Received May 14, 2022, accepted May 29, 2022, date of publication June 1, 2022, date of current version June 17, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3179592

SGMiner: A Fast and Scalable GPU-Based Frequent Pattern Miner on SSDs

KANG-WOOK CHON[1], **EUNJEONG YI**[2], **AND MIN-SOO KIM**[3], **(Member, IEEE)**
[1]Division of National Supercomputing, Korea Institute of Science and Technology Information, Daejeon 34141, Republic of Korea
[2]Department of Information and Communication Engineering, Daegu Gyeongbuk Institute of Science and Technology, Daegu 42988, Republic of Korea
[3]School of Computing, Korea Advanced Institute of Science and Technology, Daejeon 34141, Republic of Korea

Corresponding author: Min-Soo Kim (minsoo.k@kaist.ac.kr)

This work was supported by the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2019-0-01267, GPU-based Ultrafast Multi-type Graph Database Engine SW), the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2022-2020-0-01795) supervised by the IITP (Institute of Information & Communications Technology Planning & Evaluation), the KAIST UP Program, and the institutional R&D program of KISTI (K-22-L02-C04-S01).

ABSTRACT Frequent itemset mining is extensively employed as an essential data mining technique. Nevertheless, as the data size grows, the applicability of this method decreases owing to the relatively poor performance of the existing methods. Though numerous efficient sequential frequent itemset mining methods have been developed, the performance that can be achieved is clearly limited by the fact that they exploit only one thread. To overcome these limitations, a number of parallel methods using multi-core central processing units (CPUs), multiple machines or many-core graphic processing units (GPU) have been proposed. However, these methods are relatively slow in performance and have low scalability, mainly owing to large memory requirements for intermediate data, significant disk I/Os, and heavy computation. In this study, to resolve the aforementioned problems, we propose SGMiner, which is a new, fast, and scalable GPU- and disk-based method on a single machine equipped with multiple graphic processing units (GPUs) and multiple solid-state drives (SSDs) for extracting frequent patterns. It is based on an algorithm similar to the Apriori algorithm and neither has intermediate data nor large disk I/O overheads owing to its exploitation of SSDs. Moreover, we propose storing transaction databases, namely bitmap transaction chunks, in SSDs, streaming the chunks to GPU device memory via the main memory with reduced I/O overhead, and performing fast support counting with GPUs based on the chunks. In addition, when exploiting multiple GPUs and SSDs, it proposes a concept of replicating bitmap transaction chunks stored in SSDs to GPUs in a streaming fashion. This could allow an almost equal workload to be distributed evenly across multiple GPUs with reduced I/O overheads. The experiments we conducted demonstrate that SGMiner outperforms the existing methods in terms of scalability and performance with enhanced robustness.

INDEX TERMS Big data, frequent pattern mining, parallel algorithm, GPUs, scalable algorithm, disk-based algorithm.

I. INTRODUCTION

Frequent itemset mining is a popular application in many areas, including market basket analysis, bioinformatics, and recommendation systems. For instance, as a key module, a number of commercial recommendation systems utilize frequent itemset mining [1]–[9]. However, the existing frequent itemset mining methods are time-consuming and often terminate abnormally owing to the lack of memory while finding frequent patterns from large-scale datasets. As real-world datasets grow in size, strategies for mining frequent itemsets that are fast and scalable have become more relevant than ever. Many sequential frequent itemset mining methods have been devised for efficiently mining frequent itemsets [10]–[19]. However, all the sequential frequent itemset mining methods have the drawback of scalability because they exploit a single thread on a single machine. Here, scalability refers to the ability to process a large amount of data or process the data more quickly by exploiting a larger number of machines or numerous computing units (e.g., CPUs and GPUs).

Several distributed methods have been proposed for efficiently handling large-scale datasets. Such an approach theoretically could find patterns in large-scale datasets of the same size as the disk size of the distributed systems for extracting various data types such as traditional

The associate editor coordinating the review of this manuscript and approving it for publication was Xiaowen Chu.

transaction database [20]–[29], healthcare data [30], and sequential database [26], [31]. Most distributed methods are straightforward expansions of sequential methods that utilize the distributed frameworks, i.e., Spark [32] and MapReduce [33]. In this study, we consider distributed methods based on the MapReduce framework owing to their ability to process a larger amount of data than that capable by the total memory of the distributed system by performing the processing task in a massively parallel way [4], [6], [21], [26]. Although the distributed frequent itemset mining methods are performed differently according to their baseline methods (i.e., the sequential methods), they usually partition the search space for the patterns into several chunks. Subsequently, each chunk is assigned to a machine. The chunks tend to vary widely in size, causing different-sized workloads on each machine in the distributed system. As a result, the distributed frequent itemset mining methods show relatively poor scalability with the number of machines utilized. In addition, they all require high network communication overheads. As the volume of data transmitted across machines grows, these network overheads can severely degrade speed and scalability [4], [5].

Meanwhile, continuous advances in GPU technology are increasingly improving the theoretical computing power of modern computers. As the GPU's theoretical computing performance is far higher than that of the CPU, it becomes increasingly important to utilize the GPU in many issues requiring high-performance computing, including frequent itemset mining. Owing to the advantages of exploiting GPUs, several GPU-based methods have been proposed, which largely improve performance. However, most existing GPU-based methods are incapable of handling the huge size of the input and intermediate data. This is because, to process data using GPUs, the data to be processed should be much smaller than the GPU device memory size, which is usually limited to a few gigabytes. GMiner [5] is the only method that can simultaneously exploit several GPUs while handling the larger datasets than the capacity of GPU device memory. It searches frequent patterns from the first level of the search space (i.e., enumeration tree) instead of storing and using the intermediate patterns that are generated at the mining tasks. However, it is still incapable of processing large-scale datasets that exceed the capacity of the main memory.

In this study, we present SGMiner, a fast and scalable, GPU- and disk-based, frequent itemset mining method on a single machine. SGMiner can efficiently process transaction datasets larger than the capacity of the GPU device memory and the main memory as well as quickly extract frequent patterns by exploiting GPUs. It is based on the Apriori method, including iterative candidate generation and counting steps. The method performs the candidate generation step using a CPU, whereas it performs the counting step using GPUs. SGMiner stores a transaction database in a peripheral component interconnect express solid-state drives (PCIe SSDs) and executes the algorithm for the counting step by harnessing thousands of GPU cores while streaming the transaction database via the PCIe interface. Specifically, SGMiner initially copies the data related to candidate itemsets to the GPU device memory and performs the algorithm for the counting step by executing a customized GPU kernel function to each chunk of transaction database that is loaded from the secondary memory (e.g., SSDs) to the main memory and then copied to the GPU device memory in a streaming fashion. To efficiently read pieces of transaction databases from secondary memory, we generate a CPU thread, called *dispatcher thread*, which is dedicated to the data read. Then, the *dispatcher thread* asynchronously loads pieces of transaction databases while one of the other CPU threads performs the GPU data pipeline on the already loaded pieces of transaction databases in the main memory. Here, notably, the overall performance can be reduced by overlapping the time to load the data and the time to perform the GPU data pipeline. In GPUs, asynchronous data transfer can be performed by utilizing asynchronous GPU streams, which could decrease memory access delay from GPUs to the main memory, making better use of the computing power of the GPUs more efficiently. Here, we call the process of loading pieces of transaction database from SSDs and transmitting them to GPUs as *cross-level asynchronous I/O*. SGMiner divides a set of candidate itemsets into multiple partitions, all of which are the same size. Subsequently, it replicates the same *bitmap transaction chunks* to all the GPUs in a streaming fashion and distributes different partitions of candidate itemset to each GPU so that each GPU takes the same amount of workloads. Because there is no communication overhead across multiple machines, SGMiner can outperform the existing methods. Additionally, this method achieves higher scalability compared with the existing methods since there are no data replications across machines and as transaction databases (i.e., *bitmap transaction chunks*) are stored on the secondary memory (i.e., SSD) in a single machine. SGMiner shows good scalability concerning the number of GPUs and SSDs and achieves a stable speed-up ratio as the number of GPUs and SSDs increases, owing to uniform distribution of the units of candidate itemsets to GPUs, which could perform frequent itemset extraction independently of each other. The main contributions of this paper are as follows:

- We propose SGMiner, a fast and scalable GPU- and disk-based method for frequent itemset mining that exploits multiple GPUs and SSDs.
- We present a concept of *cross-level asynchronous I/O* for decreasing the overheads for loading *bitmap transaction chunks* from SSDs to GPU device memory via the main memory and running time by GPU-based massive bitwise operations only using bitmap transaction chunks.
- We present a concept of replicating bitmap transaction chunks stored in SSDs to GPUs in a streaming fashion for fully exploiting multiple GPUs and SSDs.
- We analyze the space cost and time cost of SGMiner.
- Through experiments, we show the superiority of SGMiner compared with the state-of-the-art methods across a wide range of benchmarks.

The rest of this study is organized as follows: In Section II, we discuss the related work. We present the SGMiner method in Section III and IV. Section V presents the strategy of exploiting GPUs and the analysis of space and time costs. Section VI shows the results of the experimental evaluation. Then, we summarize and conclude this study in Section VII.

II. RELATED WORK

The frequent itemset mining problem is formally defined as extracting all itemsets that occur *minsup* times at least as a subset of transactions in a given transaction database $TD = \{td_1, td_2, \ldots, td_n\}$, where td_i is a subset of distinct items from *TD* and *minsup* is predefined. In this study, we are concerned about the number of occurrences to be a support of an itemset. We divide the existing methods of frequent itemset mining into four groups: (1) sequential methods (CPU), (2) multi-threaded methods (CPU-based), (3) disk-based methods (distributed), and (4) GPU-based methods. In Table 1, we summarize the existing methods and their characteristics, including the performance tendency of the proposed method. In the table, the performance bottlenecks are colored in red. Overall, the proposed method, SGMiner, includes outstanding factors in terms of performance, whereas all the competitors include performance bottlenecks.

We discuss the details of the existing methods in Section II-A– II-D. Section II-A presents representative sequential methods that are widely used to devise multi-threaded methods, disk-based (distributed) methods, and GPU-based methods. Section II-B presents multi-threaded methods, whereas Section II-C presents disk-based (distributed) methods. GPU-based methods are presented in Section II-D.

A. SEQUENTIAL METHODS

A number of sequential methods have been devised to extract frequent patterns efficiently and effectively. Representative methods include SSFIM [27], Apriori [13], Eclat [14], [45], LCM [19], FP-Growth [12]. Apriori algorithm [13] uses anti-monotone property for efficiency. That is, if a L-itemset is not frequent, its supersets cannot be frequent. To extract frequent $(L+1)$-itemsets, it creates candidate $(L+1)$-itemsets using frequent L-itemsets and tests them [13]. Eclat algorithm [14], [45] builds equivalence classes, which partition the database into a number of independent sub-databases. It efficiently computes supports using set intersection operations by adopting a vertical data format. LCM [19] is a variant of Eclat; it combines a number of methods (i.e., occurrence deliver, bitmap database, prefix tree) from the existing methods. As a result, in the FIMI04, LCM performed best. FP-Growth [12] constructs an FP-tree from a database; it retains the itemset association information, and extracts frequent patterns by recursively accessing the FP-tree without generating candidate itemsets. FP-Growth* [17] is one of representative implementations of FP-Growth which utilizes additional array data structures to decrease the number of tree traversals. The excellence of FP-Growth* was proven in the FIMI03. SSFIM [27] extracts frequent patterns by scanning an input database once. As this method enumerates and counts all candidate itemsets that occur in each transaction, it is capable of generating a fixed number of candidates regardless of *minsups*, intuitively saving the costs in terms of running time with big data.

As all the representative sequential methods are implemented while assuming that input and intermediate data are much smaller than the size of main memory, there are many difficulties in processing large-scale data.

B. MULTI-THREADED METHODS

Several multi-threaded methods have been considered for further enhancing performance by exploiting multiple CPU threads [36]–[38]. FP-Array [36] is a parallel FP-Growth method on modern hardware. It devises a cache-conscious data structure, which improves data locality performance, and provides a lock-free method for efficiently parallelizing the FP-Growth algorithm. When utilizing eight CPU cores, this method shows improved performance by as much as six times in recent studies [5]. MC-Eclat [38] is an Eclat-based parallel method. It significantly reduces the processing time of extracting patterns on small datasets. ShaFEM [37] is a hybrid method between FP-Growth and Eclat. This method could dynamically switch between FP-Growth and Eclat according to the density of datasets. In addition, it parallelizes the mining process without the locking requirement between threads and thus, achieves improvement in performance.

Like sequential methods, multi-threaded methods also assume that data required for the mining operation is much smaller than the size of the main memory. Thus, they have difficulties in extracting patterns from large-scale datasets. Additionally, they fail in extracting frequent patterns owing to the lack of memory on the datasets that sequential methods are able to handle. This is due to the fact that they typically have higher memory requirements than sequential methods, owing to the large memory requirements for each independent thread.

C. DISK-BASED (DISTRIBUTED) METHODS

In practice, distributed methods, which utilize a number of machines, can handle large-scale datasets. For extracting frequent patterns from large-scale datasets, numerous distributed algorithms based on a shared-nothing framework have been developed. Lin *et al.* [39] proposed SPC, FPC, and DPC based on Apriori-based methods using the MapReduce framework. SPC repeats candidate generation and support counting, as a MapReduce round. Each mapper creates and computes candidate itemsets and their supports. At the reduce step, supports of candidate itemsets are aggregated and tested. For each MapReduce round, FPC processes L-itemsets, $(L+1)$-itemsets, and $(L+2)$-itemsets together. Depending on the number of candidates, DPC dynamically collects candidate itemsets of sequential multiple lengths in a single MapReduce round. Moens *et al.* [46] proposed a hybrid approach between Apriori and Eclat algorithms, called BigFIM. This method utilizes the distributed Apriori algorithm to extract short frequent itemsets and creates

TABLE 1. Comparison of the proposed method SGMiner and existing methods.

		Parallel			
	Sequential (CPU)	CPU		GPU	GPU+Disk (single PC)
		Multi-threaded	Distributed		
Computational power	X	△	O	◎	◎
Difficulty of workload balancing	X	△	O	△	△
Network overheads	X	X	O	X	X
Scalability	X	X	O	X	O
Representative methods	Apriori (Borgelt) [7], Eclat (Borgelt) [7], Eclat (Goethals) [34], LCM [19], FP-Growth* [35]. SSFIM [27], negFIN [15], dFIN [11], PrePost+ [10]	FP-Array [36], ShaFEM [37], MC-Eclat [38]	SPC [39], BigFIM [6], PFP [40], Partition [41], BIGMiner [4]	GPApriori [42], GMiner [5], Frontier-Expansion [43], GFPG-LLMA [44]	SGMiner (proposed method)

conditional databases similar to the equivalence class. Each machine uses the sequential Eclat algorithm to compute each conditional database. Its support count is much faster than that with SPC because of its usage of the sequential algorithm. PFP [40], [47], which is included in MLlib of spark, is a distributed method of FP-Growth. Each machine builds an independent FP-tree using conditional databases and finds frequent itemsets by traversing its own FP-tree. Chon and Kim [4] propose BIGMiner, which is based on Apriori methods on the MapReduce framework. BIGMiner finds frequent itemsets of short lengths and then generates transaction bitmaps in vertical layout format in the pre-computation step. To find frequent itemsets, it repeats to generate candidate itemsets and tests them. Instead of no workload skewness, BIGMiner has a problem where if the length of frequent itemsets, calculated in pre-computation, is long, the size of bitmaps increases exponentially. Moreover, when processing small datasets or high minimum supports, BIGMiner has a worse performance than that of the existing methods due to the high overhead associated with launching iterative MapReduce jobs.

The representative methods based on distributed systems have the advantage of reducing I/O time by reading input data from multiple nodes at the same time. However, the methods of finding frequent itemsets after creating multiple independent conditional databases such as PFP or BigFIM tend to fail if the size of conditional databases becomes larger than the size of the machine's memory. Moreover, Apriori-based methods have the disadvantage of taking a long time to support counting; hence, there is scope to increase performance by utilizing the latest hardware such as GPU and FPGA.

D. GPU-BASED METHODS

GPU has different characteristics compared with CPUs, i.e., it uses the single instruction multiple threads (SIMT) and coalesced memory access models. It is difficult to apply this model into efficient frequent itemset mining methods that exploit complex data structure, e.g., FP-Tree. As a consequence, most GPU-based methods depend on Eclat or Apriori algorithm and exploit data converted to bitmap format. Then, they efficiently perform support calculations of candidate itemsets through bitwise AND operations. Fang *et al.* [48] suggested a pure bitmap implementation (PBI) and a Trie-based variant (TBI). Fang *et al.* encode the data to an $j \times k$ binary matrix where j and k is the number of itemsets and the number of transactions, respectively. To calculate supports, the GPU performs an intersection operation on rows. GPApriori [42] parallelizes the support counting step using a GPU. GPApriori generates a static bitmap to represent all 1-itemsets and their tidsets. Its candidate generation step is performed on the CPU, and the support counting step is parallelized on the GPU. Silvestri and Orlando [49] proposed a parallelized dynamic counting itemset (DCI) algorithm on GPU. Chon *et al.* [5] proposed a GMiner method for large-scale data processing. GMiner divides transaction bitmaps into multiple partitions, transmits one partition at a time to the GPU, materializes only a small bitmap in the main memory, and finds frequent itemsets. Frontier-Expansion is based on the Eclat algorithm [43]. This method exploits the vertical data layout and performs candidate generation and counting steps while it traverses the itemset search space in a depth-first search manner as in Eclat. Furthermore, it proposes an optimization strategy for GPU memory allocation to increase GPU memory utilization. GFPG-LLMA is a parallel FP-Growth method on a single GPU [44]. It devises the data structure for representing the FP-Tree in a GPU-friendly manner and extracts frequent patterns via an iterative solution that utilizes a GPU, instead of invoking the recursive functions. Consequently, this method improves the performance when processing small datasets or high minimum supports.

GPU-based methods are implemented based on the assumption that input and intermediate data can be loaded into the main memory (GPU device memory), making it difficult to process large-scale data; only GMiner and Frontier-Expansion can find patterns in data larger than the

GPU device memory. Even though these methods have better scalability in terms of the size of datasets handled than other GPU-based methods, none of them can process datasets larger than the capacity of the main memory.

III. SGMiner METHOD

A. OVERVIEW

SGMiner is composed of two phases: the building bitmap transaction chunks and finding F_L phases. The overall execution flow of SGMiner is depicted in Figure 1. For each phase, this figure includes the execution order of tasks. The transactions of frequent 1-itemsets F_1 are stored in secondary memory (SSDs) during the first phase of SGMiner. Here, such transactions are stored in a vertical layout in the form of a bitmap format to enhance the performance of support counting. Specifically, we adopt the *bitmap transaction chunk* [4] for the proposed method for efficient GPU- and disk-based frequent itemset mining. We present the process to generate bitmap transaction chunks with the limited capacity of the main memory in Section III-B. As in the Apriori method, the second phase of SGMiner is to iteratively perform two steps, namely candidate generation and counting. Henceforward, these steps are referred to as an iteration. Notice here that the counting step is substantially more expensive in terms of computations than the candidate generation step [5], [13], [20]. Thus, SGMiner is mainly concerned with the counting step in terms of avoiding the failure of entire mining tasks arising from the shortage of main memory. This is achieved by streaming the data from secondary memory to GPUs and accelerating the counting step by utilizing the computing power of GPUs. In particular, SGMiner enhances the performance of testing candidate itemsets in the second phase by only utilizing the results from the building bitmap transaction chunks phase, i.e., *bitmap transaction chunks*, with multiple GPUs while it streams *bitmap transaction chunks* stored in secondary memory (i.e., SSDs). Here, notice that the size of candidates is significantly smaller than the size of *bitmap transaction chunks*. Therefore, we are not concerned about storing the frequent itemsets on SSDs and loading them into the main memory whenever necessary. In this phase, SGMiner computes the supports of candidate itemsets using GPUs; it aggregates supports on different *bitmap transaction chunks* and discards infrequent itemsets. The details of this phase are described in Section III-C.

B. BUILDING BITMAP TRANSACTION CHUNKS PHASE

As distributed methods use a number of machines, network communication overhead is inevitable; this overhead results in the distributed methods not being able to complete extraction of frequent patterns within a reasonable time [5]. Conversely, the methods on a single machine avoid such network communication overheads, but they cannot process data that is bigger than the capacity of the main memory. We use the data pipeline on a single machine with multiple SSDs to reduce both the large network communication overheads and the lack of handling for large-scale datasets. We reduce the usage of main memory by storing the partitioned transactions encoded in bitmaps (i.e., *bitmap transaction chunks*) into SSDs and load *bitmap transaction chunks* into main memory while adjusting the number of *bitmap transaction chunks* to be loaded at the same time within the capacity of the main memory. Therefore, the proposed method could find frequent patterns without the failures caused by the shortage of main memory.

Now, we explain the process of building *bitmap transaction chunks* with limited size of the main memory. Given a set of frequent itemsets F_1 and an input transaction database TD in the horizontal format, here, suppose the transaction database TD is divided into $TD_1, TD_2, \ldots, TD_R$ so that each partition TD_i could be much smaller than the capacity of the main memory. SGMiner reads a sub-database TD_i and creates the *bitmap transaction chunk* that is equivalent to TD_i, called a *bitmap transaction chunk* BC_i. A BC_i is composed of bit vectors of frequent 1-itemsets, each of which represents $32 \times W$ transactions. The size of each *bitmap transaction chunk* BC_i is $|F_1| \times 32 \times W$ in bits, where W is the user-defined variable. Thus, the size of BC_i can be easily adjusted while considering the capacity of the main memory. The i-th bit in the bit vector of an item x indicates whether the i-th transaction includes x or not. After converting all the partitioned databases $TD_{1:R}$, the total number of *bitmap transaction chunks* is $\frac{|TD|}{W+1}$, which is denoted as R. When exploiting multiple SSDs, all the *bitmap transaction chunks* $BC_{1:R}$ are equally distributed into SSDs using the SSD ID to decrease the disk I/O. For instance, in Figure 4, when the number of *bitmap transaction chunks* R is six, and the number of SSDs M is two, BC_1, BC_3, and BC_5 are stored in SSD_1, whereas BC_2, BC_4, and BC_6 are stored in SSD_2. Figure 2 describes a pictorial example of creating *bitmap transaction chunks*. This phase consists of three steps. We assume that there are two equally-partitioned transaction databases TD_1 and TD_2 in the secondary memory and $F_1 = \{A, B, D, E, F\}$. Here, Step 1 copies a partitioned database TD_i to $TDBuf$ in main memory. Subsequently, Step 3 creates *bitmap transaction chunks* by loading and reading TD_i. For each transaction t of TD_i, it makes the corresponding bit in the bit vector of x one, where an itemset x is included in t. After all the itemsets in F_1 are processed, it stores $BCBuf_{MM}$ to secondary memory. A *bitmap transaction chunk* BC_i is composed of the bit vectors of the length $|TD_i|$, where bit vectors correspond to F_1. $BC_i[x]$ denotes a bit vector of an itemset x in BC_i. In Definition 1, we formalize the concept of physical pointers to access the bit vector of x in BC_i

Definition 1 (Position Address): We formalize the physical pointer of an itemset x as the position address of x, denoted by $PA(x)$, as the difference between the start offset of BC_k and that of $BC_k[x]$ in bytes, for a single *bitmap transaction chunk* BC_k.

We exploit this concept to efficiently access a memory offset of bit vectors associated with an itemset in a *bitmap transaction chunk*. In this study, for an itemset x, $PA(x)$ is

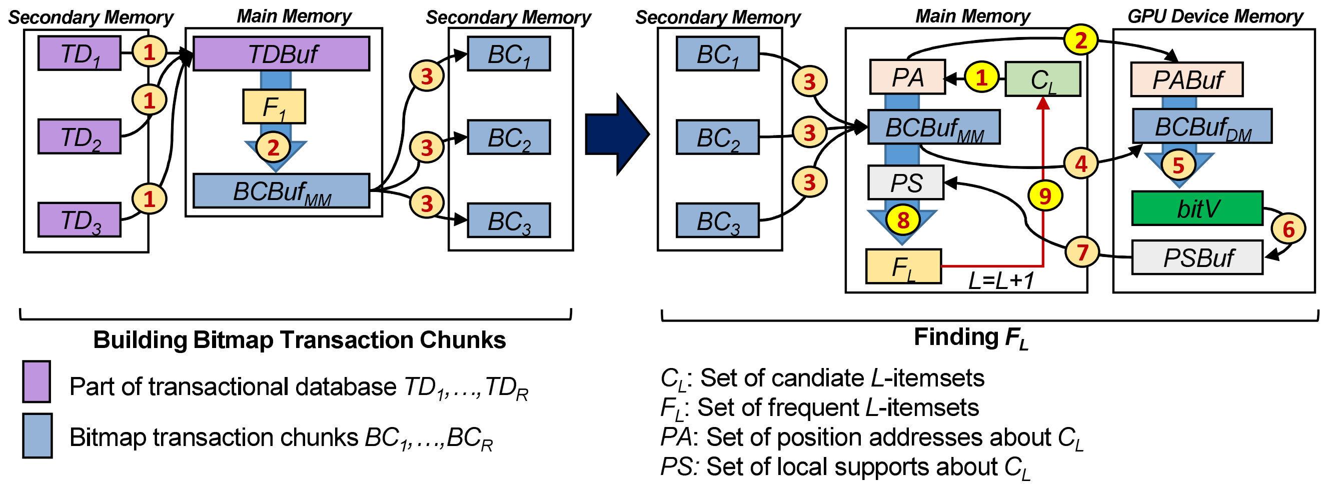

Secondary Memory
Main Memory
Secondary Memory
TD1
TD2
TD3
TDBuf
F1
BCBufMM
BC1
BC2
BC3
Building Bitmap Transaction Chunks
Part of transactional database TD1,…,TDR
Bitmap transaction chunks BC1,…,BCR
Secondary Memory
Main Memory
GPU Device Memory
PA
CL
BCBufMM
PS
FL
L=L+1
PABuf
BCBufDM
bitV
PSBuf
Finding FL
CL: Set of candiate L-itemsets
FL: Set of frequent L-itemsets
PA: Set of position addresses about CL
PS: Set of local supports about CL

FIGURE 1. Overview of SGMiner.

regarded as an ID. Notably, in all the *bitmap transaction chunks*, the position address $PA(x)$ of x is the same. This is mainly owing to the same size of *bitmap transaction chunks* and the storage of bit vectors in a consecutive space.

C. FINDING F_L PHASES

As in the Apriori method, the second phase of SGMiner repeats the candidate-generation-and-counting approach. The existing Apriori-based methods with vertical data layout usually retain considerable intermediate data in main memory. Thus, these methods hinder the mining of large-scale datasets.

SGMiner solves this problem of extracting frequent itemsets from considerable datasets without decreasing the performance beyond the capacity of main memory and GPU memory. The proposed method checks all candidate itemsets by solely exploiting the *bitmap transaction chunks* that consist of the bit vectors of frequent 1-itemsets and thus does not create any intermediate data throughout the whole mining tasks.

Additionally, we propose a new GPU- and disk-based itemset mining method, called *cross-level asynchronous I/O*, to enhance testing of candidate itemsets while decreasing the disk I/O overhead. It asynchronously transmits the *bitmap transaction chunks* to the main memory through the PCIe bus at the L-th iteration. For streaming *bitmap transaction chunks*, SGMiner creates and utilizes two CPU threads, namely, *dispatcher thread* and *pipeline thread*. The *dispatcher thread* continuously reads *bitmap transaction chunks* from secondary memory (SSDs) to the *bitmap transaction chunk* buffers $BCBuf_{MM}$ in the main memory if space is available. The *pipeline thread* carries out the data pipeline on GPUs for counting the supports of candidates. The *pipeline thread* generates the candidate L-itemsets C_L and transmits the candidate itemsets to GPUs at level L. Specifically, it transmits solely the position addresses in terms of C_L to the GPUs. The proposed method transmits $PA(C_L) = \{PA(X)|x \in C_L\}$ to the GPUs. Henceforward, we denote $PA(C_L)$ as PA for simplicity. If the size of PA is much larger than the capacity of GPU device memory, PA could be partitioned into $PA_{1:Q}$ to make each partition fit into GPU device memory. Then, it copies each PA_j, which could be regarded as the outer operand, to GPUs ($1 \leq j \leq Q$). Subsequently, for each PA_j, it transmits each partition of the inner operand, i.e., *bitmap transaction chunks*, BC_k to the GPUs ($1 \leq k \leq R$) in a streaming fashion. For each $\langle PA_j, BC_k\rangle$, SGMiner computes the *local supports* of $x \in PA_j$, using BC_k. We refer to the *local supports* for $\langle PA_j, BC_k\rangle$ as $PS_{j,k}$. In Definition 2, we formalize the *local support* of itemset x. To calculate the *local support*, SGMiner performs bitwise AND operations exploiting multiple GPUs. For bit vectors of $\{BC_k(i)|i \in x\}$, each GPU block calculates the bit vector of a candidate itemset x by performing bitwise AND operations $|x|$ - 1 times. Here, $BC_k(x)$ denotes the bit vector of a candidate itemset x. The GPU block then counts the occurrence of 1s in $BC_k(x)$ and stores its result in *PSBuf* in GPU device memory. Subsequently, it streams the calculated $PS_{j,k}$ to PS of the main memory and then aggregates them (i.e., $\sigma(c)(c \in PA_j)$).

Definition 2 (Local Support): We formalize $\sigma_x(BC_k)$ as the local support of an itemset x within a *bitmap transaction chunk* BC_k. The support of x on the whole set of *bitmap transaction chunks* $BC_{1:R}$ is $\sigma(x) = \sum_{k=1}^{R} \sigma_x(BC_k)$.

Figure 3 presents the timeline of SGMiner in terms of the copy operations from secondary memory to main memory, the copy operations from main memory to GPU device memory, and the execution of the GPU kernel function. We assume that the maximum number of *bitmap transaction chunks*, denoted as P, that are able to be loaded into the main memory is four. In terms of GPU, we assume that one GPU is exploited with four GPU streams; the use of several asynchronous GPU streams decreases the elapsed time for the data transfer between main memory (host) and GPU device memory (GPU).

The *dispatcher thread* is dedicated to loading *bitmap transaction chunks* while considering the size of memory, while

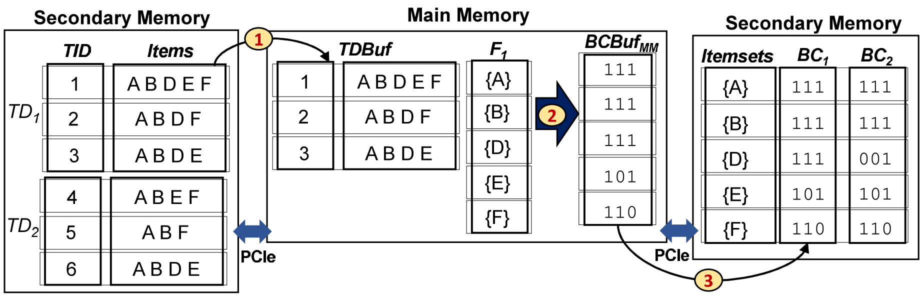


FIGURE 2. Example of building bitmap transaction chunks.

the *pipeline thread* performs the data pipeline on GPUs for calculating *local supports*. We note that these CPU threads handle different *bitmap transaction chunks* at a time, which can lead to overlapping operations performed by each thread.

Now, we will explain the data pipeline on GPUs performed by the *pipeline thread*. The *pipeline thread* initially transfers PA_j to $PABuf$. Then, it starts several GPU streams, and each GPU stream repetitively performs the following three operations, while increasing k: (1) transmitting BC_k to $BCBuf_{DM}$, (2) executing the GPU kernel function that is denoted as K for calculating $PS_{j,k}$, and (3) transmitting $PS_{j,k}$ to main memory. Here, m denotes the number of GPU streams and can be configured by users; by default, we set m to be four. In the current GPU architecture, the previously mentioned operations, i.e., transmitting to GPU memory, executing a GPU kernel, and transmitting to main memory, could overlap each other [50]. Therefore, a significant portion of the time to transmit data between the GPU device memory (GPU) and the main memory (host) could decrease. After completing m streams, all threads in GPUs are synchronized by invoking the *cudaStreamSyncronize* function in order to calculate accurate *local support* for the corresponding m *bitmap transaction chunks*.

After the GPU data pipeline for P *bitmap transaction chunks* is terminated, the *dispatcher thread* reads the remaining *bitmap transaction chunks* again, and the *pipeline thread* initiates GPU pipeline operations for the *bitmap transaction chunks* being loaded. The above steps are completed after calculating *local support* for all the *bitmap transaction chunks* stored in the secondary memory.

Figure 4 illustrates a pictorial example of SGMiner for extracting F_3 for *minsup* set to 3. In this figure, the number of all the *BCs* is six, and the maximum number of *BCs* that can be loaded into main memory at a time is three. First, the candidate itemsets $C_3 = \{\{A, B, D\}, \{A, B, E\}, \{A, B, F\}, \{A, D, E\}, \{A, D, F\}, \{A, E, F\}, \{B, D, E\}, \{B, D, F\}, \{B, E, F\}, \{D, E, F\}\}$ are generated. The *position addresses* of C_3 are divided to two chunks, i.e., PA_1 and PA_2. It starts to load BC_1, BC_2, and BC_3 into $BCBuf_{MM}$ of the main memory asynchronously. While loading *BCs*, PA_1 is synchronously transferred to $PABuf$ of GPU device memory. When loading BC_1 is completed, BC_1 is streamed to $BCBuf_{DM}$. For the pair $\langle PA_1, BC_1\rangle$, GPU blocks calculate the local support of candidate itemsets in terms of PA_1 by performing bitwise AND operations. For example, a single GPU block computes the *local support* of candidate itemset $x = \{A, B, D\}$. The GPU block performs two bitwise AND operations between $BC_1(0)$, $BC_1(1)$, and $BC_1(2)$, which are the bit vector of $\{A\}$, $\{B\}$, and $\{D\}$, respectively. Then, it is easily able to calculate the *local support* by counting the occurrence of 1s in the result 101. After calculating $PS_{1,1}$, $PSBuf$ is copied back to PS of the main memory. When copying the *local supports* into PS of the main memory, SGMiner starts to load BC_4 from SSD_2 to $BCBuf_{MM}$ and aggregate $PS(c)$ to $\sigma(c)$ ($c \in PA_1$). Likewise when calculating the *local supports* in terms of PA_1, SGMiner computes the *local support* of PA_2 by streaming $BC_{1:6}$ into GPU device memory.

IV. SGMiner ALGORITHM

SGMiner is implemented as the building *bitmap transaction chunks* phase and finding F_L phase. The first phase is building *bitmap transaction chunks*. As a result of the first phase, each *bitmap transaction chunk* is recorded to SSDs. The second phase is to find frequent L-itemsets by exploiting the *bitmap transaction chunks* that are the result of the first phase.

Algorithm 1 Generating Bitmap Transaction Chunks

```
Input: $F_1$, $TD$
Output: $BC_{1:R}$ in SSDs /* bitmap transaction chunks */
1:  $R \leftarrow T/(W \times 32) + 1$;
2:  for $i \leftarrow 1$ to $R$ do
3:      for $j \leftarrow 1$ to $W \times 32$ do
4:          for each $x \in TD[i \times W \times 32 + j]$ do
5:              if $x \in F_1$ then
6:                  Set $BC_i(x)[j]$ to 1;
7:              end if
8:          end for
9:      end for
10:     Store $BC_i$ to $SSD_{i\%M}$;
11: end for
```

Algorithm 1 shows the pseudo code for building *bitmap transaction chunks*. For the inputs, SGMiner takes frequent 1-itemsets F_1 and an input database TD in the horizontal format. The number of *bitmap transaction chunks*, denoted as R, is computed by the width of *bitmap transaction chunks* W, which is a user-defined variable. As it scans TD, for each 1-itemset x in $t \in TD$, if $x \in F_1$, j-th bit of $BC_i(x)$ is set

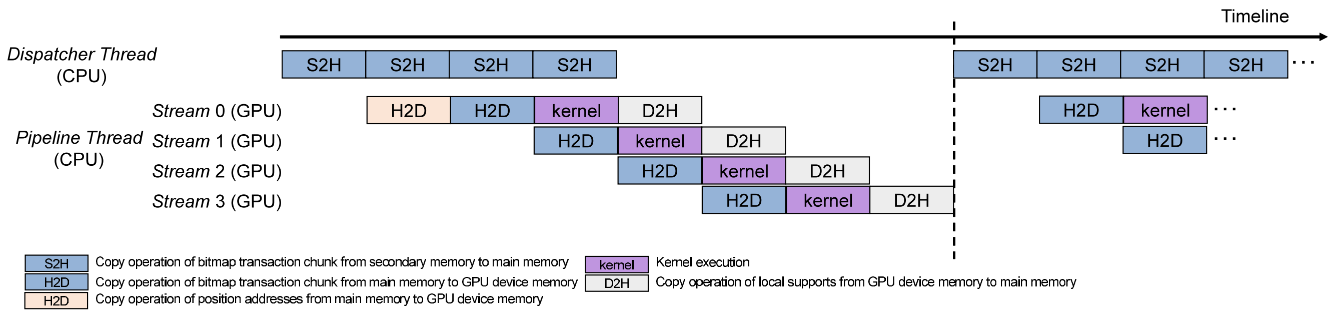

Timeline
Dispatcher Thread (CPU)
Pipeline Thread (CPU)
Stream 0 (GPU)
Stream 1 (GPU)
Stream 2 (GPU)
Stream 3 (GPU)
S2H
H2D
kernel
D2H
S2H Copy operation of bitmap transaction chunk from secondary memory to main memory
H2D Copy operation of bitmap transaction chunk from main memory to GPU device memory
H2D Copy operation of position addresses from main memory to GPU device memory
kernel Kernel execution
D2H Copy operation of local supports from GPU device memory to main memory

FIGURE 3. Example of asynchronous streaming bitmap transaction chunks.

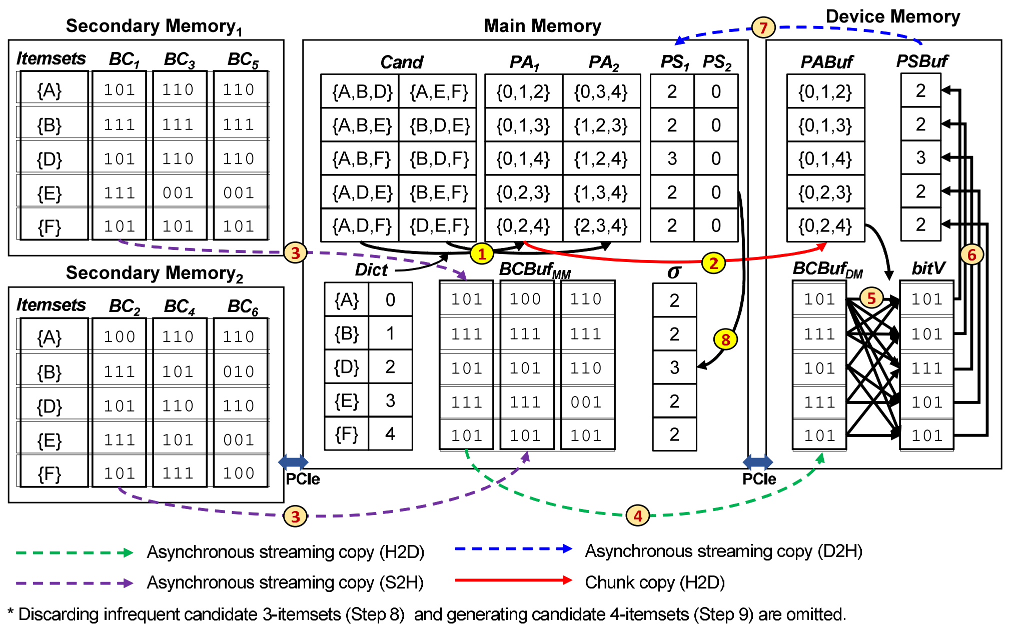

Secondary Memory₁
Secondary Memory₂
Main Memory
Device Memory
Itemsets
BC₁ BC₃ BC₅
BC₂ BC₄ BC₆
Cand
PA₁ PA₂
PS₁ PS₂
Dict
BCBuf_MM
σ
PABuf
PSBuf
BCBuf_DM
bitV
PCIe
Asynchronous streaming copy (H2D)
Asynchronous streaming copy (S2H)
Asynchronous streaming copy (D2H)
Chunk copy (H2D)
* Discarding infrequent candidate 3-itemsets (Step 8) and generating candidate 4-itemsets (Step 9) are omitted.

FIGURE 4. Example of finding F_3.

to be 1, where j is the index in the corresponding bit vector $BC_i(x)$. After scanning $W \times 32$ transactions, BC_i is stored to SSDs. To exploit multiple SSDs, $BC_{1:R}$ is equally distributed using the remainder as SSD IDs.

Algorithm 2 describes the pseudo code for the finding F_L phase. The algorithm initially allocates two buffers, namely $BCBuf_{MM}$ and PS, to the main memory (MM). Afterwards, it creates three buffers, i.e., $PSBuf$, $BCBuf_{DM}$, and $PABuf$, on GPU device memory (DM) (Lines 1-2). After this, this algorithm constructs the dictionary *dict* to be utilized in mapping x to $PA(x)$, where $x \in F_1$ (Line 3). Afterwards, it computes the maximum number of BCs, denoted as P, that could be loaded into MM (Line 4). The main loop performs repeatedly two steps: candidate generation step (Lines 9–10) and counting step (Lines 12–26) in a similar way to Apriori. However, our algorithm significantly reduces the running time of the counting by asynchronously streaming *bitmap transaction chunks* of F_1 from SSDs to main memory and from main memory to GPU device memory to overcome the limitations of the main memory and the GPU device memory, while also leveraging GPUs' massive parallelism for quick and efficient parallel computation of *local supports* (Lines 12–25). Notice that our algorithm executes the GPU kernel function K multiple times instead of executing it only once (Line 18). This is mainly because of the limited number of GPU blocks that can be specified when calling K. The K function is able to compute local support of a candidates exploiting a GPU block. Here, *maxBlock* denotes the number of available GPU blocks. When *maxBlock* = 8K, a single execution of K is able to compute local supports for 8K itemsets. Therefore, if $|PA_j|$ = 100M, it requires calling the K function 12,500 times. In terms of copying data, our algorithm first copies PA_j, which is regarded as the outer operand, to $PABuf$ of DM (Line 13). Then, if there are available spaces in $BCBuf_{MM}$ of the main memory, it performs the data pipeline on GPUs by loading BC_k from $SSD_{k\%R}$ to $BCBuf_{MM}$, executing the GPU kernel function K, and copying the result of K back to $PS[k\%P]$ of MM (Lines 15–19). Note that a single data pipeline can cover P *bitmap transaction chunks*, and, for different *bitmap transaction chunks*, three operations, i.e., transmitting to

GPU device memory, executing GPU kernel, and transmitting to main memory, could be overlapped with one another. After completing the data pipeline on GPUs in terms of P *bitmap transaction chunks*, it synchronizes the GPU threads for exact *local supports* and aggregates the *local supports* with those previously computed (Lines 21–22). Then, it sends a new asynchronous read request in order to load $BC_{k\%R}$ stored in SSDs and performs the GPU data pipeline for the *bitmap transaction chunks* to be loaded. After processing all the *bitmap transaction chunks*, it discards infrequent candidate L-itemsets at level L (Line 26).

Algorithm 2 Finding F_L

```
Input: $F_1, BC_{1:R}$
Output: $F$ /* frequent itemsets */
 1: Allocate $\{BCBuf_{MM}, PS\}$ on MM;
 2: Allocate $\{BCBuf_{DM}, PABuf, PSBuf\}$ on DM;
 3: $dict \leftarrow$ dictionary mapping $x$ to $PA(x) (x \in F_1)$;
 4: $P \leftarrow$ computing the maximum number of $BC$s in $MM$;
 5: $L \leftarrow 1$;
 6: while $|F_L| > 0$ do
 7:     $L \leftarrow L + 1$;
 8:     /* Candidate generation using CPU */
 9:     $C_L \leftarrow$ generate candidates with $F_{L-1}$;
10:     Convert $C_L$ to $PA_{1:Q}$ with $dict$;
11:     /* Counting using GPUs */
12:     for $j \leftarrow 1$ to $Q$ do
13:         Copy $PA_j$ into $PABuf$ of $DM$;
14:         for $k \leftarrow 1$ to $R$ do
15:             if $BCBuf_{MM}$ is not full then
16:                 Async load $BC_k$ from $SSD_{k\%R}$ to $BCBuf_{MM}$;
17:                 Async copy $BC_k$ to $BCBuf_{DM}$;
18:                 Call $K(PA_j, BC_k)$;
19:                 Async copy $PSBuf$ of $DM$ into $PS[k\%P]$ of $MM$;
20:             else
21:                 Thread synchronization of GPUs;
22:                 $\sigma(c) \leftarrow \sum_{q=1}^{P} PS[c][q]$, for $\forall c \in PA_j$;
23:             end if
24:         end for
25:     end for
26:     $F_L \leftarrow \{c | c \in C_L \wedge \sigma(c) \geq minsup\}$;
27: end while
28: $F \leftarrow \cup F_L$;
29: return $F$;
```

Algorithm 3 concerns the pseudo code with regard to the GPU kernel function K of SGMiner. It takes a pair of PA_j and BC_k, *completeIdx*, and *maxThread* as inputs. *completeIdx* denotes the index of the last candidates that are completed in PA_j. This variable is used to detect the portion of PA_j to be addressed in the current call of the function K. For instance, when $|PA_j| = 1{,}000$ and $maxBlock = 100$, *completeIdx* becomes 100 in the second call of K. *maxThread* denotes the maximum number of threads in a GPU block and can be specified when invoking K with *maxBlock*. *BlkID* and *ThrID* denote the system-generated IDs for the GPU block and the GPU thread, respectively. As a large number of GPU blocks run simultaneously, some GPU blocks may not have a corresponding set of candidate itemsets to test. For example, if $|PA_j| = 1000$ and $maxBlock = 2000$, some blocks do not have a set of itemsets; hence, 1000 GPU blocks cannot run GPU kernel functions. Therefore, if there is no itemset to be processed by the current GPU block, the GPU kernel function is immediately terminated (line 1–2). To improve efficiency, the GPU kernel function creates two variables (i.e., *cand* and *supp*), which are frequently accessed, in shared memory on GPUs. The variable *cand* contains a set of items whose *local support* will be computed by the GPU block *BlkID*. The vector *supp* is initially set to 0 (Lines 4–5). The main loop of K repeats the bitwise AND operation at the same time (Lines 5-8). In the modern GPU architecture, a GPU thread is able to carry out bitwise operations effectively at a single precise width (namely, 32 bits). The GPU block may simultaneously carry out bitwise operations up to $32 \times maxThread$ bits. In practice, the width of the *bitmap transaction chunk*, denoted as W, is significantly larger than $maxThread \times 32$ bits. For each $x \in PA_j$, it extracts the bit vectors in terms of x from BC_k, performs the bitwise AND operation between the extracted bit vectors, and store the result into *bitV* (Line 7). The GPU kernel repeatedly performs this process $\frac{W}{maxThread \times 32}$ times. To compute the *local supports*, it counts the occurrence of 1s in *bitV* by CUDA built-in *popCnt* function and stores them in the *supp* array (Lines 8–9). At last, the GPU kernel function adds all the values of the *supp* array as a single *local support* of BC_k using the parallel summation reduction algorithm *parallelReduction*, which takes an array as an input and recursively adds all of the array's values to the first element (Line 11).

Algorithm 3 K

```
Input: $PA_j$; /* j-th chunk of $PA$*/
Input: $BC_k$; /* k-th bitmap transaction chunk*/
Input: $completeIdx$; /* index of the last candidate itemset completed in $PA_j$*/
Input: $maxThread$; /* the number of threads on GPUs*/
Output: $PSBuf$; /* local supports of <$PA_j$, $BC_k$> */
 1: if $completeIdx + BlkID \geq |PA_j|$ then
 2:     return;
 3: end if
 4: $cand \leftarrow PA_j[completeIdx + BlkID]$;
 5: $supp[0 : maxThread] \leftarrow 0$;
 6: for $q \leftarrow 0$; $q < \frac{W}{maxThread * 32}$; $q \leftarrow q + 1$ do
 7:     $bitV \leftarrow \cap_{q \in cand} BC_k[q][w \times maxThread + ThrID]$;
 8:     $supp[ThrID] \leftarrow supp[ThrID] + popCnt(bitV)$;
 9:     $syncthreads()$;
10: end for
11: $PSBuf[completeIdx + BlkID] \leftarrow parallelReduction(supp[])$;
```

V. MULTIPLE GPUs AND COST ANALYSIS

A. EXPLOITING MULTIPLE GPUs

We depict the details of exploiting GPUs in this section. We are concerned about how to set the configuration of GPUs (i.e., the number threads on GPUs) for decreasing the running time of the GPU kernel function. Then, we present the strategy for exploiting multiple GPUs.

GPU threads could not run separately in applications different from CPUs. Instead, each GPU deals with a group of 32 threads, namely warp. More specifically, several GPU threads in a warp can be simultaneously exploited in a single instruction multiple thread (SIMT) way. Notice that each GPU block is composed of several warps. The GPU kernel function in SGMiner incorporates the *parallelReduction* function as described in Section IV. Nevertheless, since it uses several branch and bound operations, the running time grows as the number of threads on each GPU grows when exploiting GPUs. Such a performance tendency is more pronounced while increasing the number of GPU threads. Thus, by default we set the number of threads on GPUs to be 64. Here, it should be noted that we omit the discussion in terms of the number of GPU blocks. The reason is mainly owing to a limit on the number of GPU blocks under the moderun GPU architecture [5]. Instead, in Section VI, we show the performance tendency while changing the number of GPU blocks.

Now, we will discuss the method to utilize multiple GPUs. SGMiner could be readily scaled to utilize multiple GPUs, which can reduce running time further. When using several GPUs, SGMiner replicates the same *bitmap transaction chunks*, which are streamed from secondary memory (i.e., SSDs), to all the GPUs, while it transmits the different parts of the outer operand (i.e., *position addresses*) to different GPUs. Figure 5 illustrates the data flow of our proposed method for exploiting multiple GPUs. When exploiting two GPUs, this method involves copying PA_1 to GPU_1 and PA_2 to GPU_2, respectively. Then, this method involves copying the same BC_8 to two GPUs. Afterwards, the GPU kernel function on GPU_1 computes the *local supports* of PA_1, while that on GPU_2 computes the *local supports* of PA_2. Since there are no common itemsets in PA_1 and PA_2, the results of these GPU kernel functions could be computed and transmitted back to the PS of main memory without conflicts. Since workloads that compute the *local supports* in terms of PA_1 and PA_2 are independent, scalability greatly increases with respect to the number of GPUs. In addition, there is no workload imbalance problem, which is a common problem with parallel and distributed computing methods. This is because the performed tasks do not use anomalous or complicated data structures, and the amount of workload caused by these tasks is proportional to the size of PA_j and PA_k, which are similar in size. Thus, regardless of the datasets, the proposed approach shows a stable speed-up while it grows the number of GPUs.

B. SPACE COST ANALYSIS

In SGMiner, it is required for storing the *bitmap transaction chunks* in the disk space (i.e., SSDs). Total *bitmap transaction chunks* require $|F_1| \times |W| \times R$ bytes in SSD, where F_1, $|W|$, and R indicate the number of frequent 1-itemsets, width of *bitmap transaction chunks*, and number of *bitmap transaction chunks*, respectively. When exploiting multiple SSDs, the required disk space is $\frac{|F_1|\times|W|\times R}{M}$ bytes for each SSD, where M denotes the number of SSDs utilized.

The space of main memory for $BCBuf_{MM}$ is $|F_1| \times |W| \times P$ bytes, where P is the number of *bitmap transaction chunks* that could be loaded into the main memory. We note that although the total size of *bitmap transaction chunks* could be bigger than the size of heap memory, the required memory space is only $|F_1| \times |W| \times P$ bytes during the entire mining task. This is because the proposed method does load one or more *bitmap transaction chunks* whose sizes are less than the capacity of the main memory instead of loading all the *bitmap transaction chunks*. For instance, suppose that $|TD| =$ 200M, $|F_1| = 500$, and $W = 8{,}192$. The total size of *bitmap transaction chunks* is $500 \times 200{,}000{,}000 \approx 93.2$ GB, whereas the size of each *bitmap transaction chunk* is $500 \times 8{,}192 \approx$ 4 MB. Therefore, assuming that the number of *bitmap transaction chunks* which could be loaded to $BCBuf_{MM}$ of the main memory is ten (i.e., $P = 10$), the required space is approximately 40 MB in main memory regardless of the volume of input database. The memory requirement for $PA_{1:Q}$ is $maxCand \times maxlen \times 4$ bytes, where $maxCand$ and $maxlen$ are denoted as the maximum number of candidates for each iteration and the maximum length of candidates, respectively. For instance, if $maxCand = 100$M and $maxlen =$ 25, the memory space for $PA_{1:Q}$ required is $100{,}000{,}000 \times 25 \times 4 \approx 9.31$ GB. The memory space required for PS is $maxCand \times S \times 4$ bytes, where S is denoted as the number of GPU streams for each GPU. Suppose that if $maxCand =$ 100M and $S = 4$, the memory space for PS is $100{,}000{,}000 \times 4 \times 4 \approx 1.5$ GB. Using a disk-based approach, we can easily reduce the amount of memory required for $PA_{1:Q}$ and PS when the number of candidate itemsets is large. Specifically, it stores $PA_{1:Q}$ and PS in SSDs, loads the subset of $PA_{1:Q}$ and PS into the main memory, and then processes them sequentially, while carefully considering the capacity of the main memory.

C. TIME COST ANALYSIS

We describe the cost models of SGMiner with respect to the disk I/O and computations on GPUs. Equation 1 describes the total cost for the entire tasks (namely, all the iterations). Here, the terms $cost_{disk}(L)$, $cost_{copy}(L)$, and $cost_{gpu}(L)$ represent the cost of the disk I/O, the cost of the data transfer between main memory and GPU device memory, and the cost of the computations on GPU, respectively, at level L.

$$\sum_{L=1}^{\#\,iterations} cost_{disk}(L) + cost_{copy}(L) + cost_{gpu}(L). \qquad (1)$$

Equation 2 presents the cost of reading the data from SSDs into main memory. Here, the *position addresses* $PA_{1:Q}$ do not require the disk I/O by default. Therefore, the terms in terms of $PA_{1:Q}$ are omitted. Here, $\frac{|BC_k|}{speed_{disk}}$ indicates the amount of time to load a single *bitmap transaction chunk*, and this operation is repeated $\frac{R}{P}$ times, where $speed_{disk}$, R, and P represent the disk bandwidth, the number of all the *bitmap transaction chunks*, and the maximum number of *bitmap transaction chunks* that could be loaded into main memory, respectively. In case of exploiting M SSDs, $\frac{|BC_k|}{speed_{disk}} \times \frac{R}{P}$ could be divided by M, as M BCs are concurrently loaded into the main memory.

$$cost_{disk} = \frac{1}{M} \times \left\{ \frac{|BC_k|}{speed_{disk}} \times \frac{R}{P} \right\}. \qquad (2)$$

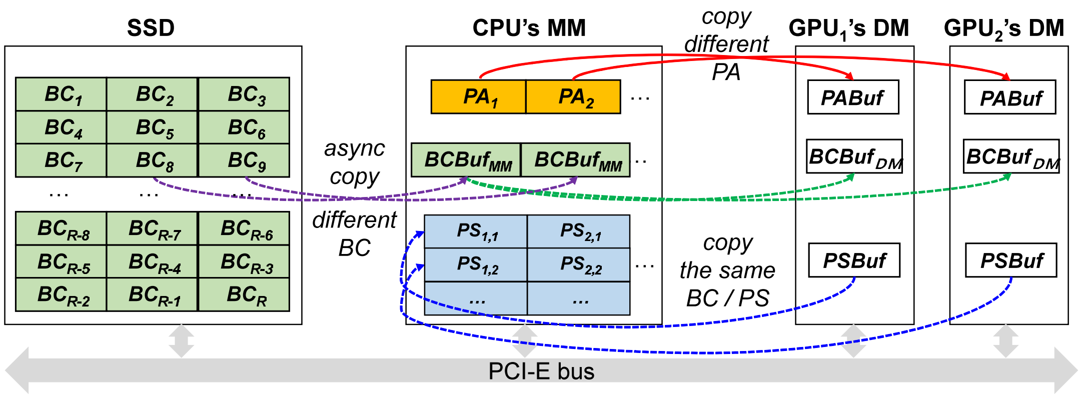


FIGURE 5. Strategy of exploiting multiple GPUs.

Equation 3 presents the cost of the data transmission from the main memory to GPU device memory. Here, $comm_1$ and $comm_2$ denote the communication throughput between main memory and GPU device memory in a chunk copy fashion and that in a streaming copy fashion, respectively. The term $\frac{|PA_{1:Q}|}{comm_1 \times N}$ denotes the total cost of transmitting the *position addresses* $PA_{1:Q}$ to GPU device memory. This cost could decrease by N times since it concurrently transmits the data to N GPUs. In brackets, the terms denote the cost of asynchronously transmitting the data to compute *local supports*. For all the *position addresses* $PA_{1:Q}$, N GPUs could process the different *position addresses* PA_k at the same time. Thus, the data transfer in terms of the bracket is repeated $\frac{Q}{N}$ times, and its cost also increases proportional to $\frac{Q}{N}$. More specifically, the term $\frac{|BC_{1:R}|}{comm_2}$ indicates the cost of streaming the *bitmap transaction chunks* and *local supports*. This cost could not be decreased by utilizing multiple GPUs as the proposed method replicates all the *bitmap transaction chunks* $BC_{1:R}$ to all the GPUs. The term $\frac{|PS_{j,R}|}{comm_2}$ represents the cost of copying the *local supports* on the last *bitmap transaction chunk* BC_R to main memory, which is unable to concealed by streaming.

$$cost_{copy} = \frac{|PA_{1:Q}|}{comm_1 \times N} + \frac{Q}{N} \times \left\{ \frac{|BC_{1:R}|}{comm_2} + \frac{|PS_{j,R}|}{comm_2} \right\}. \quad (3)$$

Equation 4 presents the cost of the GPU computations (i.e., counting the *local supports* on all the *bitmap transaction chunks* $BC_{1:R}$ and all the *position addresses* $PA_{1:Q}$). In the equation, the terms in brackets indicate the GPU computations in terms of a single partition of *position addresses* PA_j. Here, it is repeated $\frac{Q}{N}$ times, as in processing $PA_{1:Q}$ using N GPUs in Equation 3. More specifically, $t_{call}(m)$ is the overheads of calling a GPU kernel function m times. SGMiner calls the GPU kernel function $R \times \lceil \frac{PA_j}{maxBlk} \rceil$ times for each PA_j, where *maxBlk* is the number of GPU blocks. This is because each GPU block counts the *local support* of a single candidate itemset. The term $t_{ker}(BC_R)$ represents the execution time for the GPU kernel function for the last *bitmap transaction chunk* BC_R, which cannot be overlapped with asynchronous data transfer by streaming.

$$cost_{gpu} = \left\{ t_{call}(R \times \lceil \frac{PA_j}{maxBlk} \rceil) + t_{ker}(BC_R) \right\} \times \frac{Q}{N}. \quad (4)$$

VI. PERFORMANCE EVALUATION

We perform the experiments of SGMiner compared to the existing methods, which are mentioned in Table 1. We describe the results of experimental evaluation in three categories. First, we show the performance comparison between SGMiner, the representative sequential methods (i.e., Eclat by Borgelt [7], Eclat by Goethals [34], LCM [19], Apriori by Borgelt [7], FP-Growth* [17], SSFIM [27], negFIN [15], dFIN [11], PrePost+ [10]), the representative parallel methods (i.e., FP-Array [36], ShaFEM [37], MC-Eclat [51], GPApriori [42], GMiner [5], Frontier-Expansion [43], GFPG-LLMA [44]), and the representative disk-based (distributed) methods (i.e., SPC [39], BigFIM [6], PFP [40], BIGMiner [4], Partition [41]), using real datasets and synthetic datasets with different settings of *minsups*. Second, we show the scalability of the representative frequent itemset mining methods while growing the input data sizes and the scalability of SGMiner while varying the computing units (i.e., GPUs and SSDs). Third, we show the performance tendency of SGMiner while it changes a range of settings.

A. DATASETS

For the performance evaluation, we use both synthetic and real datasets. The used datasets are described in Table 2. We utilize Webdocs [52], which is the largest dataset in [53]. This dataset is considerably utilized for performance comparison between the frequent itemset mining methods. However, this dataset is not sufficiently large to test disk-based methods, therefore we utilize the Yahoo dataset, which is one of the largest graph datasets [54]. There have been a number of studies that use graph datasets for performance comparison of pattern mining methods, including [55]. For the Yahoo dataset, we transform the adjacency list of a single vertex in that dataset into a single transaction so that it could be utilized for the performance comparison of frequent itemset mining methods. We create multiple synthetic datasets by

TABLE 2. Datasets used in the experimental evaluation.

Parameter	Webdocs	Yahoo	Q-Scale
T_{avg}	177.2	11.1	200
L_{avg}	N/A	N/A	25
$\|TD\|(\times \mathbf{10}^6)$	1.69	650	[10, 25,50,75,100]
$\|I\|(\times \mathbf{10}^3)$	5,268	1,413,511	10
$Size(GB)$	1.4	60	[9,14,28,42,55]
$Name$	N/A	N/A	[Q10M,Q25M,Q50M,Q75M,Q100M]

exploiting the IBM Quest Dataset Generator [13], which takes four major parameters: the number of items $|I|$, the length of maximal pattern L_{avg}, the number of transactions $|TD|$, and the average number of items in transactions T_{avg}. We create several synthetic datasets, namely Q-Scale, while varying such parameters. Here, we notice that the real datasets (namely, Yahoo and Webdocs) are sparse, while the datasets in Q-Scale are dense.

B. ENVIRONMENTS

In order to evaluate Eclat (Borgelt), Eclat (Goethals), LCM, Apriori (Borgelt), FP-Growth*, negFIN, PrePost+, GPApriori, GMiner, GFPG-LLMA, and Frontier-Expansion, we use the latest source codes provided by the authors. For evaluating dFIN, we download and use the implementation presented in [56]. For a fair comparison here, note that we slightly changed the source codes of negFIN, PrePost+, and dFIN, which are running on Windows, to make them run on Linux. For FP-Array, MC-Eclat, and ShaFEM, we download and utilize the implementations provided in [57] and [58]. In order to evaluate SSFIM, we use best-effort reimplementation based on the original study. In order to evaluate the distributed methods in Table 1, we implement the representative distributed methods on Apache Hadoop [59] excluding BigFIM and PFP. We download and use the source codes of BigFIM and PFP in [6] and [60], respectively. Here, note that we slightly extend the implementation of PFP for extracting all the frequent itemsets, since the open-sourced code of PFP, which is included in Mahout library, extracts solely top-k frequent patterns. We compile all the methods on a single machine with gcc 9.4.0 and CUDA 11.4. For evaluating the distributed methods, we utilize Java 1.8 and Apache Hadoop 1.2.1 for all five distributed frequent itemset mining methods.

We conduct all our experiments of five GPU-based frequent itemset mining methods (including SGMiner) and 12 CPU-based frequent itemset mining methods on the same server. The server is equipped with two Intel 6-Core 3.50GHz CPUs, two NVIDIA GTX 1080 of 8 GB GPU device memory, 64 GB of main memory, and two PCIe SSDs of 1 TB. Here, multiple CPUs and GPUs in the server are connected by using the PCIe 3.0 x16 connection. We also have conducted all the experiments of the five distributed frequent itemset mining methods on the same cluster, which is composed of one primary node and 20 secondary nodes that are connected by a 1 Gbps network. Each node includes Intel quad-core 3.40GHz CPU, 32 GB of main memory, and two 3 TB HDDs.

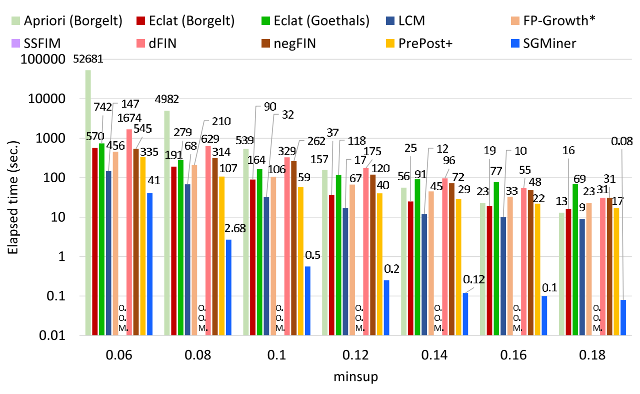
Now, we describe the detailed settings in terms of GPU exploited for the experiments of the GPU-based frequent itemset mining methods, i.e., GPApriori, Frontier-Expansion, GFPG-LLMA, and GMiner. By default, we use a single GPU for measuring the performance of the GPU-based methods. In terms of GPU threads and GPU blocks, we apply the best parameter for each method by trial-and-error-based tuning except for GFPG-LLMA, for which we use the default configuration. This is because this method dynamically changes the number of GPU blocks and threads according to the program logic. Here, Thr_{GPU}, Blk_{GPU}, and Str_{GPU} denote the number of threads on GPUs, the number of GPU blocks, and the number of GPU streams, respectively. We set Thr_{GPU} to 32 and Blk_{GPU} to 16,384 for GPApriori. In addition, we set Thr_{GPU} to 32 and Blk_{GPU} to 4,096 for Frontier-Expansion. We set Thr_{GPU}, Blk_{GPU}, and Str_{GPU} to 32, 8,192, and 4, respectively, with respect to GMiner. Now, we will describe the configuration options of SGMiner. By default, we set the number of GPUs to 1 and the number of SSDs to 1. In terms of *bitmap transaction chunks*, we set the width of *bitmap transaction chunks* to 8,192 by default. When loading the *bitmap transaction chunks* to GPU device memory, it uses both asynchronous copy from secondary memory to main memory (S2H async) and asynchronous copy from main memory to GPU device memory (H2D async) by default. In terms of GPU configuration, the default configuration includes $Thr_{GPU} = 32$, $Blk_{GPU} = 8{,}192$, and $Str_{GPU} = 2$.

For fair comparisons, we measure the running time without loading and finalization time, except for SGMiner. For SGMiner, we measure the running time it takes to read the first *bitmap transaction chunk* from secondary memory and return the frequent itemsets. For the methods that utilize GPUs, we measure the running time including all time spent transmitting data between GPU device memory and main memory.

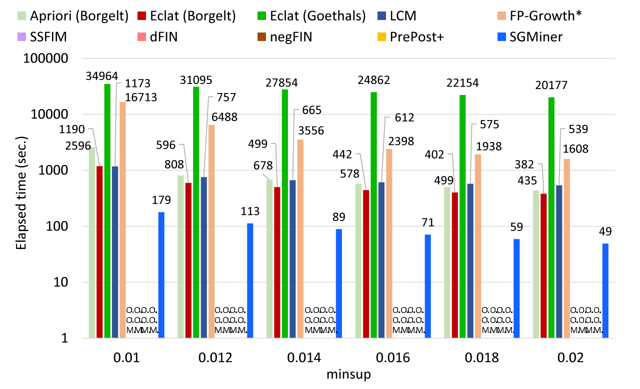
C. RESULTS

1) PERFORMANCE COMPARISON

Figures 6 (a) and (b) show the performance comparison results of SGMiner compared with the representative sequential frequent itemset mining methods, namely Eclat (Borgelt), Eclat (Goethals), LCM, Apriori (Borgelt), SSFIM, dFIN, FP-Growth*, negFIN, PrePost+, on both the Webdocs and Q10M datasets, which are small enough to load main memory, while changing *minsups*. In these figures, the X-axis represents the range of *minsups*. Here, we utilize the same range of *minsups* as in the previous work [5], [38]. The Y-axis shows the running time in log-scale. O.O.M. indicates running out of memory, i.e., a failure due to the lack of main memory. As we see figures, the running times of all methods increase as *minsup* decreases. The performance gaps between SGMiner and other frequent itemset mining methods significantly grow as *minsup* decreases. For both tested datasets (Webdocs and Q10M), SGMiner significantly and consistently outperforms other frequent itemset mining methods. As shown in Figure 6 (a), SGMiner shows 162-1284× speed-up over Apriori (Borgelt), 13-200× speed-up over Eclat (Borgelt), 18-862× speed-up over Eclat



(a) Comparison with sequential methods (Webdocs).



(b) Comparison with sequential methods (Q10M).

FIGURE 6. Performance comparison with the sequential methods.

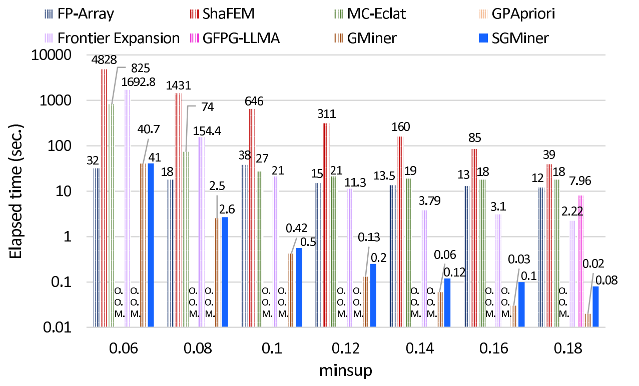
(Goethals), 3.4-112× speed-up over LCM, 11-28× speed-up over FP-Growth*, 40-387× speed-up over dFIN, 1387× speed-up over negFIN, and 8-212× speed-up over PrePost+. Here, SSFIM encounters O.O.M. errors since this method tends to consume a large amount of memory for enumerating all the combinations of long transactions. The superiority of SGMiner is mainly owing to reducing the overhead of I/O by asynchronously copying *bitmap transaction chunks* from SSDs to GPU device memory and efficiently performing massive bitwise computations by fully exploiting the computing power of GPUs as described in Section III. In Figure 6 (b), LCM, FPGrowth*, SSFIM, dFIN, negFIN, and PrePost+ show that they were running out of memory. They show O.O.M. errors since these methods are prone to high memory requirements for further enhancing performance when compared to other methods.

Figures 7 (a) and (b) describe the comparison results of SGMiner over the multi-threaded methods, namely FP-Array, ShaFEM, and MC-Eclat, and GPU-based methods, namely Frontier-Expansion, GMiner, GFPG-LLMA, and GPApriori, on the same datasets and *minsups* as those used in the performance comparison with the sequential methods. Note that we exploit 16 CPU threads for the multi-threaded methods. In these figures, GMiner shows the best performance except when *minsup* = 0.06 as this method does not require the overhead of disk I/O and simultaneously increases the performance by the support counting with highly optimized method on GPU. When *minsup* = 0.06, SGMiner outperforms all other methods except FP-Array, as shown in Figure 7 (a), despite the overhead of disk I/O. SGMiner shows 7-139× speed-up over FP-Array when the range of *minsups* is [0.08, 0.18]. In addition, our SGMiner shows 117-850× speed-up, 20-180× speed-up, 31-45× speed-up, and 99x speed-up over ShaFEM, MC-Eclat, Frontier-Expansion, and GFPG-LLMA, respectively. For both datasets, GPApriori shows that there is insufficient memory at all the test *minsups*, mainly owing to high memory requirements for building a static bitmap. In Figure 7 (b), SGMiner shows the second-best performance. Our SGMiner achieves 247-1117× speed-up and 2.8-10× speed-up over ShaFEM and MC-Eclat, respectively. The GPU-based methods, except for GMiner, easily encounter O.O.M. errors due to larger intermediate data than the capacity of the GPU device memory. FP-Array easily shows running out of memory since it requires high memory requirements to create independent memory space for each CPU thread.

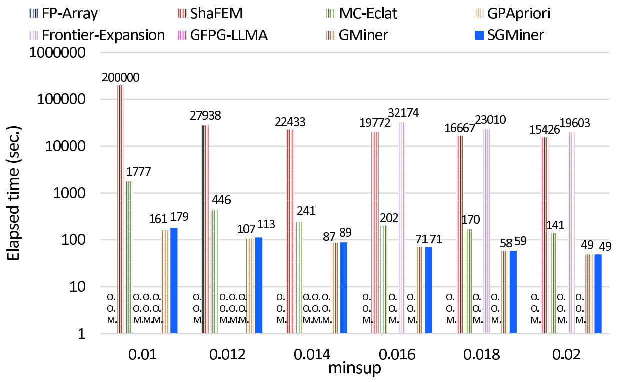
Figures 8 (a) and (b) show the comparison results over the representative distributed frequent itemset mining methods, namely Partition, SPC, BigFIM, PFP, and BIGMiner, on the Webdocs and Yahoo datasets, while changing the *minsups*. As shown in Figure 8 (a), SGMiner consistently and significantly outperforms the other methods. SGMiner shows 1909-5055× speed-up, 1451-3476× speed-up, 36000× speed-up, 10945-17306× speed-up, and 268-1316× speed-up over SPC, BigFIM, PFP, Partition, and BIGMiner, respectively. All the distributed methods, except for SPC and BIGMiner, encounter O.O.M. errors, because of the large volume of intermediate data created during the mining tasks. SPC does not meet O.O.M. errors due to its small memory requirement, but shows the poor performance owing to its inefficient support counting. For the Webdocs dataset, BIGMiner runs faster than the distributed frequent itemset mining methods as this method avoids workload skewness and has efficient support counting method of BIGMiner, which pre-computes the bit vectors of short itemsets and repeatedly reuses the pre-computed bit vectors. As shown in Figure 8 (b), SGMiner achieves 768943× speed-up, 6.9-7.3× speed-up, and 5.9-225× speed-up compared to SPC, PFP, and BIGMiner, respectively. All the distributed methods encounter O.O.M. errors or cannot extract frequent itemsets within a reasonable time at the low *minsup* (i.e., 0.00076). Only SGMiner was able to find frequent itemsets with a low *minsup* (i.e., 0.00076) in a reasonable amount of time. This is due to a number of overheads (e.g., distributing code, data, and JVM setup, and etc.) in the execution of distributed methods running on Apache Hadoop, and various problems arise due to the nature of distributed methods (e.g., workload skewness, large communication overheads across the network, etc.) as described in Section II-C.

2) SCALABILITY TEST

We compare the scalability of all 22 methods as it grows the size of datasets. Additionally, we show the scalability

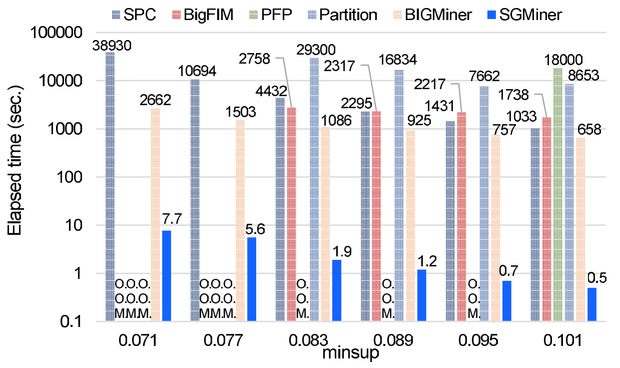


(a) Comparison with parallel methods (Webdocs).

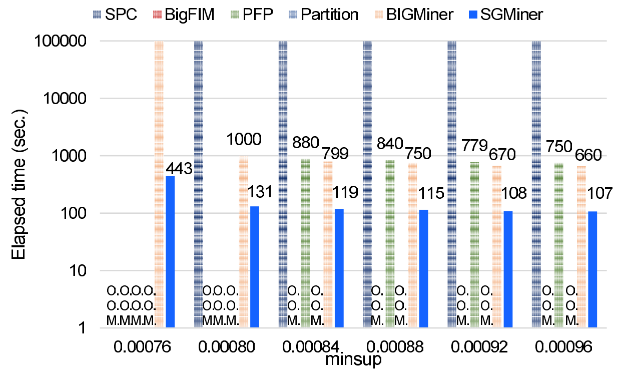


(b) Comparison with parallel methods (Q10M).

FIGURE 7. Performance comparison with the parallel methods.



(a) Comparison with distributed methods (Webdocs).



(b) Comparison with distributed methods (Yahoo).

FIGURE 8. Performance comparison with the distributed methods.

of SGMiner while we change the number of computing units (i.e., SSDs and GPUs). Figure 9 describes the elapsed time of the 22 methods in log-scale while increasing the size of database $|TD|$ from 25 million to 100 million (i.e., Q-Scale in Table 2) with $minsup = 0.01$. As we can see, the performance gap between our method and other methods significantly increases as the input size grows. For $Q25M$, among the methods on a single machine, only two sequential methods, namely Apriori (Borgelt) and Eclat (Goethals), could complete the entire mining operations, while other methods result in O.O.M. errors owing to their high memory requirements for efficiency. However, Apriori (Borgelt) and Eclat (Goethals) show poor performance due to a clear limit on computing power. From $Q50M$, all the methods running on a single machine encounter O.O.M. errors. This is because the memory requirements exponentially increase as the data size grows.

Among the distributed methods, BIGMiner outperforms other methods owing to its efficient calculation of supports by bitwise AND operations, and no workload skewness, while SPC runs slower than all other distributed frequent itemset mining methods, primarily owing to its inefficient and slow support counting. The other three distributed frequent itemset mining methods, namely BigFIM, Partition, and PFP encounter O.O.M. errors owing to their large memory requirements for the intermediate data. SGMiner shows the best scalability with all the tested datasets and outperforms other methods. As explained in Section III-B, SGMiner stores the *bitmap transaction chunks* in a secondary memory and processes multiple *bitmap transaction chunks* while considering the limit of the main memory and GPU device memory. Thus, SGMiner could handle datasets larger than the capacity of the main memory regardless of $|TD|$. Moreover, SGMiner efficiently performs the bitwise computation (i.e., support counting) using thousands of cores and, at the same time, decreases the overhead of disk I/O owing to the *cross-level asynchronous I/O*.

Figures 10 (a), (b), and (c) present the running time while we change the number of SSDs, the running time while we change the number of GPUs, and the speed-up ratio varying the number of SSDs, respectively, on the Webdocs dataset ($minsup = 0.06$), Q100M dataset ($minsup = 0.01$), and Yahoo dataset ($minsup = 0.00076$). Figure 10 (a) shows the performance tendency as we vary the number of SSDs, including the running times of SGMiner using HDD. SGMiner with one SSD performs 5.3× and 5.7× faster than SGMiner with one HDD on the Webdocs and Yahoo datasets, respectively. This is because the performance of SGMiner with one HDD is totally bound by the I/O overheads of the HDD. For the Webdocs dataset, SGMiner using one SSD slightly outperforms that using one HDD by 1.1×. This is owing to the small I/O overheads. SGMiner using 2 SSDs shows 1.12×, 1.1×, and 1.01× speed-up compared with that using one SSDs. This is because the performance of SGMiner using

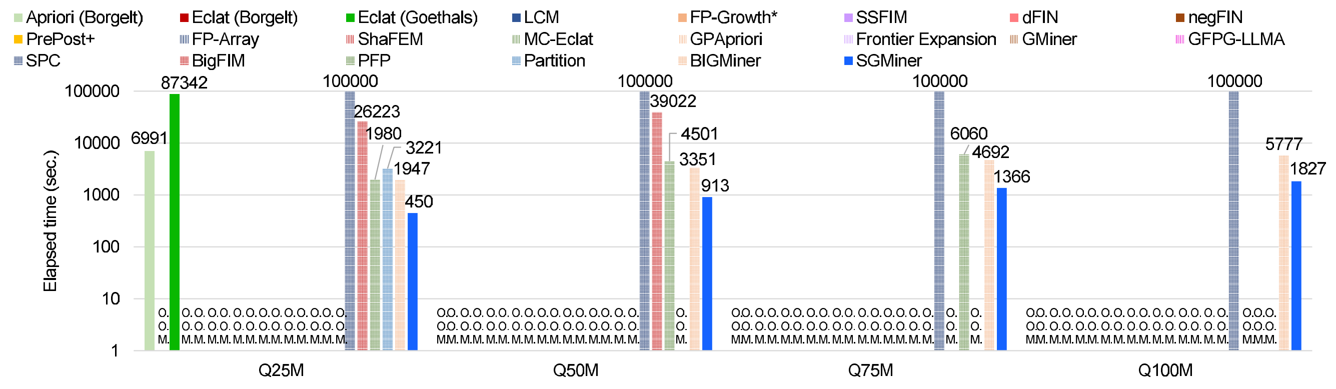


FIGURE 9. Results increasing the number of transactions (Q-Scale).

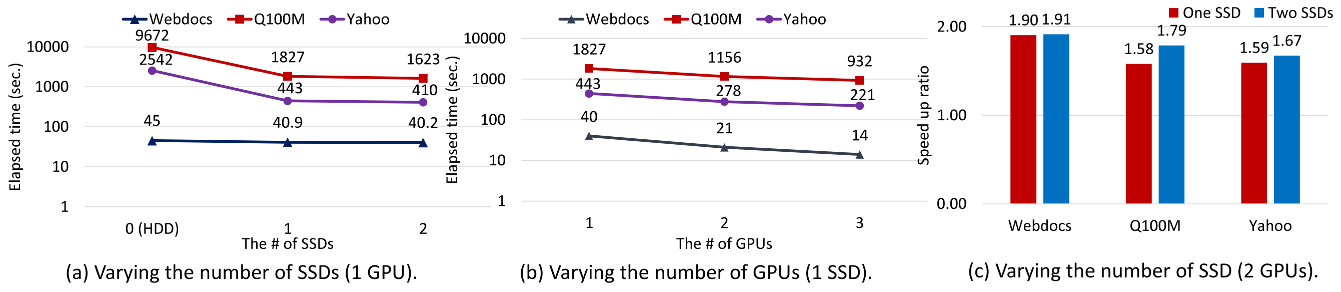


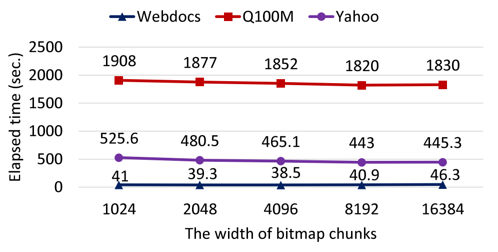
FIGURE 10. Results varying the computing units.

one SSD or two SSDs is bound by both I/O performance and computation performance. Figure 10 (b) shows the scalability while increasing the number of GPUs. SGMiner shows 1.96×, 2×, and 2.85× speed-up when increasing the number of GPUs from one to three. Here, we notice that the speed-up ratio decreases as the data size grows, since the overhead of disk I/O (i.e., loading P *bitmap transaction chunks*, where P is the number of *bitmap transaction chunks* that are able to be loaded into main memory) and synchronization (i.e., aggregating *local supports* for every group consisting of P *bitmap transaction chunks*) grows as the number of *bitmap transaction chunks* increases, as described in Section V-C. Figure 10 (c) presents the speed-up by varying the number of SSDs with two GPUs. Here, Y-axis shows $\frac{T_2}{T_1}$, where T_N is the elapsed time for the entire mining operations of SGMiner with N GPUs. Notice that the speed-up ratio grows while the number of SSDs grows. This is because the disk I/O overheads decrease by fully exploiting the bandwidth of multiple SSDs, as explained in Section V-C.

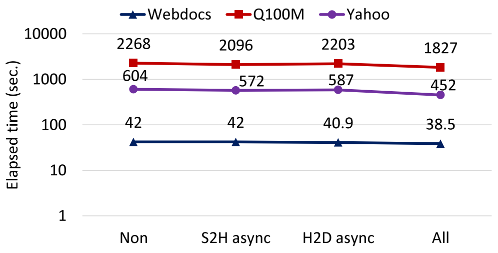
3) CHARACTERISTICS OF SGMiner

Figure 11 (a) describes the running times while we vary the width of *bitmap transaction chunks* on the Webdocs dataset (*minsup* = 0.06), the Q100M dataset (*minsup* = 0.01), and the Yahoo dataset (*minsup* = 0.00076). Here, the X-axis and Y-axis indicate the width of *bitmap transaction chunks* in four bytes and the elapsed time, respectively. As shown in the figure, 8,192 × 4 bytes = 32,768 bytes for the width of *bitmap transaction chunks* cause the best performance for all the datasets. Therefore, we use this value for SGMiner by default. Figure 11 (b) shows the elapsed times, in log-scale, while varying the strategies for copying *bitmap transaction chunks* on the Webdocs dataset (*minsup* = 0.06), the Q100M dataset (*minsup* = 0.01), and the Yahoo dataset (*minsup* = 0.00076). Here, S2H async and H2D async represent asynchronously copying *bitmap transaction chunks* from the secondary memory to the host (main memory) and that from the host (main memory) to the GPU (GPU device memory), respectively. In the X-axis, *Non* and *All* indicate copying *bitmap transaction chunks* without S2H async and H2D async and that with both S2H async and H2D async, respectively. As shown in the figure, the overhead of copying *bitmap transaction chunks* decreases while it copies the *bitmap transaction chunks* in an asynchronous fashion for all the datasets. This performance tendency is more marked as we deal with the large-scale dataset. For instance, SGMiner with S2H async and H2D async shows a 1.09× speed-up for the Webdocs dataset, a 1.24× speed-up for the Yahoo dataset, and a 1.33× speed-up for the Q100M dataset.

Figures 12 (a), (b), and (c) show the performance of different GPU configurations, namely the number of GPU blocks, the number of GPU threads, and the number of GPU streams, on the Webdocs dataset (*minsup* = 0.06), the Q100M dataset (*minsup* = 0.01), and the Yahoo dataset (*minsup* = 0.00076). Figure 12 (a) depicts the elapsed times as the number of GPU blocks is varied. Here, 8,192 GPU blocks cause the best performance; therefore, by default, we set this value for SGMiner. Figure 12 (b) depicts the elapsed times as the



(a) Varying the width of bitmap transaction chunks.



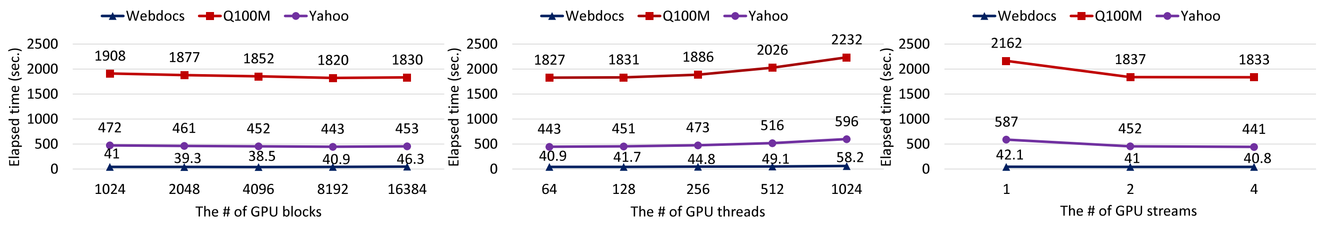
(b) Varying the strategies for copying bitmap transaction chunks.

FIGURE 11. Results varying the configurations of bitmap transaction chunks.

number of GPU threads is varied. Regardless of the tested datasets, the overall performance decreases while increasing the number of GPU threads, since each GPU mainly performs a number of branch and bound operations, which degrade the performance with the large number of GPU threads, as explained in Section III. Figure 12 (c) depicts the elapsed times as the number of GPU streams is varied. As shown in the figure, for all the datasets, the elapsed times are reduced while we increase the number of GPU streams. SGMiner, with two GPU streams, significantly improves the performance compared with that with one GPU stream, owing to hiding the overhead in transmitting data between the GPU device memory and the main memory. SGMiner along with four GPU streams slightly improves the performance compared to that with two GPU streams. This is owing to the fact that utilizing more GPU streams may increase the amount of computations that overlap with copying data.

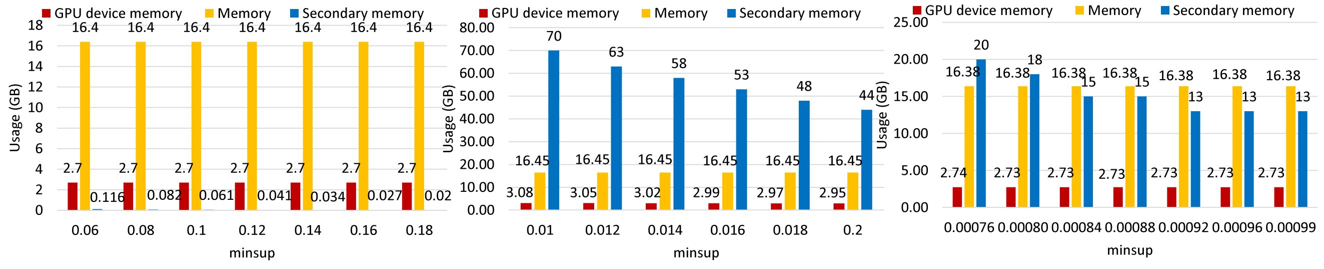
Figures 13 (a), (b), and (c) present the space usage for the datasets, namely the Webdocs dataset, the Q100M dataset, and the Yahoo dataset, varying the *minsups*. As shown in the figures, SGMiner shows a similar main memory requirement and GPU device memory requirement regardless of the sizes of datasets. For instance, for all the datasets and tested *minsups*, our SGMiner requires 16.3 GB-16.4 GB of main memory and 2.7 GB-3.08 GB of GPU device memory, whereas this method requires different secondary memory requirements for different datasets (0.02 GB-70 GB). As described in Section V-B, most of the memory requirements relate to buffers for candidate itemsets and their *local support*, but there are negligible memory requirements for *bitmap transaction chunks*. SGMiner can avoid running out of main memory due to a huge number of candidates by storing and partially processing candidate itemsets in secondary memory. Thus, SGMiner could scale with the data size by avoiding the lack of main memory and GPU device memory.

Figures 14 (a) and (b) show the profiling results of SGMiner on the Webdocs ($minsup = 0.06$), Q100M ($minsup = 0.01$), and Yahoo ($minsup = 0.00076$) datasets. Figure 14 (a) shows the time breakdown of extracting frequent itemsets (i.e., the candidate generation and counting steps). As mentioned in Section III-A, the counting step is computationally highly intensive compared to the candidate generation step. That is, the counting step accounts for approximately 99% of the execution time, and approximately 1% of the time is spent during the candidate generation step. Figure 14 (b) shows the time breakdown in terms of the counting step. We measure the time to copy the data from GPU device memory to main memory, i.e., memcpy (D2H), the time to copy the data from main memory to GPU device memory, i.e., memcpy (H2D), and the time to execute GPU kernel function, i.e., GPU kernel, which are the major performance factors in exploiting GPUs [50]. Here, we measure all the presented operations using the nvprof profiling tool that collects and views profiling data in terms of utilizing GPUs. As shown in the figure, the ratios of the GPU kernel function to the entire running time are much higher than those of memcpy (H2D) and memcpy (D2H) for all the datasets, while the ratios of the memcpy (D2H) to the entire running time are small (i.e., approximately 1%) due to the relatively small size of the data resulting from GPU kernel functions (i.e., the supports of candidate itemsets). However, the ratios of the memcpy (H2D) to the entire running time grow as the size of input data increases (i.e., the number of *bitmap transaction chunks*, *R*, increases). This tendency is mainly due to increasing the synchronization overheads for aggregating the local supports for each group of *P bitmap transaction chunks*, where *P* is the maximum number of *bitmap transaction chunks* that could be loaded into the main



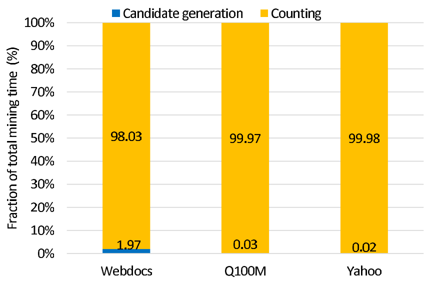
(a) Varying the number of GPU blocks. (b) Varying the number of GPU threads. (c) Varying the number of GPU streams.

FIGURE 12. Results varying the configurations of GPUs.

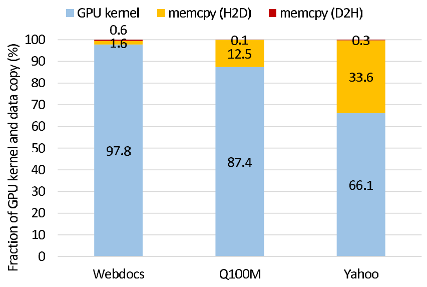


(a) Space usage (Webdocs). (b) Space usage (Q100M). (c) Space usage (Yahoo).

FIGURE 13. **Space usage varying the used datasets.**



(a) Time breakdown of candidate generation and counting steps.



(b) Time breakdown of GPU kernel and data copy.

FIGURE 14. **Results of performance profiling varying the used datasets.**

memory. In addition, the portion of hiding the cost of copying *bitmap transaction chunks* decreases as the size of input data increases, because the time to copy the first *bitmap transaction chunks* of *P bitmap transaction chunks* could not be hidden (i.e., the cost of copying $\frac{R}{P}$ *bitmap transaction chunks* to GPU device memory could not be hidden), as explained in Section III-C.

VII. CONCLUSION

We propose SGMiner, a GPU- and disk-based frequent itemset mining method running on a single machine equipped with multiple GPUs and SSDs. By storing *bitmap transaction chunks* in SSDs, carefully streaming the *bitmap transaction chunks* from SSDs to GPU device memory through main memory, and performing GPU-based massive bitwise operations only exploiting *bitmap transaction chunks*, SGMiner significantly decreases the overhead of I/O and running time. SGMiner shows scalability in terms of the size of datasets up to 6.5 billion transactions and linear scalability with the number of GPUs and SSDs. Through the tested benchmark using real and synthetic datasets, we demonstrate the superiority of SGMiner over the state-of-the-art methods. Future work includes extending the work to exploit modern GPU communication techniques (e.g., NVLink and GPUDirect RDMA) to further decrease the I/O overheads.

REFERENCES

[1] C. C. Aggarwal and J. Han, *Frequent Pattern Mining*. Cham, Switzerland: Springer, 2014.

[2] W. Lin, S. A. Alvarez, and C. Ruiz, "Efficient adaptive-support association rule mining for recommender systems," *Data Mining Knowl. Discovery*, vol. 6, no. 1, pp. 83–105, Jan. 2002.

[3] J. J. Sandvig, B. Mobasher, and R. Burke, "Robustness of collaborative recommendation based on association rule mining," in *Proc. ACM Conf. Rec. Syst. (RecSys)*, 2007, pp. 105–112.

[4] K.-W. Chon and M.-S. Kim, "BIGMiner: A fast and scalable distributed frequent pattern miner for big data," *Cluster Comput.*, vol. 21, no. 3, pp. 1507–1520, Sep. 2018.

[5] K.-W. Chon, S.-H. Hwang, and M.-S. Kim, "GMiner: A fast GPU-based frequent itemset mining method for large-scale data," *Inf. Sci.*, vols. 439–440, pp. 19–38, May 2018.

[6] (2013). *BigFIM*. [Online]. Available: https://gitlab.com/adrem/bigfim-sa

[7] C. Borgelt, "Efficient implementations of Apriori and Eclat," in *Proc. FIMI, IEEE-ICDM Workshop Frequent Itemset Mining Implementations, CEUR Workshop*, 2003, vol. 90. [Online]. Available: CEUR-WS.org

[8] (2013). *Frontier Expansion Source Code*. [Online]. Available: https://github.com/zhangfan0726/fim_gpu

[9] Y. Djenouri, H. Drias, and Z. Habbas, "Bees swarm optimisation using multiple strategies for association rule mining," *Int. J. Bio-Inspired Comput.*, vol. 6, no. 4, pp. 239–249, 2014.

[10] Z.-H. Deng and S.-L. Lv, "PrePost+: An efficient N-lists-based algorithm for mining frequent itemsets via children–parent equivalence pruning," *Expert Syst. Appl.*, vol. 42, no. 13, pp. 5424–5432, Aug. 2015.

[11] Z.-H. Deng, "DiffNodesets: An efficient structure for fast mining frequent itemsets," *Appl. Soft Comput.*, vol. 41, pp. 214–223, Apr. 2016.

[12] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *ACM SIGMOD Rec.*, vol. 29, no. 2, pp. 1–12, 2000.

[13] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proc. VLDB*, 1994, pp. 487–499.

[14] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "New algorithms for fast discovery of association rules," in *Proc. KDD*, vol. 97, 1997, pp. 283–286.

[15] N. Aryabarzan, B. Minaei-Bidgoli, and M. Teshnehlab, "negFIN: An efficient algorithm for fast mining frequent itemsets," *Expert Syst. Appl.*, vol. 105, pp. 129–143, Sep. 2018.

[16] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," *ACM SIGMOD Rec.*, vol. 26, no. 2, pp. 255–264, 1997.

[17] G. Grahne and J. Zhu, "Efficiently using prefix-trees in mining frequent itemset," in *Proc. FIMI, IEEE-ICDM Workshop Frequent Itemset Mining Implementations, CEUR Workshop*, 2003, vol. 90. [Online]. Available: CEUR-WS.org

[18] J. S. Park, M.-S. Chen, and P. S. Yu, "An effective hash-based algorithm for mining association rules," *ACM SIGMOD Rec.*, vol. 24, no. 2, pp. 175–186, 1995.

[19] T. Uno, M. Kiyomi, and H. Arimura, "LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets," in *Proc. FIMI, IEEE-ICDM Workshop Frequent Itemset Mining Implementations, CEUR Workshop*, 2004, vol. 126. [Online]. Available: CEUR-WS.org
[20] H. Yu, J. Wen, H. Wang, and L. Jun, "An improved Apriori algorithm based on the Boolean matrix and Hadoop," *Proc. Eng.*, vol. 15, pp. 1827–1831, Jan. 2011.
[21] N. Li, L. Zeng, Q. He, and Z. Shi, "Parallel implementation of Apriori algorithm based on MapReduce," in *Proc. 13th ACIS Int. Conf. Softw. Eng., Artif. Intell., Netw. Parallel/Distrib. Comput.*, Aug. 2012, pp. 236–241.
[22] F. Kovacs and J. Illés, "Frequent itemset mining on Hadoop," in *Proc. IEEE 9th Int. Conf. Comput. Cybern. (ICCC)*, Jul. 2013, pp. 241–245.
[23] K. K. Sethi and D. Ramesh, "HFIM: A spark-based hybrid frequent itemset mining algorithm for big data processing," *J. Supercomput.*, vol. 73, pp. 1–17, Jan. 2017.
[24] H. Qiu, R. Gu, C. Yuan, and Y. Huang, "YAFIM: A parallel frequent itemset mining algorithm with spark," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, May 2014, pp. 1664–1671.
[25] F. Zhang, M. Liu, F. Gui, W. Shen, A. Shami, and Y. Ma, "A distributed frequent itemset mining algorithm using spark for big data analytics," *Cluster Comput.*, vol. 18, no. 4, pp. 1493–1501, Dec. 2015.
[26] J. C.-W. Lin, Y. Djenouri, G. Srivastava, Y. Li, and P. S. Yu, "Scalable mining of high-utility sequential patterns with three-tier MapReduce model," *ACM Trans. Knowl. Discovery From Data*, vol. 16, no. 3, pp. 1–26, Jun. 2022.
[27] Y. Djenouri, D. Djenouri, J. C.-W. Lin, and A. Belhadi, "Frequent itemset mining in big data with effective single scan algorithms," *IEEE Access*, vol. 6, pp. 68013–68026, 2018.
[28] Z. Dong, "Research of big data information mining and analysis: Technology based on Hadoop technology," in *Proc. Int. Conf. Big Data, Inf. Comput. Netw. (BDICN)*, Jan. 2022, pp. 173–176.
[29] M. Yimin, G. Junhao, D. S. Mwakapesa, Y. A. Nanehkaran, Z. Chi, D. Xiaoheng, and C. Zhigang, "PFIMD: A parallel MapReduce-based algorithm for frequent itemset mining," *Multimedia Syst.*, vol. 27, no. 4, pp. 709–722, Aug. 2021.
[30] M. Sornalakshmi, S. Balamurali, M. Venkatesulu, M. N. Krishnan, L. K. Ramasamy, S. Kadry, and S. Lim, "An efficient apriori algorithm for frequent pattern mining using mapreduce in healthcare data," *Bull. Electr. Eng. Informat.*, vol. 10, no. 1, pp. 390–403, Feb. 2021.
[31] J. M.-T. Wu, G. Srivastava, M. Wei, U. Yun, and J. C.-W. Lin, "Fuzzy high-utility pattern mining in parallel and distributed Hadoop framework," *Inf. Sci.*, vol. 553, pp. 31–48, Apr. 2021.
[32] (2014). *Apache Spark*. [Online]. Available: http://spark.apache.org/
[33] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[34] B. Goethals, "Survey on frequent pattern mining," Univ. Helsinki, Helsinki, Finland, Tech. Rep., 2003, vol. 19, pp. 840–852.
[35] G. Grahne and J. Zhu, "Mining frequent itemsets from secondary memory," in *Proc. 4th IEEE Int. Conf. Data Mining (ICDM)*, Nov. 2004, pp. 91–98.
[36] L. Liu, E. Li, Y. Zhang, and Z. Tang, "Optimization of frequent itemset mining on multiple-core processor," in *Proc. PVLDB*, 2007, pp. 1275–1285.
[37] L. Vu and G. Alaghband, "Novel parallel method for mining frequent patterns on multi-core shared memory systems," in *Proc. Int. Workshop Data-Intensive Scalable Comput. Syst. (DISCS)*, 2013, pp. 49–54.
[38] B. Schlegel, T. Karnagel, T. Kiefer, and W. Lehner, "Scalable frequent itemset mining on many-core processors," in *Proc. 9th Int. Workshop Data Manage. New Hardw. (DaMoN)*, 2013, p. 3.
[39] M.-Y. Lin, P.-Y. Lee, and S.-C. Hsueh, "Apriori-based frequent itemset mining algorithms on MapReduce," in *Proc. 6th Int. Conf. Ubiquitous Inf. Manage. Commun. (ICUIMC)*, 2012, p. 76.
[40] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, "Pfp: Parallel fp-growth for query recommendation," in *Proc. RecSys*, 2008, pp. 107–114.
[41] L. Wang, "An efficient algorithm of frequent itemsets mining based on MapReduce," *J. Inf. Comput. Sci.*, vol. 11, no. 8, pp. 2809–2816, May 2014.
[42] F. Zhang, Y. Zhang, and J. D. Bakos, "GPApriori: GPU-accelerated frequent itemset mining," in *Proc. CLUSTER*, 2011, pp. 590–594, doi: 10.1109/CLUSTER.2011.61.
[43] F. Zhang, Y. Zhang, and J. Bakos, "Accelerating frequent itemset mining on graphics processing units," *J. Supercomput.*, vol. 66, no. 1, pp. 94–117, 2013.
[44] Y.-C. Wu, M.-Y. Yeh, and T.-W. Kuo, "Fast frequent pattern mining without candidate generations on GPU by low latency memory allocation," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2019, pp. 1407–1416.
[45] M. J. Zaki and K. Gouda, "Fast vertical mining using diffsets," in *Proc. 9th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2003, pp. 326–335.
[46] S. Moens, E. Aksehirli, and B. Goethals, "Frequent itemset mining for big data," in *Proc. IEEE Int. Conf. Big Data*, Oct. 2013, pp. 111–118.
[47] (2014). *Apache Spark MLlib*. [Online]. Available: http://spark.apache.org/mllib/
[48] W. Fang, M. Lu, X. Xiao, B. He, and Q. Luo, "Frequent itemset mining on graphics processors," in *Proc. 5th Int. Workshop Data Manage. New Hardw. (DaMoN)*, 2009, pp. 34–42.
[49] C. Silvestri and S. Orlando, "GpuDCI: Exploiting GPUs in frequent itemset mining," in *Proc. 20th Euromicro Int. Conf. Parallel, Distrib. Netw.-based Process.*, Feb. 2012, pp. 416–425.
[50] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim, "GTS: A fast and scalable graph processing method based on streaming topology to GPUs," in *Proc. Int. Conf. Manage. Data*, Jun. 2016, pp. 447–461.
[51] B. Schlegel, "Frequent itemset mining on multiprocessor systems," Dept. Database Technol. Group, Technische Universität Dresden, Dresden, Germany, Tech. Rep., 2013.
[52] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri, "WebDocs: A real-life huge transactional dataset," in *Proc. FIMI, IEEE-ICDM Workshop Frequent Itemset Mining Implementations, CEUR Workshop*, 2004, vol. 126. [Online]. Available: CEUR-WS.org
[53] (2005). *FIMI Repository*. [Online]. Available: http://fimi.ua.ac.be
[54] (2009). *Yahoo Webscope. Yahoo! Altavista Web Page Hyperlink Connectivity Graph*. [Online]. Available: http://webscope.sandbox.yahoo.com
[55] G. Buehrer, R. L. de Oliveira, D. Fuhry, and S. Parthasarathy, "Towards a parameter-free and parallel itemset mining algorithm in linearithmic time," in *Proc. IEEE 31st Int. Conf. Data Eng. (ICDE)*, Apr. 2015, pp. 1071–1082.
[56] (2016). *dFIN Source Code*. [Online]. Available: https://github.com/aryabarzan/dFIN
[57] S. Zalewski, "Mining frequent intra-and inter-transaction itemsets on multi-core processors," Dept. Comput. Inf. Sci., Norwegian Univ. Sci. Technol., Trondheim, Norway, Tech. Rep., 2015.
[58] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proc. PACT*, 2008, pp. 72–81.
[59] (2006). *Apache Hadoop*. [Online]. Available: http://hadoop.apache.org
[60] (2013). *Apache Mahout*. [Online]. Available: http://mahout.apache.org

KANG-WOOK CHON received the Ph.D. degree from the Department of Information and Communication Engineering, DGIST, in 2018. He worked as a Software Engineer for developing big data systems at SKT, from 2018 to 2020. He is currently working with the Division of National Supercomputing, KISTI. His research interests include scalable data mining, machine learning, and bioinformatics on parallel computing and distributed computing.

EUNJEONG YI received the B.S. and M.S. degrees from the Department of Information and Communication Engineering, DGIST. Her research interests include data mining and machine learning on heterogeneous data types.

MIN-SOO KIM (Member, IEEE) received the Ph.D. degree in computer science from KAIST, in 2006. He worked at UIUC as a Postdoctoral Fellow on data mining. He worked at IBM Almaden Research for a project of developing IBM Smart Analytics Optimizer for DB2 for z/OS. He worked at the Department of Information and Communication Engineering, DGIST, from 2011 to 2020. He is currently working with the School of Computing, KAIST. His research interests include databases, data mining, machine learning, and bioinformatics.