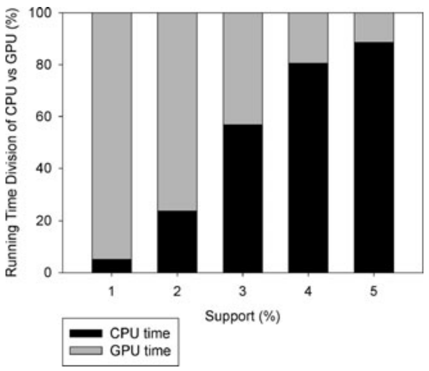
Expansion achieves a speedup scales with dataset size, which is up to 10× compared with FPGrowth and tidset Eclat, 3× to bitset Eclat, and over 30× with the CPU version.

Figure 12 demonstrates the performance and CPU-GPU workload with respect to item number variation. The dataset size is set to 3000 K, minimum support is set to 1 % and GPU number is set to 1. The speed up ratio of Frontier Expansion grows quadratically with denser datasets (less item number).

Figure 13 demonstrates the performance and CPU-GPU workload with respect to transaction length variation. The dataset size is set to 3000 K, minimum support is



(a)



(b)

Fig. 14 Performance comparison between Eclat, FPGrowth and Frontier Expansion with variation on minimum support. The other parameters are set to: Transaction Number = 3000 K, density = 0.13, GPU number = 4

set to 1 % and GPU number is set to 1. It is shown in the figures that GPU Frontier Expansion performs better on the datasets with longer transactions.

Figure 14 demonstrates the performance with respect to support threshold variation. Minimum support determines the frequency threshold of FIM, which is used by support counting to prune infrequent candidates from new candidate generation. Minimum support affects both the number of output itemsets and the search depth of the algorithm. Given a fixed-size dataset, a run with lower minimum support usually requires more time.

Our results demonstrate that Frontier Expansion achieves a higher speedup over Eclat and FPGrowth for low support ratios, which is the circumstance under which sequential algorithms become expensive. This is because lower support ratios result in more execution time required for support counting (GPU kernel), which is highly parallelized.

6.2 Performance on real datasets

Figures 15, 16, 17 show the performance study of Frontier Expansion on the three real datasets. The minimum support thresholds in the figures are scaled to a proper interval that is sufficient to demonstrate the speedup variation. On dataset accident, GPU Frontier Expansion achieves the best performance—3× compared to FPGrowth and over 30×compared to other algorithms when support is lower than 30 %. However, it is slightly slower than FPGrowth on the other two datasets, chess and connect.

6.3 Trade off between speedup and memory occupation

As described in Sect. 5, the expansion size ε is a parameter that controls the tradeoff between running time and memory occupation. Figure 18 gives an analysis of ε. In the figure, we can clearly see the memory usage is inversely correlated with the running time when ε changes.

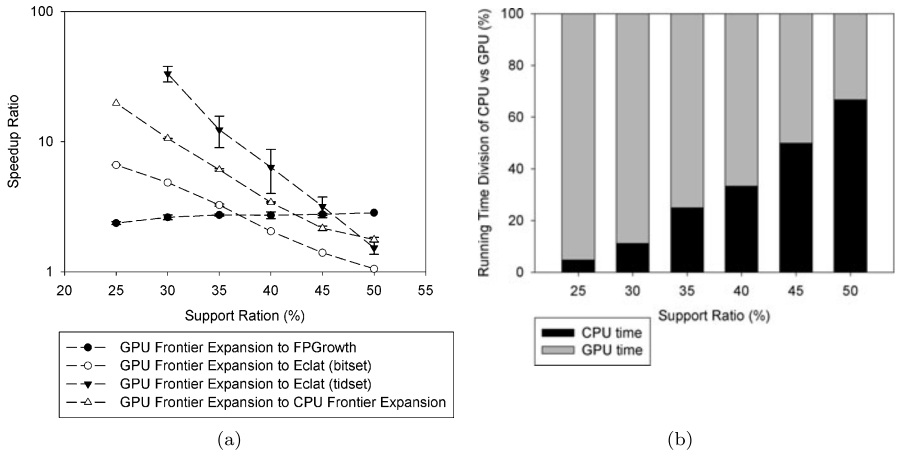


Fig. 15 Performance comparison between Eclat, FPGrowth and Frontier Expansion on dataset accident

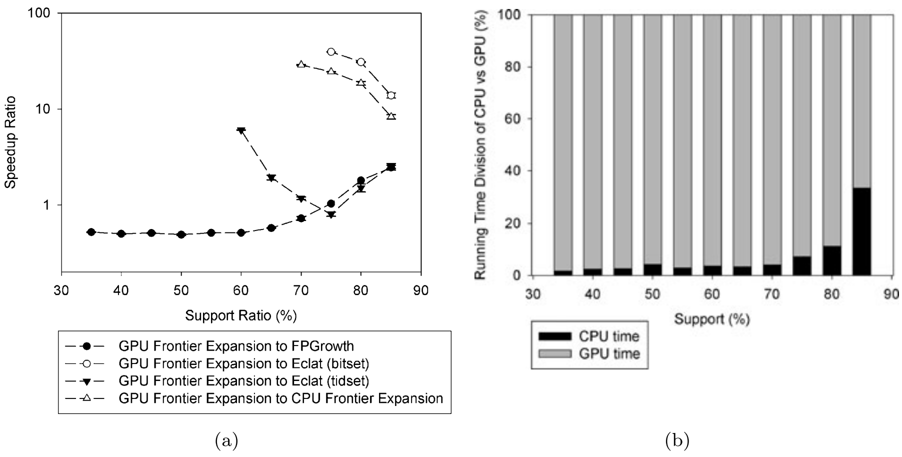
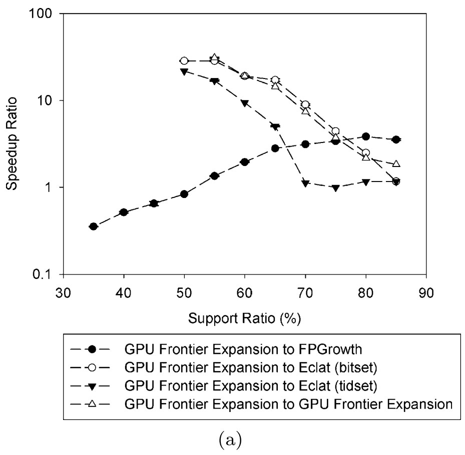


Fig. 16 Performance comparison between Eclat, FPGrowth and Frontier Expansion on dataset connect

6.4 Work load analysis

Figure 19 shows the relationship between the total running time T_{all} and the support counting kernel work load factor $T_{\text{kernel}}/T_{\text{all}}$. To correctly show the work load distribution before acceleration, we performed an experiment where we performed support counting on the CPU instead of the GPU, in order to determine the amount of execution time spent in support counting before acceleration. In the figure we can see that the workload on the support counting kernel rises with the total running time and eventually becomes the bottleneck of the program.



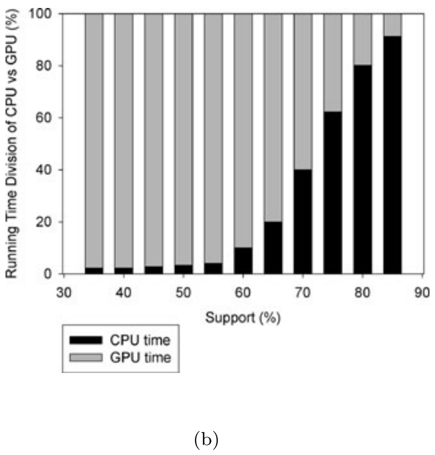
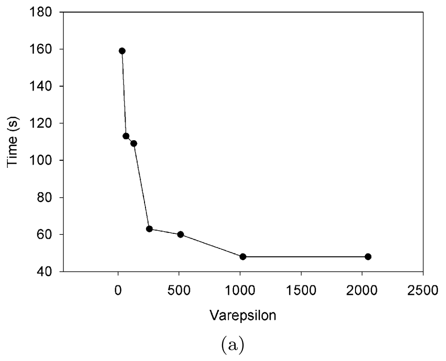


Fig. 17 Performance comparison between Eclat, FPGrowth and Frontier Expansion on dataset chess



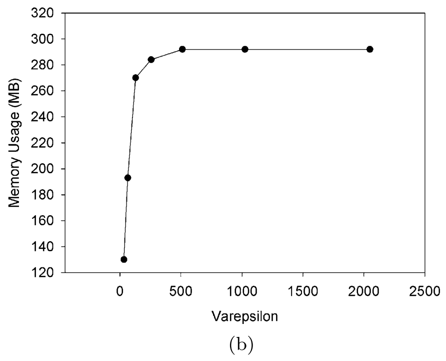
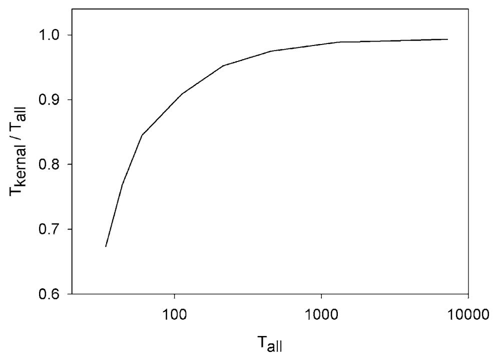


Fig. 18 Analysis of ε on dataset T40I10D500 K

Fig. 19 Work load shifting curve



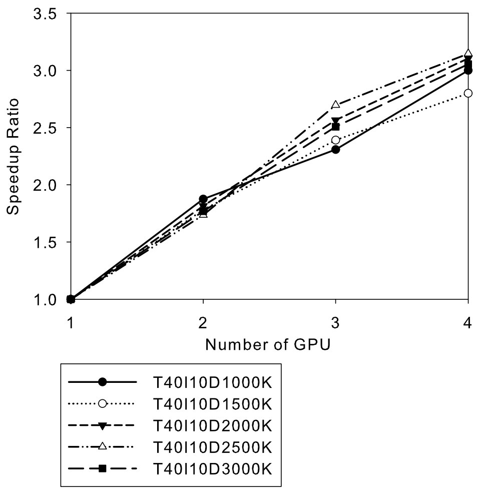


Fig. 20 Multiple GPU scalability results on T20I10D3000 K, T30I10D3000 K, T40I10D1000 K, T40I10D2000 K and T40I10D3000 K

Table 3 Real Dataset selection and characteristic

Name	Support ratio	Utilization rate
T40I10D500 K	1 %	0.99
accident	25 %	0.86
chess	50 %	0.81
connect	60 %	0.91

6.5 Multi-GPU scalability

Factors that influence scalability of our algorithm over multiple GPUs includes the following:

1. Synchronization between consumer and producer threads: the job FIFO is protected by a mutex and sometimes causes threads to wait.
2. Computation/communication ratio: for this application, the GPU is limited by memory bandwidth.
3. Unbalanced branch size of the tree: we use a producer-consumer model to schedule the GPUs, using a first-come, first served order. However, some jobs are of an abnormally large size and remain in computation after some jobs have finished. This limits the parallelism in the final stages of computation.

As shown in Fig. 20, despite these factors, Frontier Expansion is able to gain up to 3× speed up on four GPUs.

6.6 Multi-GPU utilization

Table 3 shows the utilization rate of multi-GPU Frontier Expansion on dataset T40I10D500 K (synthetic), accident, chess and connect (real). Let T_i equal the active running time of consumer thread i ($i \in \{1, \dots, N\}$). The utilization rate (a.k.a. UR) is defined by Eq. (1). The utilization rate could be brought down by an unbalanced

Table 4 Characteristic of the large datasets

Name	Trans#	Avg Trans Len	Size
T100I20D5 M	5 M	100	1.8 GB
T100I20D10 M	10 M	100	3.5 GB
T100I20D15 M	15 M	100	5.3 GB
T80I20D10 M	10 M	80	2.9 GB
T90I20D10 M	10 M	90	3.2 GB
T110I20D10 M	10 M	110	3.9 GB
T120I20D10 M	10 M	120	4.2 GB

Table 5 Speed up on large dataset, four core GPU vs. four core CPU

Dataset	Time (CPU)	Time (GPU)	Speed up
T100I20D5 M	5273	839	6.28
T100I20D10 M	11149	1797	6.20
T100I20D15 M	13341	2111	6.32
T80I20D10 M	2450	1104	2.21
T90I20D10 M	4922	1281	3.84
T110I20D10 M	23733	1395	17.0
T120I20D10 M	57296	1824	31.4

search tree (see Sect. 5 for details of the tree modeling), which is caused by bad data distribution and emerged more in real datasets. An over-weighted branch of the tree could make the other finished consumer threads wait before joining. We have

$$UR = \frac{\sum_i (T_i)}{N \times \max_i (T_i)} \tag{1}$$

6.7 Results on large dataset

We also tested Frontier Expansion on a set of large datasets. Each of these datasets is too large to be processed by the current FIM implementations (Borgelt Eclat and FPGrowth). Table 4 lists the datasets and their characteristics.

The results are shown in Table 5. Because the Borgelt Eclat and FPgrowth are not able to process the datasets, we compare the performance of GPU and CPU Frontier Expansion only. For this experiment, we compare the performance of four GPUs with four CPUs to demonstrate the GPU speed up.

The results show that on the large datasets, the GPU version is able to achieve a speed up to 30×. As the density increases, the speed up ratio grows non-linearly. The algorithm performs a stable speed up when the density is fixed, from the first three rows of the table we can discover that the speed up ratios on 5 M, 10 M and 15 M are relatively constant.

7 Conclusion

In this paper we describe Frontier Expansion, consisting of both a GPU kernel for FIM support counting based on bitset representation and a technique for dynamically managing frontier size during candidate search, allowing us to place bounds on memory use while maximizing the amount of exploitable parallelism.

We evaluated this approach for real datasets and synthetic datasets by varying dataset size, density, and the target support threshold. Our results indicate that our approach can achieve up to 30× speedup over state-of-the-art CPU-only serial implementations for datasets with high density. These results demonstrate that GPU acceleration can be beneficial for a non-uniform dynamic application and establishes a new state-of-the-art for GPU-accelerated frequent itemset mining.

We provide source code of Frontier Expansion on our research project website http://tachyon.cse.sc.edu/gpufim.html. Also we have developed a GPU mining webserver for public usage. It allows users to upload a data file, specify the minimum support, and will display the FIM results and the running time comparison of Frontier Expansion with a set of current CPU-based FIM algorithm. We are currently adding new features (noisy data preprocessing, analysis and visualization) to the web server and will also organize our other related research work on GPU data mining in the same page.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-0844951 and CCF-091560.

References

1. Agrawal R, Shafer JC (1996) Parallel mining of association rules. IEEE Trans Knowl Data Eng 8:962–969
2. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules. In: Proc of 20th intl conf on VLDB, pp 487–499
3. Ansari E, Dastghaibifard G (2008) Distributed frequent itemset mining using trie data structure. Int J Comput Sci 35(3):377–381
4. Aouad LM, Na L-k (2007) Distributed frequent itemsets mining in heterogeneous platforms. J Eng Comput Arch 1(2), ISSN: 1934–7197
5. Bart G (2004) Frequent itemset mining dataset repository. http://fimi.ua.ac.be/data/
6. Bodon F (2005) A trie-based APRIORI implementation for mining frequent item sequences. In: Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations, OSDM '05. ACM Press, New York, pp 56–65
7. Borgelt C (2003) Efficient implementations of apriori and eclat. In: Proc 1st IEEE ICDM workshop on frequent item set mining implementations (FIMI 2003), pp 90–99
8. Borgelt C, Kruse R (2002) Induction of association rules: apriori implementation. In: 15th conference on computational statistics, pp 395–400
9. Burdick D, Calimlim M (2001) Mafia: a maximal frequent itemset algorithm for transactional databases. In: Proceedings 17th international conference on data engineering, pp 443–452
10. Craus M (2008) A new parallel algorithm for the frequent itemset mining problem. In: International symposium on parallel and distributed computing, 2008, ISPDC '08, pp 165–170
11. Fang W, Lu M (2009) Frequent itemset mining on graphics processors. In: Proceedings of the fifth international workshop on data management on new hardware, DaMoN '09. ACM Press, New York, pp 34–42
12. Fiat A, Shporer S (2003) AIM: another itemset miner. In: IEEE ICDM workshop on frequent itemset mining implementations (FIMI'03)

13. Goethals B, Zaki MJ (2004) Advances in frequent itemset mining implementations: report on fimi'03. ACM SIGKDD Explor Newsl 6(1):109–117
14. Han J, Pei J (2004) Mining frequent patterns without candidate generation: a Frequent-Pattern tree approach. Data Min Knowl Discov 8:53–87
15. Kosters WA, Pijls W (2003) APRIORI, a depth first implementation. In: Proc of the workshop on frequent itemset mining implementations
16. Liu L, Li E (2007) Optimization of frequent itemset mining on Multiple-Core processor. In: VLDB '07, pp 1275–1285
17. NVIDIA (2011) NVIDIA CUDA compute unified device architecture programming guide. NVIDIA, Santa Clara
18. Parthasarathy S, Zaki MJ (1996) Parallel data mining for association rules on shared-memory multiprocessors. In: Proc Supercomputing'96, pp 43–64
19. Pramudiono I, Kitsuregawa M (2003) Parallel FP-Growth on PC cluster. In: Advances in knowledge discovery and data mining. Lecture notes in computer science, vol 2637. Springer, Berlin/Heidelberg, pp 467–473
20. Salvatore O, Claudio L (2003) kdci: a multi-strategy algorithm for mining frequent sets. In: Goethals B, Zaki MJ (eds) FIMI 03, frequent itemset mining implementations. Proceedings of the ICDM 2003 workshop on frequent itemset mining implementations, 19 December 2003, Melbourne, Florida, USA, CEUR-WS.org, CEUR workshop proceedings, vol 90
21. Sucahyo YG, Gopalan RP (2003) Efficiently mining frequent patterns from dense datasets using a cluster of computers. In: Australian conference on artificial intelligence'03, pp 233–244
22. Ye Y, Chiang C (2006) A parallel apriori algorithm for frequent itemsets mining. In: Fourth international conference on software engineering research, management and applications, pp 87–94
23. Zaki MJ, Gouda K (2003) Fast vertical mining using diffsets. In: Proc SIGKDD, pp 326–335
24. Zaki MJ, Parthasarathyi S (1997) New algorithms for fast discovery of association rules. In: 3rd intl conf on knowledge discovery and data mining. AAAI Press, Menlo Park, pp 283–286
25. Zhang F, Zhang Y, Bakos J (2012) Gpapriori: Gpu-accelerated frequent itemset mining. In: IEEE international conference on cluster computing, pp 590–594