Sequential Pattern Mining with the Micron Automata Processor

Ke Wang, Elaheh Sadredini, Kevin Skadron
Department of Computer Science
University of Virginia
Charlottesville, VA, 22904 USA
{kewang, elaheh, skadron}@virginia.edu

ABSTRACT

Sequential pattern mining (SPM) is a widely used data mining technique for discovering common sequences of events in large databases. When compared with the simple set mining problem and string mining problem, the hierarchical structure of sequential pattern mining (due to the need to consider frequent subsets within each itemset, as well as order among itemsets) and the resulting large permutation space makes SPM extremely expensive on conventional processor architectures. We propose a hardware-accelerated solution of the SPM using Micron's Automata Processor (AP), a hardware implementation of non-deterministic finite automata (NFAs). The Generalized Sequential Pattern (GSP) algorithm for SPM searching exposes massive parallelism, and is therefore well-suited for AP acceleration. We implement the multi-pass pruning strategy of the GSP via the AP's fast reconfigurability. A generalized automaton structure is proposed by flattening sequential patterns to simple strings to reduce compilation time and to minimize overhead of reconfiguration. Up to 90X and 29X speedups are achieved by the AP-accelerated GSP on six real-world datasets, when compared with the optimized multicore CPU and GPU GSP implementations, respectively. The proposed CPU-AP solution also outperforms the state-of-the-art PrefixSpan and SPADE algorithms on multicore CPU by up to 452X and 49X speedups. The AP advantage grows further with larger datasets.

CCS Concepts

•Information systems → Association rules; •Computer systems organization → Multiple instruction, single data; •Hardware → Emerging architectures;

Keywords

Automata Processor; sequential pattern mining, Apriori

1. INTRODUCTION

Sequential Pattern Mining (SPM) is a data-mining technique that identifies strong and interesting sequential relations among variables in structured databases. SPM has become an important data mining technique with broad application domains, such as customer purchase patterning analysis, correlation analysis of storage system, web log analysis, software bug tracking and software API usage tracking [2]. For example, when a person buys a pen, appropriate recommendations for paper and ink may increase sales of a store. SPM is the right technique to mine sequential relations from the records of transactions.

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 recognize frequent set mining (FSM), SPM needs to deal with permutations among the frequent itemsets. This dramatically increases the number of patterns to consider 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 the sequential pattern mining. The three most competitive algorithms today are Generalized Sequential Pattern (*GSP*) [15], Sequential PAttern Discovery using Equivalence classes (*SPADE*) [19] and *PrefixSpan* [12]. *SPADE* and *PrefixSpan* are generally favored today, and perform better than *GSP* on conventional single-core CPUs in average cases; but *GSP* 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., [4, 8, 14, 18]. 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. [6, 20, 21], we are not aware of any previous work for a hardware-accelerated solution for SPM.

Micron's new Automata Processor (AP) [5] offers an appealing accelerator architecture for SPM. 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 [5], DNA motif search-

CF'16, May 16-19, 2016, Como, Italy

DOI: http://dx.doi.org/10.1145/2903150.2903172

ing [13], and frequent set mining [16].

In this paper, we propose a CPU-AP heterogeneous computing solution to accelerate SPM based on the *GSP* algorithm framework, whose multipass algorithm to build up successively larger candidate itemsets and sequences is best suited to the AP's highly parallel pattern-matching architecture, which can check a large number of candidate patterns in parallel. The sequential patterns are identified and counted by an NFA-counter automaton structure on the AP chip. The key idea of designing such an NFA for SPM is to flatten sequential patterns to simple 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. To our knowledge, this is the first automaton design to identify hierarchical sequential patterns.

On multiple real-world and synthetic datasets, we compare the performance of the proposed AP-accelerated *GSP* versus CPU and GPU implementations of *GSP*, as well as Java multi-threaded implementations of *SPADE* and *PrefixSpan* [7]. The performance analysis of the AP-accelerated *GSP* shows up to 90X speedup over a multicore CPU *GSP* and up to 29X speedup 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 solution also shows good performance scaling as the size of the input dataset grows, achieving even better speedup over *SPADE* and *PrefixSpan*. Our file 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).

Overall, this paper makes three principal contributions:

1. We show how to map SPM to NFAs, and develop a CPU-AP computing infrastructure for *GSP*.
2. We design a novel automaton structure for the sequential pattern matching and counting in *GSP*. This structure flattens the hierarchical patterns to strings and adopts a multiple-entry scheme to reduce the automaton design space for candidate sequential patterns.
3. Our AP SPM solution shows performance improvement and broader capability over multicore and GPU implementations of *GSP* SPM, and also outperforms *SPADE* and *PrefixScan* (especially for larger datasets).

2. SEQUENTIAL PATTERN MINING

2.1 Introduction to SPM

Sequential pattern mining (SPM) was first described by Agrawal and Srikant [3]. 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. FSM 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 $s_1 =< t_1 t_2 ... t_m >$ called to be a subsequence of $s_2 =< r_1 r_2 ... r_j >$, if there are integers $1 \preceq k_1 \prec k_2 \prec .. \prec k_{m-1} \prec k_m \preceq j$ such that $t_1 \subseteq r_{k_1}, t_2 \subseteq r_{k_2}, ..., t_m \subseteq r_{k_m}$. 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 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 at 1-sequence 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

Table 1: Example of candidate generation

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

times the input matches a sequence candidate. The occurrence of each candidate pattern is recorded 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* and how this solution compares with other parallel or accelerator implementations of SPM. 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. AUTOMATA PROCESSOR

3.1 Architecture

The AP chip has three types of functional elements - the state transition element (STE), counters, and Boolean elements [5].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. The AP uses a homogeneous NFA representation [5] 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.

Micron's current generation AP-D480 boards use AP chips built on 50nm DRAM technology, running at an input symbol (8-bit) rate of 133 MHz. A D480 chip has 192 blocks, with 256 STEs, 4 counters and 12 Boolean elements per block [5]. We assume an AP board with 32 AP chips, so that all AP chips process input data stream in parallel.

3.2 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 1's in positions corresponding to the elements that report and 0's 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.

3.3 Programming and configuration

The Micron's AP SDK provides Automata Network Markup Language (ANML), an XML-like language for describing automata networks, as well as C, Java and Python binding interfaces 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*) [1]. 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 [13]. 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 [13], 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.

4. MAPPING SPM ONTO THE AP

As we discussed in Sec. 2.2, *GSP* maps to the AP architecture naturally and the sequential pattern matching-and-counting step is the performance bottleneck of the *GSP* on conventional architectures. Therefore, we propose to use the AP to accelerate the matching-and-counting step.

4.1 Automaton for Matching and Counting

The hierarchical patterns in SPM, sequences of itemsets, are more complex than strings or individual itemsets as studied in the previous works [13, 16]. Within itemsets of a sequence, items of interest may be discontinuous, i.e. we may only be interested in some frequent subset of an itemset [16]. While, one input sequence may have irrelevant itemsets in between interesting itemsets. The matching part of the automaton for SPM should identify the frequent 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 proposed automaton design for hierarchical pattern matching. Furthermore, in order to maxi-

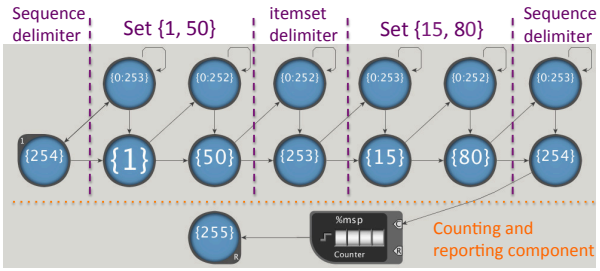
mize benefit from the high parallelism of NFAs, and the Micron AP in particular, an appropriate automaton structure must be as compact as possible, to maximize the number of such structures that can be accommodated in a single pass.

4.1.1 Flattening the Hierarchy of Sequential Patterns

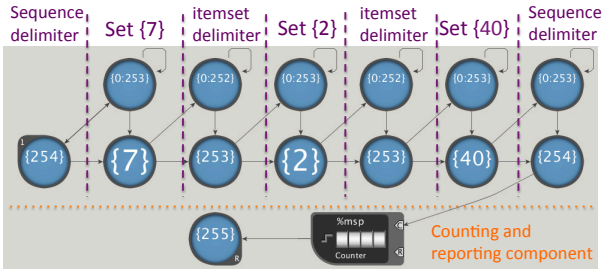
To match sequences of itemsets, we first convert sets into strings with a pre-defined order. And then we introduce a delimiter of itemset to bound and connect these strings (converted from itemsets) within a sequential pattern. The sequence of strings is also a string. Keeping this observation in mind, 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. In the case of more than 253 frequent items, two conjunctive STEs are used to represent an item and support up to 64,009 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 conjunctive sequences. In Figure 1, the counting and reporting component is shown below the orange dotted line. The I/O optimization strategy proposed in [16] 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 [16], the NFA for SPM is divided into several itemsets, demarcated by the itemset delimiter. 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 (with itemsets or between itemsets). Each "position holder" has a self-activation connection and match all valid symbols (excluding the delimeters). 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 patterns, 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 a sequence of sequences or a pattern of more than a two-level hierarchy, can be implemented using



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



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

Figure 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.

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 1.

4.1.2 Multiple-entry NFAs

In each *GSP* level, there could be 0 to $k-1$ delimiters in actual patterns, 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 delimeters). Because candidate sequences are generated at runtime, the number of patterns to be checked of a given encoded length is not known before runtime. We need a further step to reduce the automaton design space of the candidates for each *GSP* 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. [16]. 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

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.

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".

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).

4.1.3 Macro Selection and Capacity

The flattening strategy and multiple-entry strategy introduced in Sec 4.1.1 and 4.1.2 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 4K 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. For this case, counters are more likely a limiting factor of the capacity.

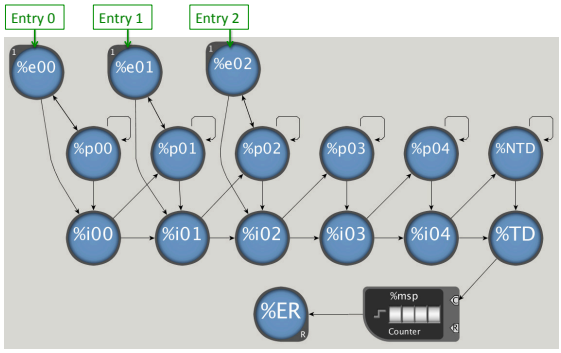
The actual capacity of a macro may be limited by STEs, counters, or routing resources of the AP chip. We have developed a library of macro structures described in Section 4.1.2 and compiled all these macros with the newest Micron AP compiler (v1.6-5). Table 2 and 3 show the actual capacities of macros 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.

4.2 Program Infrastructure

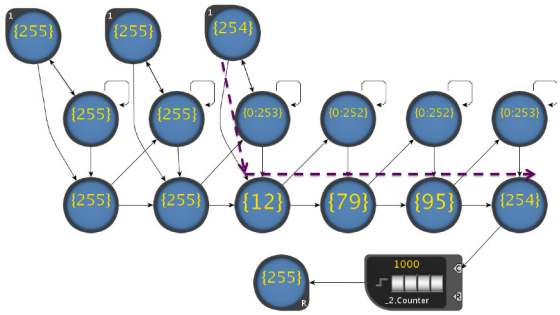
Figure 3 shows the complete workflow of the AP-accelerated SPM proposed in this paper. The data pre-processing step creates a data stream from the input dataset and makes the data stream compatible with the AP interface. Pre-processing consists of the following steps:

1. Filter out infrequent items from input sequences
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

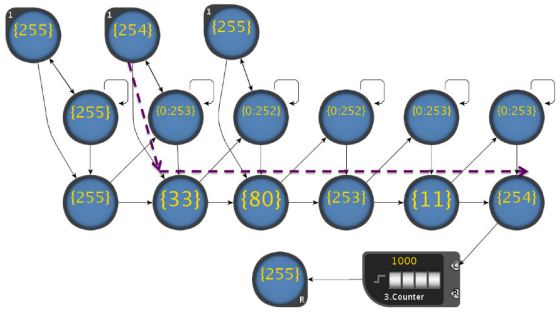
Step 1 helps to avoid unnecessary computing on infrequent items and reduces the dictionary size of items. Depending on the number of frequent items, the items can be encoded



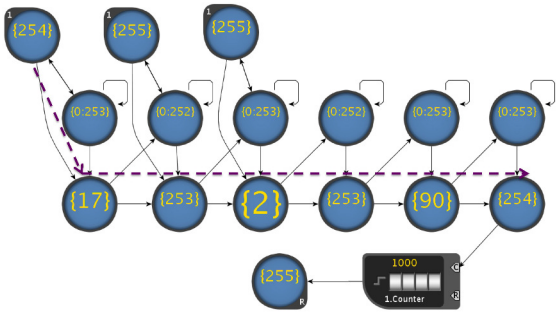
(a) AP macro for sequential pattern



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



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



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

Figure 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.

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 capacities of patterns. 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

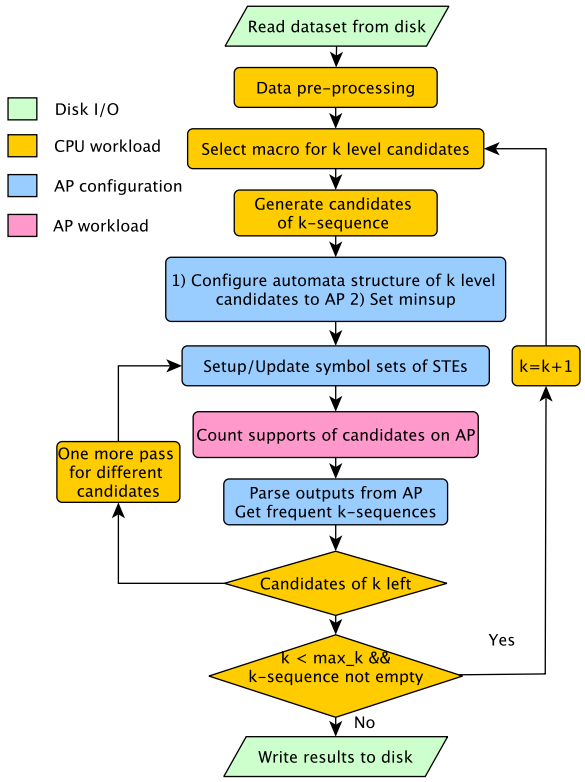


Figure 3: The workflow of AP-accelerated SPM

(in any given order) to fit the automaton design described in Section 4.1. 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 4.1.3), and is configured onto the AP board. The candidates are generated on the CPU and are filled into the selected automaton template macro. The input data formulated in pre-processing is then streamed into the AP board for counting.

5. EXPERIMENTAL RESULTS

The performance of our AP implementation is evaluated using CPU timers, stated configuration latencies, and an AP simulator in the AP SDK [1, 11], 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 3).

5.1 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 [7] 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* [7]. 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.

5.2 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 database for GPU and AP implementations to optimize the performance of candidate pattern generation.

GSP-multi-core: Work flow is the same as the GSP-CPU implementation except that the matching and counting step is parallelized using OpenMP .

The CPU version adopts the data structure of linked-list to accelerate the pruning and counting operations to achieve the best overall performance.

5.3 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)

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.

5.4 Datasets

Six public real-world datasets for sequential pattern mining found on *spmf* [7] website are tested. The details of these datasets are shown in Table 4.

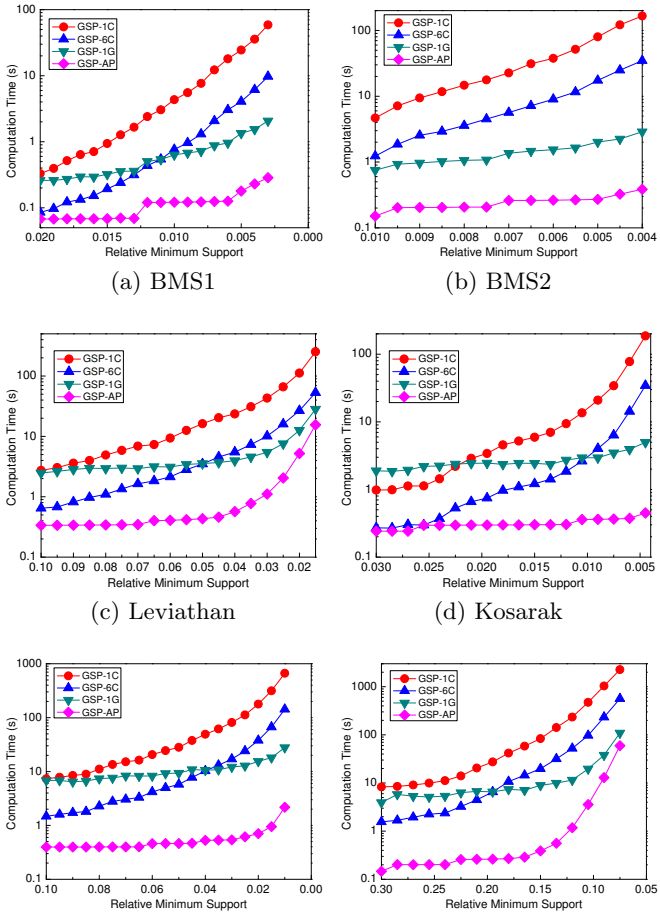
5.5 GSP-AP vs. Other GSP Implementations

Figure 4 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.

Table 4: **Datasets**

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.



(a) BMS1 (b) BMS2

(c) Leviathan (d) Kosarak

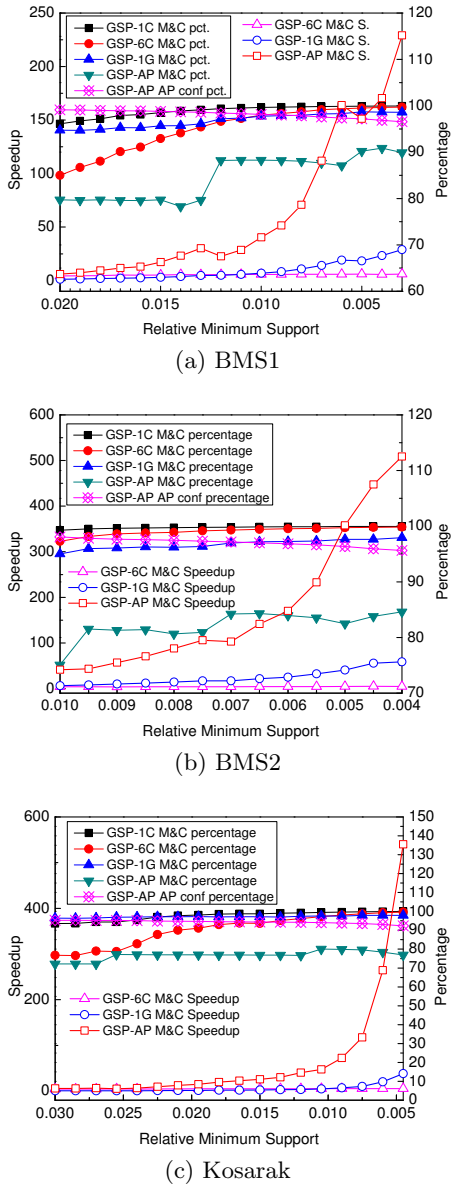
(e) Bible (f) FIFA

Figure 4: The performance comparison among GSP-1C, GSP-6C, GSP-1G and GSP-AP on six benchmarks.

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.

5.6 Timing Breakdown and Speedup Analysis

To better understand the performance shown in Figure 4, profiling results are shown in Figures 5 and 6. 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



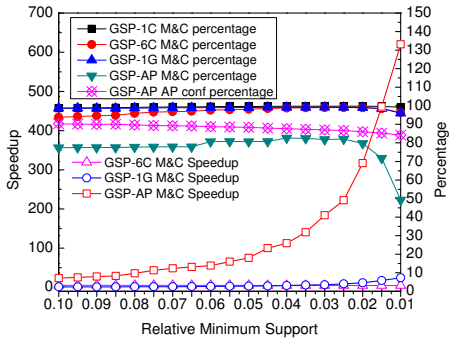
(a) BMS1

(b) BMS2

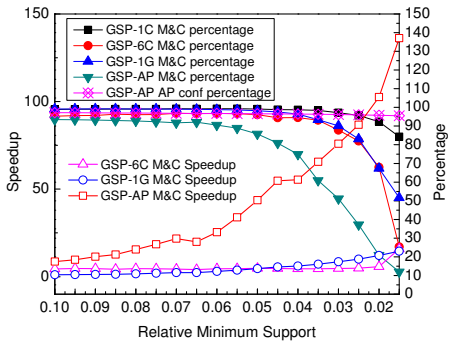
(c) Kosarak

Figure 5: The timing breakdown and speedup analysis on GSP implementations. The "M&C percentage" means the percentage of matching and counting steps within the total *GSP* execution time. The "AP conf percentage" means the percentage of AP configuration time, including both routing configuration time and symbol replacement time, in total AP matching and counting time.

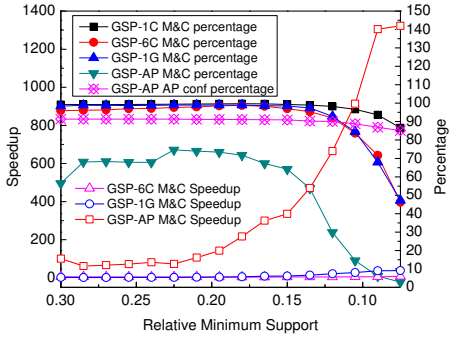
sequential matching and counting implementation. The smaller the minimum support, the more candidates are generated, and the larger the speedups achieved for both GPU and AP versions. On one hand, it shows the performance boost of massive complex-pattern matching achieved by the AP. On the other hand, 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 5.7). Amdahl's law has even more severe impact on the AP version than on GPU



(a) Bible



(b) Leviathan



(c) FIFA

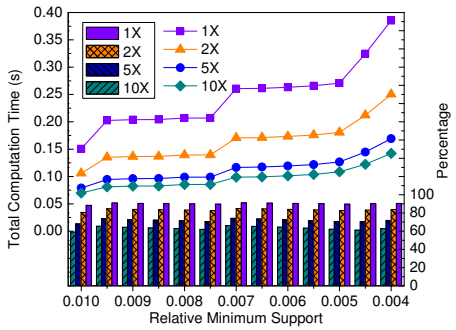
Figure 6: The timing breakdown and speedup analysis on GSP implementations.

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%.

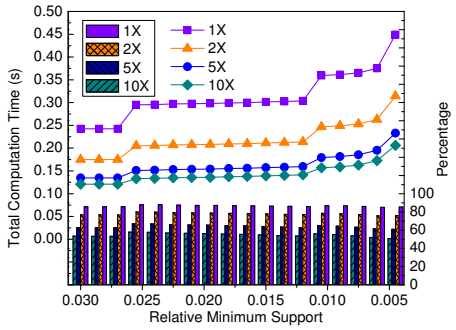
From Figures 5 and 6 we observe that configuration time dominates the total AP matching and counting time, 80%-90% of the AP time for all cases. Fortunately, the latency of symbol replacement could be significantly reduced in future generations of the AP, because symbol replacement is simply a series of DRAM writes, and this should be much faster. We hypothesize that the current times assume some conservative buffering. Reducing symbol replacement could improve the overall performance greatly. Figure 7 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.

5.7 GSP-AP vs. Other SPM Algorithms

The *PrefixSpan* and *SPADE* are two advanced algorithms



(a) BMS2

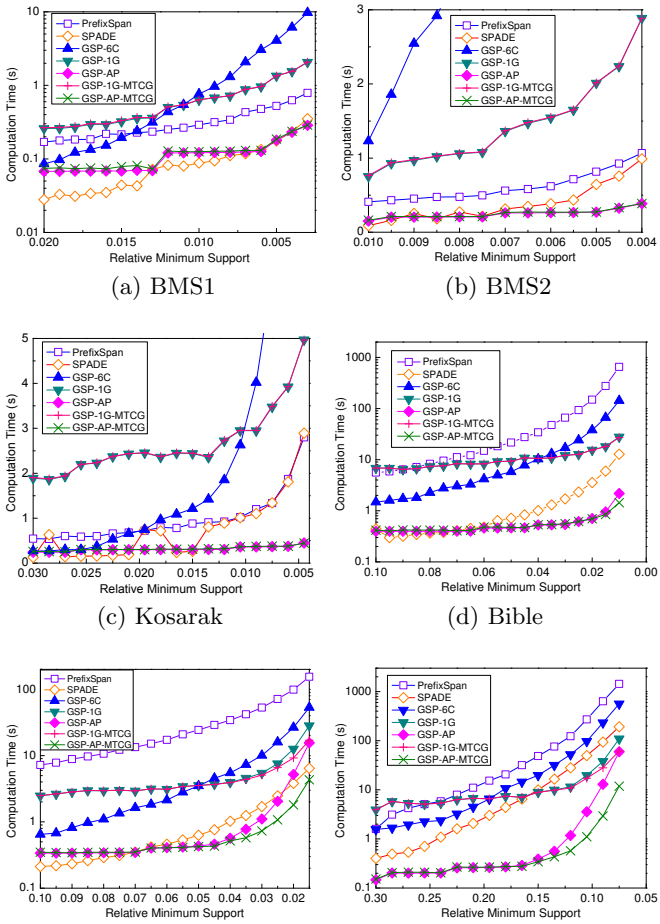


(b) Kosarak

Figure 7: 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.

which outperform the *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. 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. Figure 8 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 discussed in Section 5.6, the performance of 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 AP and GPU, GSP-AP-MTCG and GSP-1G-MTCG. The performance improvements are clear in Bible, FIFA and Leviathan who 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.



(a) BMS1 (b) BMS2

(c) Kosarak (d) Bible

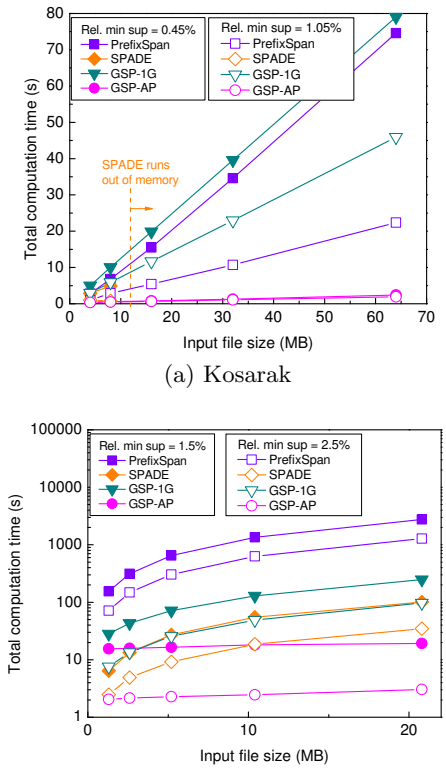
(e) Leviathan (f) FIFA

Figure 8: The performance comparison among GSP-GPU, GSP-AP, PrefixSpan and SPADE.

5.8 Performance Scaling with Data Size

In this era of "big data", mining must accommodate ever-larger data sets. 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 scaling of performance 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 9 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 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 paper. In the case of



(a) Kosarak

(b) Leviathan

Figure 9: Performance scaling with input data size on Kosarak and Leviathan.

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*.

6. RELATED WORK

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

6.1 Sequential Algorithms

Generalized Sequential Pattern *GSP* [15] follows the multi-pass *candidate generation–pruning* scheme of *Apriori* algorithm and inherits the horizontal data format and breadth-first-search scheme from it. Also in the family of the *Apriori* algorithm, Sequential PAttern Discovery using Equivalence classes (*SPADE*) [19] was derived from the concept of *equivalence class* [17] for sequential pattern mining, and adopts the vertical data representation. To avoid the multiple passes of candidate generation and pruning steps, *PrefixSpan* [12] algorithm extended the idea of the pattern growth paradigm [9] to sequential pattern mining.

6.2 Parallel Implementations

Shintani and Kitsuregawa [14] proposed three parallel *GSP* algorithms on distributed memory systems. These algorithms show good scaling properties on an IBM SP2 cluster. Zaki *et al.* [18] 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 [8] 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.* [4] 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.

6.3 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. In ref. [10], the authors present a parallel *GSP* implementation on GPU but they relax the problem of sequential pattern mining to 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.

Micron's AP shows great potential in boosting performance of massive pattern matching applications. We show in this paper that the proposed AP-accelerated solution for sequential pattern mining using *GSP* has great performance advantage over other parallel and hardware-accelerated implementations.

7. CONCLUSIONS AND FUTURE WORK

We present a hardware-accelerated solution for sequential pattern mining (SPM), using Micron's new Automata Processor (AP), which provides native hardware implementation of non-deterministic finite automata. Our proposed solution adopts the algorithm framework of the Generalized Sequential Pattern (*GSP*), based on the downward closure property of frequent sequential patterns. We derive a compact automaton design for matching and counting frequent sequences. A key insight that enables the use of automata for SPM is that we can flatten hierarchical patterns of sequences into strings by using delimiters and place-holders. A multiple-entry NFA strategy is proposed to accommodate variable-structured sequences. Together, 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 solution 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 improvements can be achieved by hardware support to minimize symbol replacement latency. The AP advantage increases with larger datasets, showing good scaling properties for larger datasets while the alternatives scale poorly.

8. ACKNOWLEDGMENTS

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. The authors would like to thank to Prof. Samira Khan, Dept. of Computer Science, University of Virginia for her valuable comments on the manuscript.

References

[1] Micron Automata Processor website, 2015. http://www.micronautomata.com/documentation.

[2] C. C. Aggarwal and J. Han, editors. *Frequent Pattern Mining*. Springer International Publishing, Cham, 2014.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. ICDE'95*, pages 3–14. IEEE, 1995.

[4] S. Cong, J. Han, J. Hoeflinger, and D. Padua. A sampling-based framework for parallel data mining. In *Proc. PPoPP '05*. ACM, 2005.

[5] P. Dlugosch et al. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE TPDS*, 25(12):3088–3098, 2014.

[6] W. Fang et al. Frequent itemset mining on graphics processors. In *Proc. DaMoN '09*, 2009.

[7] P. Fournier-Viger et al. Spmf: A java open-source pattern mining library. *Journal of Machine Learning Research*, 15:3569–3573, 2014.

[8] V. Guralnik and G. Karypis. Parallel tree-projection-based sequence mining algorithms. *Parallel Comput.*, 30(4):443–472, Apr. 2004.

[9] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. SIGMOD '00*. ACM, 2000.

[10] K. Hryniów. Parallel pattern mining-application of gsp algorithm for graphics processing units. In *ICCC '12*, pages 233–236. IEEE, 2012.

[11] H. Noyes. Micron automata processor architecture: Reconfigurable and massively parallel automata processing. In *Proc. of Fifth International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2014. Keynote presentation.

[12] J. Pei et al. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Trans. on Knowl. and Data Eng.*, 16(11):1424–1440, 2004.

[13] I. Roy and S. Aluru. Discovering motifs in biological sequences using the micron automata processor. *IEEE/ACM T COMPUT BI*, 13(1):99–111, 2016.

[14] T. Shintani and M. Kitsuregawa. Mining algorithms for sequential patterns in parallel: Hash based approach. In *Proceedings of the Second Pacific−Asia Conference on Knowledge Discovery and Data mining*, pages 283–294, 1998.

[15] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. EDBT '96*, 1996.

[16] K. Wang, Y. Qi, J. Fox, M. Stan, and K. Skadron. Association rule mining with the micron automata processor. In *Proc. IPDPS '15*, 2015.

[17] M. J. Zaki. Scalable algorithms for association mining. *IEEE Trans. on Knowl. and Data Eng.*, 12(3):372–390, 2000.

[18] M. J. Zaki. Parallel sequence mining on shared-memory machines. *J. Parallel Distrib. Comput.*, 61(3):401–426, 2001.

[19] M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Mach. Learn.*, 42(1-2):31–60, 2001.

[20] F. Zhang, Y. Zhang, and J. D. Bakos. Accelerating frequent itemset mining on graphics processing units. *J. Supercomput.*, 66(1):94–117, 2013.

[21] Y. Zu et al. GPU-based NFA implementation for memory efficient high speed regular expression matching. In *Proc. PPoPP '12*, pages 129–140. ACM, 2012.