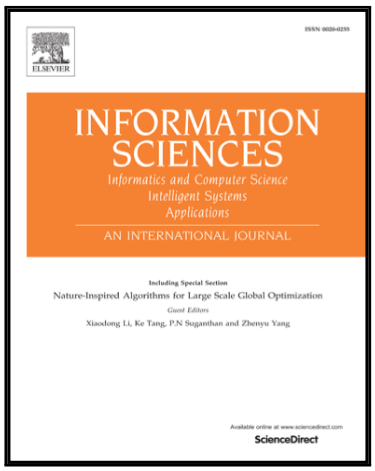


GMiner: A Fast GPU-based Frequent Itemset Mining Method for Large-scale Data

Kang-Wook Chon, Sang-Hyun Hwang, Min-Soo Kim

PII: S0020-0255(18)30069-0
DOI: 10.1016/j.ins.2018.01.046
Reference: INS 13400

To appear in: *Information Sciences*

Received date: 19 March 2017
Revised date: 16 January 2018
Accepted date: 25 January 2018

Please cite this article as: Kang-Wook Chon, Sang-Hyun Hwang, Min-Soo Kim, GMiner: A Fast GPU-based Frequent Itemset Mining Method for Large-scale Data, *Information Sciences* (2018), doi: 10.1016/j.ins.2018.01.046

GMiner: A Fast GPU-based Frequent Itemset Mining Method for Large-scale Data

Kang-Wook Chon[a], Sang-Hyun Hwang[a], Min-Soo Kim*,[a]

[a]*DGIST (Daegu Gyeongbuk Institute of Science and Technology), Daegu, Republic of Korea*

Abstract

Frequent itemset mining is widely used as a fundamental data mining technique. However, as the data size increases, the relatively slow performances of the existing methods hinder its applicability. Although many sequential frequent itemset mining methods have been proposed, there is a clear limit to the performance that can be achieved using a single thread. To overcome this limitation, various parallel methods using multi-core CPU, multiple machine, or many-core graphic processing unit (GPU) approaches have been proposed. However, these methods still have drawbacks, including relatively slow performance, data size limitations, and poor scalability due to workload skewness. In this paper, we propose a fast GPU-based frequent itemset mining method called GMiner for large-scale data. GMiner achieves very fast performance by fully exploiting the computational power of GPUs and is suitable for large-scale data. The method performs mining tasks in a counterintuitive way: it mines the patterns from the first level of the enumeration tree rather than storing and utilizing the patterns at the intermediate levels of the tree. This approach is quite effective in terms of both performance and memory use in the GPU architecture. In addition, GMiner solves the workload skewness problem from which the existing parallel methods suffer; as a result, its performance increases almost linearly as the number of GPUs increases. Through extensive experiments, we demonstrate that GMiner significantly outperforms other representative sequential and parallel methods in most cases, by orders of magnitude

☆Fully documented templates are available in the elsarticle package on CTAN.

*Corresponding author

Email addresses: kw.chon@dgist.ac.kr (Kang-Wook Chon), sanghyun@dgist.ac.kr (Sang-Hyun Hwang), mskim@dgist.ac.kr (Min-Soo Kim*,)

on the tested benchmarks.

Key words: frequent itemset mining, graphics processing unit, parallel algorithm, workload skewness

1. Introduction

As a fundamental data mining technique, frequent itemset mining is widely used in a wide range of disciplines such as market basket analysis, web usage mining, social network analysis, intrusion detection, bioinformatics, and recommendation systems. However, the deluge of data generated by automated systems for diagnostic or analysis purposes makes it difficult or even impossible to apply mining techniques in many real-world applications. The existing methods often fail to find frequent itemsets in such big data within a reasonable amount of time. Thus, in terms of computational time, itemset mining is still a challenging problem that has not yet been completely solved.

Many sequential frequent itemset mining methods such as Apriori [2], Eclat [35], FP-Growth [14], and LCM [30] use a single CPU thread. However, these single-threaded applications all have a fundamental mining performance limit because CPU clock speed is generally no longer increasing. To overcome the single-thread performance limit, multiple parallel frequent itemset mining methods have been proposed. These methods can be categorized into three main groups: (1) (CPU-based) multi-threaded methods, (2) distributed methods, and (3) graphic processing unit (GPU)-based methods. We omit the term "multi-thread" from the GPU-based methods because they are obviously multi-threaded. The first group focuses on accelerating the performance of the single-threaded methods by exploiting multi-core CPUs [20, 24, 26, 27, 29], while the second group tries to accelerate the performance by exploiting multiple machines [12, 17, 19]. Details about these methods are available in recent survey studies [9, 31].

The third group, namely, the GPU-based methods, focuses on accelerating the performance by exploiting many-core GPUs [7, 15, 18, 28, 37–39]. Due to the higher theoretical computing performance of GPUs for certain types of tasks compared with CPUs, it has become increasingly important to exploit the capabilities of GPUs in a

wide range of problems, including frequent pattern mining. However, existing GPU-based methods all suffer from data size limitations due to limited GPU memory. GPU memory tends to be much smaller than main memory. Most of the methods can only find frequent patterns in data loaded into GPU memory, which includes the input transaction data and intermediate data generated at the intermediate levels of the pattern space. To the best of our knowledge, Frontier Expansion [38] is the only method in this group that can handle larger input transaction data than GPU memory while simultaneously exploiting multiple GPUs. However, it still cannot address the same data sizes as CPU-based methods, because it cannot store sufficiently large amounts of data at intermediate levels of the pattern space in GPU memory.

Most existing parallel methods of the above three groups also suffer from the problem of workload skewness. Workload skewness is extremely common and significantly affects parallel computing performance. The existing parallel methods usually divide the search space of the patterns to be explored into multiple chunks (e.g., equivalence classes) and assign each chunk to a processor (or machine). Each subtree of the enumeration tree tends to have a different workload size. As a result, these methods are not particularly scalable in terms of the number of CPUs, machines, or GPUs. That is, their performance does not increase proportionally as the number of processors increases.

In this paper, we propose a fast GPU-based frequent itemset mining method called GMiner for large-scale data. Our GMiner method achieves high speed by fully exploiting the computational power of GPUs. It can also address the same data sizes as CPU-based methods-that is, it solves the main drawback of the existing GPU-based methods. GMiner achieves this by mining the patterns from the first level of the enumeration tree rather than storing and utilizing the patterns at intermediate levels of the tree. This strategy might look simple, but it is quite effective in terms of performance and memory usage for GPU-based methods. We call this strategy the Traversal from the First Level (TFL) strategy. The TFL strategy does not store any projected database or frequent itemsets from the intermediate levels of the enumeration tree in GPU memory; instead, it finds all the frequent itemsets using only the frequent itemsets from the first level, denoted as F_1. This strategy reduces the amount of GPU memory used and simultaneously and paradoxically improves the performance. This

result seems somewhat counterintuitive but makes sense in a GPU architecture, where the gap between processor speed and memory speed is quite large. In most cases, mining the frequent n-itemsets by performing a large amount of computation based on a small F_1 set is faster than mining the same result by performing a smaller amount of computation based on a large set of frequent (n-1)-itemsets under the GPU architecture. Using the TFL strategy, GMiner improves the performances of the representative parallel methods, including multi-threaded, distributed, and GPU-based methods, by orders of magnitude. In addition to the TFL strategy, we also propose a strategy called Hopping from the Intermediate Level (HIL), to further improve the performance on datasets that contain long patterns. Intuitively, the HIL strategy reduces the required computation by utilizing more GPU memory, thereby improving the performance for long patterns. In addition to fast mining with efficient memory usage, GMiner solves the workload skewness problem of the existing parallel methods. As a result, GMiner's performance increases almost linearly as the number of GPUs increases. To solve the workload skewness problem, we propose the concepts of a transaction block and a relative memory address. The former is a fixed-size chunk of bitwise representations for transactions, while the latter is an array representation for candidate itemsets. For parallel processing, GMiner does not divide the search space of the enumeration tree into sub-trees; instead, it divides an array of relative memory addresses into multiple subarrays, all of which have the same size. Then, GMiner stores a subarray in each GPU and performs mining by streaming transaction blocks to all the GPUs so that each GPU is assigned almost the same workload. The main contributions of this paper are as follows:

- We propose a new, fast GPU-based frequent itemset mining method named GMiner that fully exploits the GPU architecture by performing a large amount of computation on a small amount of data (i.e., frequent 1-itemsets).
- We propose a strategy called HIL that can further improve the performance on datasets that contain long patterns by performing a moderate amount of computation based on a moderate amount of data.
- We propose a method to solve the workload skewness problem by splitting an

array of relative memory addresses for candidate itemsets among GPUs and streaming transaction blocks to all GPUs.

- Through experiments, we demonstrate that GMiner significantly outperforms most of the state-of-the-art methods that have been addressed in recent studies [4, 9, 25, 31, 33] on two kinds of benchmarks.

The source code for GMiner is available at **https://infolab.dgist.ac.kr/GMiner**. The remainder of this paper is organized as follows. Section 2 discusses the related work. We propose the TFL strategy in Section 3, and in Section 4, we propose the HIL strategy. In Section 5 we present a method that exploits multiple GPUs and the cost model of GMiner. Section 6 presents the results of experimental evaluations, and Section 7 summarizes and concludes this paper.

2. Related Work

The frequent itemset mining problem is usually defined as the problem of determining all itemsets $\mathcal{F}$ that occur as a subset of at least a pre-defined fraction $minsup$ of the transactions in a given transaction database $D = \{t_1, t_2, ..., t_n\}$, where each transaction t_i is a subset of items from $\mathcal{I}$ [1, 13]. In this paper, we mainly use the number of occurrences, instead of a fraction, as the support of an itemset. Many sequential and parallel frequent itemset mining methods have been proposed. We categorize the parallel methods into three groups: (1) (CPU-based) multi-threaded methods, (2) distributed methods, and (3) GPU-based methods. Their characteristics and representative methods are summarized in Table 1 and are explained in detail in Sections 2.1-2.4.

2.1. Sequential Methods

Many sequential methods have been proposed for frequent pattern mining. The representative methods include Apriori [2], Eclat [35], LCM [30], and FP-Growth [14]. Apriori is based on the anti-monotone property: if a k-itemset is not frequent, then its supersets can never become frequent. Apriori repeatedly generates candidate (k+1)-itemsets C_{k+1} from the frequent k-itemsets F_k (where $k \geq 1$) and computes the support of C_{k+1} over the database D for testing. Borgelt [6] is a well-known implementation

Table 1: Categorization of the existing frequent itemset mining methods.

	sequential	**parallel**		
		CPU		**GPU**
	(CPU)	**multi-threaded**	**distributed**	
relative computational power	low	medium	high	high
difficulty of workload balancing	N/A	medium	high	high
network communication overhead	X	X	O	X
processor memory limit	X	X	X	O
representative methods (used in experimental study)	Apriori (Borgelt) [6], Eclat (Borgelt) [6], Eclat (Goethals) [10], LCM [30], FP-Growth* [11]	FP-Aray [20], ShaFEM [32], MC-Eclat [26]	MLlib [3]	TBI [7], GPApriori [37], Frontier Expansion [38]

of Apriori that exploits a prefix tree to represent the transaction database and finds frequent itemsets directly with the prefix tree to calculate support efficiently. Eclat [35] uses the equivalence class concept to partition the search space into multiple independent subspaces (*i.e.*, subproblems). Its vertical data format makes it possible to perform support counting efficiently by set intersection. Goethals et al. [10] and Borgelt [6] are well-known implementations of Eclat that optimize it using the diffset [34] representation for candidate itemsets and transactions. The superiority of both methods to other vertical methods has been demonstrated on the Frequent Itemset Mining Implementations (FIMI) competitions (i.e., FIMI03 and FIMI04) [8]. LCM is a variation of Eclat that combines various techniques such as a bitmapped database, prefix tree, and the occurrence deliver technique. As a result, LCM achieved the overall best performance among sequential methods in the FIMI04 competition. FP-Growth [14] builds an FP-Tree from the database and recursively finds frequent itemsets by traversing the FP-Tree without explicit candidate generation. It outperforms the Apriori-based methods in many cases. FP-Growth* is a well-known implementation of FP-Growth that reduces the number of tree traversals by exploiting additional array data structures. FP-Growth*'s superiority was demonstrated in the FIMI03 competition.

2.2. Multi-threaded Methods

Many efforts have been made to parallelize sequential methods using multiple threads to improve the performance [20, 26, 32]. FP-Array [20], based on FP-Growth,

utilizes a cache-conscious FP-Array built from a compact FP-Tree and a lock-free tree construction algorithm. In an experimental study, FP-Array improved the performance by up to six times on eight CPU cores. MC-Eclat [26] is a parallel method based on Eclat. MC-Eclat utilizes three parallel mining approaches, namely, independent, shared, and hybrid mining, and it greatly improves the performance on relatively small datasets. ShaFEM [32] is a parallel method that dynamically chooses mining strategies based on dataset density. In detail, it switches between FP-Growth and Eclat based on dataset characteristics. In many cases, multi-threaded methods greatly improve the performance compared to sequential methods. However, they fail in pattern mining due to out-of-memory failures on some datasets that sequential methods handle successfully and tend to require more memory than the sequential methods due to the large amounts of memory used by the independent threads.

2.3. Distributed Methods

In theory, distributed methods that exploit many machines can address large-scale data. Several distributed methods [3, 19, 22] have been proposed, all of which are based on a shared-nothing framework such as Hadoop or Spark. Lin et al. [19] proposed parallel methods based on Hadoop for the Apriori approach. Moens et al. [22] proposed Dist-Eclat and BigFIM. Dist-Eclat is based on the Eclat approach and BigFIM is a hybrid approach between Apriori and Eclat. MLlib of Spark [3] includes a parallel version of FP-growth called PFP. PFP is an in-memory distributed method that runs on a cluster of machines. It builds independent FP-Trees and then performs frequent itemset mining independently on each FP-Tree in each machine. Although the distributed methods should be able to handle larger data, or greatly improve the performance by adding more machines, they do not show such results in many cases due to workload skewness. According to the experimental results (which will be presented in Section 6), distributed methods can result in even worse performance than do multi-threaded methods that use a single machine due to the large amount of network communication overhead.

2.4. GPU-based Methods

Modern GPUs have many computing cores that allow multiple simultaneous executions of a kernel, which is a user-defined function. In addition, using GPUs in a single machine does not involve network communication overhead. GPUs have radically different characteristics than CPUs, including the Single Instruction, Multiple Threads (SIMT) model and the importance of coalesced memory access. These differences make it difficult to apply most parallel methods using complex data structures (e.g., FP-Array) to GPUs directly and efficiently. Thus, most GPU-based methods have been proposed based on Apriori [7, 15, 18, 28, 31, 37].

Fang et al. [7] presented two GPU-based methods: Pure Bitmap Implementation (PBI) and Trie-Based Implementation (TBI). These methods represent a transaction database as a n x m binary matrix, where n is the number of itemsets and m is the number of transactions, thereby making it suitable for the GPU architecture. These methods perform intersection operations on rows of the binary matrix using a GPU to count support. PBI and TBI outperform the existing sequential Apriori methods, such as the Apriori implementation written by Borgelt [6], by factors of 2-10. However, according to Fang et al. [7], these methods are outperformed by the existing parallel FP-Growth methods by factors of 4-16 on the PARSEC benchmark [5]. TBI is superior to PBI in terms of the number of candidate itemsets that can be handled simultaneously; therefore, we compare TBI with our method in Section 6.

Zhang et al. [37] presented GPApriori, which generates a so-called static bitmap that represents all the distinct 1-itemsets and their tidsets. Similar to other GPU-based Apriori methods, GPApriori uses a GPU only to parallelize the support counting step. The candidate generation step is performed using CPUs. GPApriori adopts multiple optimizations, such as pre-loading candidate itemsets into the shared GPU memory and using hand-tuned GPU block sizes. Consequently, it shows a speed-up of up to 80 times on a small dataset that can fit into GPU memory compared with some sequential Apriori methods (e.g., that of Borgelt [6]). However, according to Zhang et al. [37], GPApriori could not outperform state-of-the-art sequential methods such as FP-Growth* [11], Eclat [6], and LCM [30].

In [28], the authors proposed a parallel version of the Dynamic Counting Itemset al-

gorithm (DCI) [23], a variation of Apriori in which two major DCI operations, namely, intersection and computation, are parallelized using a GPU. They proposed two strategies: a transaction-wise approach (called tw) and a candidate-wise approach (called cw). The tw strategy uses all GPU cores for the same candidate simultaneously, and each thread oversees a part of the data, while the cw strategy handles many candidate itemsets simultaneously. We omit these methods in Table 1 and in our experiments, because the tw strategy is almost the same as TBI, and the cw strategy works for only very-small datasets [28].

The above three Apriori-based methods, which use GPUs, have a common serious drawback: they cannot handle datasets larger than GPU memory. Therefore, using them for real large-scale datasets is difficult because GPU memory is quite limited (e.g., to a few GB). In addition, the above methods did not outperform the representative sequential methods (e.g., LCM) as well as the representative multi-threaded methods (e.g., FP-Array) [7, 28, 37].

According to the recent survey papers on frequent itemset mining [9, 31], Frontier Expansion [38] is the only GPU-based method that can handle datasets larger than GPU memory. Frontier Expansion is based on Eclat rather than Apriori, and it utilizes multiple GPUs. The authors showed that it outperforms the sequential Eclat and FP-Growth methods [38], which were previously known to be the fastest methods in their categories. However, it fails to outperform some state-of-the-art multi-threaded methods such as FP-Array (as shown by our experimental results in Section 6). We found that Frontier Expansion's failure is due to three major drawbacks: (1) it stores a large amount of intermediate-level data in GPU memory (wasting GPU clock cycles); (2) it has a large data transfer overhead between main memory and GPU memory; and (3) it is not scalable in terms of the number of GPUs. We will explain how the proposed GMiner method solves these drawbacks in Sections 3-5.

3. TFL Strategy

For fast frequent itemset mining, even for large-scale data, GMiner uses the *Traversal from the First Level* (TFL) strategy of mining the patterns from the first level, i.e., F_1, of the enumeration tree. The TFL strategy does not store any projected database or

frequent itemsets from the intermediate levels of the enumeration tree in GPU memory; instead, it finds the entire frequent itemsets using only F_1. This approach significantly reduces GPU memory usage; thus, it can address large-scale data without encountering out-of-memory problems. In addition, to eliminate the data transfer overhead between main memory and GPU memory, GMiner performs pattern mining while streaming transaction databases from main memory to GPU memory. Here, GMiner splits the transaction database into blocks and streams them to GPUs. This block-based streaming approach allows us to solve the workload skewness problem, as explained in Section 5. Sections 3.1 and 3.2 explain the transaction blocks and the block-based streaming approach, respectively. Section 3.3 presents the algorithm that implements the TFL strategy.

3.1. Transaction Blocks

It is important that the data structures are simple and use a regular memory access pattern to fully exploit the computational power of GPUs in terms of workload balance among thousands of GPU cores and coalesced memory access. In general, compared with CPUs, the arithmetic and logic units (ALUs) and memory scheme of GPUs are not efficient for handling complex or variable-sized data structures, including sets, lists, maps, and their combinations. Furthermore, GPUs have only limited memory, which is a major obstacle for frequent itemset mining on large-scale and/or dense datasets using GPUs.

For computational efficiency, GMiner adopts a vertical bitmap layout for data representation. The horizontal layout and vertical tidset layout are too complex and irregular to maximize GPU computational efficiency. Frequent itemset mining using the vertical bitmap layout relies heavily on bitwise `AND` operations among large-scale bitmaps, where GPUs have an overwhelming advantage over CPUs.

Moreover, the vertical bitmap layout allows us to easily partition the input database vertically into subdatabases, each of which can fit in main memory or GPU memory. Hereafter, we denote an input database D in the vertical bitmap layout as a *transaction bitmap*. We define the vertical partitioning of a transaction bitmap in Definition 1.

Definition 1. ***(Transaction bitmap partition)*** *We vertically divide the transaction bitmap TB into R non-overlapping partitions of the same width and denote them by $TB_{1:R}$, where TB_k denotes the k-th transaction bitmap partition ($1 \leq k \leq R$).*

As in other frequent itemset mining methods, GMiner begins by mining the frequent 1-itemsets $|F_1|$; therefore, the size of TB is $|F_1|$ x $|D|$ in bits, where $|D|$ is the total number of transactions. When we denote the width of a single partition of the transaction bitmap as W, the size of TB_k becomes $|F_1|$ x W. If the number of transactions of the last partition TB_R is less than W, GMiner pads the partition with 0 values to guarantee the width of W.

The parameter W should be set to a sufficiently small value to fit each TB_k into GPU memory. For instance, we typically set W to 262,144 transactions in our experimental evaluation, which equals to 262,144/8=32 KB for each 1-itemset. We consider each TB_k of size $|F_1|$ x W as a transaction block. The transaction blocks are allocated consecutively in main memory (or stored as chunks in secondary storage similar to a disk page).

A frequent 1-itemset x ($x \in F_1$) has a bit vector of length $|D|$ in TB, which is subdivided into R bit vectors of length W. We denote a bit vector of x within TB_k as $TB_k(x)$. As mentioned above, TB contains only the bit vectors for frequent 1-itemsets. Thus, if x is a frequent n-itemset, x has n bit vectors in TB_k, i.e., $\{TB_k(i)|i \in x\}$. We define a set of physical pointers to the bit vectors for a frequent itemset x in the transaction bitmap in Definition 2.

Definition 2. ***(Relative memory address)*** *We define a relative memory address of an item i, denoted as $RA(i)$, as the distance in bytes from the starting memory address of TB_k to that of $TB_k(i)$, for a transaction block TB_k. Then, we define a set of relative memory addresses of a frequent itemset x, denoted as $RA(x)$, as $\{RA(i)|i \in x\}$.*

This concept facilitates the fast access to a memory location of an itemset (or memory locations of itemsets) within a single transaction block in main memory or GPU memory. $RA(x)$ is used as an identifier for an itemset x in GMiner. We denote the number of items in x as $|x|$ and the number of distinct memory addresses of $RA(x)$

as $|RA(x)|$. Then, $|x| = |RA(x)|$, because each item $i \in x$ has its own unique memory address in TB_k. We note that $RA(x)$ for a frequent itemset x does not change across all TB_k $(1 \leq k \leq R)$; that is, it always has the same relative memory addresses because the size of TB_k is fixed.

3.2. Nested-Loop Streaming

GMiner finds frequent itemsets using the candidate generation and testing approach with breadth-first search (BFS), as in Apriori, which repeats two major steps, namely, candidate generation and testing (support counting), at each level of an itemset lattice. Generally, the testing step is more computationally intensive than the candidate generation step. Thus, GMiner focuses on accelerating the testing step by exploiting GPUs. The candidate generation step is performed using CPUs.

GMiner uses BFS traversal rather than DFS traversal (e.g., equivalence classes) to fully exploit the massive parallelism of GPUs and achieve better workload balance. When using BFS traversal, the number of frequent itemsets at a certain level could become too large to be stored in the limited GPU memory and used for support counting of the candidate itemsets of the next level. The increase in the number of transactions makes the problem more difficult. Therefore, existing GPU-based methods for mining large-scale datasets (such as Frontier Expansion [38]) use a DFS approach that tests only the frequent and candidate itemsets of an equivalence class within GPU memory. However, the use of this DFS approach on GPUs could degrade the performance of itemset mining due to lack of parallelism and workload skewness, which will be shown in Section 6.

Our proposed TFL strategy solves the issue of mining frequent itemsets in large-scale datasets without degrading the performance within limited GPU memory. We call an entire set of frequent 1-itemsets the *first level* in the itemset lattice and call other levels in the itemset lattice *intermediate levels*. Most of the existing frequent itemset mining methods materialize frequent itemsets in intermediate levels to reduce computational overhead, but this approach greatly increases the space overhead. For example, AprioriTid materializes n-itemsets when finding n + 1-itemsets, and Eclat materializes the itemsets that have the same prefix. However, this approach can suffer

from a lack of main memory due to the large amount of intermediate data. Moreover, this tendency is more marked when exploiting GPUs, because GPU memory is limited compared to main memory. The proposed TFL strategy tests all the candidate itemsets of intermediate levels using only the first level, i.e., F_1. This feature is based on the observation that GPUs have high computational power, especially for massive bitwise operations, but relatively small device memory. Our observation indicates that, in the GPU architecture, testing the candidate n + 1-itemsets using frequent 1-itemsets tends to be much faster than testing the candidate n + 1-itemsets using frequent n-itemsets (i.e., F_n). This speed difference occurs because copying F_n to GPU memory incurs a much larger data transfer overhead than does copying only F_1, and simultaneously, accessing F_n in GPU memory incurs more non-coalesced memory access than does accessing F_1.

For mining large-scale databases, we also propose a new itemset mining technique on GPUs called *nested-loop streaming*. Here, a single series of candidate generation and testing steps constitutes an *iteration*. GMiner performs nested-loop streaming at each iteration. This technique copies the candidate itemsets to GPUs as the outer operand. Specifically, it copies only the relative memory addresses of the itemsets to the GPUs rather than the itemsets themselves. We denote the candidate itemsets at level L as C_L. The proposed technique copies $RA(C_L) = \{RA(x)|x \in C_L\}$ to the GPUs (hereafter, when there is no ambiguity, we simply denote $RA(C_L)$ as RA). The technique also copies transaction blocks of the first level (i.e., $TB_{1:R}$) to GPUs as the inner operand. We note that the outer operand, RA, or the inner operand, $TB_{1:R}$, or both, might not fit in GPU memory. Thus, the proposed technique partitions the outer operand RA into $RA_{1:Q}$ and copies each RA_j to the GPUs individually $(1 \leq j \leq Q)$. Then, for each RA_j, it streams each piece of the inner operand, i.e., transaction block, TB_k to the GPUs $(1 \leq k \leq R)$. In most intermediate levels, the outer operand, RA, is much smaller than the inner operand, TB. In particular, when the entire RA can be kept in GPU memory (i.e., Q = 1), streaming TB_k to the GPUs becomes a major operation of this technique.

For each pair $\langle RA_j, TB_k \rangle$, GMiner calculates the partial supports of $x \in RA_j$ within TB_k. We denote the *partial supports* for $\langle RA_j, TB_k \rangle$ as $PS_{j,k}$. We formally

define the partial support of itemset x in Definition 3.

Definition 3. ***(Partial support)*** *We define $\sigma_x(TB_k)$ as the partial support of an itemset x within a given transaction block TB_k. The full support of x on the entire transaction bitmap $TB_{1:R}$ becomes $\sigma(x) = \sum_{k=1}^{R} \sigma_x(TB_k)$.*

To calculate the partial support $\sigma_x(TB_k)$ for an itemset $x = \{i_1, \ldots, i_n\}$, GMiner simply performs bitwise AND operation $n-1$ times among bit vectors of $\{TB_k(i)|i \in x\}$ and counts the number of 1s in the resultant bit vector. GMiner can efficiently access to the locations of the bit vectors $TB_k(x)$ because $RA(x)$ contains the relative memory addresses of x in TB_k in GPU memory. We denote the function to apply a series of $n-1$ bitwise AND operations for the itemset x as $\bigcap\{TB_k(x)\}$. We also denote the function that counts the number of 1s in a given bit vector by $count(\cdot)$. Then, $\sigma_x(TB_k) = count(\bigcap\{TB_k(x)\})$.

Figure 1 shows the basic data flow of GMiner with the nested-loop streaming technique. In Figure 1, the outer operand $RA_{1:Q}$ and inner operand $TB_{1:R}$ are stored in main memory ($Q = 1$). The buffer for RA_j, called $RABuf$, and the buffer for TB_k, called $TBBuf$, are stored in GPU memory. Here, we allocate $RABuf$ and $TBBuf$ to GPU global memory. GMiner copies each TB_k to $TBBuf$ in a streaming fashion via the PCI-E bus, after copying RA_j to $RABuf$. To store the partial support values for all candidate itemsets in RA in each transaction block TB_k, GMiner maintains a two-dimensional array of size $|RA|$ x R in main memory, denoted as $PSArray$, where $|RA|$ is the number of candidate itemsets. GMiner also allocates the buffer for partial support in GPU global memory, denoted as $PSBuf$. The partial support values calculated on the GPU cores are first stored in $PSBuf$ in GPU global memory, and then copied back to the $PSArray$ in main memory.

3.3. TFL Algorithm

In this section, we present the algorithm that implements the TFL strategy. We first explain the overall procedure of the algorithm using the example shown in Figure 1. The TFL strategy performs a total of seven steps. We denote the set of candidate itemsets at the current level in the itemset lattice as C_L. In Step 1, the TFL strategy

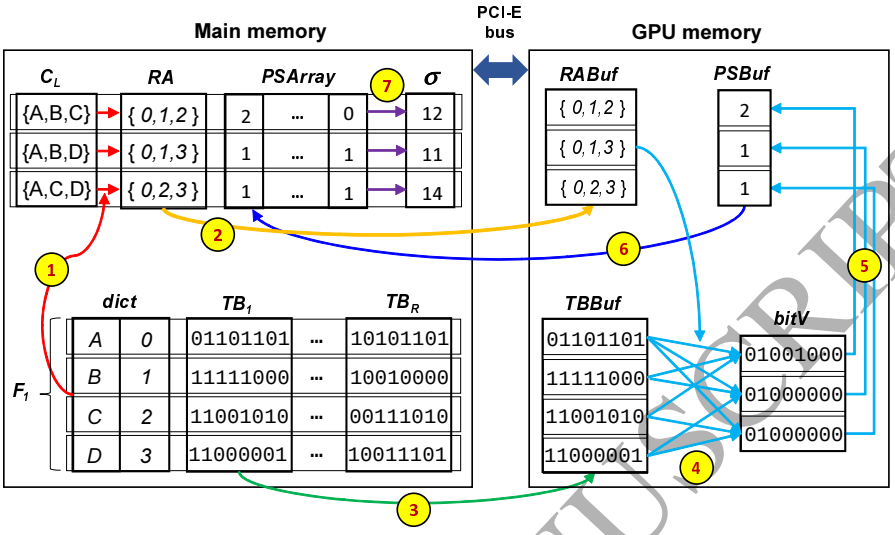


Figure 1: Example of the TFL strategy.

converts C_L to RA by mapping each itemset x in C_L to its relative memory address $RA(x)$ using $dict$. Here, $dict$ is a dictionary that maps a frequent 1-itemset $x \in F_1$ to $RA(x)$ within a transaction block TB_k. If the size of RA is larger than that of $RABuf$ in GPU memory, then RA is logically divided into Q partitions, i.e., $RA_{1:Q}$, such that each partition can fit in $RABuf$. In Step 2, it copies a partition RA_j to $RABuf$ in GPU memory. In Step 3, it copies each transaction block TB_k to $TBBuf$ in GPU memory in a streaming fashion. In Steps 4-5, the GPU kernel function for the bitwise `AND` operations, denoted as K_{TFL}, calculates the partial supports of candidate itemsets in $RABuf$ and stores the values in $PSBuf$. In Step 6, the TFL strategy copies the partial supports in $PSBuf$ back to $PSArray$ in main memory. Here, it copies the values of TB_k to the k-th column of $PSArray$. In Step 7, it aggregates the partial supports of each itemset x in $PSArray$ to obtain $\sigma(x)$. After Step 7, GMiner finds the frequent L-itemsets F_L for which the support values are greater than or equal to a given threshold $minsup$, as in the existing frequent itemset mining methods.

Algorithm 1 shows the pseudo code for the algorithm. During initialization, the algorithm loads a transaction database D into main memory (MM) and allocates $PSArray$ to MM. Then, it allocates three buffers, namely, $TBBuf$, $RABuf$, and $PSBuf$, to

GPU global memory (DM) (Lines 1-3). Next, it converts D to a set of transaction blocks $TB_{1:R}$ using F_1 such that each transaction block can fit in $TBBuf$ (Lines 4-5). After the dictionary dict used to map x to $RA(x)$ has been constructed (Line 6), it remains fixed during itemset mining because the TFL strategy uses only $TB_{1:R}$ for F_1 as input data. The main loop consists of a generating step (Lines 10-11) and a testing step (Lines 12-20), as in the Apriori algorithm; however, compared to the Apriori algorithm, our algorithm significantly improves the testing step performance by streaming the transaction blocks of F_1 to overcome the limitations imposed by GPU memory, while simultaneously exploiting GPU computing for fast and massively parallel calculation of partial supports (Lines 12-18).

We note that the kernel function K_{TFL} is usually called multiple times instead of a single time (Line 16). This is due to a limit on the number of GPU blocks, which we can specify when calling K_{TFL}. The K_{TFL} function can calculate a partial support of a single itemset using a single GPU block. If we set the maximum number of GPU blocks, denoted as $maxBlk$, to 16 K, a single call to K_{TFL} can simultaneously calculate partial supports for 16 K itemsets. Thus, if $|RA_j|$=100 M, we must call the K_{TFL} function $\lceil \frac{100\,M}{16\,K} \rceil \approx 6,250$ times. That is, for the same transaction block TB_k in $TBBuf$, GMiner executes the kernel function repeatedly while changing the affected portions of RA_j. When copying data, RA is the outer operand, and TB is the inner operand. However, when calling the kernel function, TB_k is the inner operand, and RA_j is the outer operand.

Next, we present the pseudo code for the GPU kernel function of GMiner in Algorithm 2. This function is used not only in the TFL strategy but also in the HIL strategy in Section 4. It takes a pair of RA_j and TB_k, along with $doneIdx$ and $maxThr$, as inputs. Here, $doneIdx$ is the index of the last candidate that was processed in RA_j. This value is required to identify the portion of RA_j that the current call of K_{TFL} should process. For example, if $|RA_j|$ = 10000 and $maxBlk$ = 1000, $doneIdx$ in the second call of K_{TFL} becomes 1000. The input $maxThr$ is the maximum number of threads in a single GPU block, which we can specify when calling K_{TFL}, as with $maxBlk$. BID and TID are the IDs of the current GPU block and GPU thread, respectively, which are automatically determined system variables. Because many GPU blocks ex-

Algorithm 1: The TFL strategy

Input : D; /* transaction database */

Input : $minsup$; /* minimum support */

Output : $\mathcal{F}$; /* frequent itemsets */

Load D into MM;
Allocate $PSArray$ on MM;
Allocate $\{TBBuf, RABuf, PSBuf\}$ on DM;
$F_1 \leftarrow$ find all frequent 1-itemsets;
Build $TB_{1:R}$ using D and F_1 on MM;
$dict \leftarrow$ dictionary mapping x to $RA(x)$ $(x \in F_1)$;
$L \leftarrow 1$;
while $|F_L| > 0$ **do**
    $L \leftarrow L + 1$;
    `/* Generating candidates using CPUs */`
    $C_L \leftarrow$ generate candidate itemsets using F_{L-1};
    Convert C_L to $RA_{1:Q}$ using $dict$;
    `/* Testing using GPUs */`
    for $j \leftarrow 1$ *to* Q **do**
        Copy RA_j into $RABuf$ of DM;
        for $k \leftarrow 1$ *to* R **do**
            Copy TB_k into $TBBuf$ of DM;
            Call $K_{TFL}(RA_j, TB_k)$; /* $\lceil \frac{|RA_j|}{maxBlk} \rceil$ times */
            Copy $PSBuf$ into $PSArray$ of MM;
    Thread synchronization of $GPUs$;
    $\sigma(c) \leftarrow \sum_{k=1}^{R} PSArray[c][k]$, for $\forall c \in C_L$;
    $F_L \leftarrow \{c | c \in C_L \bigwedge \sigma(c) \geq minsup\}$;
$\mathcal{F} \leftarrow \bigcup F_L$;
Return $\mathcal{F}$;

ecute concurrently, some might have no corresponding candidate itemsets to test. For instance, when $|RA_j|$ = 100 and $maxBlk$ = 200, 100 GPU blocks should not execute the kernel function because some blocks would have no itemsets. Thus, when the current GPU block has no itemset, the kernel function returns immediately (Lines 1-2). The kernel function prepares two frequently-accessed variables, namely, can and sup, in the shared memory in GPUs to improve performance. The variable can contains the itemset for which the current GPU block BID will calculate the partial support, and the vector sup is initialized to zero.

The main loop of K_{TFL} performs bitwise `AND` operations simultaneously and repeatedly (Lines 5-8). Under current GPU architectures, a single GPU thread can efficiently perform bitwise `AND` operations for single-precision widths (i.e., 32 bits). That is, a single GPU block can perform bitwise `AND` operations up to $maxThr \times 32$ bits simultaneously. However, the width of a transaction block W might be considerably larger than $maxThr \times 32$ bits.

Figure 2 shows an example of K_{TFL}, when $maxThr = 2$, and $can = 0, 1, 3$. Here, we assume that a GPU thread can perform bitwise `AND` for 4 bits for simplicity. Because the length of candidate itemset is 3, threads 1 and 2 perform bitwise `AND` operations twice over $\{TB(0), TB(1), TB(3)\}$ and store the resultant bits in $bitV$. The kernel repeats this process $\frac{W}{maxThr*32}$ times. The number of 1s in $bitV$ can easily be counted using the $popCount$ function and stored in the sup vector. In CUDA, the $popCount$ function is denoted as $popc()$. The partial support values are accumulated in the sup vector $\frac{W}{maxThr*32}$ times, as shown in Figure 2. Finally, the kernel function aggregates the values in sup into a single partial support value in TB_k for the candidate itemset can using a $parallelReduction$ function (Line 9).

3.4. Exploiting GPUs

In this section, we present the details of the GMiner implementation that exploits GPUs. First, we discuss how to allocate and utilize the GPU memory. In particular, we consider a method to avoid the out-of-memory issue when handling large-scale data using GPUs. Second, we explain how to set GPU threads and improve the GPU kernel function performance. Third, we explain the details of the nested-loop streaming,

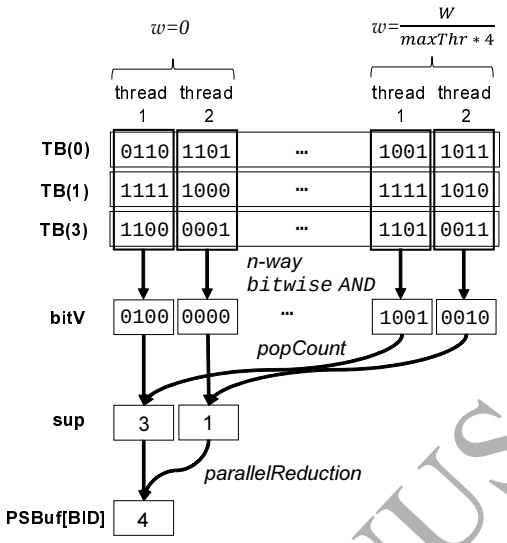


Figure 2: The GPU kernel function K_{TFL} (a GPU block takes an itemset{A,B,D}).

process, including both synchronization and host-device transfer.

First, we allocate three types of buffers to GPU memory only once; subsequently we use them repeatedly for the entire mining task. The out-of-memory issue of current GPU-based methods indicates that the overall mining tasks fail due to increasing data sizes that do not fit into GPU global memory. To avoid this out-of-memory issue, we allocate the buffers (i.e., $TBBuf$, $RABuf$, and $PSBuf$) to GPU global memory once while considering the GPU memory capacity, and then use them repeatedly. When the data (i.e., relative addresses and transaction bitmap) are larger than the size of the corresponding buffers, our method divides the data (i.e., relative addresses and transaction bitmap) into multiple partitions, each of which then fits into the corresponding buffer, and then copies each partition to the buffer individually. Consequently, our method avoids the out-of-memory issue and simultaneously reduces the buffer allocation overhead in GPU memory. In contrast, other GPU-based methods repeatedly allocate the buffers to GPU memory during the mining task.

Second, we exploit the shared memory of GPUs. Each GPU follows the single instruction multiple thread (SIMT) model and handles threads in a warp, which is a group of 32 threads. Multiple warps form a GPU block, and threads in the same GPU

Algorithm 2: K_{TFL}: Kernel function for partial supports

Input : RA_j; /* j-th partition of RA */

Input : TB_k; /* k-th transaction block */

Input : $doneIdx$; /*index of last candidates done in RA_j*/

Input : $maxThr$; /*max number of threads in GPU block*/

Variable: can; /* shared variable for a candidate */

Variable: sup; /* shared variable for a partial support */

1 **if** $doneIdx + BID \geq |RA_j|$ **then**

2 return;

3 $can \leftarrow RA_j[doneIdx + BID]$;

4 $sup[0 : maxThr] \leftarrow 0$;

5 **for** $i \leftarrow 0$; $i < \frac{W}{maxThr*32}$; $i \leftarrow i + 1$ **do**

6 $bitV \leftarrow \bigcap_{i \in can} TB_k[i][w * maxThr + TID]$;

7 $sup[TID] \leftarrow sup[TID] + popCount(bitV)$;

8 $syncthreads()$;

9 $PSBuf[doneIdx + BID] \leftarrow parallelReduction(sup[])$;

block can quickly communicate with one another using shared memory and built-in primitives. Frequently accessing GPU global memory to update variables is generally prohibitively expensive. To avoid this cost, our method uses shared memory to store the number of 1s in the bit vectors corresponding to the candidate itemset x. After computing the partial support for the corresponding transaction block, our method stores the partial support of x in the corresponding location of $PSBuf$. As a result, our method improves performance by accessing the GPU global memory only once. We also consider the number of GPU threads for the GPU kernel function. As discussed in Section 3.3, our GPU kernel function includes the $parallelReduction$ function. However, this function uses multiple brand-and-bound operations, which degrade the performance when using GPUs. This performance degradation becomes more marked as the number of GPU threads increases. Therefore, by default, we set the number of GPU threads

to 32, because they might be scheduled together in the GPU architecture.

Third, we exploit multiple asynchronous GPU streams. This approach reduces the data transmission overhead between main memory and GPU memory. Figure 3 shows the timeline of the copy operations of the transaction blocks. A CPU thread first transfers RA_j to $RABuf$. Then, it starts multiple GPU streams, each of which performs the following series of operations repeatedly, while incrementing k: (1) copying TB_k to $TBBuf$, (2) executing the GPU kernel function, denoted as K, to calculate $PS_{j,k}$, and (3) copying $PS_{j,k}$ back to main memory. We denote the number of GPU streams as m. Then, this scheme requires the size of $TBBuf$ to equal m transaction blocks and the size of $PSBuf$ to be m x $|RA_j|$, where $|RA_j|$ denotes the number of candidate itemsets in RA_j. In general, the above three kinds of operations, namely, copying to GPU memory, kernel execution, and copying to main memory, can overlap with one another in the current GPU architecture [16]; thus, a large portion of the copying time between GPU memory and main memory becomes hidden. After processing m streams, all the GPU threads are synchronized by calling the $cudaStreamSynchronize$ function to compute the exact partial supports for the corresponding m transaction blocks. Here, the number of GPU streams m is specified by the user; we used m = 4 as the default.

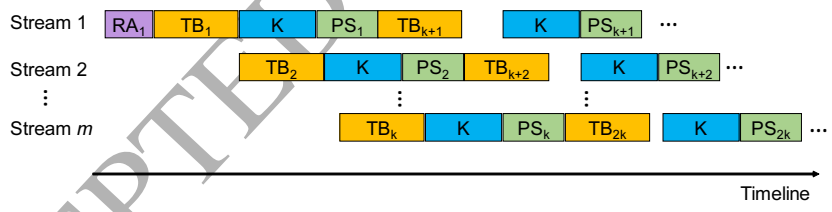


Figure 3: Multiple asynchronous GPU streams of GMiner

4. HIL Strategy

The TFL strategy in GMiner can find all the frequent itemsets for a large-scale database using GPUs that have only a limited amount of GPU memory. Although it shows outstanding performance in most cases, its performance might degrade if the lengths of the frequent itemsets were to become very long. To solve this issue, we

propose the *hopping from intermediate level* (HIL) strategy, which increases the scalability of the testing step in terms of the lengths of itemsets by utilizing more memory. We first present the data structure for storing some frequent itemsets at low intermediate levels, which are called *fragment blocks*, in Section 4.1, and then present the HIL algorithm in Section 4.2.

4.1. Fragment Blocks

The HIL strategy horizontally partitions each transaction block TB_k into disjoint *fragment blocks*. We define the *fragment size* as the number of frequent 1-itemsets that belong to a single fragment block. The fragment size is fixed, and we denote it as H. Thus, there are a total of $S = \lceil \frac{|F_1|}{H} \rceil$ fragment blocks in each transaction block.

The HIL strategy materializes all frequent itemsets within each fragment block. Here, materialization of itemset x means creating a bit vector for x in the corresponding fragment block. Because the fragment size is H, up to a maximum of $2^{|H|} - 1$ frequent itemsets can be materialized in each fragment block. Thus, we set the height of a fragment block to $2^{|H|} - 1$, instead of H.

The HIL strategy vertically and horizontally partitions the transaction bitmap TB into fragment blocks, each of which has the width W and height $2^{|H|} - 1$. A fragment block has its own ID within a transaction block, denoted as FID. Each fragment block is allocated consecutively in main memory (or stored as a chunk in secondary storage, similar to a disk page). We denote the l-th fragment block of TB_k as $TB_{k,l}$. A fragment block $TB_{k,l}$ consists of $2^{|H|} - 1$ bit vectors, and we denote the set of itemsets corresponding to those bit vectors as $itemsets(TB_{k,l})$. Figure 4 shows an example of the HIL strategy in which there are a total of $R \times 3$ fragment blocks and $H = 2$. The transaction block TB_1 is partitioned into three fragment blocks $\{TB_{1,1}, TB_{1,2}, TB_{1,3}\}$, and each fragment block contains $2^2 - 1 = 3$ bit vectors. In Figure 4, $itemsets(TB_{1,2})$ becomes $\{\{C\}, \{D\}, \{C, D\}\}$. The bit vectors of the n-itemsets ($n > 1$) in each fragment block are initialized with 0s before materialization.

In terms of space complexity, the HIL strategy requires the space of $O(\frac{|F_1|}{H} \times (2^H - 1) \times |D|)$ bits for the transaction bitmap. Compared with the full materialization of frequent itemsets at intermediate levels, the fragment blocks require a much smaller

amount of memory. For example, suppose that $|D| = 10M$, and $|F_1| = 500$. Then, full materialization up to third level might require up to $(\binom{500}{1} + \binom{500}{2} + \binom{500}{3}) \times$ 10,000,000 $\approx$ 23 TB. In contrast, the fragment blocks with $H = 5$ require only $\frac{500}{5} \times (2^5 - 1) \times$ 10,000,000 $\approx$ 3.6 GB.

For materialization, the HIL strategy performs frequent itemset mining from levels 2 to H for each of $R \times S$ fragment blocks. In general, the number of fragment blocks increases as the size of database increases and can become very large. For fast materialization of many blocks, we utilize the same nested-loop streaming technique as was proposed in Section 3. Because the fragment blocks are not correlated, they can be materialized independently.

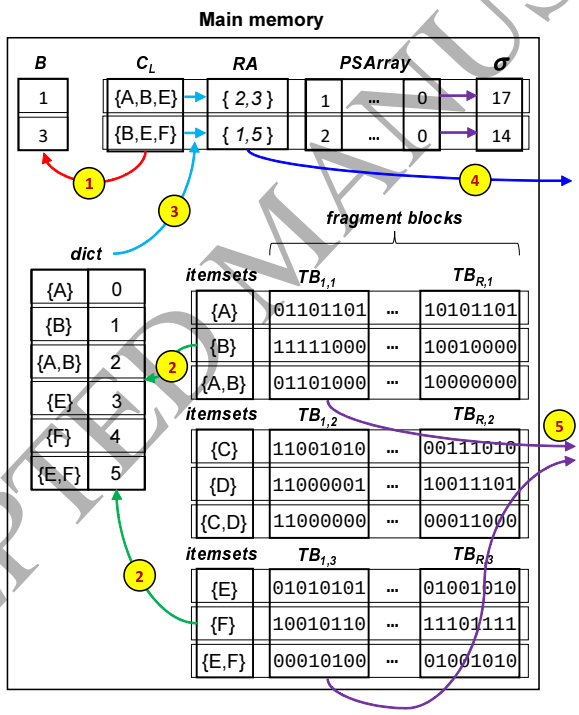


Figure 4: Example of the HIL strategy.

Algorithm 3 presents the algorithm to materialize the fragment blocks. The algorithm takes all the fragment blocks as input. It first calculates the maximum number of fragment blocks that can be materialized simultaneously using $RABuf$ on DM (Line 1); we denote this number as Q. Then, it executes the main loop Q times.

The algorithm streams the fragment blocks of FIDs between the $start$ and end of all transaction blocks, i.e., $FB_{[1:R,begin:end]}$. That is, a total of $R \times \frac{S}{Q}$ fragment blocks are transferred to GPU memory in a streaming fashion. To map the itemsets in those blocks to their relative memory addresses, we build the $dict$ (Lines 5-6). Then, the algorithm maps only the itemsets for $F_H - F_1$ because we do not need to materialize F_1 (Line 7). The algorithm executes the kernel function K_{HIL} simultaneously as it streams the fragment blocks (Lines 9-12). Here, K_{HIL} is basically the same with K_{TFL}, but it stores $bitV$ vectors in the corresponding positions in the fragment blocks in $TBBuf$, instead of calculating partial supports; thus we omit the pseudo code for K_{HIL}. After the call to K_{HIL} completes, the updated fragment blocks $TB_{k,[start:end]}$ are copied back to main memory. This materialization scheme, which uses GPU computing, is very fast; its elapsed time is almost negligible, as shown in Section 6.

Algorithm 3: Materialize fragment blocks

Input : $TB_{1:R,1:S}$; /* fragment blocks */

1 $Q \leftarrow$ integer value satisfying $\frac{S}{Q} \times (2^{|H|} - 1) < |RABuf|$;
2 **for** $j \leftarrow 1$ *to* Q **do**
3 $\quad start \leftarrow (j-1) \times \frac{S}{Q} + 1$;
4 $\quad end \leftarrow j \times \frac{S}{Q}$;
5 $\quad F_H \leftarrow \bigcup_{l=begin}^{l=end} itemsets(TB_{j,l})$;
6 $\quad dict \leftarrow$ dictionary mapping x to $RA(x)$ $(x \in F_H)$;
7 $\quad$ Convert $F_H - F_1$ to RA_j using $dict$;
8 $\quad$ Copy RA_j into $RABuf$ of DM;
9 $\quad$ **for** $k \leftarrow 1$ *to* R **do**
10 $\quad\quad$ Copy $TB_{k,[start:end]}$ into $TBBuf$ of DM;
11 $\quad\quad$ Call $K_{HIL}(RA_j, TB_{k,[start:end]})$;
12 $\quad\quad$ Copy $TB_{k,[start:end]}$ on DM back to MM;
13 Thread synchronization of $GPUs$;

4.2. HIL Algorithm

The HIL strategy tries to reduce the number of bitwise AND operations by utilizing fragment blocks, which are a type of precomputed results. Different from the TFL strategy, the HIL strategy determines the set of fragments at each level dynamically to reduce the amount of transaction bitmap transferred to GPUs. Algorithm 4 presents the pseudo code of the HIL strategy. It first materializes the fragment blocks using Algorithm 3 (Lines 1-2). After generating the candidate itemsets C_L, the algorithm finds the minimal set of fragments, denoted as B, that contain all the itemsets in C_L (Line 7). When the level, L, is low, most of fragments would be chosen as the set B. However, as level L increases, the number of fragments that contain C_L decreases; thus, we can reduce the overhead involved in transferring fragment blocks. Because the set of fragments changes at each level, the relative memory addresses of candidate itemsets in C_L also change. Thus, the algorithm builds $dict$ using only the itemsets in B at each level and converts C_L to $RA_{1:Q}$ (Lines 8-9). When streaming the transaction bitmap to the GPUs, the algorithm copies only the relevant fragment blocks $TB_{k,l \in B}$ instead of the entire transaction block TB_k.

In the HIL strategy example shown in Figure 4, we assume that $C_L = \{\{A, B, E\}, \{B, E, F\}\}$. Then, we can easily identify the fragments $B = \{1, 3\}$ that contain all the itemsets in C_L. The dictionary $dict$ is built using the first and third fragments; thus, $RA(\{E\})$ becomes 3 instead of 6. When converting an itemset $x \in C_L$ to $RA(x)$, $RA(x)$ in the HIL strategy is shorter than in the TFL strategy. That is, fewer bitwise AND operations are required to obtain the partial supports. For instance, the length of $RA(\{A, B, E\})$ is two (i.e., $\{2, 3\}$), whereas it is three in the TFL strategy. As the fragment size H increases, the length of $RA(x)$ tends to decrease. Each RA_j is copied to $RABuf$ in GPU memory; then, the first set of fragment blocks $\{TB_{1,1}, TB_{1,3}\}$ in B are streamed to $TBBuf$ in GPU memory. Next, the second set of fragment blocks $\{TB_{2,1}, TB_{2,3}\}$ are streamed.

Algorithm 4: HIL Algorithm

Input : D; /* transaction database */
Input : $minsup$; /* minimum support */
Input : H; /* fragment size */
Output : $\mathcal{F}$; /* frequent itemsets */

`/* Lines 1-4 in Algorithm 1 */`
Build $TB_{1:R,1:S}$ using D, F_1, and H on MM;
Materialize the fragment blocks of $TB_{1:R,1:S}$;
$L \leftarrow 1$;
while $|F_L| > 0$ **do**
  $L \leftarrow L + 1$;
  `/* Generating candidates using CPUs */`
  $C_L \leftarrow$ generate candidate itemsets using F_{L-1};
  $B \leftarrow$ set of fragment blocks containing C_L;
  $dict \leftarrow$ dictionary mapping $x \in itemsets(B)$ to $RA(x)$;
  Convert C_L to $RA_{1:Q}$ using $dict$;
  `/* Testing using GPUs */`
  for $j \leftarrow 1$ *to* Q **do**
    Copy RA_j into $RABuf$ of DM;
    for $k \leftarrow 1$ *to* R **do**
      Copy $TB_{k,l \in B}$ into $TBBuf$ of DM;
      Call $K_{TFL}(RA_j, TB_{k,l \in B})$;
      Copy $PSBuf$ into $PSArray$ of MM;
  `/* Lines 18-20 in Algorithm 1 */`
$\mathcal{F} \leftarrow \bigcup F_L$;
Return $\mathcal{F}$;

5. Multiple GPUs and Cost Model

5.1. Exploiting Multiple GPUs

GMiner can be easily extended to exploit multiple GPUs and further improve the performance. When exploiting multiple GPUs, GMiner copies the different portion of the outer operand to the different GPUs and copies the same transaction (or fragment) blocks to all the GPUs. We call this scheme as the *transaction bitmap sharing* scheme. This scheme can be applied to both TFL and HIL strategies.

Figure 5 shows the data flow of our scheme. When there are two GPUs, the scheme copies RA_1 to GPU_1 and RA_2 to GPU_2. Then, it copies the same TB_1 to both GPU_1 and GPU_2. The kernel function on GPU_1 calculates the partial supports in RA_1, while that on GPU_2 calculates the partial supports in RA_2. Note that because RA_1 and RA_2 have no common itemsets, the results of both kernel functions can be copied back to $PSArray$ in main memory without conflicts.

This scheme is highly scalable in terms of the number of GPUs used because RA_j and RA_k are independent tasks ($j \neq k$). In addition, it does not have the problem of workload imbalances, which is a typical issue in distributed and parallel computing methods. The computation is not skewed when RA_j and RA_k are the same size, because the computation heavily relies on the number of bitwise `AND` operations and does not use complex or irregular data structures. Therefore, regardless of the characteristics of the processed datasets, the proposed scheme achieves a stable speed-up ratio when using multiple GPUs.

5.2. Cost Model

In this section, we present the cost models of GMiner to understand its performance tendencies. In particular, we present the model for the TFL strategy and skip that of the HIL strategy due to their similarity. Here, we consider the factors that significantly affect the performance. The cost model of the TFL strategy is given by

$$\sum_{L=1}^{\#iteration} (\frac{|RA_{1:Q}|}{c1 \times N} + \frac{Q}{N} \times \{\frac{|TB_{1:R}|}{c2} + t_{call}(R \times \lceil \frac{|RA_j|}{maxBlk} \rceil) + t_{kernel}(TB_R) + \frac{|PS_{j,R}|}{c2}\}). \quad (1)$$

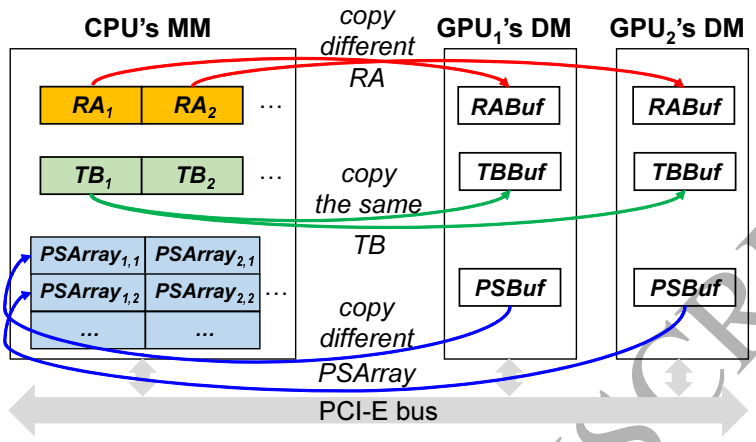


Figure 5: Data flow of GMiner using multiple GPUs.

where, $c1$ and $c2$ respectively represent the communication rate between main memory and GPU memory in chunk copy mode (approximately 12GB/sec in PCI-E 3.0 x16 interface, in practice) and that in streaming copy mode (approximately 8GB/sec in PCI-E 3.0 x16 interface, in practice), and N represents the number of GPUs. The term $\frac{|RA_{1:Q}|}{c1 \times N}$ represents the total amount of time required to copy the outer operands, i.e., $RA_{1:Q}$ to GPU memory. It is divided by N because the data is transferred concurrently to N GPUs. The terms in brackets represent the costs of streaming the inner operands and computing partial supports. In detail, the term $\frac{TB_{1:R}}{c2}$ represents the cost of streaming the transaction blocks. It cannot be reduced by using multiple GPUs due to the characteristics of the transaction bitmap sharing scheme. Here, $t_{call}(n)$ is the time overhead for calling a kernel function n times. The TFL strategy calls the kernel function $R \times \lceil \frac{|RA_j|}{maxBlk} \rceil$ times for each RA_j. The term $t_{kernel}(TB_R)$ indicates the kernel execution time for the last single transaction block, which cannot be hidden by data streaming. Likewise, the term $\frac{|PS_{j,R}|}{c2}$ indicates the cost for copying the last partial supports for RA_j back to main memory, which also cannot be hidden by streaming.

6. Performance Evaluation

In this section, we present the experimental evaluation of GMiner compared with other representative methods summarized in Table 1. We present experimental results

in three categories. First, we evaluate the performance of GMiner compared with the representative sequential (i.e., single-threaded) methods, Apriori by Borgelt [6], Eclat by Borgelt [6], Eclat by Goethals [10], FP-Growth* [11], and LCM [30]. Second, we evaluate the performance of GMiner compared with the representative parallel (i.e., multi-threaded, distributed, and GPU-based) methods. The considered multi-threaded methods are MC-Eclat[26], ShaFEM[32], and FP-Array[20]; the distributed method is the implementation of FP-Growth with MLlib [3] of Apache Spark; and the GPU-based methods are TBI [7], GPApriori [37], and Frontier Expansion [38]. Third, we examine the performance characteristics of GMiner while varying a wide range of settings and compare them with those of the TFL and HIL strategies.

6.1. Experimental Setup

For experiments, we use both a real dataset and synthetic datasets, as presented in Table 2. As the real dataset, we use the largest dataset from FIMI Repository [8], called Webdocs [21], which has been widely used for the performance evaluation of frequent itemset mining. As synthetic datasets, we use datasets generated by using IBM Quest Dataset Generator [2], which accepts four major parameters: T_{avg}, the average number of items per transaction; M_{avg}, the average length of the maximal pattern; $|D|$, the number of transactions; and $|I|$, the number of distinct items. In Table 2, the values of $|D|$ are in millions (M) and the values of $|I|$ are in thousands (K). We generate the default synthetic dataset, which is called Quest-Def, and multiple variations by changing parameters. We note that Webdocs is relatively sparse, whereas Quest-Def is relatively dense.

To evaluate LCM, FP-Growth*, Eclat (Borgelt), Eclat (Goethals), Apriori (Borgelt), GPApriori, TBI, and Frontier Expansion, we download and compile their latest source code. To evaluate MC-Eclat, ShaFEM, and FP-Array, we use the implementations presented in [36] and [5]. To evaluate Apache Spark MLlib's FP-Growth, we download and use the source code from [3]. All the methods in experiments are compiled with the same optimization option, namely -O3, with gcc 4.9. We perform all experiments on Ubuntu 14.04.3 LTS with the same GPU Toolkit, namely, CUDA 7.5. For the distributed methods, we use Scala 2.11.7, Spark 1.5.0, and Hadoop 1.2.1. When

Table 2: Statistics of transaction datasets used in the experiments.

Name	Webdocs	Quest-Def	Quest-Scale1	Quest-Scale2
$\mathbf{T_{avg}}$	177.2	200	200	200
$\mathbf{M_{avg}}$	N/A	25	25	25
$\|\mathbf{D}\|$ (M)	1.692	10	[1,5,10,15]	10
$\|\mathbf{I}\|$ (K)	5,268	10	10	[5,10,15,20]
size (GB)	1.4	9	[0.9,4.6,9.1,14]	[8.9,9.1,9.8,11]

measuring the elapsed times of GPU-based methods such as TBI, GPApriori, Frontier Expansion, and GMiner, to perform a fair comparison, we include all the time spent transferring data between GPU memory and main memory. We set the number of GPU streams for GMiner to four as the default.

We conduct all the experiments on a machine with two Intel 8-core CPUs running at 2.90 GHz (a total of 16 cores), 128GB of main memory, and four NVIDIA GTX 1080 GPUs with 2,560 cores running at 1.7 GHz, 8 GB of device memory, and 96 KB of shared memory. The NVIDIA GTX 1080 GPU follows the Pascal GPU architecture and supports CUDA 8.0, which includes many new features, such as improved compiler performance. The CPUs and GPUs are connected via PCI-E 3.0 x16 interface. We conduct all the experiments that involve distributed methods on a cluster of eleven machines, one master and ten slaves, each of which is equipped with an Intel quad-core CPU running at 3.40 GHz, 32 GB of main memory, and 4 TB HDDs. That is, the cluster slaves have a total of 40 CPU cores and 320 GB main memory.

We present the detail settings used for the GPU-based methods, namely, TBI [7], GPApriori [37], Frontier Expansion [38], and GMiner. For both TBI and GPApriori, the experimental settings, such as the number of GPU threads and the number of GPU blocks, are not given in their papers. Therefore, for both methods, we set the number of GPU threads and the number of GPU blocks to 32 and 16,384, respectively, which were the best parameters found through trial-and-error-based tuning. For Frontier Expansion, 256 GPU threads and 2,048 GPU blocks were used in the original study [38]. However, we found that these parameters did not yield the best performance in our

experimental environments. Therefore, we set the number of GPU threads and the number of GPU blocks to 32 and 4,096, respectively, which were the best parameters we found. For GMiner, we set the number of GPU threads, the number of GPU blocks, and the number of GPU streams to 32, 16,384, and 4, respectively. For GMiner, we set the width of transaction blocks to 8,192 in four bytes.

6.2. Comparison with Sequential Methods

Figures 6(a)-(b) present the speed-up ratios of GMiner over the representative sequential methods, namely, FP-Growth*, LCM, Eclat (Borgelt), Eclat (Goethals), and Apriori (Borgelt), on both the Webdocs and Quest-Def datasets while varying $minsup$. The speed-up ratios on the Y-axis are shown in log-scale, and O.O.M. means an out-of-memory error. We use the same X-axis range as in [26]. In the figures, although the elapsed times of all the methods decrease as $minsup$ increases, the gaps in elapsed time between GMiner and all the other methods increase slightly; thus, the speed-up ratios also increase slightly.

For both the Webdocs and Quest-Def datasets, GMiner consistently and significantly outperforms all other methods. In Figure 6(a), GMiner outperforms LCM, FP-Growth*, Eclat (Borgelt), and Apriori (Borgelt) by factors of $7-100$, $23-494$, $13-90$, and $124-3094$, respectively. Among the existing methods, LCM shows the overall best performance on both datasets. The large performance gap between GMiner and the existing methods is mainly due to the TFL strategy, which fully exploits fast and massive bitwise computation using thousands of cores and simultaneously reduces the memory access overhead by using relative memory addresses on the transaction blocks of F_1 and nested-loop streaming, as explained in Section 3. In Figure 6(b), LCM and Eclat (Borgelt) encounter O.O.M. errors because they tend to consume more memory when improving the performance than do the other existing methods.

6.3. Comparison with CPU-based Parallel Methods

Figures 7(a)-(b) present the speed-up ratios of GMiner over the representative CPU-based parallel methods, namely, MC-Eclat, ShaFEM, FP-Array, and MLlib, on both the Webdocs and Quest-Def datasets. MC-Eclat, ShaFEM, and FP-Array are

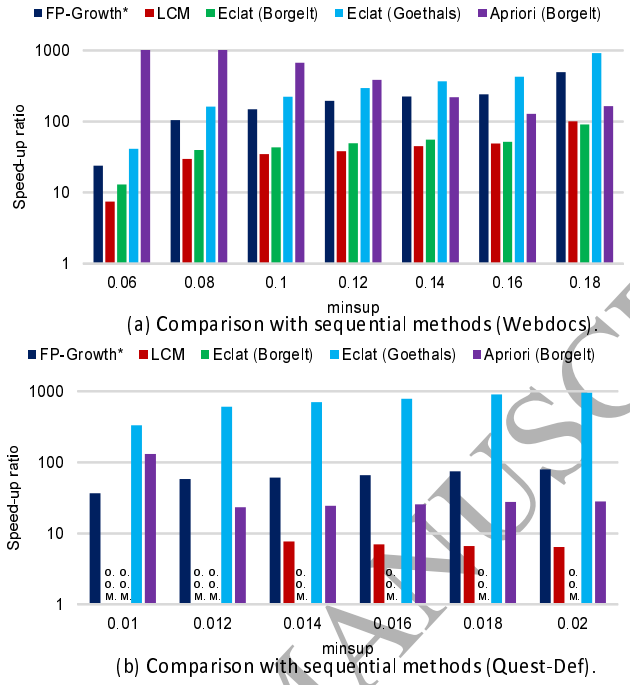


Figure 6: Performance comparison with sequential frequent itemset mining methods.

CPU-based parallel methods that run on a single machine, and MLlib is a distributed method that runs on Spark. On both the Webdocs and Quest-Def datasets, GMiner still consistently and significantly outperforms all other methods, except for FP-Array at $minsup = 0.06$ on the Webdocs dataset. In Figure 7(a), GMiner outperforms MC-Eclat and FP-Array by factors of $3.3 - 94$ and $0.45 - 2.8$, respectively.

Among the existing three multi-threaded methods, FP-Array achieves the best overall performance on the Webdocs dataset, while MC-Eclat achieves the best overall performance on the Quest-Def dataset. FP-Array usually results in O.O.M. errors on the Quest-Def dataset, because it requires more memory than do MC-Eclat and ShaFEM. As stated earlier, FP-Array is a multi-threaded version of FP-Growth*; therefore, it requires considerably more memory than does FP-Growth*. However, FP-Array is much faster than FP-Growth* on a given dataset and with the same $minsup$ value. Between the Webdocs and Quest-Def datasets, Quest-Def usually requires more memory because it is denser. ShaFEM results in the worst performance among multi-threaded

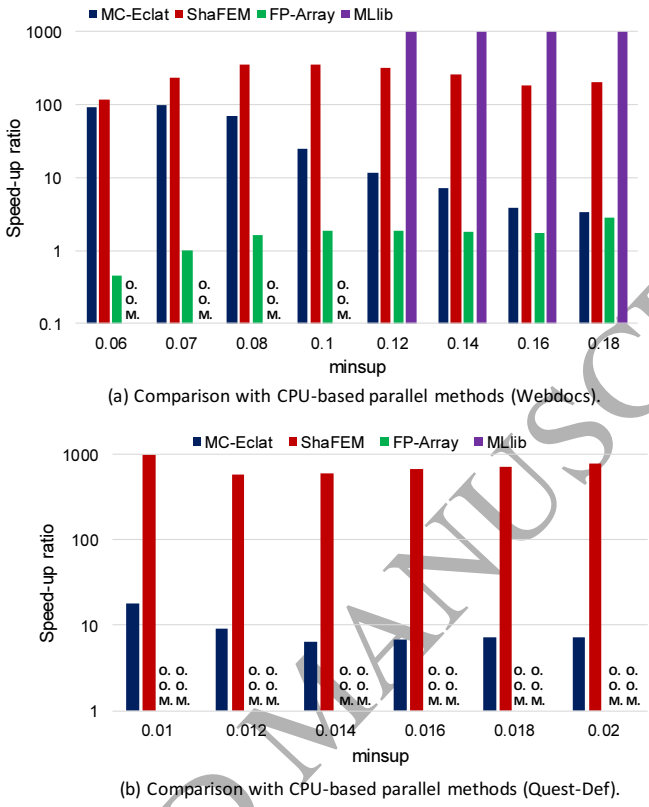


Figure 7: Performance comparison with CPU-based parallel frequent itemset mining methods.

methods. We note that a distributed method, namely, MLlib, achieves the worst performance among the existing parallel methods or fails to find frequent itemsets, although it utilizes a total of 40 CPU cores and 320 GB of main memory. MLlib is based on FP-Growth, where each conditional database is processed by the FP-Growth method in each machine. In some cases, the degree of workload skewness (i.e., imbalance) is extremely high; in such cases, a single conditional database has almost the same size as the original database. In addition, MLlib's FP-Growth is implemented on top of Apache Spark; thus, it tends to use more memory than did the original FP-Growth and incur additional overhead. These results suggest that improving the performance of frequent itemset mining using the sequential methods is non-trivial, and a parallel method must be devised carefully to achieve that goal.

6.4. Comparison with GPU-based Parallel Methods

Figures 8(a)-(b) present the speed-up ratios of GMiner over the representative GPU-based parallel methods, including Frontier Expansion, TBI, and GPApriori, on both the Webdocs and Quest-Def datasets. Note that the existing GPU-based methods outperform the CPU-based methods, but only when the size of data to be copied to GPU memory is quite small. In contrast, the performances of the GPU-based methods degrade compared with CPU-based methods as the data size increases. This is because the data transmission overhead between main memory and GPU memory can significantly affect the performance. GMiner does not have the drawbacks of the existing GPU-based methods. For instance, GMiner reduces the overhead of data transmission between the host and GPU devices by exploiting multiple GPU streams, as explained in Section 3, while other methods do not hide the overhead. As a result, GMiner outperforms the state-of-the-art CPU-based methods, as shown in Figures 6 and 7.

Frontier Expansion achieves a performance similar to MC-Eclat for Webdocs, but is outperformed by MC-Eclat on Quest-Def. In many cases, Frontier Expansion fails to find frequent itemsets due to O.O.M. errors on Quest-Def. This result occurs because Frontier Expansion tries to maintain frequent itemsets at the intermediate level in GPU memory and Quest-Def is denser than Webdocs; therefore, it must store considerably more intermediate data in GPU memory. TBI outperforms Frontier Expansion at the $minsup$ values of 0.1-0.18 on Webdocs, because TBI processes more candidate itemsets simultaneously. However, TBI shows O.O.M. errors at the $minsup$ values of 0.06-0.08, while Frontier Expansion successfully completes the pattern mining at the same $minsup$ values. This is because TBI tries to maintain a larger number of frequent intermediate-level itemsets in GPU memory than does Frontier Expansion. GPApriori results in O.O.M. errors in all cases on both Webdocs and Quest-Def; that is, the size of its static bitmap is larger than the capacity of main memory (i.e., 128 GB). This result occurs because GPApriori generates a static bitmap for the whole input database without pruning the infrequent 1-itemsets during initialization.

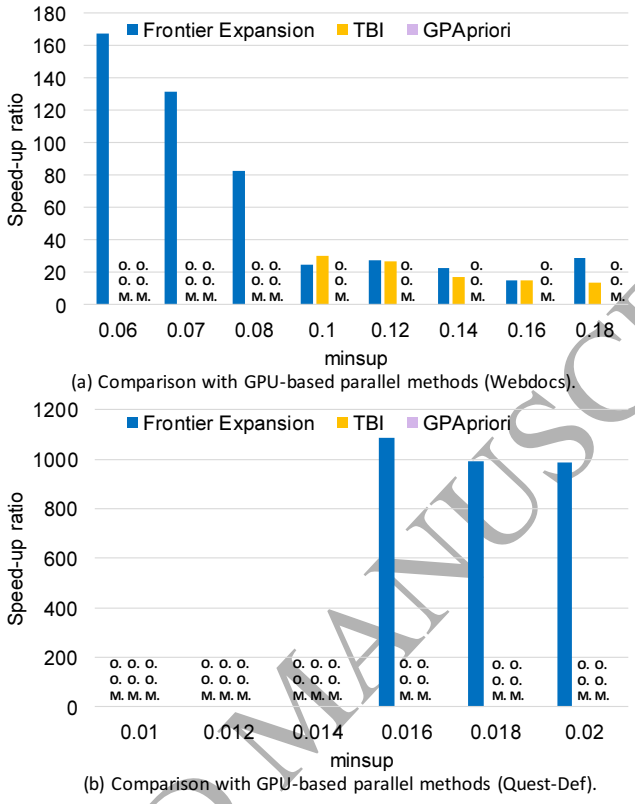


Figure 8: Performance comparison with GPU-based parallel frequent itemset mining methods.

6.5. Scalability test

Figures 9(a)-(b) present the speed-up ratios of GMiner over the representative sequential and parallel methods, namely, FP-Growth*, LCM, Eclat (Borgelt), Eclat (Goethals), Apriori (Borgelt), MC-Eclat, FP-Array, and Frontier Expansion, on the Quest-Def dataset. In these experiments, we omit GPApriori, TBI, ShaFEM, and ML-lib because they resulted in relatively poor performances in the experiments in Sections 6.2, 6.3, and 6.4. Figure 9(a) shows the results obtained while varying the number of transactions (i.e., Quest-Scale1 in Table 2). As the number of transactions increases, the speed-up ratios also increase, i.e., GMiner improves the performance more compared to the existing methods. Figure 9(b) shows the results obtained while varying the number of distinct itemsets (i.e., Quest-Scale2 in Table 2). As the number of dis-

tinct itemsets (i.e., $|I|$) increases, the speed-up ratios remain approximately constant. GMiner consistently outperforms the existing methods, regardless of the value of $|I|$.

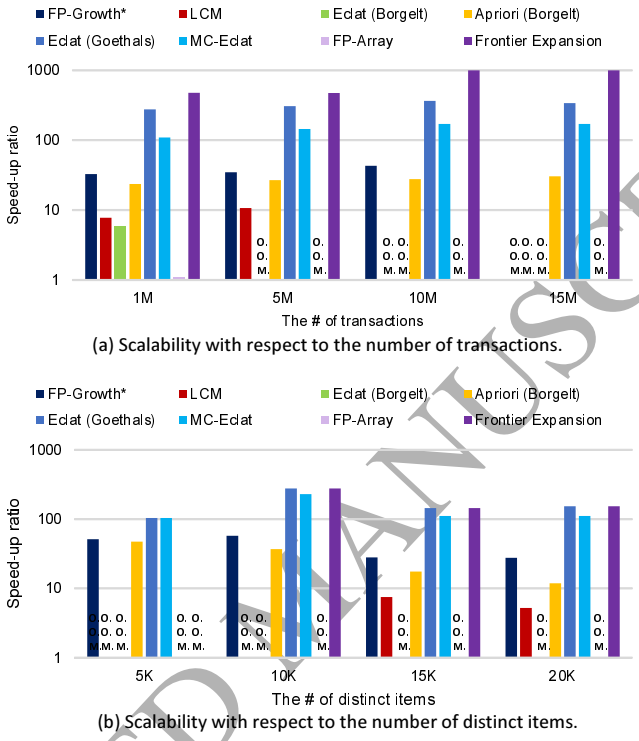
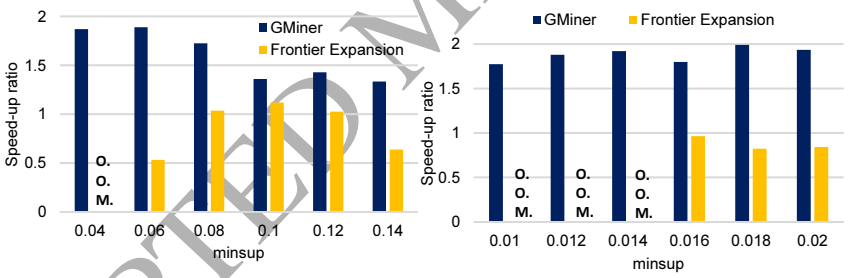


Figure 9: Scalability test varying the number of transactions and distinct items.

Figures 10(a)-(b) present the speed-up ratios obtained when using two GPUs for the GPU-based methods, namely, Frontier Expansion and GMiner. Here, the theoretical maximum speed-up ratio is two. On both the Webdocs and Quest-Def datasets, GMiner achieves ratios close to this maximum value in most cases. In contrast, Frontier Expansion shows much lower speed-up ratios below one in most cases, which means that using two GPUs degrades its performance compared to using a single GPU. When the range of $minsup$ is within [0.1, 0.14] on Webdocs, the speed-up ratios of GMiner degrade slightly because the total elapsed time is too short ($\leq 1sec.$) and the time spent in support counting using GPUs is relatively small. However, except when the workload is small, GMiner achieves almost the maximum speed-up ratio because it shares the transaction bitmap among all the GPUs and assigns equal-sized indepen-

dent RA_j to each GPU, as explained in Section 5.1. In contrast, Frontier Expansion assigns a subtree of the enumeration tree (i.e., equivalence class) to each GPU. Because it is likely that those subtrees have different amounts of workload (that is, a workload skewness problem occurs), its scalability can degrade greatly. In addition, Frontier Expansion requires a larger overhead when exploiting multiple GPUs than does GMiner because it divides the search space of patterns into equivalence classes.

Figures 11(a)-(b) present the speed-up ratios of GMiner when using multiple GPUs. We define the speed-up ratio as $\frac{T_1}{T_M}$, where T_1 and T_M are the running times using a single GPU and M GPUs, respectively. Figure 11(a) shows the ratios obtained while varying the number of transactions (i.e., $|D|$) among 1 M, 5 M, 10 M, and 15 M. Figure 11(b) shows the ratios obtained while varying the number of distinct items (i.e., $|I|$) among 5 K, 10 K, 15 K, and 20 K. The results show that regardless of the dataset characteristics, the speed-up ratios of GMiner increase almost linearly. We note that there is a small gap between the number of GPUs and the ideal speed-up ratio; this gap occurs mainly because of synchronization overhead among the GPUs.



(a) Speed-up ratio when using two GPUs (Webdocs). (b) Speed-up ratio when using two GPUs (Quest-Def).

Figure 10: Speed-up ratios when using multiple GPUs for GPU-based methods.

6.6. *Characteristics of GMiner*

Figure 12 presents the elapsed times obtained while varying the width of TB (i.e., W) and the number of GPU blocks, on both datasets. As shown in the figure, 8 K $\times$ 32 = 262,144 bits for the width of TB and 16 K GPU blocks yield the best overall performance; thus, we applied these values as the default settings for GMiner. Because GMiner sets the number of threads per GPU block to 64, the total number of threads used in GMiner is 64 $\times$ 16 K = 1 million.

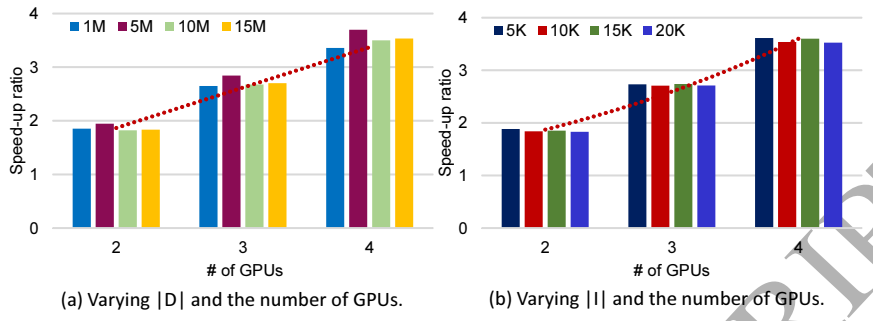


Figure 11: Speed-up ratios of GMiner when using multiple GPUs and varying the characteristics of datasets.

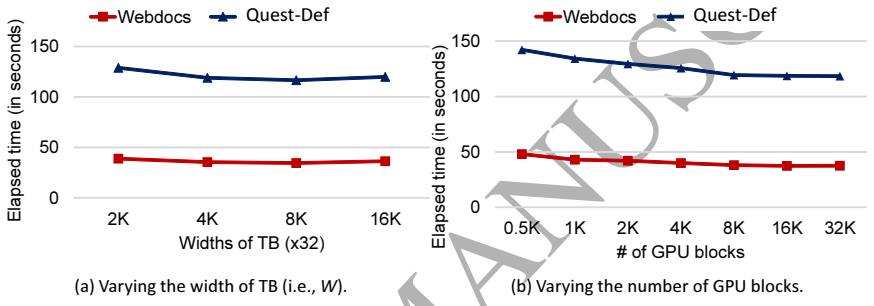


Figure 12: Finding optimal W and the number of GPU blocks.

Figure 13 presents the various characteristics used in the HIL strategy of GMiner. Figure 13(a) shows the memory usage as a function of the fragment size (H), which increases almost exponentially as H increases. Figure 13(b) shows the elapsed time for fragment materialization (described in Algorithm 3) which increases in proportion to the amount of memory usage in Figure 13(a). We note that the materialization time is very short compared with the total running time in Figure 13(d) and is almost negligible. Figure 13(c) shows the ratio of the number of fragment blocks copied to GPU memory for support counting to the total number of fragment blocks in each iteration. When $H = 1$, i.e., using the TFL strategy, the transaction blocks for F_1 are copied to GPU memory at every iteration, so the usage ratio is 100 %. However, when $H > 1$, i.e., using the HIL strategy, only a necessary subset of fragment blocks is copied to GPU memory, as described in Algorithm 4 resulting in a usage ratio of approximately 55 %, regardless of fragment size. Figure 13(d) shows the total running time, which is minimized when $H = 5$.

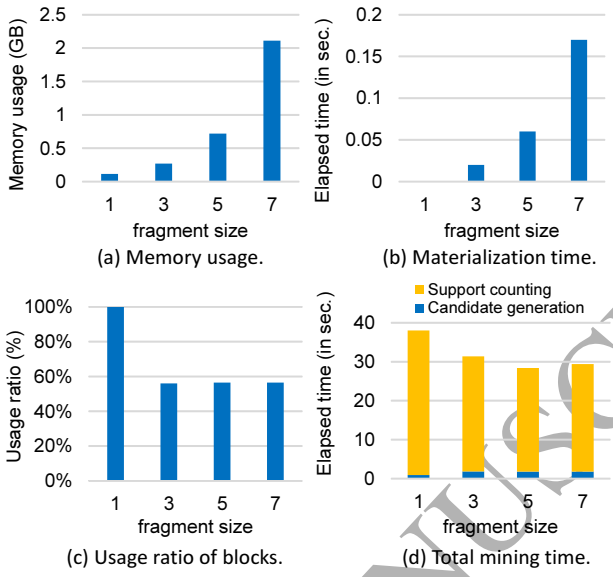


Figure 13: Characteristics of the HIL strategy.

We note that the support counting step still takes most of the running time even though GMiner exploits GPUs for support counting. Based on these results, we set H = 5 as the default for the HIL strategy.

Figure 14 presents the comparison results between the TFL and HIL strategies. For this experiment, we added a long pattern of length n to the Quest-Def dataset; as a result, the dataset contains many long patterns of lengths n - 1, n - 2, and so on. Figure 14(a) shows the total mining time (as a multiple of 1000 sec.) as a function of the length of n. The HIL strategy outperforms the TFL strategy from n = 22; subsequently, the performance gap between two strategies increases as n increases, as explained in Section 4. The HIL strategy reduces the number of bitwise AND operations, while using more memory. Figure 14(b) shows the trade-off between running time and memory usage as a function of H in the HIL strategy. In this figure, H = 7 yields slightly improved performance, but greatly increases the memory usage. Thus, we suggest that H = 5 is a good setting for databases that contain long patterns.

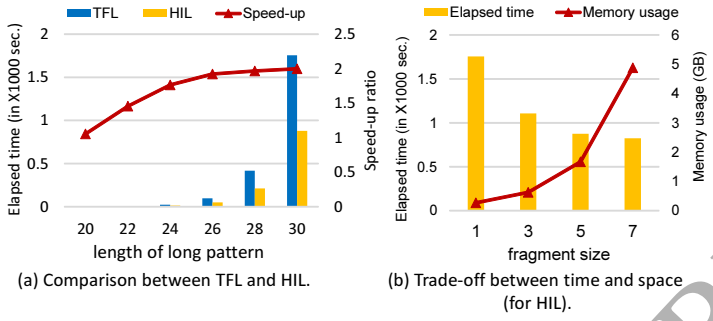


Figure 14: Evaluation of the HIL strategy for database containing long patterns.

7. Summary

In this paper, we proposed a fast GPU-based frequent itemset mining method for large-scale datasets called GMiner. In detail, we proposed the TFL strategy, which fully exploits the computational power of GPUs by performing a large amount of computation on a small amount of data, and the HIL strategy, which can further improve the performance on datasets that contain long patterns by performing a moderate amount of computation on a moderate amount of data. GMiner solves the workload skewness problem the existing parallel methods suffer from by splitting an array of relative memory addresses for candidate itemsets among the GPUs and streaming transaction blocks to all the GPUs. Through extensive experiments, we demonstrated that GMiner significantly outperforms most of the state-of-the-art methods that have been addressed in recent studies [4, 9, 25, 31, 33] on two kinds of benchmarks, and its performance is scalable in terms of the number of GPUs.

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT and Future Planning(2017R1E1A1A01077630) and Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1502-10.

References

[1] C. C. Aggarwal, J. Han, Frequent pattern mining, Springer, 2014.

[2] R. Agrawal, R. Srikant, Fast Algorithms for Mining Association Rules in Large Databases, in: VLDB, 487–499, URL http://www.vldb.org/conf/1994/P487.PDF, 1994.

[3] Apache Spark MLlib, http://spark.apache.org/mllib/, 2017.

[4] E. Baralis, T. Cerquitelli, S. Chiusano, A. Grand, Scalable out-of-core itemset mining, Information Sciences 293 (2015) 146–162.

[5] C. Bienia, S. Kumar, J. P. Singh, K. Li, The PARSEC benchmark suite: characterization and architectural implications, in: PACT, ACM, 72–81, 2008.

[6] C. Borgelt, Efficient implementations of apriori and eclat, in: FIMI, 2003.

[7] W. Fang, M. Lu, X. Xiao, B. He, Q. Luo, Frequent itemset mining on graphics processors, in: DaMon, ACM, 34–42, 2009.

[8] FIMI Repository, http://fimi.ua.ac.be, 2005.

[9] P. Fournier-Viger, J. C.-W. Lin, B. Vo, T. T. Chi, J. Zhang, H. B. Le, A survey of itemset mining, Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 2017.

[10] B. Goethals, Survey on frequent pattern mining, Univ. of Helsinki, 2003.

[11] G. Grahne, J. Zhu, Efficiently Using Prefix-trees in Mining Frequent Itemsets., in: FIMI, vol. 90, 2003.

[12] F. Gui, Y. Ma, F. Zhang, M. Liu, F. Li, W. Shen, H. Bai, A distributed frequent itemset mining algorithm based on Spark, in: CSCWD, 271–275, URL http://dx.doi.org/10.1109/CSCWD.2015.7230970, 2015.

[13] J. Han, J. Pei, M. Kamber, Data mining: concepts and techniques, Elsevier, 2011.

[14] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, in: ACM SIGMOD Record, vol. 29, ACM, 1–12, 2000.

[15] Y.-S. Huang, K.-M. Yu, L.-W. Zhou, C.-H. Hsu, S.-H. Liu, Accelerating Parallel Frequent Itemset Mining on Graphics Processors with Sorting, in: Network and Parallel Computing, Springer, 245–256, 2013.

[16] M.-S. Kim, K. An, H. Park, H. Seo, J. Kim, GTS: A Fast and Scalable Graph Processing Method based on Streaming Topology to GPUs, in: SIGMOD, ACM, 447–461, 2016.

[17] H. Li, Y. Wang, D. Zhang, M. Zhang, E. Y. Chang, Pfp: parallel fp-growth for query recommendation, in: RecSys, ACM, 107–114, 2008.

[18] C. Lin, K. Yu, W. Ouyang, J. Zhou, An OpenCL Candidate Slicing Frequent Pattern Mining algorithm on graphic processing units, in: SMC, 2344–2349, URL http://dx.doi.org/10.1109/ICSMC.2011.6084028, 2011.

[19] M.-Y. Lin, P.-Y. Lee, S.-C. Hsueh, Apriori-based frequent itemset mining algorithms on MapReduce, in: ICUIMC, ACM, 76, 2012.

[20] L. Liu, E. Li, Y. Zhang, Z. Tang, Optimization of frequent itemset mining on multiple-core processor, in: PVLDB, VLDB Endowment, 1275–1285, 2007.

[21] C. Lucchese, S. Orlando, R. Perego, F. Silvestri, WebDocs: a real-life huge transactional dataset., in: FIMI, vol. 126, 2004.

[22] S. Moens, E. Aksehirli, B. Goethals, Frequent itemset mining for big data, in: Big Data, IEEE, 111–118, 2013.

[23] S. Orlando, kDCI: a multi-strategy algorithm for mining frequent sets, in: Proc. IEEE ICDM'03 Workshop FIMI'03, 2003.

[24] S. Parthasarathy, M. J. Zaki, M. Ogihara, W. Li, Parallel data mining for association rules on shared-memory systems, Knowledge and Information Systems 3 (1) (2001) 1–29.

[25] B. Schlegel, Frequent itemset mining on multiprocessor systems, Dissertation, Technischen Universität Dresden, 2013.

[26] B. Schlegel, T. Karnagel, T. Kiefer, W. Lehner, Scalable frequent itemset mining on many-core processors, in: DaMon, ACM, 3, 2013.

[27] B. Schlegel, T. Kiefer, T. Kissinger, W. Lehner, PcApriori: scalable Apriori for multiprocessor systems, in: SSDBM, ACM, 20, 2013.

[28] C. Silvestri, S. Orlando, gpuDCI: Exploiting GPUs in Frequent Itemset Mining, in: PDP, 416–425, URL http://dx.doi.org/10.1109/PDP.2012.94, 2012.

[29] G. Teodoro, N. Mariano, W. M. Jr., R. Ferreira, Tree Projection-Based Frequent Itemset Mining on Multicore CPUs and GPUs, in: SBAC-PAD, 47–54, URL http://dx.doi.org/10.1109/SBAC-PAD.2010.15, 2010.

[30] T. Uno, M. Kiyomi, H. Arimura, LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets, in: FIMI, vol. 126, 2004.

[31] R. L. Uy, M. T. C. Suarez, Survey on the Current Status of Serial and Parallel Algorithms of Frequent Itemset Mining, Manila Journal of Science 9 (2016) 115–135.

[32] L. Vu, G. Alaghband, Novel parallel method for mining frequent patterns on multi-core shared memory systems, in: DISCS, ACM, 49–54, 2013.

[33] K. Wang, Y. Qi, J. J. Fox, M. R. Stan, K. Skadron, Association Rule Mining with the Micron Automata Processor, in: IPDPS, 689–699, URL http://dx.doi.org/10.1109/IPDPS.2015.101, 2015.

[34] M. J. Zaki, K. Gouda, Fast vertical mining using diffsets, in: SIGKDD, ACM, 326–335, 2003.

[35] M. J. Zaki, S. Parthasarathy, M. Ogihara, W. Li, et al., New Algorithms for Fast Discovery of Association Rules., in: KDD, vol. 97, 283–286, 1997.

[36] S. Zalewski, Mining Frequent Intra-and Inter-Transaction Itemsets on Multi-Core Processors, MS thesis, NTNU, 2015.

[37] F. Zhang, Y. Zhang, J. D. Bakos, GPApriori: GPU-Accelerated Frequent Itemset Mining, in: CLUSTER, 590–594, URL http://dx.doi.org/10.1109/CLUSTER.2011.61, 2011.

[38] F. Zhang, Y. Zhang, J. D. Bakos, Accelerating frequent itemset mining on graphics processing units, The Journal of Supercomputing 66 (1) (2013) 94–117, URL http://dx.doi.org/10.1007/s11227-013-0887-x.

[39] J. Zhou, K. Yu, B. Wu, Parallel frequent patterns mining algorithm on GPU, in: SMC, 435–440, URL http://dx.doi.org/10.1109/ICSMC.2010.5641778, 2010.