Noname manuscript No.
(will be inserted by the editor)

Hierarchical Pattern Mining with the Automata Processor

Ke Wang · Elaheh Sadredini · Kevin Skadron

Received: date / Accepted: date

Abstract Mining complex patterns with hierarchical structures becomes more and more important to understand the underlying information in large and unstructured databases. When compared with a set-mining problem or a string-mining problem, the computation complexity to recognize a pattern with hierarchical structure, and the large associated search space, make hierarchical pattern mining (HPM) extremely expensive on conventional processor architectures. We propose a flexible, hardware-accelerated framework for mining hierarchical patterns with Apriori-based algorithms, which leads to multi-pass pruning strategies but exposes massive parallelism. Under this framework, we implemented two widely used HPM techniques, sequential pattern mining (SPM) and disjunctive rule mining (DRM) on the Automata Processor (AP), a hardware implementation of non-deterministic finite automata (NFAs).

Two automaton-design strategies for matching and counting different types of hierarchical patterns, called *linear design* and *reduction design*, are proposed in this paper. To generalize automaton structure for SPM, the *linear design* strategy is proposed by flattening sequential patterns to plain strings to produce automaton design space and to minimize the overhead of reconfiguration. Up to 90X and 29X speedups are achieved by the AP-accelerated algorithm on six real-world datasets, when compared with the optimized multicore CPU

Ke Wang
85 Engineers Way, Charlottesville, VA 22904, USA
Fax: +1 434-982-2214
E-mail: kw5na@virginia.edu

Elaheh Sadredini
85 Engineers Way, Charlottesville, VA 22904, USA
Fax: +1 434-982-2214
E-mail: elaheh@virginia.edu

Kevin Skadron
85 Engineers Way, Charlottesville, VA 22904, USA
Fax: +1 434-982-2214
E-mail: skadron@virginia.edu

and GPU GSP implementations, respectively. The proposed CPU-AP solution also outperforms the state-of-the-art PrefixSpan and SPADE algorithms on a multicore CPU by up to 452X and 49X speedups. The AP advantage grows further with larger datasets.

For DRM, the *reduction design* strategy is adopted by applying reduction operation of *AND*, with on-chip Boolean units, on several parallel substructures for recognizing disjunctive items. This strategy allows implicit *OR* reduction on alternative items within a disjunctive item by utilizing bit-wise parallelism feature of the on-chip state units. The experiments show up to 614X speedups of the proposed CPU-AP DRM solution over a sequential CPU algorithm on two real-world datasets. The experiments also show significant increase of CPU matching-and-counting time when increasing d-rule size or the number of alternative items. However, in practical cases, the AP solution runs hundreds of times faster in matching and counting than the CPU solution, and keeps constant processing time despite the increasing complexity of disjunctive rules.

Keywords data mining · Automata Processor · sequential pattern mining · disjunctive rule mining · finite automaton

1 Introduction

Pattern mining, a subfield of data mining, is a process of analyzing data from different perspectives to identify strong and interesting relations among variables in datasets. The traditional pattern-mining techniques based on simple pattern structures, such as itemset mining and sub-string mining, are not capable of capturing hidden relations among variables in the datasets. Mining patterns with complicated structures becomes more and more important in the 'Big Data' era. Two mining techniques for hierarchical patterns, sequential pattern mining (SPM) and disjunctive rule mining (DRM) have attracted a lot of attention in the field of data mining.

Sequential Pattern Mining (SPM) is a data-mining technique which identifies strong and interesting sequential relations among variables in structured datasets. SPM has become an important data mining technique with broad application domains, such as customer purchase patterning analysis, correlation analysis of storage systems, web log analysis, software bug tracking, and software API usage tracking [3]. For example, a college student at the University of Virginia (UVA) buys textbooks according to his classes during his college years. Since every class has pre-requisite classes, a student normally follows the prerequisite order to buy and study textbooks accordingly. The UVA bookstore could study the frequent sequential patterns from the records of book purchases and give every student good recommendations for his/her next step of learning. SPM is the right technique to mine sequential relations from the records of transactions. To be precise, a sequential pattern refers to a hierarchical pattern consisting of a sequence of frequent transactions (itemsets) with a particular ordering among these itemsets. In addition to frequent itemset mining (FIM), SPM needs to capture permutations among the frequent itemsets. This dramatically increases the number of patterns to be considered and hence the computational cost relative to simple set mining or string mining operations. In addition, as the sizes of interesting datasets keeps growing, higher performance becomes critical to make SPM practical.

Many algorithms have been developed to improve the performance of sequential pattern mining. The three most competitive algorithms today are Generalized Sequential Pattern (*GSP*) [21], Sequential PAttern Discovery using Equivalence classes (*SPADE*) [27] and *PrefixSpan* [17]. *SPADE* and *PrefixSpan* are generally favored today, and perform better than *GSP* on conventional single-core CPUs in average cases. However the *GSP* is based on *Apriori* algorithm, which exposes massive parallelism and may be a better candidate for highly parallel architectures. Several parallel algorithms have been proposed to accelerate SPM on distributed-memory systems, e.g., [8, 12, 20, 26]. Increasing throughput per node via hardware acceleration is desirable for throughput as well as energy efficiency, but even though hardware accelerators have been widely used in frequent set mining and string matching applications, e.g. [10, 28, 29], we are not aware of any previous hardware-accelerated solution for SPM.

Disjunctive rule mining is derived from frequent itemset mining, but allows “alternatives” for each item. For example, let’s continue the bookstore story mentioned earlier. Each class recommends several reference books. Each student tends to select one or two reference books to buy together with the textbook for each class. Since the reference books are labeled as separate items, the strong relation between the textbook and one specific reference book may not be captured by traditional frequent itemset mining, but could be recognized by disjunctive rule mining when considering possible alternatives. The UVA bookstore could calculate the disjunctive rules from the records of book purchases and give every student good recommendations of reference books for each class. Several CPU algorithms [7, 15, 19] were proposed to mine disjunctive rules effectively. However, no hardware-accelerated disjunctive rule mining method has been proposed yet.

The new Automata Processor (AP) [9] offers an appealing accelerator architecture for hierarchical pattern mining. The AP architecture exploits the very high and natural level of parallelism found in DRAM to achieve native-hardware implementation of non-deterministic finite automata (NFAs). The use of DRAM to implement the NFA states provides a high capacity: the first-generation boards, with 32 chips, provide approximately 1.5M automaton states. *All* of these states can process an input symbol and activate successor states in a single clock cycle, providing extraordinary parallelism for pattern matching. The AP’s hierarchical, configurable routing mechanism allows rich fan-in and fan-out among states. These capabilities allow the AP to perform complex symbolic pattern matching and test input streams against a large number of candidate patterns in parallel. The AP has already been successfully applied to several applications, including regular expression matching [9], DNA motif searching [18], and frequent set mining [6, 22, 24]. In our previous work on SPM [23], we showed that the AP can also achieve impressive speedups for mining hierarchical patterns. This paper extends that prior work with additional capabilities and analysis.

Specifically, we describe CPU-AP heterogeneous computing solutions to accelerate both SPM and DRM under the *Apriori*-based algorithm framework, whose multipass algorithms to build up successively larger candidate hierarchical patterns are best suited to the AP's highly parallel pattern-matching architecture, which can check a large number of candidate patterns in parallel. This paper extends the prior AP-SPM work [23] with disjunctive capabilities and describes a flexible framework for mining hierarchical patterns such as sequential patterns and disjunctive rules with hardware accelerators. Designing compact NFAs is a critical step to achieve good performance of AP-accelerated SPM and DRM. The key idea of designing an NFA for SPM is to flatten sequential patterns to strings by adding an itemset delimiter and a sequence delimiter. This strategy greatly reduces the automaton design space so that the template automaton for SPM can be compiled before runtime and replicated to make full use of the capacity and massive parallelism of the AP board. The proposed NFA design for recognizing disjunctive rules utilizes the on-chip Boolean units to calculate *AND* relations among disjunctive items ("d-item" in short, an item allowing several alternatives) but takes full use of the bitwise parallelism appearing in the state unites of the AP chips to calculate *OR* relations of items within a d-item.

On multiple real-world and synthetic datasets, we compare the performance of the proposed AP-accelerated SPM against CPU and GPU implementations of *GSP*, an *Apriori* based algorithm, as well as Java multi-threaded implementations of *SPADE* and *PrefixSpan* [11]. The performance analysis of the AP-accelerated SPM shows up to 90X speedup over the multicore CPU *GSP* and up to 29X speedups over the GPU *GSP* version. The proposed approach also outperforms the Java multi-threaded implementations of *SPADE* and *PrefixSpan* by up to 452X and 49X speedups. The proposed AP-accelerated SPM also shows good performance scaling as the size of the input dataset grows, achieving even better speedup over *SPADE* and *PrefixSpan*. Our input size scaling experiments also show that *SPADE* fails at some datasets larger than 10MB (a small dataset size, thus limiting utility of SPADE in today's "big data" era).

The proposed CPU-AP DRM solution shows up to 614X speedups over sequential CPU algorithm on two real-world datasets. The experiments also show a significant increase of CPU matching-and-counting time when increasing the d-rule size or the number of alternative items but constant AP processing time with increasing complexity of disjunctive patterns. This analysis extends the prior analysis [23] with Boolean-based pattern matching including analysis of disjunctive features.

Overall, this paper makes four principal contributions:

1. We develop a flexible CPU-AP computing infrastructure for mining hierarchical patterns based on *Apriori* algorithm.

2. We propose a novel automaton design strategy, called *linear design*, to generate automata for matching and counting hierarchical patterns and apply it on SPM. This strategy flattens the hierarchical structure of patterns to strings and adopts a multiple-entry scheme to reduce the automaton design space for candidate patterns.
3. We propose another novel automaton design strategy, called *reduction design*, for the disjunctive rule matching and counting. This strategy takes full use of the bit-wise parallelism of the state units on the AP chips to discover the optionality of items on a lower level and utilize Boolean units on the AP chip to identify occurrences of items on a higher level.
4. Our AP SPM and DRM solutions show performance improvement and broader capability over multicore and GPU implementations of *GSP* SPM, and also outperforms state-of-the-art SPM algorithms *SPADE* and *PrefixSpan* (especially for larger datasets).

2 Sequential Pattern Mining

2.1 Introduction to SPM

Sequential pattern mining (SPM) was first described by Agrawal and Srikant [4]. SPM finds frequent sequences of frequent itemsets. All the items in one itemset have the same transaction time or happen within a certain window of time, but in SPM, the order among itemsets/transactions matters. In short, SPM looks for frequent permutations of frequent itemsets, which in turn are frequent combinations of items. FIM takes care of the items that are purchased together; for example, “7% of customers buy laptop, flash drive, and software packages together”; whereas in SPM, the sequence in which the items are purchased matters, e.g., “6% of customers buy laptop first, then flash drive, and then software packages”.

In a mathematical description, we define $I = i_1, i_2, ..., i_m$ as a set of items, where i_k is usually represented by an integer, call item ID. Let $s =< t_1 t_2 ... t_n >$ denotes a sequential pattern (or sequence), where t_k is a transaction and also can be called as an itemset. We define an element of a sequence by $t_j = \{x_1, x_2, ..., x_m\}$ where $x_k \in I$. In a sequence, one item may occur just once in one transaction but may appear in many transactions. We also assume that the order within a transaction (itemset) does not matter, so the items within one transaction can be *lexicographically ordered* in preprocessing stage. We define the *size* of a sequence as the number of *items* in it. A sequence with a size k is called a k-sequence. Sequence $s_1 =< t_1 t_2 ... t_m >$ is called a subsequence of $s_2 =< r_1 r_2 ... r_j >$ if there are integers $1 \leq k_1 < k_2 < .. < k_{m-1} < k_m \leq j$ such that $t_1 \subseteq r_{k1}, t_2 \subseteq r_{k2}, ..., t_m \subseteq r_{km}$. Such a sequence s_j is called a sequential pattern. The support for a sequence is the number of total data sequences that contains this sequence. A sequence is known as frequent *iff* its support is greater than a given threshold value called minimum support, *minsup*. The goal of SPM is to find out all the sequential patterns whose supports are greater than *minsup*.

2.2 Generalized Sequential Pattern framework

The *GSP* method, a member in the *Apriori* family, is based on the downward-closure property and represents the dataset in a *horizontal* format. The downward-closure property means all the subsequences of a frequent sequence are also frequent and thus for an infrequent sequence, all its supersequences must also be infrequent. In *GSP*, candidates of (k+1)-sequences are generated from known frequent k-sequences by adding one more possible frequent item. The mining begins with 1-sequences and the size of candidate sequences increases by one with each pass. In each pass, the *GSP* algorithm has two major operations: 1) Candidate Generation: generating candidates of frequent (k+1)-sequences from known frequent k-sequences 2) Matching and Counting: Matching candidate sequences and counting support.

2.2.1 Sequence Candidates Generation

In *GSP*, the candidates of (k+1)-sequences are generated by joining two k-sequences that have the same contiguous subsequence. c is a contiguous subsequence of sequence $s =< t_1 t_2 ... t_n >$ if one of these conditions hold:

1. c is derived from s by deleting one item from either t_1 or t_n
2. c is derived from s by deleting an item from an transaction t_i which has at least two items
3. c is a contiguous subsequence of c', and c' is a contiguous subsequence of s

Candidate sequences are generated in two steps as follows.

Joining phase Two k-sequence candidates (s_1 and s_2) can be joined if the subsequence formed by dropping the first item in s_1 is the same as the subsequence formed by dropping the last items in s_2. Consider frequent 3-sequences $s_1 =< \{A, B\} \{C\} >$ and $s_2 =< \{B\} \{C\} \{E\} >$ in Table 1; dropping the first items in s_1 results in $< \{B\} \{C\} >$ and dropping the last element in s_2 results in $< \{B\} \{C\}$. Therefore, s_1 and s_2 can get joined to a candidate 4-sequence $s_3 =< \{A, B\} \{C\} \{E\} >$. Note that here $\{E\}$ will not merge into the last itemset in the s_1, because it is a separate element in s_2.

Pruning Phase If a sequence has any infrequent subsequence, this phase must delete this candidate sequence. For example, in Table 1, candidate $< \{A, B\} \{C\} \{E\} >$ gets pruned because subsequence $< \{B\} \{C\} \{E\} >$ is not a frequent 3-sequence.

2.2.2 Matching and Counting

The matching-and-counting stage will count how many times an input matches a sequence candidate. The occurrence of each candidate pattern is recorded

Table 1: Example of candidate generation

Frequent 3-sequences	Candidate 4-sequences	
	Joined	Pruned
$< \{B\}\{C\}\{E\} >$	$< \{A,B\}\{C\}\{E\} >$	$< \{A,B\}\{C,D\} >$
$< \{A,B\}\{C\} >$	$< \{A,B\}\{C,D\} >$	
$< \{B\}\{C,D\} >$		
$< \{A\}\{C,D\} >$		
$< \{A,B\}\{D\} >$		

and compared with the minimum support number. The matching-and-counting stage is the performance bottleneck for *GSP*, but it exposes massive parallelism. The high density of on-chip state elements and fine-granularity communication found on the AP allows many candidate sequences (patterns) to be matched in parallel, and make AP a promising hardware performance booster for matching and counting operations of *GSP*. For this reason, the *GSP* algorithm becomes a natural choice for mapping SPM onto the AP. In the rest of this paper, we will show how to utilize the AP to speed up the matching-and-counting stage of *GSP* (Section 5) and how this solution compares with other parallel or accelerator implementations of SPM (Section 8). For comparison purpose, we also propose OpenMP and CUDA implementations for multicore CPU and GPU to speed up the matching and counting of *GSP*.

3 Disjunctive Rule Mining

3.1 Introduction to DRM

Disjunctive Rule Mining (DRM) is derived from frequent set mining [5]. In the DRM problem, we define $I = i_1, i_2, ..., i_m$ as a set of interesting items. Let $T = t_1, t_2, ..., t_n$ be a dataset of transactions, where each transaction t_j is a subset of I. Define $x_j = \{i_{s1}, i_{s2}, ..., i_{sl}\}$ to be a set of items in I, called an itemset in traditional frequent set mining. Define $y_j = \{d_{s1}, d_{s2}, ..., d_{sl}\}$ to be a set of disjunctive items (d-items), called a disjunctive rule in DRM, where each d-item $d_{sj} =< i_{m1}, i_{m2}...i_{mw} >$ is a set of alternative items. w is the max number of alternative items could appear in one d-item. Duplicate items are not allowed in a disjunctive rule.

The d-rule with k d-items is called k-d-rule. A d-item d_r is said to cover an item i_s *iff* $i_s \in d_r$. A transaction t_p is said to cover the d-rule y_q *iff* each d-item y_q covers one item of t_p. The support of y_q, $Sup(y_q)$, is the number of transactions that cover it. A d-rule is known as frequent *iff* its support is greater than a given threshold value called minimum support, *minsup*. The goal of disjunctive rule mining is to find out all d-rules which supports are greater than *minsup*.

3.2 Apriori and downward-closure

In DRM, the downward-closure means all the subsets of a frequent d-rule are also frequent and thus for an infrequent d-rule, all its supersets must also be infrequent. The downward-closure is valid when considering the relations among d-rules. On the contrary, an upward-closure is valid when considering the items within a d-item. In another word, for a frequent d-rule y_q, a new d-rule y'_q with one more alternative item in any d-item must be frequent.

3.2.1 Algorithm framework

Given both downward-closure and upward-closure properties described above, we implement an *Apriori* like algorithm. The mining begins at 2-d-rules and increases the size of d-rules by one with each outer iteration. In each inner iteration, infrequent d-rules are picked up to generate new d-rules by adding one alternative item to any possible d-item. The algorithm will take the following steps:

1. Candidate Generation:
   (a) d-rule size-increase iteration: generating candidates of frequent (k+1)-d-rules from the known frequent k-d-rules
   (b) d-item size-increase iteration: generating candidates of frequent k-d-rules from the known *infrequent* k-d-rules
2. Matching and Counting: matching candidate d-rules and counting supports

3.2.2 Matching and Counting

The matching-and-counting stage counts how many times the input matches a d-rule. The occurrence of each candidate d-rule is recorded and compared with the minimum support number. When considering both the *OR* relations among items of each d-item and the *AND* relations among d-items, the CPU implementation takes a longer time to match a d-rule than a itemset. In addition to the high capacity of state units on the AP chips, which allows massively parallel matching and counting on a large number of d-rules, the sub-set matching nature of AP state units makes it no extra cost in calculating *OR* relations among items of each d-item. Therefore, in practical cases, the runtime of CPU-implemented matching and counting will increase along the complexity of a d-rule (d-rule size and the total number of alternative items). In contrast, the AP matching-and-counting time will keep constant and is two orders of magnitude faster than the CPU version (see Section 9).

4 Automata Processor

4.1 Architecture

The AP chip has three types of functional elements - the state transition element (STE), counters, and Boolean elements [9]. The STE is the central feature of the AP chip and is the element with the highest population density. An STE holds a *subset* of 8-bit symbols via a DRAM column and represents an NFA state, activated or deactivated, via an one-bit register along with the matching rule for that state. The AP architecture achieves fine-granularity (bit-wise) parallelism at the scale of entire row of the memory, every cycle. The high parallelism is achieved in two places:

1. Activating the row in each subarray that corresponds to the matching rules for the entire system in response to the input symbol
2. Operating directly on the row buffer to complete the process of checking the input symbol against all possible transitions, by doing the *OR* in each column of the matching rule with the active vector

Therefore, when any symbol of the symbol set compiled on one activated STE is seen, the STE "matches" and produces a high signal. In another word, the *OR* operations are implicitly applied among the symbols complied on an STE. This feature, as a part of bit-wise parallelism, could be utilized for efficient disjunctive rule mining. The AP uses a homogeneous NFA representation [9] for a more natural match to the hardware operation. In terms of Flynn's taxonomy, the AP is therefore a very unusual multiple-instruction, single-data (MISD) architecture: each state (column) holds unique responses (instructions) to potential inputs, and they all respond in parallel to each input. Most other commercial architectures are von Neumann architectures, e.g. single CPU cores (SISD), multicore or multiprocessors (MIMD), and GPUs (SIMD).

The counter element counts the occurrence of a pattern described by the NFA connected to it and activates other elements or reports when a given threshold is reached. One counter can count up to $2^{12}-1$. Two or more counters can be daisy-chained to handle larger threshold. Counter elements are a scarce resource of the AP chip, and therefore become an important limiting factor for the capacity of the SPM automaton proposed in this work.

The Boolean element is programmable and could be configured to one of nine different Boolean gates including *AND*, *OR*, *NOT*, *SOP* (sum of product) and *POS* (product of sum). The *AND* gate supports up to seven fan-in connections. However, due to the available routing resource, it is hard to reach this limitation in practical cases.

The current generation AP-D480 boards use AP chips built on 50nm DRAM technology, running at an input symbol (8-bit) rate of 133 MHz. Each D480 chip has 192 blocks, with 256 STEs, 4 counters and 12 Boolean elements per block [9]. We assume an AP board with 32 AP chips, so that all AP chips process input data stream in parallel. The projected power consumption of a 32-chip AP board is about 155W.

4.2 Input and output

The AP takes input streams of 8-bit symbols. Any STE can be configured to accept the first symbol in the stream (called start-of-data mode, small "1" in the left-upper corner of STE in the following automaton illustrations), to accept every symbol in the input stream (called all-input mode, small "∞" in the left-upper corner of STE in the following illustrations) or to accept a symbol only upon activation.

Any type of element on the AP chip can be configured as a reporting element; one reporting element generates a one-bit signal when it matches the input symbol. If any reporting element reports on a particular cycle, the chip will generate an output vector which contains 1s in positions corresponding to the elements that report and 0s for reporting elements that do not report. Too-frequent outputs will cause AP stalls; therefore, minimizing output vectors is an important consideration for performance optimization.

4.3 Programming and configuration

The AP SDK provides the Automata Network Markup Language (ANML), an XML-like language for describing automata networks, as well as C, Java and Python bindings to describe automata networks, create input streams, parse output and manage computational tasks on the AP board. A "macro" is a container of automata for encapsulating a given functionality, similar to a function or subroutine in common programming languages.

Deploying automata onto the AP fabric involves two stages: placement-and-routing compilation (*PRC*) and *loading* (*configuration*) [2]. In the *PRC* stage, the AP compiler deduces the best element layout and generates a binary version of the automata network. In the cases of large number of topologically identical automata, macros or templates can be precompiled in *PRC* stage and composed later [18]. This shortens *PRC* time, because only a small automata network within a macro needs to be processed, and then the board can be tiled with as many of these macros as fit.

A pre-compiled automata only needs the *loading* stage. The *loading* stage, which needs about 50 milliseconds for a whole AP board [18], includes two steps: routing configuration / reconfiguration that programs the connections, and the symbol set configuration/reconfiguration that writes the matching rules for the STEs. The changing of STE rules only involves the second step of *loading*, which takes 45 milliseconds for a whole AP board. The feature of fast partial reconfiguration play a key role in a successful AP implementation of SPM: the fast symbol replacement helps to deal with the case that the total set of candidate patterns exceeds the AP board capacity; the quick routing reconfiguration enables a fast switch from k to $k+1$ level in a multiple-pass algorithm like GSP for sequence mining.

5 Mapping SPM onto the AP

The general framework for AP-accelerated hierarchical pattern mining is to generate candidate patterns on the CPU and utilize the high parallelism of the AP chips to speedup performance bottleneck of matching-and-counting steps. For SPM, we adopt the *GSP* algorithm (discussed in Sec. 2.2), a variant of the *Apriori* algorithm for SPM, to generate candidates for sequential patterns.

5.1 Automaton for Matching and Counting

The hierarchical patterns in SPM, sequences of itemsets, are more complex than strings or individual itemsets as studied in previous works [18, 22]. Within itemsets of a sequence, items of interest may be discontinuous, i.e., we may only be interested in some frequent subsets of an itemset [22]. Furthermore, one input sequence may have irrelevant itemsets in between interesting itemsets. The matching part of the automaton for SPM should identify the interesting itemsets as well as the order among the itemsets. In summary, the automaton design needs to deal with all possible continuous and discontinuous situations for both items and itemsets, and keep the order among itemsets in the same time. There is no previous work that has proposed any automaton design for hierarchical pattern matching. Furthermore, in order to maximize benefit from the high parallelism of NFAs, and the AP in particular, an appropriate automaton structure must be as compact as possible, to maximize the number of such structures that can be accommodated in a single pass.

5.1.1 Flattening the Hierarchy of Sequential Patterns

To match sequences of itemsets, we first convert sets into strings with a pre-defined order. Then we introduce a delimiter for itemsets to bound and connect these strings (converted from itemsets) within a sequential pattern. The sequence of strings is also a string. Keeping this observation in mind, the hierarchy of a sequence of itemsets is therefore flattened to a discontinuous sequence-matching problem. This is the key innovation of proposed automaton design for SPM.

Figure 1 shows the automaton design for sequential pattern matching and counting. In the examples shown here, the items are coded as digital numbers in the range from 0 to 252, with the numbers 255, 254, 253 reserved as the data-ending reporting symbol, sequence delimiter, and itemset delimiter, respectively. Other choices of these three special symbols also work well under the proposed algorithm framework. In the case of more than 253 frequent items, two consecutive STEs are used to represent an item and support up to 64,009 (253×253) frequent items, which is sufficient in all the datasets we examine; because the AP native symbol size is 8 bits, this will require two clock cycles to process each 16-bit symbol. Even larger symbol alphabets are possible by longer consecutive sequences of STEs. In Figure 1, the counting and reporting component is shown below the (orange) dotted line. The I/O optimization strategy proposed in [22] is adopted by delaying all reports from frequent patterns to the last cycle.

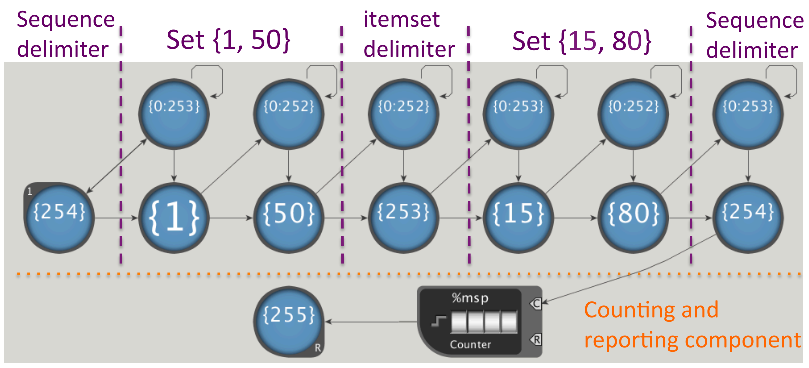
The STEs for matching sequential patterns are shown above the orange dotted line. One matching NFA is bounded by a starting sequence delimiter for starting a new sequence and an ending sequence delimiter (the same symbol) for activating the counting-and-reporting component. In contrast to the set-matching NFAs proposed in [22], the NFA for SPM is divided into several itemsets, demarcated by the itemset delimiters. Each NFA has two rows of

STEs. The bottom row is for the actual symbols in a candidate sequential pattern. The STEs in the top row, called "position holders", help to deal with the discontinuous situations (within itemsets or between itemsets). Each "position holder" has a self-activation connection and matches all valid symbols (excluding the delimiters). As long as the input symbol stays in range, the "position holder" will stay activated and keep activating the next STE in the bottom row. The key idea to implement *hierarchical pattern* matching with the flattened automaton design is to define two types of "position holder": "itemset position holder" and "item position holder". In the case of sequential pattern, the first "position holder" in each itemset should be an itemset position holder, $0 : 253$. It will stay activated before the end of a sequence and handle discontinuous itemsets within that sequence. The other "position holders" are "item position holders", $0 : 252$, which only hold the position within an input itemset. In the example shown in Figure 1a, any other itemsets except a superset of $\{1, 50\}$, will not reach the itemset delimiter. After a superset of $\{1, 50\}$ is seen, the "position holder" above STE "15" will hold the position (activate itself) until the end of the same input sequence. Namely, after a superset of $\{1, 50\}$ is seen, the itemsets other than the superset of $\{15, 80\}$ are ignored before a superset of $\{15, 80\}$ appears in the same input sequence. Note that more sophisticated hierarchical patterns, such as sequences of sequences or patterns of more than a two-level hierarchy, can be implemented using the same idea.

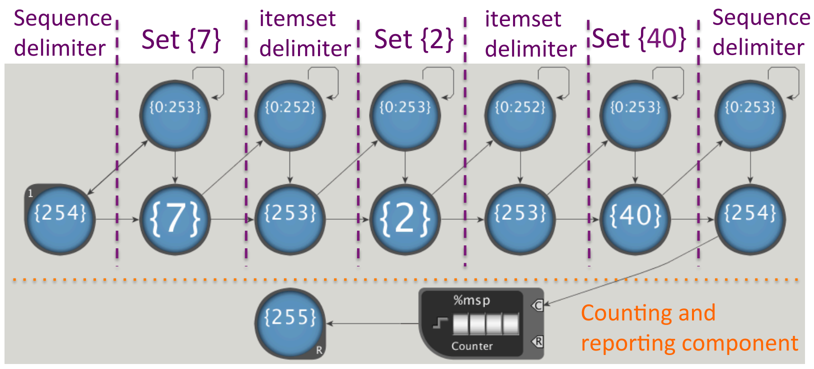
The only difference between an "item position holder" and an "itemset position holder" are their symbol set. One important advantage of the flattened automaton design is that one such automaton structure can deal with all situations of the same encoded pattern length (the encoded pattern length includes the itemset delimiters). This feature greatly reduces the design space of sequential pattern matching automata. For example, the automaton structure shown in Figure 1 can deal with all these cases: $< \{a, b, c, d, e\} >$, $< \{a\}\{b, c, d\} >$, $< \{a, b\}\{c, d\} >$, $< \{a, b, c\}\{d\} >$, $< \{a\}\{b\}\{c\} >$. We define the actual item IDs in a sequential pattern without counting delimiters as "effective items" and define the pattern that considers the itemset delimiters "encoded pattern". In this step, the automaton design space for a given length of "encoded pattern" is reduced to one. If we regard one item node in the bottom line and the position holder above it as a super node, the matching will proceed from the left to the right linearly. Therefore, we call the strategy of flatting hierarchical patterns into string patterns as *linear design* strategy.

5.1.2 Multiple-entry NFAs

In each *GSP* level, there could be 0 to $k - 1$ delimiters in actual patterns, and the encoded pattern lengths of level k can vary from k (a sequence consisting of a single itemset) to $k + k - 1$ (all the itemsets only have a single item, so we have k-1 itemset delimiters). Because candidate sequences are generated at runtime, the number of patterns to be checked for a given encoded length is not known before runtime. We need a further step to reduce the automaton



(a) Automaton for sequence $< \{1, 50\}, \{15, 80\} >$



(b) Automaton for sequence $< \{7\}, \{2\}, \{40\} >$

Fig. 1: Examples of automaton design for sequential pattern matching and counting. Blue circles and black boxes are STEs and counters, respectively. The numbers on an STE represent the symbol set that STE can match. "0:252" means any item ID in the range of ASCII 0-252. Symbols "255", "254", "253" are reserved as the input ending, sequence delimiter and itemset delimiter.

design space of the candidates for each *GSP* iteration to one single template, so that the place and routing can be done before runtime.

To solve this problem, we adopt the idea of multiple-entry NFAs for variable-size itemsets (ME-NFA-VSI) proposed by Wang et al. [22]. Figure 2 shows an example of the ME-NFA-VSI structure that can handle all possible cases of sequences of effective length 3. Figure 2a shows the ANML macro of this ME-NFA-VSI structure, leaving some parameters to be assigned for a specific sequence. %TD and %NTD are the sequence delimiter and its complement and are assigned to "254" and "0-253". %ER is the ending and reporting symbol of the input stream and is assigned to "255" in this paper. %e00 - %e02 are symbols for three entries. Only one entry is enabled for a given sequence. %i00 - %i04 are individual symbols of items and itemset delimiter. %p00 - %p04 are the corresponding "position holders".

Table 2: **Number of macros that fit into one block with 8-bit encoding**

	$k <= 10$	$10 < k <= 20$	$20 < k <= 40$
$sup < 4096$	4	2	1
$sup >= 4096$	2	2	1

Table 3: **Number of macros that fit into one block with 16-bit encoding**

	$k <= 5$	$5 < k <= 10$	$10 < k <= 20$
$sup < 4096$	4	2	1
$sup >= 4096$	2	2	1

192 AP blocks per D480 AP chip; 6144 blocks per 32-chip AP board.

To match and count a sequence of three itemsets (two itemset delimiters are introduced), the first entry is enabled by "254", the sequence delimiter, and the other two entries are blocked by "255" (Figure 2d). The sequence matching will start at the left most item symbol, and handle the cases of $< \{X\}\{Y\}\{Z\} >$. Similarly, this structure can be configured to handle other situations by enabling a different entry point (Figure 2c and 2d).

5.1.3 Macro Selection and Capacity

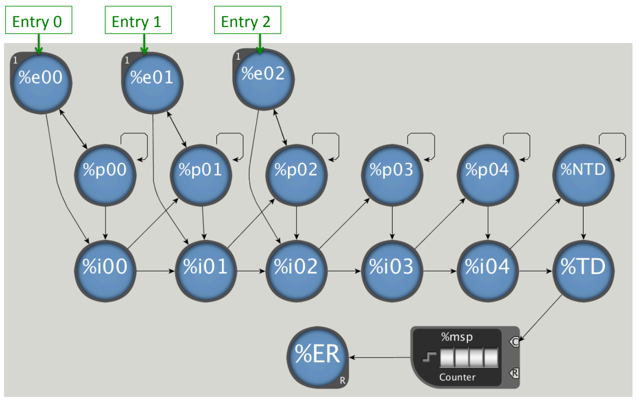
The flattening strategy and multiple-entry strategy introduced in Sec 5.1.1 and 5.1.2 shrink the automata design space (the number of different automata design) of a sequential pattern of length k from 2^{k-1} patterns to a single pattern template, which makes it possible to pre-compile a library of automata for each level k and load the appropriate one to the AP chip at runtime. In each level k, the different encoding schemes, 8-bit and 16-bit, and the support threshold (greater than 4095 or not) lead to four different automaton designs. To count a support number larger than 4095, two counters should be daisy-chained to behave as a larger counter. In this case, counters are more likely a limiting factor of the capacity.

The actual capacity of a macro may be limited by STEs, counters, or routing resources of the AP chip. We have developed a library of macro structures described in Section 5.1.2 and compiled all these macros with the newest AP compiler (v1.7.17). Table 2 and 3 show the actual capacities of macros for the different encoding schemes, support number and level range. Note that across all of our datasets, we never encountered a case of k larger than 20.

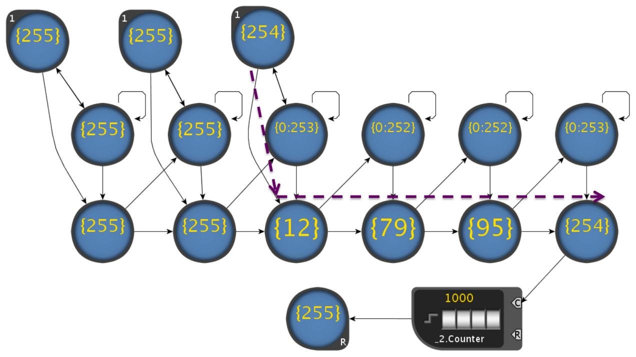
5.2 Program Infrastructure

Figure 3 shows the complete workflow of the AP-accelerated SPM proposed in this paper. The data pre-processing step creates a data stream from the input dataset and makes the data stream compatible with the AP interface. Pre-processing consists of the following steps:

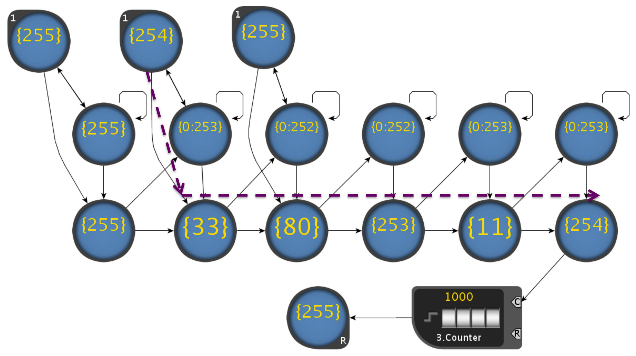
1. Filter out infrequent items from input sequences



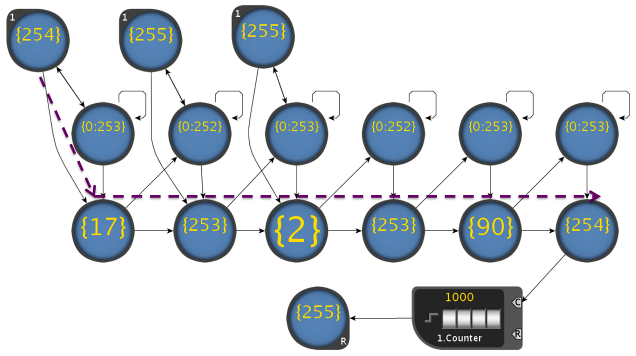
(a) AP macro for sequential pattern



(b) Automaton for sequence $< \{12, 79, 95\} >$



(c) Automaton for sequence $< \{33, 80\}\{11\} >$



(d) Automaton for sequence $< \{17\}\{2\}\{90\} >$

Fig. 2: A small example of multiple-entry NFA for all possible sequences of effective size 3. (a) is the macro of this ME-NFA-VSI with parameters.

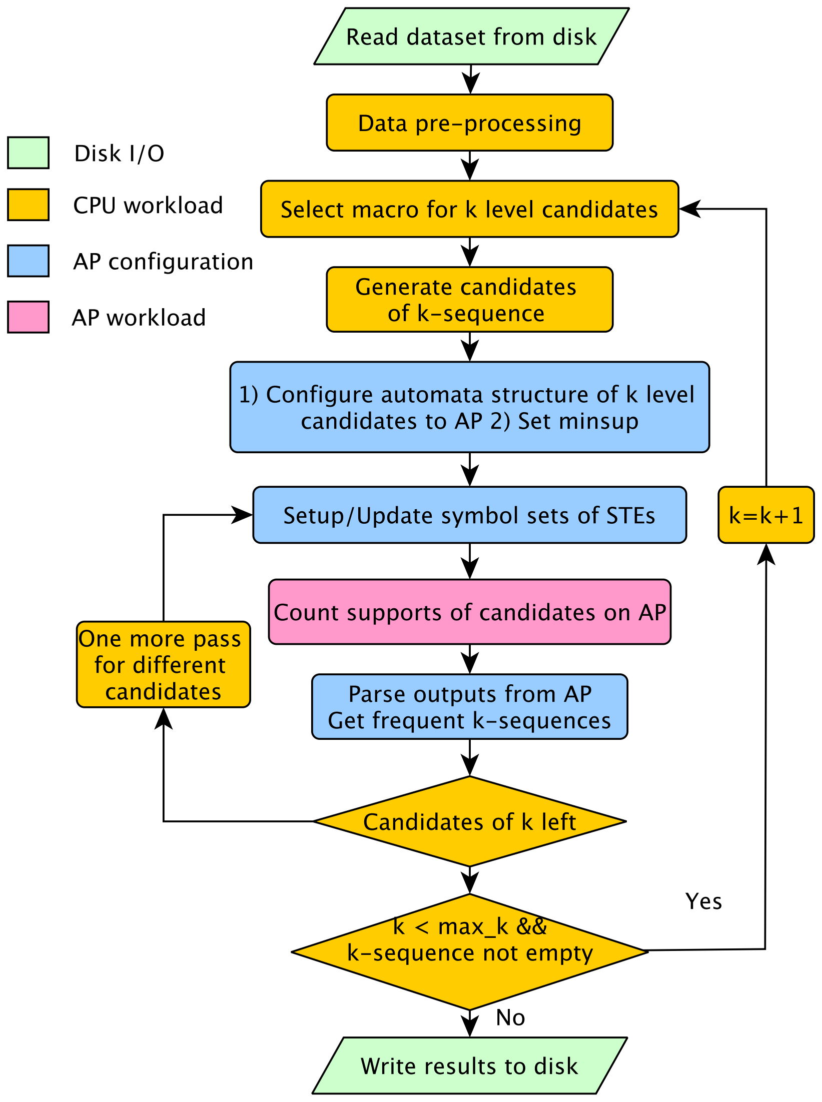

Read dataset from disk
Data pre-processing
Disk I/O
CPU workload
AP configuration
AP workload
Select macro for k level candidates
Generate candidates of k-sequence
1) Configure automata structure of k level candidates to AP 2) Set minsup
Setup/Update symbol sets of STEs
k=k+1
Count supports of candidates on AP
One more pass for different candidates
Parse outputs from AP Get frequent k-sequences
Candidates of k left
Yes
k < max_k && k-sequence not empty
No
Write results to disk

Fig. 3: The workflow of AP-accelerated SPM

2. Recode items into 8-bit or 16-bit symbols
3. Recode input sequences
4. Sort items within each itemset of input sequences, and connect itemsets and sequences

Step 1 helps to avoid unnecessary computing on infrequent items and reduces the dictionary size of items. Depending on the number of frequent items, the items can be encoded by 8-bit ($freq\_item\# < 254$) or 16-bit symbols ($254 <= freq\_item\# <= 64009$) in step 2. Different encoding schemes lead to different automaton designs and automaton capacities. Step 3 removes infrequent items from the input sequences, recodes items, and removes very short transactions (fewer than two items). Step 4 sorts items in each itemset (in any given order) to fit the automaton design described in Section 5.1. The data pre-processing is only carried out once per workflow.

Each iteration of the outer loop shown in Figure 3 explores all frequent k-sequences from the candidates generated from $(k-1)$-sequences. In the beginning of a new level, an appropriate precompiled template macro of automaton structure for sequential patterns is selected according to k, encoding scheme (8-bit or 16-bit), and the minimum support (see Section 5.1.3), and is configured onto the AP board with many instances spread out the whole board. The candidates are generated on the CPU and are filled into the instances of the selected automaton template macro. The input data formulated in pre-processing is then streamed into the AP board for matching and counting.

6 Mapping DRM onto the AP

We adopt the same flexible framework as described in Section 5 to develop a CPU-AP algorithm for disjunctive rule mining. Similarly, we generate candidates of disjunctive rules by an *Apriori* based algorithm. The performance bottleneck of this algorithm is still the matching-and-counting operation. We will show how to accelerate this bottleneck by using the AP in this section.

6.1 Program Infrastructure

Figure 4 shows the complete workflow of the AP-accelerated DRM proposed in this paper. The data pre-processing step is similar to that for frequent itemset mining [22] and SPM (Section 5.2). However the sorting operation is no longer needed. The outer loop is the d-rule size-increase iteration which generates candidates of frequent (k+1)-d-rules from known frequent k-d-rules by a d-item without any alternative. The outer loop stops when no more frequent k-d-rule is found. The inner loop is the d-item expanding iteration which generates candidates of frequent k-d-rules from known *infrequent* k-d-rules by adding one alternative item into any possible d-item each time. The outer loop stops when no more infrequent k-d-rule is found. Although the algorithm to generate d-rule candidates could be improved, the delicate candidate generation algorithm is out of the scope of this paper. We focus on accelerating matching and counting d-rules by using the AP.

6.2 Automaton for Matching and Counting

Similar to sequential patterns, a d-rule has a two-level hierarchy. In the lower level, the alternative items in one d-item follow *OR* relations, that is, any items in this d-item seen in the input stream will cause a match of this d-item.

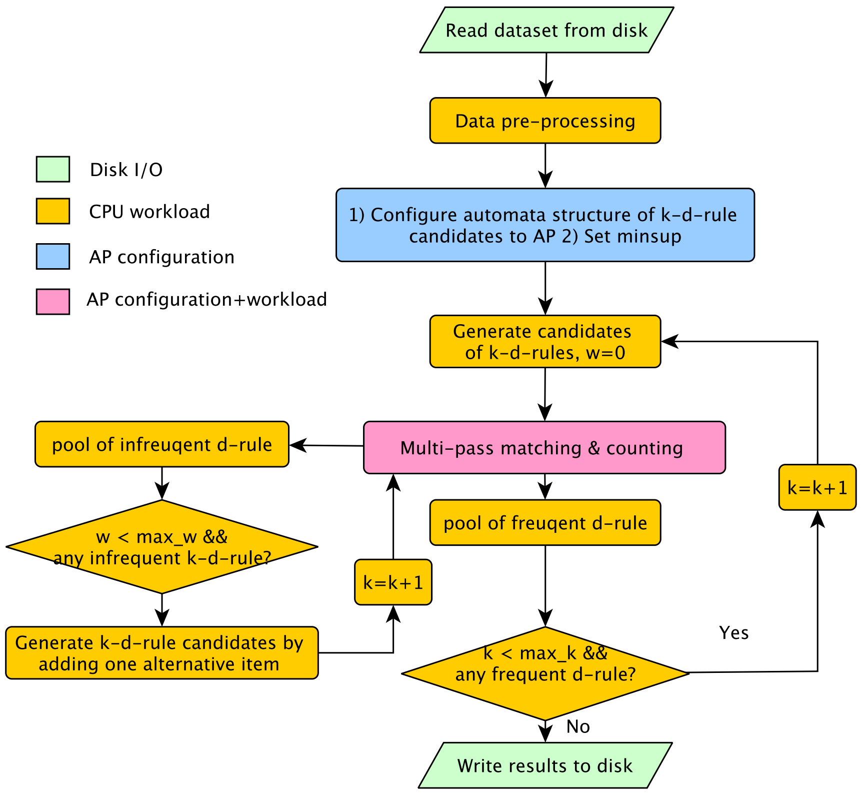

Disk I/O
CPU workload
AP configuration
AP configuration+workload
Read dataset from disk
Data pre-processing
1) Configure automata structure of k-d-rule candidates to AP 2) Set minsup
Generate candidates of k-d-rules, w=0
Multi-pass matching & counting
pool of infreuqent d-rule
w < max_w && any infrequent k-d-rule?
Generate k-d-rule candidates by adding one alternative item
k=k+1
pool of freuqent d-rule
k < max_k && any frequent d-rule?
Yes
k=k+1
No
Write results to disk

Fig. 4: The workflow of AP-accelerated DRM.

In the higher level, the d-items within a d-rule obey *AND* relations, and the d-rule matches a transaction when every its d-item matches an item in this transaction. The AP STE has a feature to hold a *symbol set* of 8-bit symbols instead of a single symbol, and any symbol of the *symbol set* seen in the input will cause a match of this STE. This feature is a part of the AP's bit-wise parallelism capability, and could be naturally utilized to accommodate a d-item on an STE without any extra cost.

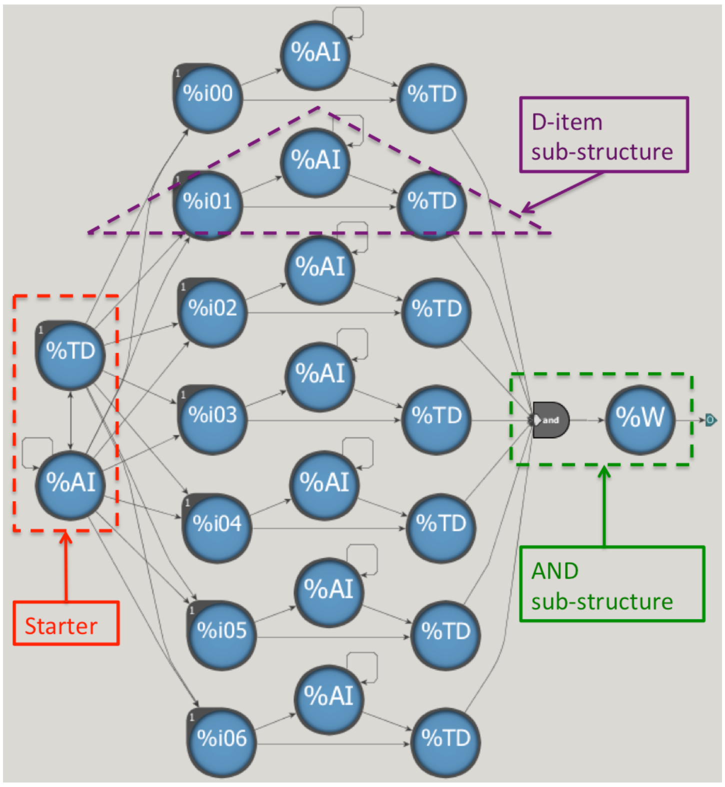
The *linear design* strategy, proposed in Section 5 for SPM, which flattens a hierarchical pattern to a plain string by pre-sorting items within each itemset and connecting transactions with a delimiter symbol, is no longer applicable to a disjunctive mining problem. This is because a pre-defined item order of input transactions will cause false negatives when two non-conjunctive items appear in one d-item. For example, a dataset has only three items, $1, 2, 3$. The *linear design* strategy only works well on two d-rules: $1or2, 3$ and $1, 2or3$ when the pre-defined order lets 2 to be next to both 1 and 3. There are only two options: $< 1, 2, 3 >$ or $< 3, 2, 1 >$. Given an input transaction $< 1, 2, 3 >$, a new d-rule $1or3, 2$ will cause a false negative on transaction $2, 3$. No pre-defined order works well with $1or2, 3$, $1, 2or3$ and $1or3, 2$ on all possible input transactions.

6.2.1 Boolean based d-rule representation

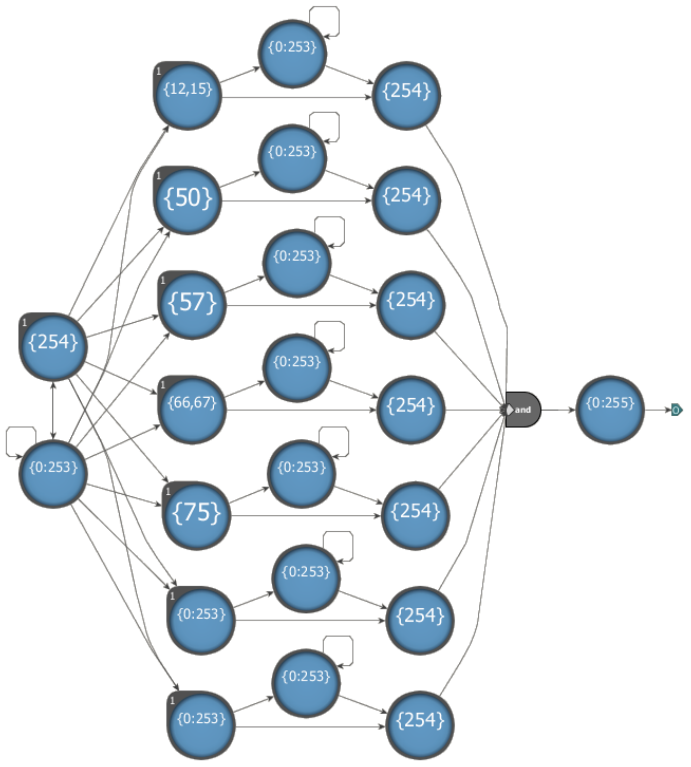
To solve the problem described above, we introduce a novel automaton design strategy, *reduction design*, which works for disjunctive rule mining without a need of pre-sorting items in transactions. The key idea of this automaton design is to take advantage of bit-wise parallelism of an STE to represent the *OR* relation of alternative items in a d-item and utilize the on-chip Boolean elements to calculate *AND* relation among d-items of a d-rule.

Figure 5a shows the automaton design for d-rule matching. The d-rule matching automaton has three major components: starter, d-item sub-structure and *AND* sub-structure. A starter has two STEs, the "%TD" STE represents the beginning of a new input transaction and activates all d-item sub-structures when matches a transaction delimiter; "%AI" STE matches any valid item and keeps activating all d-item sub-structures before the end of the current transaction. The left STE of d-item sub-structure holds the set of items in one d-item. The middle "%AI" STE holds the activated status of this d-item sub-structure until the end of the current transaction. The right "%TD" STE waits until the end of the current transaction to finish d-item matching. All the outputs of d-item sub-structure connect to the *AND* unit of an *AND* sub-structure. An *AND* unit of the current AP generation supports up to 7 fan-in connections. Therefore, we can have up to 7 d-item sub-structures in this d-rule matching automaton. This automaton structure could be pre-compiled and loaded in the runtime. When fewer d-items are needed, the rest of the d-item sub-structures can be filled with "%AI", any valid item, to feed "true" to *AND* gate. The "%W" STE wildcard, matching any 8-bit symbol, simply separates the *AND* gate and counter or logic gate connected to this d-rule matching structure by paying one cycle of delay in the end of the whole input stream. However, without the wildcard STE, the Boolean and counter elements will connect to each other, which causes the clock rate of the whole AP chip to reduce to half the normal speed. In summary, the parallel d-item sub-structures seek d-items independently, and their results are collected by a "reduction" of *AND* to obtain the final output. This automaton design strategy for pattern matching is called *reduction design.*

Figure 5a shows an example of d-rule $\{12/15, 50, 57, 66/67, 75\}$. In the examples shown in this paper, the items are coded as digital numbers in the range from 0 to 253, with the numbers 255, 254 reserved as the data-ending reporting symbol and the itemset delimiter, respectively. That is, %AI = 0-253, %TD = 254, %ER = 255 and %W = 0-255. Therefore, the two unused d-items are filled with 0-253.



(a) d-rule matching sub-structure with parameters



(b) d-rule matching example {12/15, 50, 57, 66/67, 75}

Fig. 5: d-rule matching automaton

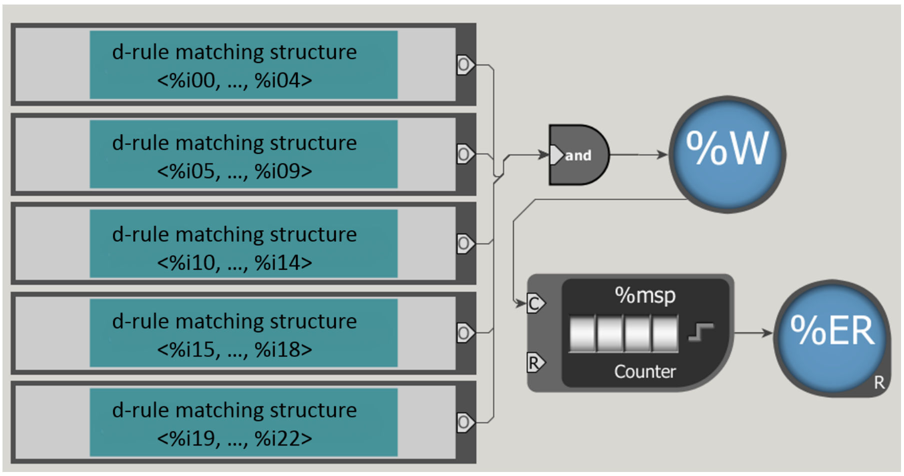

d-rule matching structure
<%i00, ..., %i04>
d-rule matching structure
<%i05, ..., %i09>
d-rule matching structure
<%i10, ..., %i14>
d-rule matching structure
<%i15, ..., %i18>
d-rule matching structure
<%i19, ..., %i22>
and
%W
%msp
Counter
C
R
%ER
R

Fig. 6: A d-rule matching-and-counting macro.

6.2.2 d-rule with larger size

In many cases, a d-rule has more than 7 d-items. We propose to use another level of *AND* (reduction) sub-structure to connect several d-rule matching automata to extend to larger d-rules, as shown in Figure 6. In theory, seven d-rule matching automata could be connected together to extend to 49 d-items. However, there is a trade-off between the total capacity of d-rule matching-and-counting macros and the size of d-rules. To fit two d-rule matching-and-counting automata in one AP block, we need to reduce the size of each d-rule automaton. An optimal configuration, shown in Figure 6 with automata sizes of 5, 5, 5, 4, 4, representing a d-rule of size 23, can fit two macros in one AP block. Further automaton optimization and advanced placement and routing optimization may allow larger d-rule with the same capacity, however 23 d-items are more than enough in practical cases of FIM and DRM we have observed. This automaton design can also be utilized for simple FIM problem. However when compared with the *linear design* strategy that supports an itemset with up to 40 items, an interesting trade-off between pattern size (number of items) and bit-wise parallelism (holding multiple symbols in one STE) is shown here.

6.2.3 Macro Selection and Capacity

Similar to SPM, one major factor in the AP performance is macro capacity, that is, how many macro instances can fit onto an AP board, which will directly affect the actual parallelism of the AP device. The different ranges of d-rule size and the max support number will lead to different automaton designs and therefore different capacities. We have developed an automaton library of DRM macro structures described in Section 6.2.1 and compiled all different

Table 4: **Number of macros that fit into one block with 8-bit encoding**

	$k <= 7$	$7 < k <= 23$	$23 < k <= 33$
$sup < 4096$	4	2	1
	$k <= 23$		$23 < k <= 33$
$sup >= 4096$	2		1

macros with the newest AP compiler (v1.7.17). Table 4 shows the capacity of DRM macros that can fit into one block for different d-rule sizes and support thresholds. In runtime, the appropriate macro should be selected and loaded onto the AP board before AP processing.

To count a support number larger than 4095, two counters should be daisy-chained to behave as a larger counter. Unlike the *linear design* for SPM, where the counter resource is a major limiting factor of the capacity, routing and STE capacity become important limitations in *reduction design* because of the routing hotspots of Boolean elements and more STE usage of a d-item sub-structure than that of an item representation in *linear design*. In order to support a larger size of symbol sets (e.g. 12 bits or 2^{12} different items) without changing the design, we need a new AP architecture, capable of handling larger symbol-set size. Alternatively, by changing the design and considering for different combinations of disjunctive items, current generation AP can also handle 16-bit encoding. In this paper, we only test the automaton design for 8-bit encoding.

7 Testing platform and datasets

The performance of our CPU-AP implementations of SPM and DRM are evaluated using CPU timers (CPU sequential parts), stated configuration latencies, and an AP simulator in the AP SDK [2, 16], assuming a 32-chip Micron D480 AP board. Because the AP advances by one 8-bit symbol every clock cycle, the number of patterns that can be placed into the board, and the number of candidates that must be checked in each stage, determines how many passes through the input are required, which allows a simple calculation to determine the total time on the AP (see hardware parameters in Section 4).

7.1 Testing Platform and Parameters

All of the above implementations are tested using the following hardware:
- CPU: Intel CPU i7-5820K (6 physical cores, 3.30GHz)
- Memory: 32GB, 1.333GHz
- GPU: Nvidia Kepler K40C, 706 MHz clock, 2888 CUDA cores, 12GB global memory
- AP: D480 board, 133 MHz clock, 32 AP chips (simulation)

Table 5: **Datasets for sequential pattern mining**

Name	#sequences	Aver. Len.	#item	Size (MB)
BMS1	59601	2.42	497	1.5
BMS2	77512	4.62	3340	3.5
Kosarak	69998	16.95	41270	4.0
Bible	36369	17.84	13905	5.4
Leviathan	5834	33.8	9025	1.3
FIFA	20450	34.74	2990	4.8

Aver. Len. = Average number of items per sequence.

7.2 Datasets

7.2.1 Datasets for SPM

Six public real-world datasets for sequential pattern mining available on the *spmf* [11] website are tested. The details of these datasets are shown in Table 5. The *BMS*(1&2), *Kosarak*, and *FIFA* are clickstream data from an e-commerce site, Hungarian news portal and the website of FIFA World Cup 98, respectively. The SPM studies on these four clickstream datasets are to discover the frequent click sequences of the corresponding applications. The *Bible* and *Leviathan* are sequence datasets generated from the Bible and the novel Leviathan (by Thomas Hobbes, 1651), considering each sentence as a sequence and each word as an item. The goal of SPM analysis on these two datasets is to discover the common word sequences in English (a natural language).

7.2.2 Datasets for DRM

One commonly-used real-world dataset, *Webdocs*, from the *Frequent Itemset Mining Dataset Repository* [1] and one real-world dataset generated by ourselves (ENWiki [22]) are tested (details are shown in Table 6). The *ENWiki* is the English Wikipedia downloaded in December 2014. We have removed all paragraphs containing non-roman characters and all MediaWiki markups. The resulting dataset contains about 1,461,281 articles, 11,507,383 sentences (defined as transactions) with 6,322,092 unique words. We construct a dictionary by ranking the words using their frequencies. Capital letters are all converted into lower case and numbers are replaced with the special "NUM" word. *Webdocs* is a collection of web html documents after filtering out html tags and most common words. In natural language processing, the idea of deducing some aspects of semantic meaning from patterns of word co-occurrence is becoming increasingly popular. The goal of disjunctive rule mining on these two datasets is to compute such co-occurred word clusters with alternative words.

8 Experimental Results for SPM

8.1 Comparison with Other Implementations

We compare the performance of the proposed AP-accelerated *GSP* (GSP-AP) versus the multi-threaded Java *GSP* implementation (GSP-JAVA) from *spmf* toolkit [11], as well as a highly optimized *GSP* single-core CPU C implementation (GSP-1C), a multicore implementation using OpenMP, (GSP-6C), and a GPU implementation (GSP-1G) of the *GSP* algorithm. We also compare the AP-accelerated *GSP* with Java multi-threaded implementations of *SPADE* and *PrefixSpan* [11]. Because GSP-1C is always faster than GSP-JAVA, we don't show the results of GSP-JAVA in this paper, but use it as a baseline to determine the feasible ranges of minimum support number.

For each benchmark, we compare the performance of the above implementations over a range of minimum support values. A lower minimum support number requires a larger search space (because more candidates survive to the next generation) and more memory usage. To finish all our experiments in a reasonable time, we select minimum support numbers that produce computation times of the GSP-JAVA in the range of 2 seconds to 2 hours. A relative minimum support number, defined as the ratio of a minimum support number to the transaction number, is adopted in the figures.

8.2 Multicore and GPU GSP

In multicore and GPU implementations of *GSP*, the most time-consuming step, the matching and counting, is parallelized using OpenMP and CUDA.
GSP-GPU: After filtering out the infrequent items, the whole dataset is transferred to the GPU global memory. Then, the algorithm iterates over two steps: (1) generating $(k+1)$-sequence candidates from the frequent k-sequences on CPU, and (2) identify the frequent $(k+1)$-sequences on GPU. In the CUDA kernel function, each thread is responsible for matching and counting one candidate in the input dataset. Once the matching-and-counting phase is done for all the candidates of $k+1$ level, the results are transferred back to the CPU for the next level. We do not consider pruning in the candidate generation step (neither in AP nor in GPU implementation) as it increases pre-processing time and decreases the overall performance. An array data structure is used to contain candidates and the input dataset for GPU and AP implementations to optimize the performance of candidate pattern generation.

Table 6: **Datasets for disjunctive rule mining**

Name	Trans#	Aver. Len.	Item#	Size (MB)
Webdocs	1692082	177.2	5267656	1434
ENWiki	11507383	70.3	6322092	2997.5

Aver. Len. – Average number of items per transaction.

GSP-multi-core: Workflow is the same as the GSP-CPU implementation except that the matching-and-counting step is parallelized using OpenMP. The CPU version uses a linked-list to accelerate the pruning and counting operations to achieve the best overall performance.

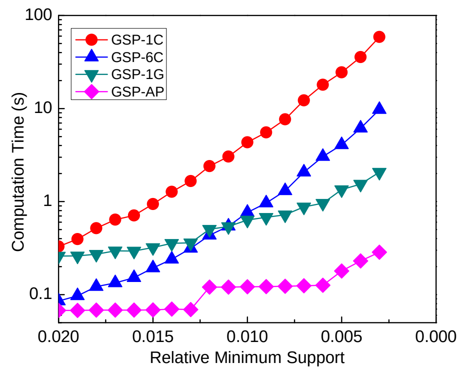
8.3 GSP-AP vs. Other GSP Implementations

Figure 7 shows the performance comparison among four different *GSP* implementations. As the minimum support number decreases, the computation time of each method increases, as a larger pattern search space is exposed. On average, the performance relationship among the four tested implementations follows this order: $GSP-1C < GSP-6C < GSP-1G < GSP-AP$. The multicore GSP-6C achieves about 3.7X-6X speedup over single-core version GSP-1C. The GPU version outperforms GSP-1C up to 63X. GSP-1G shows better performance than GSP-6C at large support numbers but loses at small ones. This indicates that more parallelism needs to be exposed for GPU implementation to compensate for the data transfer overhead between CPU and GPU. The proposed GSP-AP is the clear winner, with a max 430X (in the BMS2) speedup over single-core, up to 90X speedup over multicore, and 2-29X speedup over GPU.

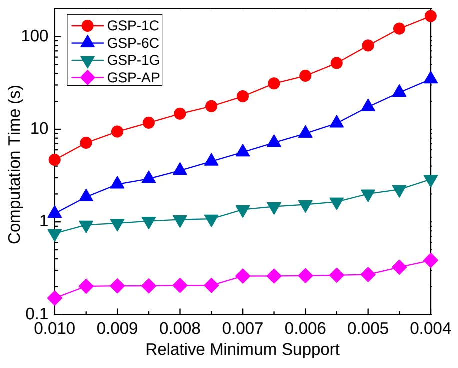
8.4 Timing Breakdown and Speedup Analysis

To better understand the performance shown in Figure 7, profiling results are shown in Figures 8 and 9. Focusing on the matching-and-counting stage, the multi-core and GPU versions achieve 5X and tens-X speedups over single-core CPU implementation, while the AP implementation achieves several hundreds to 1300 times speedups over the sequential matching and counting implementation. The smaller the minimum support, the more candidates are generated, and the larger the speedups achieved for both GPU and AP versions. On one hand, it shows the performance boost of massive complex-pattern matching achieved by the AP. On the other hand, Amdahl's law starts to take effect at small support numbers, with the percentage of time for matching and counting within the total execution time dropping, and the un-accelerated candidate-generation stage becoming dominant. This could be addressed by parallelizing candidate generation (see Section 8.5). Amdahl's law has even more severe impact on the AP version than on GPU implementation. FIFA is one typical example, where over 1300X speedup is achieved at 7.5% relative support, but the percentage of matching and counting drops to 3%.

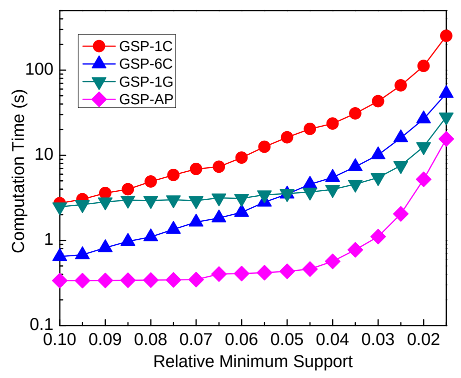
From Figures 8 and 9 we observe that configuration time dominates the total AP matching-and-counting time, 80%-90% of the AP time for all cases. Fortunately, the latency of symbol replacement could be significantly reduced in future generations of the AP, because symbol replacement is simply a series of DRAM writes, and this should be much faster. We hypothesize that the current times assume some conservative buffering. Reducing symbol replacement



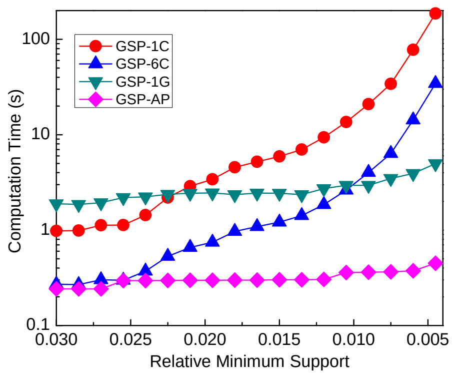
(a) BMS1



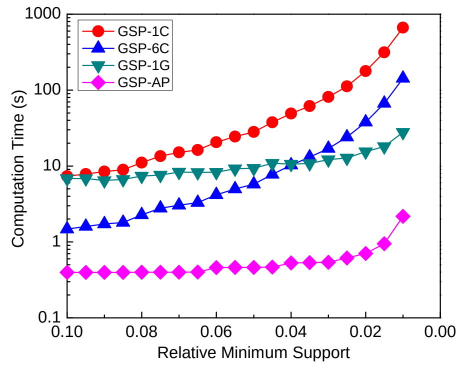
(b) BMS2



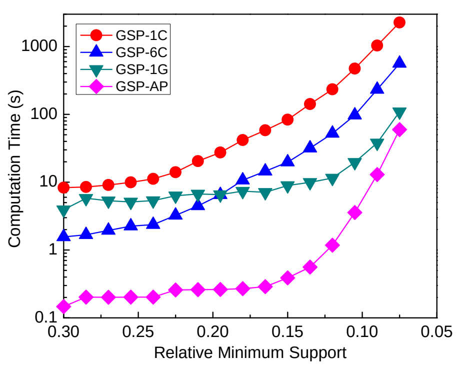
(c) Leviathan



(d) Kosarak

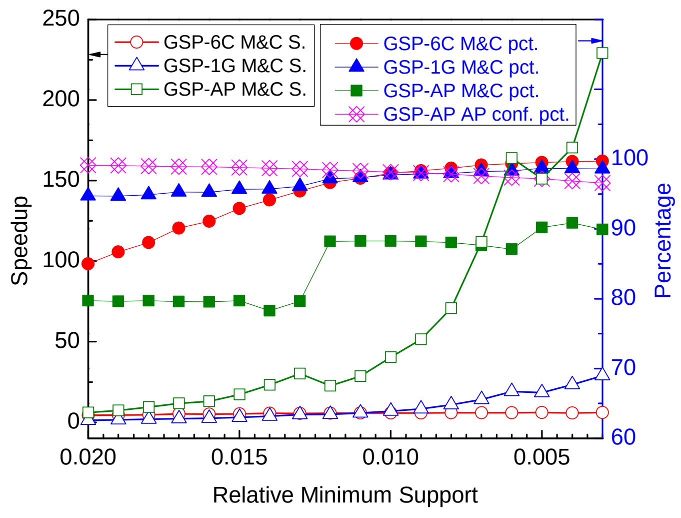


(e) Bible

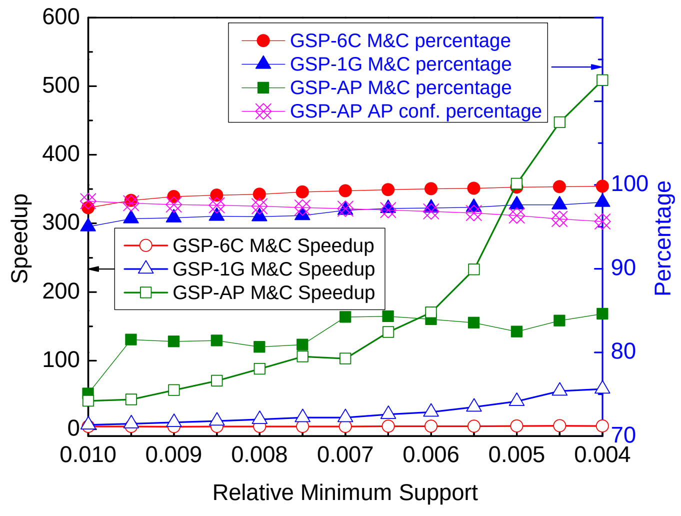


(f) FIFA

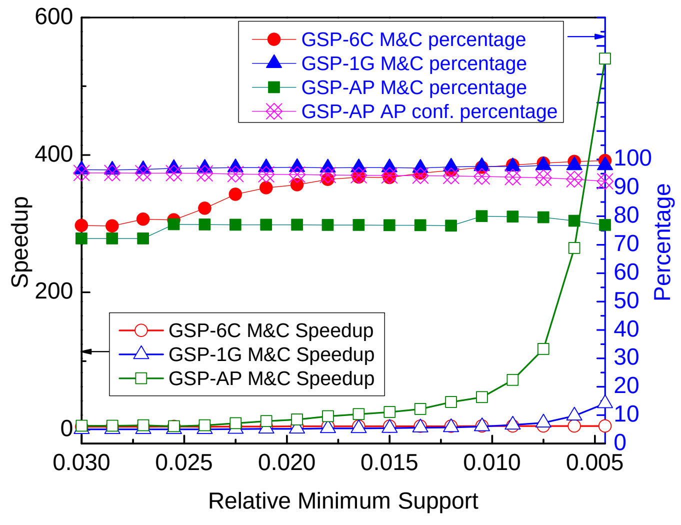
Fig. 7: The performance comparison among GSP-1C, GSP-6C, GSP-1G and GSP-AP on six benchmarks.



(a) BMS1

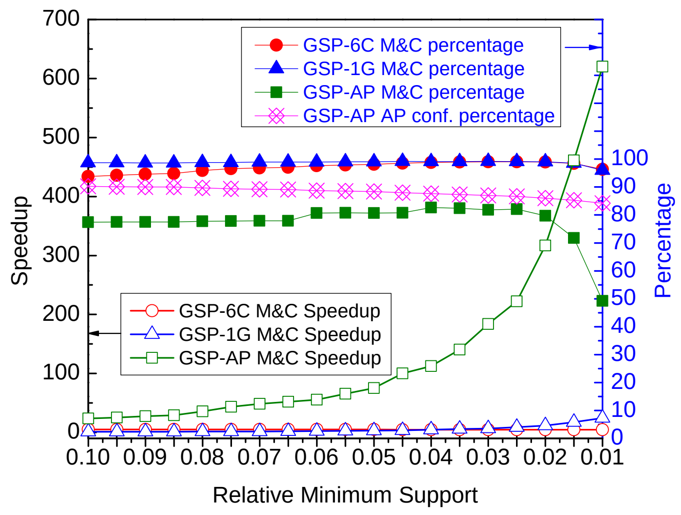


(b) BMS2

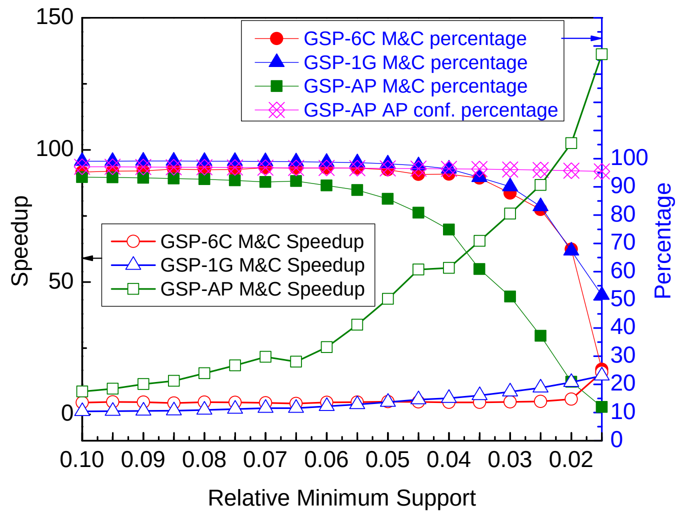


(c) Kosarak

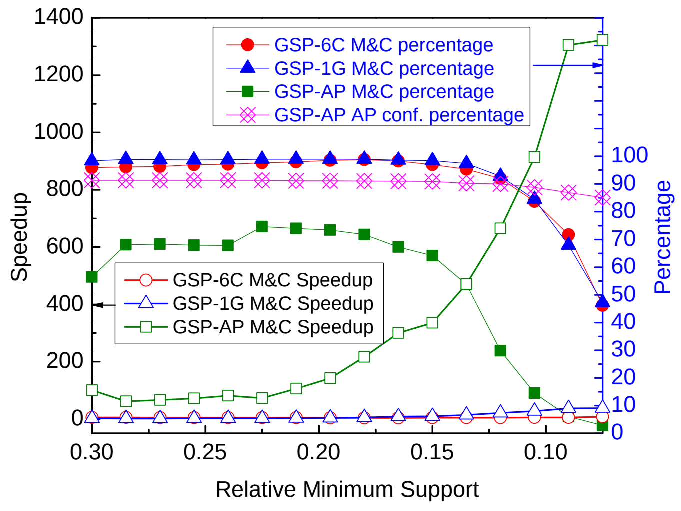
Fig. 8: The timing breakdown and speedup analysis on GSP implementations. “M&C percentage” means the percentage of matching-and-counting steps within the total *GSP* execution time. “AP conf. percentage” means the percentage of the AP configuration time, including both routing configuration time and symbol replacement time, in the total AP matching and counting time.



(a) Bible



(b) Leviathan



(c) FIFA

Fig. 9: The timing breakdown and speedup analysis on GSP implementations.

could improve the overall performance greatly. Figure 10 studies the cases of BMS2 and Kosarak, assuming 2X, 5X and 10X faster symbol replacement. Up to 2.7X speedup is achieved over current AP hardware when assuming 10X faster symbol replacement.

8.5 GSP-AP vs. Other SPM Algorithms

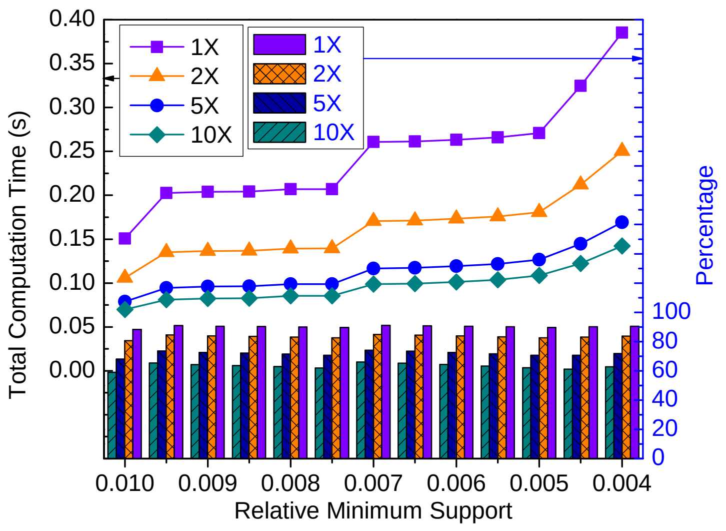
PrefixSpan and *SPADE* are two advanced algorithms which outperform *GSP* in general cases. In this paper, we test multi-threaded Java implementations of these two algorithms and evaluate them on a multi-core CPU. Figure 11 compares the performance of the Java multi-threaded implementations *PrefixSpan* and *SPADE* with hardware-accelerated *GSP* implementations. The performance of GSP-1G is in between *PrefixSpan* and *SPADE* on average. The proposed GSP-AP outperforms both *PrefixSpan* and *SPADE* in most cases, and achieves up to 300X speedup over *PrefixSpan* (in Bible) and up to 30X speedup over *SPADE* (in FIFA). As we see in the results, even multi-core *PrefixSpan* gives poor performance related to the AP. In addition, at least 50X speedup would be needed for *PrefixSpan* on the GPU to be competitive to the AP. So we do not implement it on the GPU. For *SPADE*, we again do not implement it for the GPU, because it runs out of memory for benchmarks larger than 10MB, assuming a high-end GPU with 24GB memory, such as the Nvidia K80. Smaller GPUs will fail even earlier.

As we discussed in Section 8.4, the performance of the AP and GPU solutions suffer from the increasing portion of the un-accelerated candidate-generation stage. We therefore implemented a multi-threaded candidate generation version for the AP and the GPU, GSP-AP-MTCG and GSP-1G-MTCG. The performance improvements are clear in Bible, FIFA and Leviathan, which become candidate-generation dominant at small minimum support numbers. The GSP-AP-MTCG get 452X speedup over *PrefixSpan* (in Bible) and up to 49X speedup over *SPADE* (in FIFA). The speedups of GSP-AP-MTCG over GSP-1G-MTCG become even larger because the same sequential stage is parallelized in the same way.

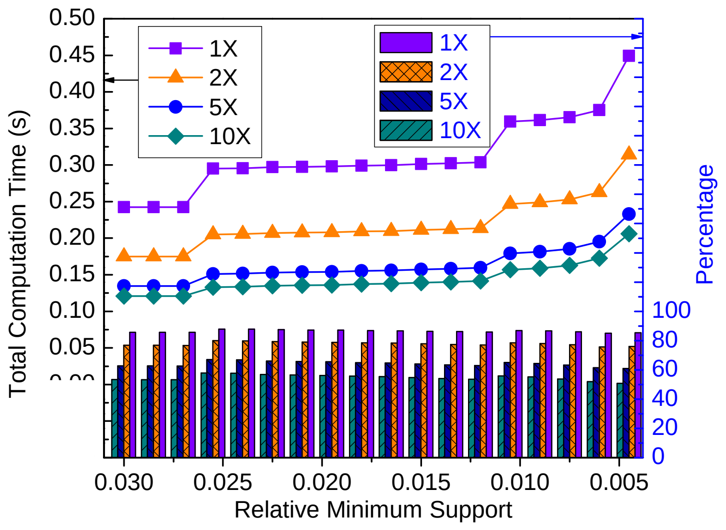
8.6 Performance Scaling with Data Size

In this era of “big data”, mining must accommodate ever-larger data sets. The size of the original datasets we adopted are all below 10MB, which may once have been representative, but are less so for the future. In this subsection, we study the trend of the performance scaling as a function of input data sizes. We enlarge the input data size by concatenating duplicates of the whole dataset with an assumption that the number of input sequences will grow much faster than the dictionary size (the number of distinct items) does.

Figure 12 shows the performance results of input data scaling on Kosarak and Leviathan. The total execution times of all tested methods, *PrefixSpan*, *SPADE*, GSP-1G and GSP-AP, increase linearly with the input data size on

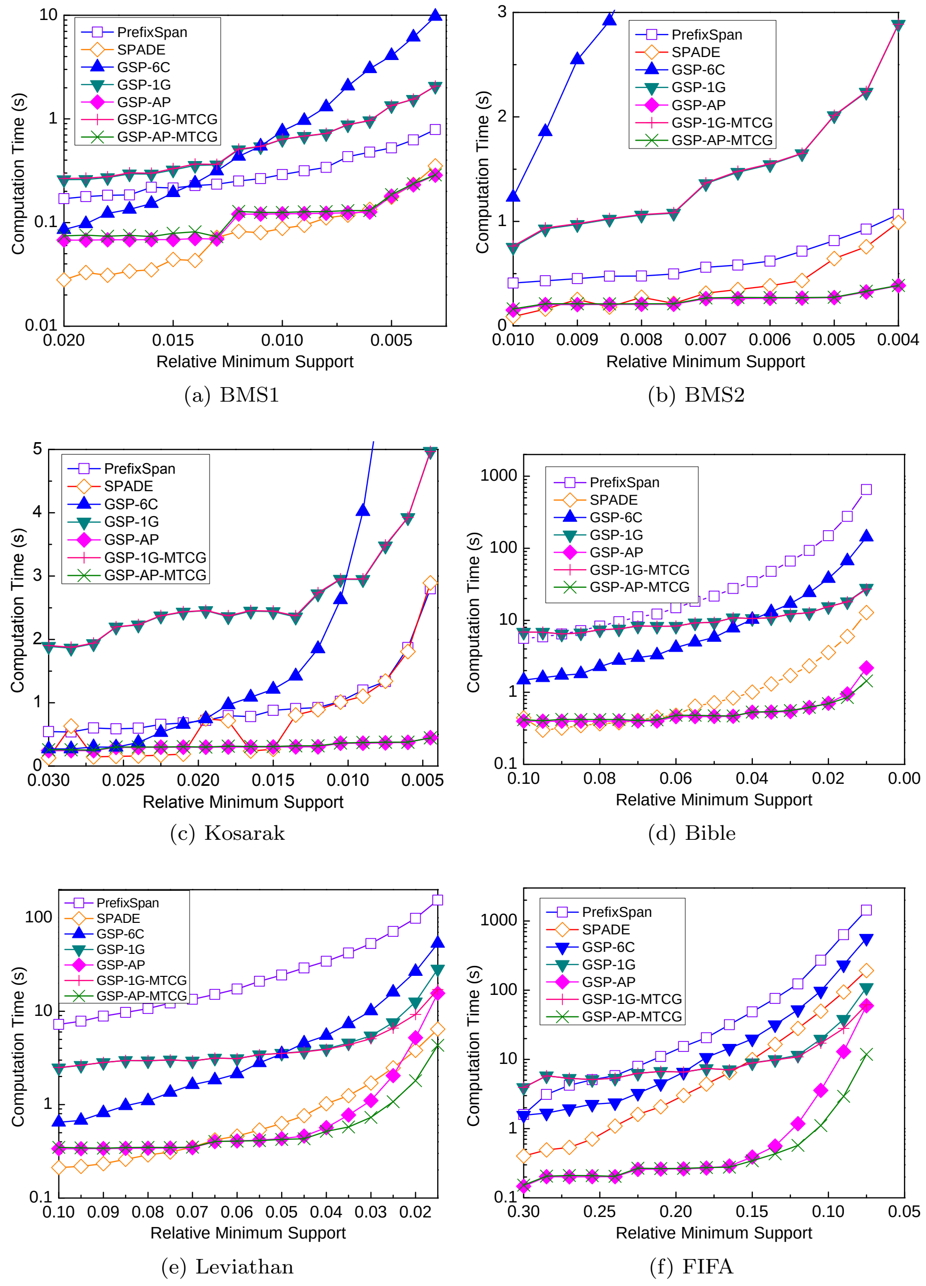


(a) BMS2



(b) Kosarak

Fig. 10: The impact of symbol replacement time on GSP-AP performance for BMS2 and Kosarak. The columns show the percentage of AP configuration time in total AP matching-and-counting time. The symbols and lines show overall all computation time.



(a) BMS1 (b) BMS2

(c) Kosarak (d) Bible

(e) Leviathan (f) FIFA

Fig. 11: The performance comparison among GSP-GPU, GSP-AP, PrefixSpan and SPADE.

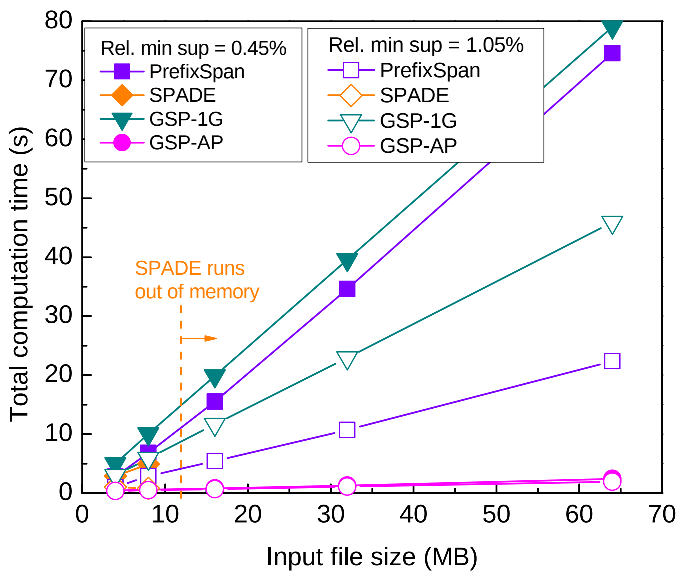
both benchmarks. The *SPADE* method runs out of memory (32GB on the CPU) for both tested minimum support numbers on Kosarak at input size larger than 10MB. Given smaller GPU on-board memory, a GPU *SPADE* would fail at even smaller datasets. The execution time of the proposed GSP-AP method scales much more favorably than other methods. Its speedup over *PrefixSpan* grows with larger data sizes, and reaches 31X at relative minimum support of 0.45%. A GPU implementation of *PrefixSpan* is unlikely to gain more speedup over the multi-threaded *PrefixSpan* shown here. For these reasons, the GPU implementations of *PrefixSpan* and *SPADE* are not needed in this analysis. In the case of Leviathan, GSP-AP shows worse performance than *SPADE* at small datasets, but outperforms it at large datasets. In this case, GSP-AP achieves up to 420X speedup over *PrefixSpan* and 11X speedup over *SPADE*.

9 Experimental Results for DRM

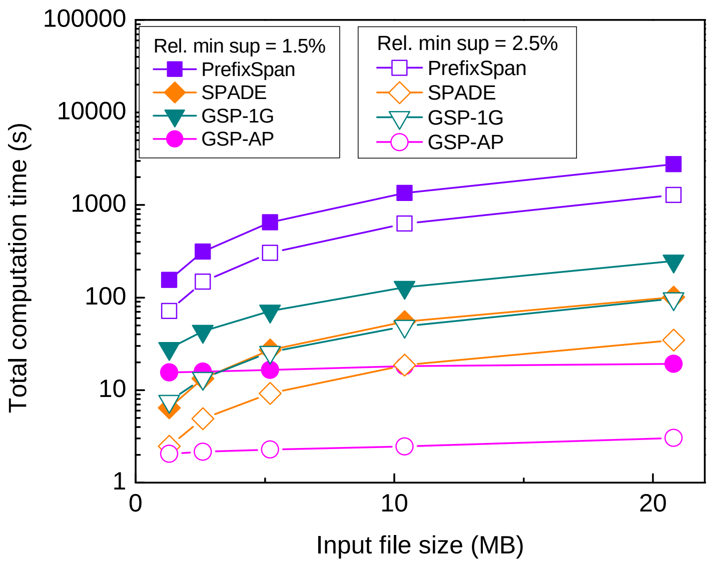
9.1 Comparison with CPU implementation

For each benchmark, we compare the performance of our CPU-AP implementation and a sequential CPU implementation over a range of minimum support values. Similar to SPM, a lower minimum support number requires longer CPU time. To finish all of our experiments in a reasonable time, we select minimum support numbers that produce computation times of the CPU-only implementation in the range of 10 seconds to 20 hours. A relative minimum support number, defined as the ratio of a minimum support number to the transaction number, is adopted in the figures. For a single d-item, the extra items beyond one item are called alternative items. The number of alternative items in one d-item is defined as the size of the d-item minus one. To avoid an extremely large search space, we only allow one alternative item (could be in any d-item) for end-to-end performance comparison, no matter how large the size of a d-rule is.

Considering disjunctive items, more candidates need to be scanned and more operations (*OR* operations between disjunctive items) need to be calculated. Therefore, the matching-and-counting step is more likely to be a performance bottleneck than that of the traditional frequent itemset mining. Figure 13 shows performance results of the CPU-AP solution and sequential CPU solution on datasets *Webdocs* and *ENWiki*. The CPU-AP DRM solution proposed by this paper achieves up to 487X and 614X speedups for end-to-end computation time measurement over the CPU solution on *Webdocs* and *ENWiki*. 2971X and 1366X speedups are achieved when comparing the d-rule matching-and-counting performances.

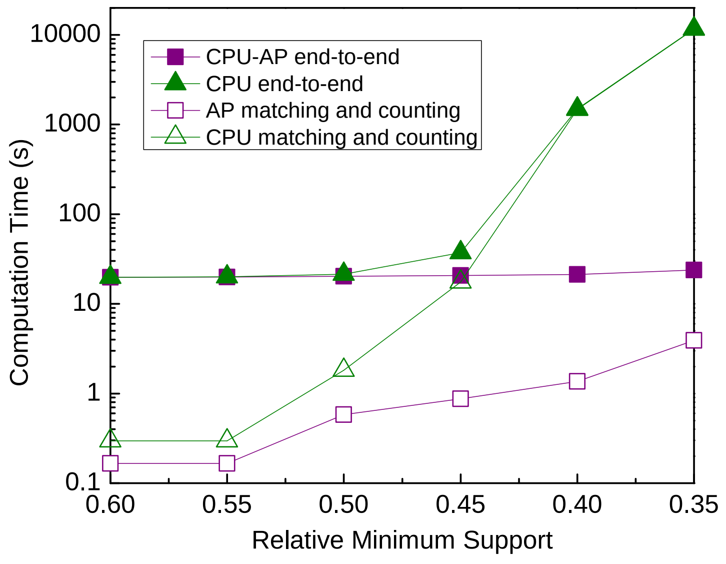


(a) Kosarak

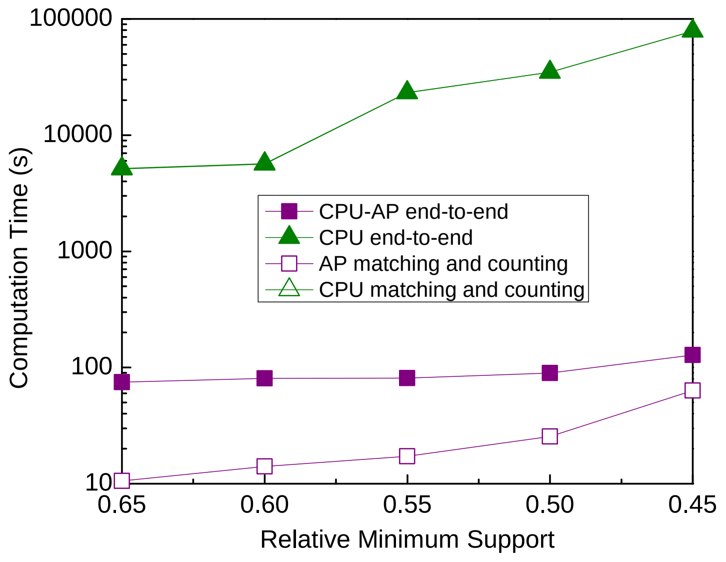


(b) Leviathan

Fig. 12: Performance scaling with input data size on Kosarak and Leviathan. (This figure is for Section 8)



(a) Webdocs



(b) ENWiki

Fig. 13: The performance comparison between CPU-AP DRM and CPU DRM.

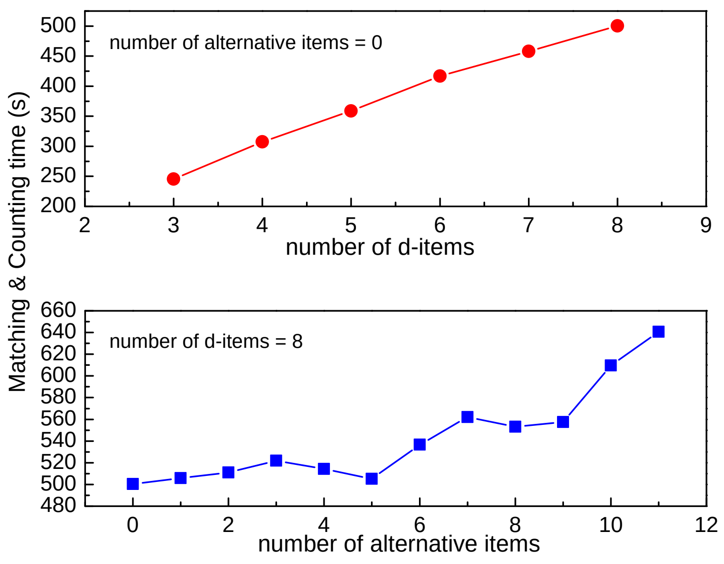


Fig. 14: The CPU time of matching and counting 20,000 d-rules against d-rule size and total number of alternative items. The corresponding AP matching and counting time is just 1.16s for all the cases shown in this figure.

9.2 Matching and Counting

Figure 14 shows the trends of the CPU matching-and-counting time varying against d-rule size (d-item numbers of a d-rule) and total number of alternative items of a d-rule. The d-rules and input stream are all generated from the benchmark *Webdocs*. Adding one more d-item to an existing d-rule causes one extra *AND* operation when recognizing this d-rule. Adding one more alternative items to an existing d-rule causes extra one *OR* operation when recognizing this d-rule. Figure 14 shows significant increase in the CPU time when increasing either d-rule size or alternative items. From the figure, one can expect even longer CPU time for larger d-rule sizes and more alternative items. The sub-linearity and the noise shown in this figure are all due to the short-circuit evaluation technique we adopted to improve the CPU matching performance. In contrast, the AP matching-and-counting time keeps invariant if d-rule size is not larger than the design capacity of the d-rule automaton macro (for example, 23). Actually, in very rare cases, the mining will not stop before d-rule size reaches 20. Therefore, in practical cases, the AP DRM matching-and-counting time only depends on the number of d-rule candidates and the input length.

10 Related Work

Because of the larger permutation space and complex hierarchical patterns involved, performance is a critical issue for applying hierarchical pattern mining techniques. Many efforts have been made to speed up hierarchical pattern mining via software and hardware.

10.1 Sequential Pattern Mining

10.1.1 Sequential Algorithms

Generalized Sequential Pattern *GSP* [21] follows the multi-pass *candidate generation–pruning* scheme of the classic *Apriori* algorithm and inherits the horizontal data format and breadth-first-search scheme from it. Also in the family of the *Apriori* algorithms, Sequential PAttern Discovery using Equivalence classes (*SPADE*) [27] was derived from the concept of *equivalence class* [25] for sequential pattern mining, and adopts the vertical data representation. To avoid the multiple passes of candidate generation and pruning steps, *PrefixSpan* [17] algorithm extended the idea of the pattern growth paradigm [13] to sequential pattern mining.

10.1.2 Parallel Implementations

Shintani and Kitsuregawa [20] proposed three parallel *GSP* algorithms on distributed memory systems. These algorithms show good scaling properties on an IBM SP2 cluster. Zaki *et al.* [26] designed *pSPADE*, a data-parallel version of SPADE for fast discovery of frequent sequences in large databases on distributed-shared memory systems, and achieved up to 7.2X speedup on a 12-processor SGI Origin 2000 cluster. Guralnik and Karypis [12] developed tree-projection-based parallel sequence mining algorithms for distributed-memory architectures and achieved up to 30X speedups on a 32-processor IBM SP cluster. Cong *et al.* [8] presented a parallel sequential pattern mining algorithm (Par-ASP) under their sampling-based framework for parallel data mining, implemented by using MPI over a 64-node Linux cluster, achieving up to 37.8X speedup.

10.1.3 Accelerators

Hardware accelerators allow a single node to achieve orders of magnitude improvements in performance and energy efficiency. General-purpose graphics processing units (GPUs) leverage high parallelism, but GPUs' single instruction multiple data (SIMD), lockstep organization means that the parallel tasks must generally be similar. Hryniów [14] presented a parallel *GSP* implementation on GPU. However this work did not accelerate sequential pattern mining but relaxed the problem to an itemset mining. To the best of our knowledge, there has been no previous work on hardware acceleration for true SPM. In particular, *SPADE* and *PrefixSpan* have not been implemented on GPU. For our analysis purpose, we implemented true *GSP* for SPM on GPU.

10.2 Disjunctive Rule Mining

Nanavati et al. [15] first introduced the concept of disjunctive rules and did conceptual and algorithmic studies on disjunctive rules of both inclusive *OR* and exclusive *OR*. Sampaio et al. [19] developed a new algorithm to induce disjunctive rules under certain restrictions to limit the search spaces of the antecedent and consequent terms. Chiang et al. [7] proposed disjunctive consequent association rules, a conceptual combination of the disjunctive rule and the sequential pattern, and illustrated the promising commercial applications of this new mining technique. However, all these existing works focused on effectiveness more than the efficiency of the implementations.

The Automata Processor shows great potential in boosting the performance of massive and complex pattern-searching applications. We show in this paper that the proposed AP-accelerated solutions for sequential pattern mining and disjunctive rule mining have great performance advantages over the CPU and other parallel and hardware-accelerated implementations.

11 Conclusions and the Future Work

We present a flexible hardware-accelerated framework for hierarchical pattern mining problems. Under this framework, sequential pattern mining (SPM) and disjunctive rule mining (DRM) are accelerated on the new Automata Processor (AP), which provides native hardware implementation of non-deterministic finite automata. Two automaton design strategies, *linear design* and *reduction design*, are proposed and tested for SPM and DRM respectively, and have shown to effectively leverage highly-parallel automata hardware such as the AP.

Our CPU-AP solution for SPM adopts the Generalized Sequential Pattern (*GSP*) algorithm from the *Apriori* family, based on the downward-closure property of frequent sequential patterns. We derive a compact automaton design for matching and counting frequent sequences. The *linear design* strategy designs automata for SPM by flatting hierarchical patterns of sequences into plain strings with delimiters and place-holders. A multiple-entry NFA strategy is proposed to accommodate variable-structured sequences. This allows a single, compact template to match any candidate sequence of a given length, so this template can be replicated to make full use of the capacity and massive parallelism of the AP. We compare *GSP* across different hardware platforms. Up to 430X, 90X, and 29X speedups are achieved by the AP-accelerated GSP on six real-world datasets, when compared with the single-threaded CPU, multicore CPU, and GPU GSP implementations. The AP-accelerated SPM also outperforms *PrefixSpan* and *SPADE* on multicore CPU by up to 300X and 30X. By parallelizing candidate generation, these speedups are further improved to 452X and 49X. Even more performance improvement can be achieved with hardware support to minimize symbol replacement latency. Our AP solution shows good scaling properties for larger datasets, while the alternatives scale poorly.

To accelerate DRM by using the AP, we present an automaton design for matching and counting disjunctive rules efficiently. This automaton design follows the *reduction design* strategy and utilizes the on-chip Boolean (AND) gates to implement the reduction operations among d-rules, and the bit-wise parallelism feature of STEs to deal with the OR operations among items in one d-item. The experiments show up to 614X speedups of the proposed CPU-AP DRM solution over sequential CPU algorithm on two real-world datasets. The experiments also demonstrate significant increase on the CPU matching-and-counting time when increasing d-rule size or the number of alternative items. In contrast, the d-rule recognition time on the AP is two orders of magnitudes faster than the CPU version and keeps invariant despite the increasing complexity of d-rules.

A field-programmable gate array (FPGA) is a good competitor to the AP architecture in symbolic data processing, because it is another reconfigurable architecture. A performance comparison between the AP and FPGA for pattern-mining algorithms is an interesting direction for future work.

Acknowledgements This work was supported in part by the Virginia CIT CRCF program under grant no. MF14S-021-IT; by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA; NSF grant EF-1124931; and a grant from Micron Technology.

References

1. (2004) Frequent itemset mining dataset repository. http://fimi.ua.ac.be/data/
2. (2015) Micron Automata Processor website. Http://www.micronautomata.com/documentation
3. Aggarwal CC, Han J (eds) (2014) Frequent Pattern Mining. Springer International Publishing, Cham
4. Agrawal R, Srikant R (1995) Mining sequential patterns. In: Proc. of the International Conference on Data Engineering (ICDE), IEEE, pp 3–14
5. Agrawal R, Imieliński T, Swami A (1993) Mining association rules between sets of items in large databases. In: Proc. of SIGMOD '93
6. Bo C, et al (2016) Entity resolution acceleration using microns automata processor. In: Proc. of the International Conference on Big Data (BigData)
7. Chiang DA, Wang YF, Wang YH, Chen ZY, Hsu MH (2011) Mining disjunctive consequent association rules. Applied Soft Computing 11(2):2129 – 2133

8. Cong S, Han J, Hoeflinger J, Padua D (2005) A sampling-based framework for parallel data mining. In: Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), ACM
9. Dlugosch P, et al (2014) An efficient and scalable semiconductor architecture for parallel automata processing. IEEE Transactions on Parallel and Distributed Systems 25(12):3088–3098
10. Fang W, et al (2009) Frequent itemset mining on graphics processors. In: Proc. International Workshop on Data Management on New Hardware (DaMoN)
11. Fournier-Viger P, et al (2014) Spmf: A java open-source pattern mining library. Journal of Machine Learning Research 15:3569–3573
12. Guralnik V, Karypis G (2004) Parallel tree-projection-based sequence mining algorithms. Parallel Comput 30(4):443–472
13. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Proc. of SIGMOD ’00, ACM
14. Hryniów K (2012) Parallel pattern mining-application of gsp algorithm for graphics processing units. In: Proc. of the International Carpathian Control Conference (ICCC), IEEE, pp 233–236
15. Nanavati AA, Chitrapura KP, Joshi S, Krishnapuram R (2001) Mining generalised disjunctive association rules. In: Proc. of the Tenth International Conference on Information and Knowledge Management (CIKM), ACM, New York, NY, USA, pp 482–489
16. Noyes H (2014) Micron automata processor architecture: Reconfigurable and massively parallel automata processing. In: Proc. of the Fifth International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, keynote presentation
17. Pei J, et al (2004) Mining sequential patterns by pattern-growth: The prefixspan approach. IEEE Transactions on Knowledge and Data Engineering (TKDE) 16(11):1424–1440
18. Roy I, Aluru S (2016) Discovering motifs in biological sequences using the micron automata processor. IEEE/ACM Transactions on Computational Biology and Bioinformatics 13(1):99–111
19. Sampaio MC, Cardoso OHB, Santos GPD, Hattori L (2008) Mining disjunctive association rules
20. Shintani T, Kitsuregawa M (1998) Mining algorithms for sequential patterns in parallel: Hash based approach. In: Proc. of the Second Pacific−Asia Conference on Knowledge Discovery and Data mining, pp 283–294
21. Srikant R, Agrawal R (1996) Mining sequential patterns: Generalizations and performance improvements. In: Proc. of the International Conference on Extending Database Technology (EDBT)

22. Wang K, Qi Y, Fox J, Stan M, Skadron K (2015) Association rule mining with the micron automata processor. In: Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)
23. Wang K, Sadredini E, Skadron K (2016) Sequential pattern mining with the micron automata processor. In: Proc. of the ACM International Conference on Computing Frontiers, ACM, New York, NY, USA, CF '16
24. Wang MH, et al (2016) Using the automata processor for fast pattern recognition in high energy physics experimentsa proof of concept. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 832:219 − 230
25. Zaki MJ (2000) Scalable algorithms for association mining. IEEE Transactions on Knowledge and Data Engineering (TKDE) 12(3):372–390
26. Zaki MJ (2001) Parallel sequence mining on shared-memory machines. Journal of Parallel and Distributed Computing 61(3):401–426
27. Zaki MJ (2001) Spade: An efficient algorithm for mining frequent sequences. Machine Learning 42(1-2):31–60
28. Zhang F, Zhang Y, Bakos JD (2013) Accelerating frequent itemset mining on graphics processing units. Journal of Supercomputing 66(1):94–117
29. Zu Y, et al (2012) GPU-based NFA implementation for memory efficient high speed regular expression matching. In: Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), ACM, pp 129–140