Efficient and versatile FPGA acceleration of support counting for stream mining of sequences and frequent itemsets

Adrien Prost-Boucle, Frédéric Pétrot, Vincent Leroy, Hande Alemdar

Eff cient and versatile FPGA acceleration of support counting for stream mining of sequences and frequent itemsets

ADRIEN PROST-BOUCLE and FRÉDÉRIC PÉTROT, TIMA Lab. - CNRS/Université Grenoble-Alpes
VINCENT LEROY and HANDE ALEMDAR, LIG Lab. - CNRS/Université Grenoble-Alpes

Stream processing has become extremely popular for analyzing huge volumes of data for a variety of applications, including IoT, social networks, retail, and software logs analysis. Streams of data are produced continuously, and are mined to extract patterns characterizing the data. A class of data mining algorithm, called generate-and-test, produces a set of candidate patterns that are then evaluated over data. The main challenges of these algorithms are to achieve high throughput, low latency and reduced power consumption. In this paper, we present a novel power-eff cient, fast, and versatile hardware architecture whose objective is to monitor a set of target patterns in order to maintain their frequency over a stream of data. This accelerator can be used to accelerate data mining algorithms including itemsets and sequences mining.

The massive f ne-grain reconf guration capability of FPGA technologies is ideal to implement the high number of pattern detection units needed for these intensive data mining applications. We have thus designed and implemented an IP that features high-density FPGA occupation and high working frequency. We provide detailed description of the IP internal micro-architecture and its actual implementation and optimization for the targeted FPGA resources. We validate our architecture by developing a co-designed implementation of the Apriori Frequent Itemset Mining (FIM) algorithm, and perform numerous experiments against existing hardware and software solutions. We demonstrate that FIM hardware acceleration is particularly eff cient for large and low-density datasets (i.e. long-tailed datasets). Our IP reaches a data throughput of 250 million items/s and monitors up to 11.6k patterns simultaneously, on a prototyping board that overall consumes 24W in the worst case. Furthermore, our hardware accelerator remains generic and can be integrated to other generate and test algorithms.

Additional Key Words and Phrases: Data Mining, Frequent Itemset Mining, Sequence Mining, Stream Mining, FPGA Architecture, Hardware Accelerator, Apriori Algorithm

1. INTRODUCTION

Extracting information from huge collections of unstructured data has taken an increasing importance in the last decades. This mining process takes place in a context in which an increasing amount of data sources are data streams, i.e. data is produced by a continuous and uninterruptible source, leading to a potentially unbounded amount of data.

Many algorithms have been developed to perform batch extraction of patterns (e.g. sets or sequences of items) from f nite datasets, including frequent pattern mining, emerging pattern mining, association rules mining, etc. However most of these algorithms require the ability to store the entire dataset in memory to scan it repeatedly for f ltering and data structure transformation with unconstrained processing time. This is notably the case for mining frequent itemsets and sequences. These algorithms have a high practical interest and have been applied successfully for a variety of applications [Cret et al. 2009; Zaiane 2014; Gu et al. 2016]. These algorithms work under the assumption that the data is organized as a sequence of transactions. Mining frequent itemsets consists of f nding groups of items that are present in an arbitrary order in a suff ciently large number of transactions. Mining sequences is a similar process but then the order of items must be respected.

Batch mining algorithms are designed for the off ine processing of a f xed collection of transactions, and thus are not directly applicable online on data streams. A practical approach for stream processing consists in splitting the stream according to a sliding window based on transaction frontiers. This

This project is being funded in part by Grenoble Alpes Métropole through the Nano2017 Esprit project.
Author's addresses: Adrien Prost-Boucle and Frédéric Pétrot: 46 Avenue Félix Viallet, 38031 Grenoble Cedex, France. Vincent Leroy and Hande Alemdar: CS 40700, 38058 Grenoble Cedex 9, France. Email: <f rstname>.<lastname>@imag.fr

$T_0 = \{a, b\}$
$T_1 = \{a, b, d\}$
$T_2 = \{a, c, d\}$
$T_3 = \{a, b, c, d\}$
$T_4 = \{b\}$
$T_5 = \{c, d\}$

(a) Example dataset

size 1	size 2	size 3
$\{a\} : 4$	$\{a, b\} : 3$	$\{a, b, d\} : 2$
$\{b\} : 4$	$\{a, c\} : 2$	$\{a, c, d\} : 2$
$\{c\} : 3$	$\{a, d\} : 3$	
$\{d\} : 4$	$\{b, d\} : 2$	
	$\{c, d\} : 2$	

(b) Frequent Itemsets

Fig. 1. Example of FIM ($\varepsilon = 2$)

constitutes *mini-batches* which enables to use traditional batch-based mining algorithms. However this requires temporary storage for these mini-batches. The storage capacity and the time data has to be retained depends on the execution time of the actual mining process. Given the interest of doing these kinds of analysis online, our goal is to specify and design a versatile and low power hardware device to accelerate pattern mining. In particular, we focus on the *generate-and-test* class of pattern mining algorithms that operate by generating candidate patterns and evaluating their frequency in the data. Our hardware accelerator loads the set of candidate patterns and maintains a count of their frequency over a stream of data. This accelerator is generic, as it can be applied to a wide range of *generate-and-test* algorithms, including itemset and sequence mining.

Among the algorithms that have been proposed for Frequent Itemset Mining (FIM), *Apriori* [Agrawal et al. 1994], *Eclat* [Zaki et al. 1997], *FP-Growth* [Han et al. 2000], and *LCM* [Uno et al. 2003] are representative of the evolution of the domain. A large body of work has been done at optimizing software implementations [Borgelt 2003], but the required CPU processing power remains very high. As such, the currently known software implementations of these algorithms cannot be used on high throughput, low latency streams, with a reasonable power budget. Apart from LCM, these algorithms feature a high level of fine-grained parallelism and hardware acceleration approaches have been studied. In this work, we evaluate our hardware accelerator with a co-designed implementation of the Apriori algorithm, as it is representative of *generate-and-test* algorithms.

The rest of the paper is organized as follows. Section 2 presents the context of this work by introducing data mining concepts and presenting the characteristics of the main Frequent Itemset Mining algorithms. Section 3 introduces the most relevant works that target hardware acceleration of these algorithms, and outlines their strengths and weaknesses. Section 4 details the specification of a novel hardware architecture aiming at accelerating pattern matching and support counting on FPGA, and Section 5 details its integration with the Apriori algorithm. The implementation is tested and compared to previously published software and hardware/software results in Section 6. Finally, Section 7 summarizes our work and draws some perspectives for enhancement.

2. PRELIMINARIES

2.1. Definition of the Frequent Itemset Mining Problem

A dataset $\mathcal{D}$ is a collection of transactions over a set of items $\mathcal{I}$. We assume the existence of a strict ordering on I. A transaction $T \in \mathcal{D}$ of size n is denoted $T = \{i_1, ..., i_n\}$ where $T \subseteq \mathcal{I}$. Table 1a provides an example dataset of 6 transactions where $\mathcal{I} = \{a, b, c, d\}$. An *itemset* P is a subset of $\mathcal{I}$. A transaction T is an occurrence of P if T contains P, i.e. $P \subseteq T$. The number of occurrences of an itemset in $\mathcal{D}$ is called its *support*, denoted $sup_{\mathcal{D}}(P)$. In our example dataset, $sup_{\mathcal{D}}(\{a, b\}) = 3$. Frequent Itemset Mining (FIM) consists of finding all itemsets whose support in $\mathcal{D}$ is at least ε, along with their exact support. The list of frequent itemsets in our sample dataset for $\varepsilon = 2$ is given in Table 1b.

2.2. Most notable algorithms

Apriori. Agrawal et al. defined the problem of FIM and proposed the Apriori algorithm to solve it [Agrawal et al. 1994]. Apriori enumerates itemsets by increasing length, also called breadth-first

exploration, finding all frequent itemsets of size K (noted K-itemset) before considering $(K+1)$-itemsets. Apriori follows the *generate-and-test* approach. $(K+1)$-itemset P' are generated by adding one frequent item e to the frequent K-itemset P (*i.e.* generate phase). P' is then a *candidate* itemset, and to decide whether it is actually frequent or not, it is necessary to count the number of dataset transactions that do contain P' (*i.e.* test phase). This operation is named *support counting* and it is the most time-consuming part of Apriori, as it requires scanning through $\mathcal{D}$ for each candidate.

The interest of Apriori comes from the fact that for any $(K+1)$-candidate P' to be frequent, all the K-itemsets included in P' must also be frequent. This property is known as the anti-monotony of the support of itemsets. By building the list of $(K+1)$-candidates from the list of actually frequent K-itemsets, the very costly support counting operation is not launched on itemsets which are known to be infrequent. While a dataset $\mathcal{D}$ potentially contains $2^{|I|}$ different itemsets, Apriori avoids in practice exponential complexity by exploiting this property to prune the search space.

Eclat. Zaki et al. proposed Eclat as an alternative to Apriori [Zaki 2000]. Contrary to Apriori, Eclat performs a depth-first exploration of itemsets, and computes the supersets of an itemset before other itemsets of the same size. Eclat uses a vertical representation of the dataset: instead of storing transactions as arrays of items, it computes for each item the list of transactions it belongs to. The transaction list of an itemset P is obtained by intersecting the list of transactions of the parent itemsets of P, and its support is the number of entries it contains. This vertical representation is effectively obtained by a transposition of the dataset structure and it must be done on the whole dataset before starting frequent itemset mining. Contrary to Apriori, Eclat does not store candidates nor previously found frequent itemsets, which makes it a generally fast and memory-efficient algorithm.

FP-Growth. Han et al. introduced the FP-Growth algorithm [Han et al. 2000]. It consists of transforming the dataset into a frequent pattern tree (FP-tree, a tree that contains all itemsets that exists in the dataset). In such a tree, each node represents an item, and the path from the root node to any other node represents an itemset. Each node also contains a counter that is the number of transactions that include the corresponding itemset. Once this is done, the tree can be recursively scanned and the counters checked to decide whether the corresponding itemset is frequent. Software implementations of FP-Growth are generally faster than Apriori but the FP-tree's high memory usage limits its application to large datasets.

LCM. LCM, proposed by Uno et al. [Uno et al. 2005], performs a depth-first itemsets enumeration similar to Eclat. It maintains both a horizontal (transaction based) and vertical (item based) representation of the dataset. The main innovation of LCM is a dataset reduction operation, which creates a significantly compressed version of the dataset before enumerating the supersets of an itemset. While the performance of each FIM algorithm varies depending on the characteristics of the dataset, LCM 3.0 is generally considered as the fastest algorithm available (see section 6.7).

3. STATE OF THE ART IN ACCELERATED FIM

3.1. Apriori

Baker and Prasanna devised an FPGA-based acceleration for Apriori [Baker and Prasanna 2005; Baker and Prasanna 2006]. The systolic array architecture is used for every step of the algorithm: finding pairs of parent itemsets to generate candidates, eliminating candidates using anti-monotony, and computing support of candidates. The hardware proposed in [Baker and Prasanna 2005] is able to load 560 candidates in a Virtex-II Pro XC2VP100 FPGA, and support is evaluated by streaming the dataset. An updated version of the support counting unit groups similar candidates around small CAMs used for item recoding. This enables to overall load more candidates, which reduces the number of iterations through the dataset [Baker and Prasanna 2006]. However they do not indicate the CPU cost of efficiently grouping candidates in order to make the best use of these CAMs.

Unfortunately the results in [Baker and Prasanna 2005; Baker and Prasanna 2006] were based on wrong estimations of the number of candidate itemsets. This has also been noted by [Thoni and Strey 2009] who re-implemented the technique proposed in [Baker and Prasanna 2006]. Although in

[Thoni and Strey 2009] no CAM-based item recoding is implemented, the obtained execution time and FPGA resource utilization per candidate are notably higher.

In [Wen et al. 2008] the FPGA-based platform, HAPPI, is proposed. They a use hardware hash table to limit CPU-FPGA communications and synchronizations. This seems to be very effective on the small synthetic datasets they used but scalability is probably limited due to the hardware nature of their hash table.

GPU acceleration of Apriori is proposed in [Nikam and Meshram 2014]. A bitmap representation of the dataset is used, which does not scale well to large datasets that have long tail distributions (i.e. many items but short transactions). They used only very small datasets in experiments (max. 19 MB) despite using a high-end workstation (two Tesla C2050 GPUs), and absolute execution times are difficult to infer from their figures.

Finally, an implementation based on Micron's Automata processor is proposed in [Wang et al. 2015]. They obtain noticeable acceleration compared to Borgelt's Apriori sortware tool and a multi-thread implementation of the Eclat algorithm. Their approach appears competitive with GPU-accelerated Eclat running on GPU NVidia Kepler K20C [Zhang et al. 2013a].

3.2. Eclat

Zhang et al. proposed to accelerate Eclat using GPU [Zhang et al. 2013a]. Candidates enumeration takes place on the CPU, while support counting operations take place on the GPU. A vertical representation of the dataset is stored in the memory of the GPU as bitsets. The intersection of transactions lists is thus a AND operation taking place on the GPU, and the support is computed by counting the number of 1 bits in the resulting bitset. The authors evaluate their solution on a Nvidia Tesla S1070 GPU, and report a 6 to 30 speedup compared to software FIM implementations.

A similar acceleration was also performed on FPGA [Zhang et al. 2013b]. The main difference with GPU acceleration stems from the lack of memory on the FPGA. Thus, bitsets are stored in main memory and the FPGA only has a small cache. The authors argue that the main bottleneck is the bandwidth of the memory controller, and proposed compression techniques to improve performance.

Shi et al. propose a different FPGA acceleration [Shi et al. 2013]. Transactions are not represented as bitsets, but as lists of integers. Hence, the intersection process, taking place on the FPGA, is more complex than with bitsets.

3.3. FP-growth

Hardware acceleration of FP-Growth with systolic trees has been proposed in [Sun et al. 2008]. They noted that the size of the systolic tree grows more than exponentially with the number of frequent items the tree can handle, which limits hardware accelerators to only a few frequent items, typically four. To overcome this limitation they proposed to generate sub-databases by performing projection of the original database on a few items. These sub-databases have to be processed iteratively with the systolic tree and the results have to be merged together.

Experiments with known datasets were performed in [Sun and Zambreno 2011] with a systolic tree limited to four frequent items. The mining times are compared to a software implementation of FP-Growth. The hardware accelerator speeds mining up when the number of frequent items is low, but there may be no acceleration at all when database projection operations are needed.

In [Bustio et al. 2016] using systolic trees is proposed for stream mining applications. They used datasets with a very low number of items (17 to 75) because of the scalability issue of the systolic tree. Mining also becomes approximate for more than 20 items.

3.4. Analysis

In this work, we focus on accelerating the Apriori algorithm. Indeed, our goal is to design a reusable hardware accelerator for data mining, and the *generate-and-test* approach of Apriori is more generic than other algorithms specifically tailored for FIM. Furthermore, Apriori requires the least data structure transformation to perform mining, even sometimes no transformation at all. All pattern matching and support counting can be performed in parallel on all candidates itemsets, which make it

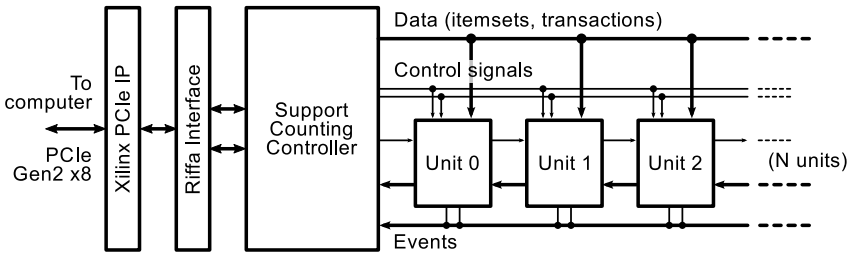

Fig. 2. Hardware accelerator architecture

a prime candidate for a massively parallel hardware accelerator. For stream mining, this also ensures only a minimal amount of buffering is needed which minimizes both hardware requirements and output latency.

Just like the hardware accelerators from previous works, our proposal relies on the knowledge of certain properties of the data to mine (e.g. number of different items), and puts some bounds to the data it can handle at once (e.g. number of transactions).

4. HARDWARE ACCELERATOR ARCHITECTURE AND IMPLEMENTATION

We propose an optimized FPGA accelerator for *generate-and-test* mining algorithms. The bottleneck of such algorithms, including Apriori, is the *support counting* phase, in which the frequency of all candidate patterns is evaluated. Hence, contrary to [Baker and Prasanna 2005; Wen et al. 2008] where hardware resources are used for mechanisms to filter candidate itemsets, we direct our efforts towards the support counting units.

Our proposal stands in three points. First, we design an optimized support counting unit, from the standpoint of FPGA area and speed. Second, we propose a low latency data delivery mechanism. And third, we optimize the placement of the compute part of the accelerator to maximize FPGA efficiency.

Improvements compared to the related works are:

— introduction of a technique to handle de-duplicated transactions in the accelerated support counting phase,
— definition of a very efficient technique to distribute data to support counting units,
— design of a highly optimized implementation for the support counting units,
— definition of an *ad-hoc* algorithm to place a very high number of units in the FPGA.

These contributions are presented in the following sections.

4.1. Hardware accelerator architecture

Figure 2 is an overview of the architecture of our hardware accelerator. It is a PCIe Gen 2 peripheral that provides hardware acceleration of support counting to the host computer.

We integrate our support counting IP with the RIFFA interface framework [Jacobsen et al. 2015]. The RIFFA interface takes care of the setup of the PCIe hard IP embedded in the FPGA and offers simple bidirectional FIFO-based communication channels to our IP. We use two RIFFA channels: the first is dedicated to the read/write operations on the configuration registers inside our IP controller, and the second is dedicated to high-throughput streamed data transfer between the computer and our IP.

The support counting operation is performed by a large number of small and identical support counting units (implementation details are given in section 4.2). Data and control signals are broadcast from the controller to the units using an *ad-hoc* distribution mechanism to cope with *fanout* issues (implementation details are given in section 4.3). Events from units are gathered with a similar *ad-hoc* mechanism (implementation details are given in section 4.4).

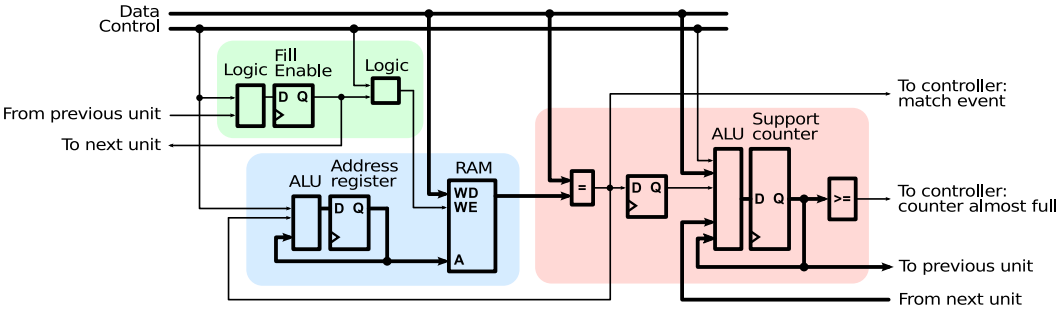

Fig. 3. Support counting unit

Each unit can be programmed at run-time with one candidate pattern, and is dedicated to counting the support of that pattern. To that end, the computer sends the pattern data to our IP controller through the RIFFA channel dedicated to high-throughput transfers. The controller broadcasts this pattern to all units, but a *write enable* token mechanism only enables one unit to store any given pattern. The *write enable* token is sent to the units with a dedicated one-bit scan chain that traverses all units.

Once the computer has transfered the set of all candidate patterns, it sends the dataset through the same RIFFA channel. Our IP controller broadcasts this data stream to all counting units, that perform pattern matching on-the-fly and increment an internal support counter if the pattern is detected in the transaction.

Then the computer asks that all support values are sent back. Our IP controller broadcasts a dedicated control signal to all counting units. The support values are then shifted one by one to the controller, using another dedicated scan chain that traverses all units. The controller transfers this data stream back to the computer through the RIFFA channel.

4.2. Support counting unit

The internal implementation of a single support counting unit is given Figure 3. A lot of care has been taken optimizing every bit of this unit in order to have as many of them as possible fit into the FPGA.

Some elements are now relatively common in the field of FPGA-accelerated FIM [Baker and Prasanna 2005; Baker and Prasanna 2006; Wen et al. 2008]: a small RAM stores the pattern, an item comparator checks for inclusion in the received dataset transactions, and a counter keeps track of the pattern support.

However, the unit we propose differs on several fundamental points:

— the absence of address comparator,
— the simplified item comparator,
— the ability to handle transaction weights,
— the absence of per-unit controller.

As proposed in [Wang et al. 2015] for FIM acceleration with Micron's Automata Processor, we use a special item value as delimiter of the end of patterns and transactions.

An illustration of how our unit works is given in Figure 4. The pattern address register is reset to zero at the beginning of each transaction. This address register is incremented each time the item is matched with the transaction item. For this only an *equal to* item comparator is needed, where the additional functionality *greater than* and/or *less than* was required in [Baker and Prasanna 2005; Wen et al. 2008]. Additionally, the end-of-pattern delimiter value can only be matched at the end of the transaction. If it is matched, it means that all items of the candidate pattern monitored by the support counting unit have also been matched in the transaction. This way, to check if the current transaction supports the pattern, it is sufficient to test if the delimiter item is matched at the end of

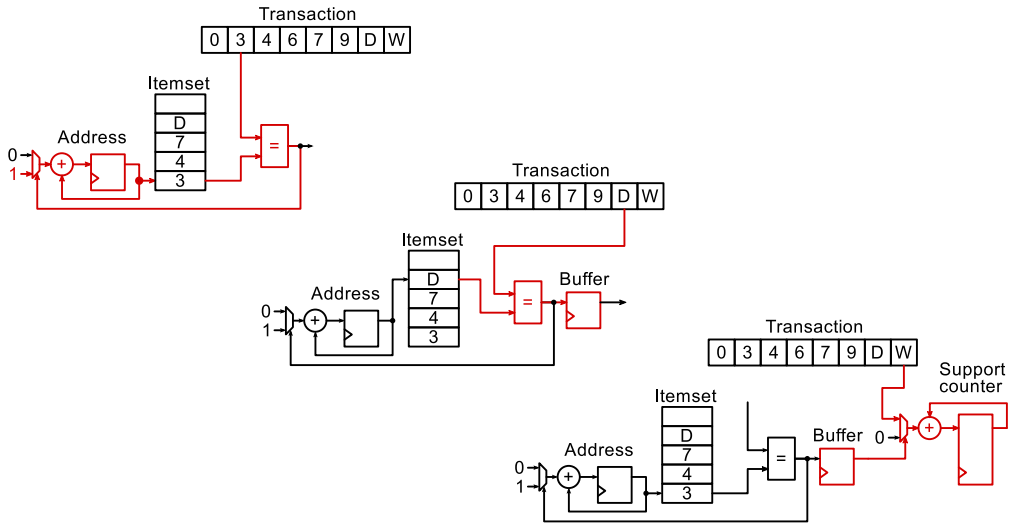

Fig. 4. Unit in action

the transaction. This test works for any pattern size, so there is also no need for another comparator on the address register, unlike proposed in [Baker and Prasanna 2006]. Hence, a single *equal to* item comparator per unit is sufficient.

This approach to detecting the presence of patterns in data transactions is generic and can be easily used in the context of both itemset and sequence mining. Indeed, to match itemsets, it is sufficient to sort candidate patterns and transactions using an arbitrary ordering on items. In the case of sequences, the order of items in candidates and transactions remains unchanged.

It is neither needed to send the pattern size to counting units, nor to store this size in the units. In the case of Apriori, all candidate patterns have the same size. However, the pattern in all counting units of the FPGA can be of different sizes, which makes the proposed implementation completely generic and able to load any set of candidate patterns.

To our knowledge, no related work has been proposed to handle transaction *weights*. Many software implementations perform de-duplication of identical transactions after filtering infrequent items. This operation enables to reduce the dataset size, sometimes by one order of magnitude depending on the characteristics of the data. This is particularly beneficial for hardware acceleration where the dataset is commonly sent one item at a time to the units. In the proposed implementation, the transaction weight is handled and is sent to units at the clock cycle that follows the end of the transaction. The units add the weight to the support counter instead of adding 1.

In fact, the proposed unit features no actual controller. Most control signals are global and are generated by the main support counting controller, as illustrated in Figure 2. This saves many hardware resources in the support counting units. Dedicated data distribution mechanisms are used to efficiently send the global signals to all units.

4.3. Mechanism for data distribution

The proposed implementation does not use a scan chain to send candidate patterns, transactions and control signals to units. This is the method of choice in related works [Baker and Prasanna 2005; Wen et al. 2008]. However despite its apparent simplicity, this solution has two main drawbacks. First, this forms a high-latency pipeline, and second, it requires a high amount of registers, which contributes to the area per unit and to power consumption. We propose an alternative solution that addresses both points.

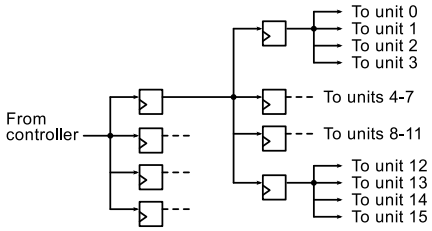

Fig. 5. Distribution tree

The proposed implementation is a pipelined fanout distribution tree, illustrated in Figure 5. It is a tree in which each node is buffered with a register, and where each register is used under a low fanout (we used the value 4). With a fanout of 4, it can be calculated that the average number of registers used per unit is roughly one third of what would be needed by a corresponding dedicated scan chain. This tree also uses no logic resources (e.g. LUT), so these registers fit nicely in the few places that are still unused after compact placement of the units. Hence, the area of this global distribution mechanism is virtually zero.

Our implementation still makes use of two scan chains: a one-bit chain to propagate a *Fill Enable* flag to enable writing patterns into the RAMs of the units, and another chain to send the support values back to the controller. The former uses exactly one register and one LUT per unit, which makes it very low-cost. The latter has a much larger width but it is entirely implemented in the registers of the support counters and the corresponding ALU, so its resource usage is also virtually zero.

The mechanism employed to write the candidate patterns into the units is also independent from the scan chain formed by the support counters. For iterations of the *generate-and-test* algorithm where the number of candidate patterns exceeds the number of available units, this enables to save time by simultaneously writing a batch of patterns and reading the support values of a previous batch of patterns (see section 5.2).

4.4. Mechanism for gathering unit events

Support counters are incremented during mining. To avoid overflow, it is necessary to read and clear them regularly. However mining is stopped when reading counters, which can be a real problem for online mining of uninterruptible streams.

In order to read counters only when strictly necessary, the hardware mining controller must know when a counter is almost full. To that end, one small comparator can be added to the matching unit to detect when the most significant bits of the counter cross a given threshold (only one LUT is enough for this). The value of the threshold can be set in the LUT at synthesis time, or at run-time at the cost of an additional dedicated small data distribution tree to send the threshold value to the units.

Similarly, it is important to know as early as possible when a pattern is found in the mined stream. That way the controlling application can store only the parts of the stream that do contain monitored patterns, for later more thorough analysis. To that end, the mining controller has to know as early as possible when any of the monitored patterns is found in the mined dataset/stream transactions. This event can also serve as wake-up signal for any controlling software, so for power-saving purposes the CPU (if any) can be kept in its lowest-power sleep mode most of the time. For FIM acceleration, this signal can also be used to reduce the dataset size between iterations of the Apriori algorithm by removing unuseful items and transactions.

So each unit can output two event signals (if enabled at synthesis time). These signals are gathered and routed back to the mining controller using a hardware tree similar to the data distribution tree described in section 4.3, but where signals pass in the other direction. Each node of the tree consists of one OR logic gate (inside one LUT) and one register. This effectively performs a pipelined OR reduction on the signals that come from all counting units. One instance of that tree is used to

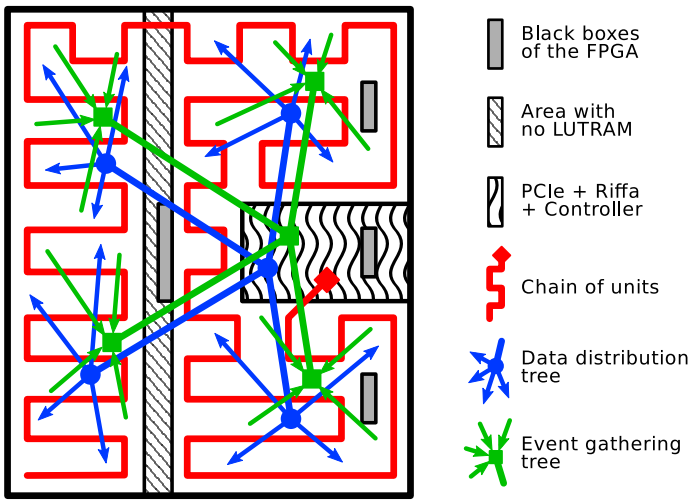

Fig. 6. Global FPGA organization

gather the signals for counter almost full, and another instance is used to gather the output of the comparators inside the units (according to whether these functionalities are enabled at synthesis time).

With this lightweight event gathering tree mechanism, the mining controller receives events with a 4 to 6 clock cycles latency, which is very short. The controlling software running on the host computer (if any) can also receive these events with only a few microseconds latency (e.g. for PCI-Express), with no interruption of the data stream.

4.5. Global FPGA organization

Our implementation is tuned for the Xilinx VC709 FPGA board, so the presented optimizations cannot be directly applied on other targets and particularly for other FPGA vendors. However, similar per-target optimizations should be feasible.

The global FPGA organization is illustrated Figure 6. The RIFFA framework [Jacobsen et al. 2015] is used as PCI-Express communication interface with the computer. Our support counting controller and the RIFFA interface are placed close to the hard PCI-Express endpoint. The controller only needs to drive one side of the chain of support counting units, so there is a lot of freedom to place that chain through the FPGA while filling all gaps as much as possible.

Unfortunately, the fully automated Vivado place-and-route suite fails to automatically infer the topology of the connections between the units and the controller, and it fails to place the units close to each other. Overall, Vivado fits 25% less matchers in our FPGA than theoretically possible, and the resulting designs work at a frequency 20% lower than our 250 MHz target (linked to PCIe interface). This under-utilizes the FPGA capabilities by around 40%, which is very significant.

In order to fully exploit the FPGA capabilities and to get closer to the theoretical limits, a large part of the design placement is predefined and is performed by *ad-hoc* scripts, which guides the Vivado place-and-route steps. This is achieved by describing the unit entity at netlist level by explicitly instantiating the LUT and slice register primitives. The internal elements of the units are then largely untouched by logic synthesis, and they can be found again later and manipulated with scripts. The automated placer is left with the task of placing the controller, the data distribution tree and the event gathering trees (if any). The very short distance between the pre-placed unit elements ensures a high working frequency and makes the task of the automated router much easier.

However there are thousands of units to place with scripts and this is only possible with a high level of regularity. This is achieved by using a particular placement layout template where two units

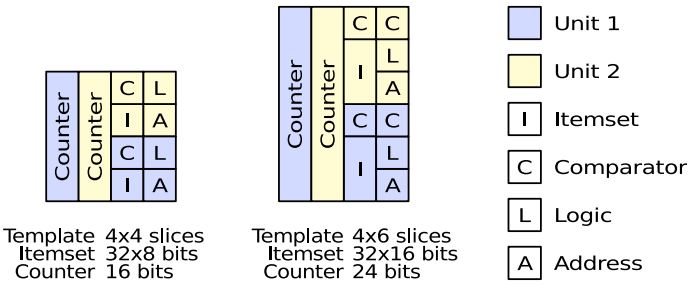

Fig. 7. Examples of placement layout templates

Table I. Unit details for various pattern heights

Template height (slices)	# Units in FPGA	Pattern size	Item (bits)	Counter (bits)	Event gathering tree
4	11636	32	8	16	no
5	9360	32	8	20	yes
		32	12	19	no
		32	12	16	yes
		32	16	16	no
		64	8	20	no
6	7788	32	12	24	yes
		32	16	24	no
		64	8	24	yes
7	6644	32	16	28	yes
		64	12	28	yes
8	5818	32	24	32	yes
		32	32	28	no
		64	16	28	yes

are packed together. The template is rectangular, 4 FPGA slices large. The optimal template height (in slices) depends on the width of the support counter and of the items to process, in bits. Examples of templates are illustrated Figure 7 for a maximum itemset size of 32, including the delimiter item. With 16-bit support counters and 8-bit items, two units fit inside a 4x4 slices template. With 24-bit support counters and 16-bit items, two units fit inside a 4x6 slices template. In both cases, strictly all LUT primitives of the template are used, which makes these two templates optimal. Approximately half of the slice registers are either used or unusable because of restrictions due to LUTRAM usage in the same slice. The remaining registers are needed for the global distribution tree, with a welcomed little margin because this tree is placed and routed with the automated Vivado design suite.

Our placement scripts are not fully automatic. We created one basic placement routine that fills a specified rectangle area by placing units using a zig-zag path. To completely fill our FPGA, this routine is called around 10 to 15 times, with manually-specified coordinates, for each supported template height value. We felt this was an acceptable enough solution to efficiently handle the FPGA black boxes and other local non-uniformities of the reconfigurable fabric. Devising a fully automatic algorithm to place our chain of units should be possible, but this is out of the scope of this paper.

This technique enables to very densely tile the available reconfigurable FPGA area. Table I presents some of the possible combinations of unit characteristics for several height values, given in FPGA slices. The pattern size includes the delimiter item. Note that many other combinations are possible, including some that are locally optimal, but they are not shown for clarity. For each combination, the corresponding FPGA configuration can be generated. This operation can take hours for each configuration because of the size of the FPGA. However it is only done once so this effort is amortized. For FIM operations, the most appropriate configuration can be selected and programmed onto the FPGA. With high-speed data transfer through PCI-Express or from on-board DDR memory,

the entire FPGA can be reconfigured in as low as 72 ms. Also once programmed, the FPGA can be used for any number of different datasets, so the FPGA programming time is generally not an issue.

There is however an area of the FPGA without LUTRAM primitives. It represents approximately 6% of the total reconfigurable FPGA area. LUTRAMs are needed for the itemset RAM of each unit, so no unit can be placed in that area. Even with this limitation, and with some area margin taken to ease placement and routing of the large design section *PCIe+RIFFA+Controller*, the global LUT usage is around 90%. This is exceptionally high compared to the usual objective of 80% for large FPGA designs. With a global working frequency of at least 250 MHz, the proposed design architecture enables to very efficiently use the FPGA resources.

4.6. Data throughput

As one item is processed per clock cycle, with our 250 MHz implementation the maximum throughput ranges from 250 MB/s for 8-bit item encoding to 1 GB/s for 32-bit item encoding, regardless of the fact that it is a dataset being processed offline or a data stream being monitored online.

We highlight the fact that during Apriori FIM, our hardware accelerator behaves strictly the same way as it would for online sequence matching on an uninterruptible data stream.

5. APPLICATION: APRIORI ACCELERATION

We demonstrate the efficiency of our architecture using the batch Apriori FIM application, because the underlying *generate-and-test* approach can be used for a variety of mining tasks, including sequence mining, whether from offline batch or online stream. There are also many implementations to compare to, in hardware as well as in software, which is not the case for stream sequence mining.

Note that the dataset filtering and sorting steps and the need for dataset re-scanning are specific to batch-based applications. However, the acceleration of the core support counting functionality of the Apriori FIM application is representative of stream mining (user-specified sequences or itemsets).

5.1. Dataset loading and filtering

In the usual software FIM world, the dataset loading steps are the following:

(1) read the dataset file from disk,
(2) remove infrequent items,
(3) sort transactions so items are always in the same order,
(4) de-duplicate identical transactions.

Some of these operations may be performed in a different order, particularly with trie-based storage of transactions.

The usual FPGA-accelerated Apriori world presents two differences:

— items are re-coded using the range 0 to $\#items - 1$,
— identical transactions are not de-duplicated.

Re-coding items enables to use hardware units tailored to just the minimum number of useful bits, and to reduce the size of data transfers between the computer and the accelerator. The reason for not de-duplicating identical transactions is generally due to hardware constraints: handling a per-transaction *weight* requires more complex data transfer protocols, hence an increased design complexity, and usually larger support counting units, which reduces the number of units that can fit in the FPGA.

However in the proposed approach, the hardware unit is able to handle per-transaction weight at virtually no hardware cost, while still having a very simple implementation. De-duplication of identical transactions can be performed and, on some datasets, the amount of data transfers is strongly reduced, which brings a very high speedup.

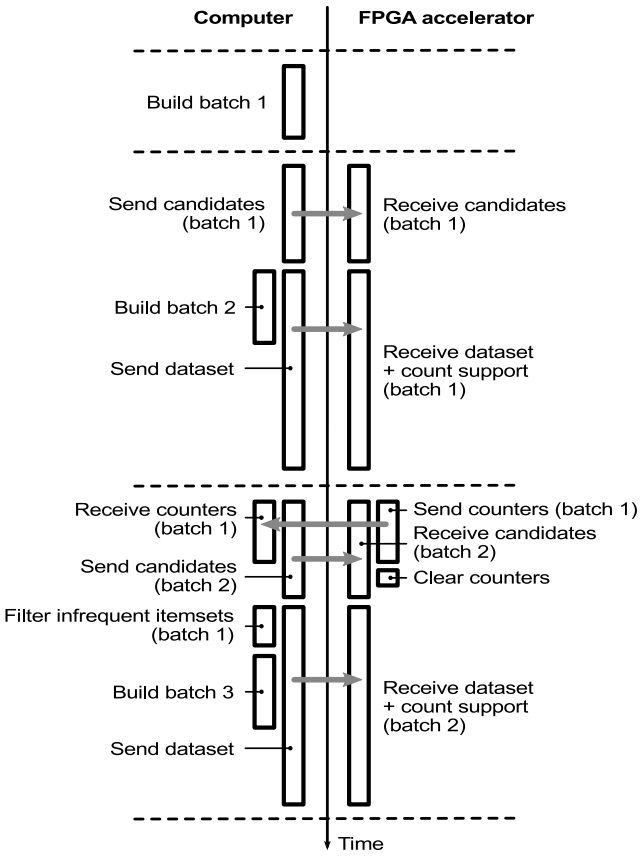

Fig. 8. Pipelined processing of batches of candidate itemsets

5.2. HW / SW parallelism

When the number of candidate $(K+1)$-itemsets is higher than the number of support counting units, the candidates are grouped into several batches that are iteratively sent to the FPGA. The order of candidates is not relevant, so they are simply grouped by following the order in which the generation algorithm produces them. Each batch is created and processed in several steps:

(1) build a batch of candidates,
(2) send the batch of candidates to the FPGA,
(3) send the dataset to the FPGA,
(4) get the support values from the FPGA,
(5) drop infrequent itemsets.

The processing of batches is highly pipelined in order to perform as many software tasks as possible during data transfers. From the software side, data transfers are actually handled by the RIFFA driver and performed by the autonomous DMA of the processor. This uses nearly no CPU time so it is done in parallel with the other software tasks.

Figure 8 illustrates how these steps are orchestrated between the software and hardware sides, and shows how the processing pipeline is initialized. Tasks are represented by boxes and data transfers are highlighted with large grey arrows.

Building batches of candidates $(K+1)$-itemsets is performed using a well-known technique, briefly described here. The frequent K-itemsets obtained at the previous iteration of the Apriori

algorithm are scanned. From two K-itemsets of the form $P' \cup \{a\}$ and $P' \cup \{b\}$, where P' is a frequent $(K-1)$-itemsets, the candidate $P' \cup \{a, b\}$ is built. This candidate is also filtered by checking that all other K-itemsets it contains are also frequent. This is done with a software hash table that contains hashes of all frequent K-itemsets, which is both very fast and scalable.

The support counting process begins by building the first batch of candidates of size $K+1$. Then this first batch is sent to the FPGA, followed by the dataset for support counting. While the dataset is being sent, the computer builds the second batch of candidates.

The support counters of the first batch are sent back to the computer, which is simultaneously sending the second batch to the FPGA.

When both operations are finished, the dataset is sent again to the FPGA for support counting of the second batch. In the meantime, the computer drops infrequent itemsets from the first batch, then it builds the third batch while the dataset is still being sent.

The processing pipeline is now initialized and batches are processed with maximum computer-accelerator parallelism. This process continues until all candidates have been either filtered out or processed in the batches.

5.3. Splitting large datasets

Similarly to the fact that there can be more candidates than there are hardware counting units, which forces to process candidates into several batches, the range of the hardware counters can be shorter than what would be needed to hold the worst-case itemset frequency.

To that end, after loading and filtering the dataset, the frequencies of all items are considered. Indeed, no itemset can be more frequent than any of its items. In case the hardware counters are too short to hold this worst-case itemset frequency, the dataset is split into several partial datasets such that for each one, the worst-case itemset frequency is guaranteed to fit in the hardware counters. At support counting time, the dataset is not sent to the FPGA in one go as previously illustrated Figure 8. Instead, the partial datasets are sent one at a time. Hardware counters are also read and cleared between partial datasets.

However, there might be transactions whose weight value can't fit in the hardware counters, even taken alone. In this situation, these transactions are isolated and each forms a partial dataset. Such special partial dataset is sent to the FPGA with transaction weight forced to 1, and when later receiving support values, the values are multiplied by the original transaction weight. This technique enables to use hardware units with short counters even for very large datasets. This enables to instantiate in the FPGA a higher number of units compared to using counters large enough, potentially speeding up the support counting operation.

6. RESULTS

6.1. Datasets

Datasets chosen from related works are used in order to compare the performance of our solution against other recent implementations. Details are given in Table II.

T10I4D100K and *T40I10D100K* are standard synthetic datasets from [FIMI Repository 2003], used in [Baker and Prasanna 2005; Baker and Prasanna 2006; Thoni and Strey 2009]. This enables to compare against competing FPGA implementations of the Apriori algorithm.

Pumsb, *accidents* and *webdocs* are standard real-world datasets from [FIMI Repository 2003], *webdocs5x* is a synthetic dataset obtained by replicating *webdocs* 5 times. They are used in [Wang et al. 2015], enabling to compare against an implementation of the Apriori algorithm using Micron Automata Processor.

T40I10D03N500K, *T40I10D03N1000K*, *T60I20D05N500K* and *T90I20D05N500K* are used in [Zhang et al. 2013b]. This enables to compare against an FPGA-accelerated implementation of the Eclat algorithm. The datasets are generated with the tool IBM Market-Basket Synthetic Data Generator [IBM 2012].

Table II. Datasets

Dataset	# Trans.	# Items	Avg. length of trans.	Size
T10I4D100K	100k	870	10.1	3.8 MB
T40I10D100K	100k	942	39.6	14.8 MB
pumsb	49046	2113	74	15.9 MB
accidents	340183	468	33.8	33.9 MB
webdocs	1692082	5267656	177.2	1.38 GB
webdocs5x	8460410	5267656	177.2	6.90 GB
T40I10D03N500K	500k	299	40	63 MB
T40I10D03N1000K	1000k	300	40	134 MB
T60I20D05N500K	500k	500	60	104 MB
T90I20D05N500K	500k	499	90	155 MB
chess	3196	75	37	334 kB
BMS-WebView-2	77512	3340	4.62	2.20 MB
connect	67557	129	43	8.80 MB
BMS-POS	515596	1657	6.53	11.3 MB
kosarak	990002	41270	8.10	30.5 MB

The five other datasets, *chess*, *BMS-WebView-2*, *connect*, *BMS-POS* and *kosarak* are used in [Sun and Zambreno 2011]. This enables to compare against an FPGA-accelerated implementation of the FP-Growth algorithm.

6.2. Workstation

All experiments are performed on a Dell Precision T3500 workstation. It is equipped with an Intel Xeon W3530 processor (2.8 GHz, 8 MB cache) and 12 GB of RAM. The processor is a 4-cores, 8-CPU model but all experiments are run using only one CPU. Our VC709 FPGA board is directly plugged in a PCIe2 8x motherboard slot. This board is equipped with the Xilinx FPGA XC7VX690T. It is also equipped with 8 GB of dedicated RAM, but this RAM is not used by our acceleration technique. For clarity, only three FPGA configurations are used:

— 11636 units, 8-bit items, 16-bit counters,
— 9360 units, 12-bit items, 19-bit counters,
— 7788 units, 16-bit items, 24-bit counters.

All configurations have itemset size 32, including the special delimiter item value, and the special even gathering tree was not used.

In order to reduce the dependency of results upon disk I/O speed, each dataset is pre-loaded into main RAM before any FIM operation, accelerated or not.

We highlight that the available PCIe data throughput largely exceeds our needs: our FPGA IP processes one item per clock cycle, so for 16-bit items only 500 MB/s are required, whereas the PCIe 2.0 8x interface provides up to 4 GB/s.

6.3. Comparison with FPGA-accelerated FP-Growth

We compare with [Sun and Zambreno 2011] as they provide results with several datasets and broad ranges of support values. They implemented a systolic tree of depth 4, which means 4 frequent items. Handling more frequent items is done by performing dataset projections. The transactions must be sorted before sending them to the systolic tree, this is a light limitation we also have.

Table III shows execution times for their approach and ours for a few support values. It is important to highlight that the datasets are small and dense, and the support values that are used lead to millions, sometimes billions of frequent itemsets. Obviously in this kind of situation only the number of itemsets can be counted. There is even a peak at 1.4×10^{17} for *BMS-WebView-2* with minimum support 2. However during generation, the Apriori algorithm stores all K-itemsets simultaneously in main RAM memory. So for extreme amounts of itemsets, RAM requirements are often excessive and our approach fails (noted *swap* in the table). Anyhow, as the number of itemsets increases, it becomes

Table III. Execution time compared to systolic tree (in seconds)

Dataset	Support	# Itemsets	Time, systolic tree	Time, VC709	Time, LCM 3.0
chess	1000	29,442,848	20	9.41	0.23
	300	5,689,107,303	1545	*swap*	10.4
BMS-WebView-2	4	60,193,073	25	30	1.07
	2	144,256,300,227,727,150	403	*swap*	7.27
connect	30000	188,117,389	78	67.5	0.16
	20000	1,408,869,383	868	*swap*	0.30
BMS-POS	1000	29,490	15	0.53	0.61
	100	5,711,447	63	20.2	2.39
pumsb	30000	16,029,969	20.2	3.66	0.24
	21000	1,447,453,702	3000	*swap*	4.88
kosarak	2000	34,483	25	1.21	0.90
	1000	711,424	52	5.45	1.45

Table IV. Execution time compared to accelerated Eclat (in seconds)

Dataset	Support	Zhang2013	VC709
T40I10D03N500K	1%	12.1	51.1
T40I10D03N1000K	2%	5.71	16.4
T60I20D05N500K	2%	8.75	45.6
T90I20D05N500K	5%	16.6	120

difficult, if not impossible, to perform analyses and draw useful conclusions from an application point of view.

Depending on the dataset, our approach achieves better or similar performance than the systolic tree. Interestingly, our speedup is the highest when the number of frequent itemsets is reasonable (datasets *BMS-POS* and *kosarak*). The bottleneck of their approach is probably the dataset projections they need to perform, which are CPU tasks. When there are relatively few itemsets to find, the cost of these projections is not amortized. So it makes sense to also compare against one of the fastest software tools, LCM 3.0 (see our comparison of SW tools in section 6.7). Unsurprisingly for these small and dense datasets, the depth-first, memory-efficient LCM 3.0 approach is much faster than the systolic tree (and faster than our approach too). Hardware acceleration is probably now only pertinent for much larger, lower-density datasets. Fortunately, these so called long-tailed dataset are very frequent, in particular in Web data and retail data [Anderson 2006].

6.4. Comparison with FPGA-accelerated Eclat

Table IV shows execution times of our VC709 platform, compared to an FPGA-accelerated implementation of the Eclat algorithm [Zhang et al. 2013b]. The configuration with 9360 units is used for all datasets except *T40I10D03N1000K* where the 11636-units configuration is used due to the low number of frequent items. For these datasets and support values, their platform seems to perform much better than ours with speedups ranging from 2.87x to 7.22x.

However, the underlying hardware is different. Our Xilinx Virtex-7 xc7vx609t FPGA is composed of 108k logic slices, they use four Altera Statix III EP3SE260 FPGAs, each having 102k Adaptive Logic Modules (ALM). For our support counting implementation, each Xilinx slice is roughly equivalent to 2 Altera ALMs. So with the amount of hardware resources available on their platform, our acceleration technique would run twice faster. They also use large on-board RAM with low-latency access. This makes their platform more complex and probably more power-consuming.

The very limited test set makes it difficult to further compare both platforms. It would also be interesting to know how fast the on-board RAM limit is reached and mining fails, like in [Wang et al. 2015] with dataset *T100D20M*. Our accelerator is not limited by on-board RAM.

6.5. Comparison with Micron Automata

In Figures 9 and 10 the support counting times with Micron Automata Processor board (noted AP) and of our FPGA-accelerated platform (noted VC709) are shown. We use counting time instead of

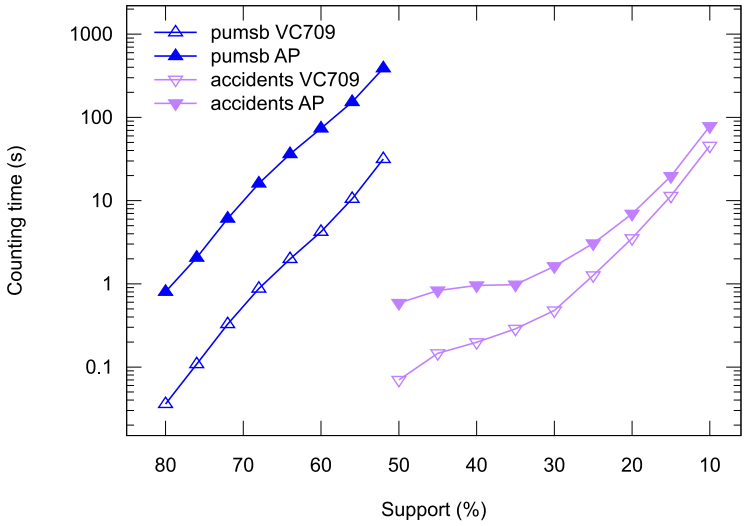

Fig. 9. VC709 vs AP - Datasets pumsb and accidents

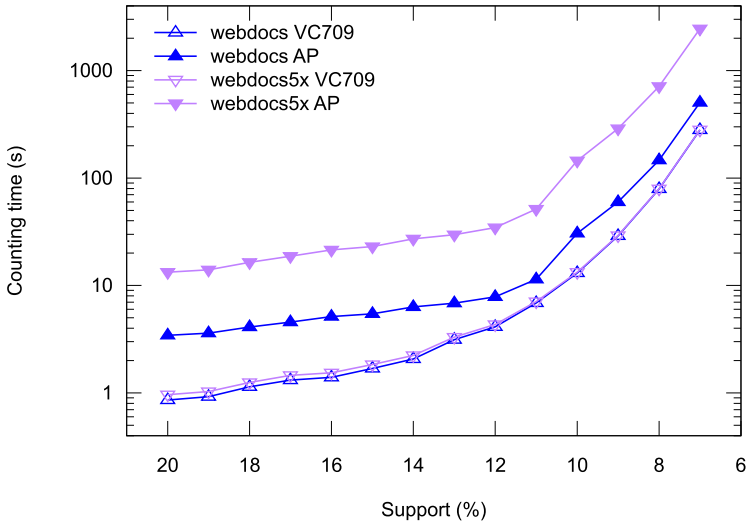

Fig. 10. VC709 vs AP - Datasets webdocs and webdocs5x

overall execution time because only counting time could be precisely extracted from the reference paper [Wang et al. 2015].

The configuration with 11636 units is used for *pumsb* and *accidents*, and 7788 units are used for *webdocs* and *webdocs5x* It can be observed that our platform is much faster than AP for all datasets and for all support values, with a maximum speedup of 12.3x for dataset *pumsb* (support 52%).

This performance gap has several explanations:

— number of counting units and clock frequency,
— symbol replacement time,

Table V. Execution time compared to previous FPGA-accelerated Apriori (support counting only)

Dataset	Support	Previous design		Proposed design		Speedup
		Time (ms)	# Passes	Time (ms)	# Passes	
T10I4D100K	150	1888	288	196	41	9.63
T10I4D100K	200	1699	261	179	37	9.49
T10I4D100K	300	1423	221	146	31	9.75
T10I4D100K	500	918	149	97.4	22	9.43
T10I4D100K	1000	344	64	38.5	9	8.94
T40I10D100K	1000	12464	532	1200	72	10.4
T40I10D100K	5000	693	42	69.6	6	9.96

— de-duplication of dataset transactions.

The AP board is clocked at 133 MHz and is configured with 18432 counting units. Our VC709 board is clocked at 250 MHz, so with the 11636-units configuration it is theoretically faster than the AP board by 18.7%. Also, when using 9-bit to 16-bit item encoding, the item rate is halved on the AP board. On our VC709 board the item rate is unchanged, so we provide a theoretical speedup of 90.9% with 9-bit to 12-bit encodings (9360 units) and 58.8% with 13-bit to 16-bit encodings (7788 units). Our FPGA also supports virtually any item width, whereas it is unknown if more that 16-bit items can be used one the AP board.

The symbol replacement time, in the worst case, is 45 ms for the AP board, but only 1.49 ms for the VC709 board with the 11636-units configuration. However this parameter should only be relevant for very small filtered datasets, *e.g. pumsb*.

The AP board can only increment support counters by 1 when counting. This is the main drawback of their implementation and perhaps of the hardware itself. Our FPGA counting unit is specifically designed to enable adding transaction *weights* to support counters. So during dataset loading, we perform de-duplication of filtered transactions. The size of the dataset *pumsb* is then reduced by the factor 4.88x for support 52%. This de-duplication feature has a direct consequence on the time needed to send the dataset to the accelerator. Similarly for the dataset *webdocs5x* (created by concatenating *webdocs* 5 times), there is a minimum guaranteed speedup of 5x thanks to de-duplication.

The advantages of the VC709 board over the AP board are not limited to counting speed. The AP board is composed of 48 Automata Processor chips, where each has a 150 mm² die size on 50 nm technology node and a 4 W power consumption [Micron Technology, Inc. 2013]. This is a total of 7200 mm² of die size and 192 W, not including other board components and power supplies. The power consumption of a development board is also announced at 300 W [Micron, Inc. 2016].

In comparison, the power consumption of our entire VC709 board, which includes power supplies and the unnecessary on-board 8 GB RAM, was measured at 24 W (peak). Xilinx does not publish the die size for our exact board FPGA and package, but for reference the packages with same footprint size 45x45 mm have maximum die size 23.85x21.65 mm which is 516 mm² [Xilinx Inc. 2015]. Our FPGA is manufactured on 28 nm node, so even if the AP chips are now manufactured on 45 nm node, it is safe to assume that the VC709 FPGA die size is smaller by an order of magnitude. So for FIM acceleration, our FPGA solution is not only much faster than the AP board, it also presents a performance per watt level one order of magnitude higher, and it uses much more common and certainly more affordable hardware.

6.6. Comparison with previous FPGA-accelerated Apriori

We compare against [Thoni and Strey 2009] because their target FPGA technology (Xilinx Virtex-5) is very close to ours (Xilinx Virtex-7). They also noted that the previous results in [Baker and Prasanna 2006] were wrong. Moreover, the results from [Baker and Prasanna 2005] can also be verified to be wrong due to erroneous estimation of the numbers of candidates. So the works from [Thoni and Strey 2009] are the most pertinent. Two datasets from [FIMI Repository 2003] are used: *T10I4D100K* and *T40I10D100K*.

We use the configuration with 9360 counting units, all working at 250 MHz. They use 1116 counting units working at 170 MHz. Like ours, their architecture is able to process one item per clock cycle. This brings a theoretical, ideal speedup of 12.3x for our platform. The results are given in Table V. We measure only counting time instead of overall execution time because it is the metric used in [Thoni and Strey 2009]. The actual average speedup is 9.6x, which is lower than the ideal speedup. This is because for the last iterations of the Apriori algorithm, the number of candidates is lower than the number of counting units, so the FPGA is under-utilized. Actually the datasets used are very small and this effect would have a much lower impact with larger ones. However if considering only the number of passes, the average speedup should be only 7.1x, but our speedup is higher than that. This is partly due to our transaction de-duplication technique, which reduces the dataset size by 11% to 12%. The low amount of details given in [Thoni and Strey 2009] makes it difficult to explain where the rest comes from, but it may be due to our design allowing to simultaneously read the support counters and send another batch of candidates.

It is also pertinent to compare the resource usage and the functionalities of the proposed counting units. Unfortunately in [Thoni and Strey 2009] and [Baker and Prasanna 2006], neither the counter width nor the maximum itemset size are given. Similarly in [Baker and Prasanna 2005] the counter width is not given. As previously shown in section 4.5 these parameters have a strong impact on the size of counting units, which makes the comparison difficult.

Our placement template (see section 4.5) is 5 slices high, which means 40 LUT6 per counting unit. This is a 29.2% improvement compared to the best case 56.5 LUT6 achieved in [Thoni and Strey 2009]. Using much shorter counters would also be possible with our solution and for instance, for the two datasets used, 15-bit support counters is the minimum that does not require dataset splitting. This would make our counting units use only 36 LUT6, which would be a 36.3% improvement. And even with our 6-slices high template (24-bit counters, 16-bit items) which is largely oversized for the needs, the resource usage is 48 LUT6, which still represents a 15.1% improvement.

The scalability of designs can also be compared. In [Thoni and Strey 2009] the CAMs are 10-bit deep which corresponds to 10-bit items. Should we use our best configuration for 10-bit items, our solution would be much more efficient in terms of logic resources utilization. Regarding memory needs, their requirement in CAM resources (hence as RAM blocks) grows linearly with the number of frequent items their hardware can handle. Their FPGA can theoretically contain 11-bit items with 80% block RAM utilization, but beyond that the RAM resources are limiting. It can be calculated that for 16-bit items the number of counting units they can implement in their FPGA is divided by at least 32. So our solution scales much better for large numbers of frequent items.

Finally, for FIM, their implementation has an advantage: it is inherently insensitive to item order, so it does not require that transactions are sorted (although in practice sorting transactions has a relatively low impact on overall mining time). However because of that insensitivity, they can't do sequence mining.

6.7. Comparison with software tools

It is very common in related works to compare the execution speed of a hardware-accelerated platform against the pure software implementation of the same algorithm being accelerated. But we consider that FIM users are mainly interested in execution speed and/or energy consumption, and not in the nature of the algorithm that is internally used. This position is relevant especially as Borgelt's Eclat actually internally selects between several variants of Eclat and LCM algorithms.

Table VI gives run times for several recent software FIM tools, and for datasets used in related works. Borgelt's Apriori and Eclat tools are often used as reference software tools in related works about hardware-accelerated FIM. For each tool, we use the latest version available at the time of this writing. Borgelt's Apriori is used in [Wang et al. 2015][Baker and Prasanna 2005][Baker and Prasanna 2006], we use version 6.19 (2015.08.18). Borgelt's Eclat is used in [Zhang et al. 2013b][Shi et al. 2013], we use version 5.8 (2015.08.18). We use version 6.8 (2015.12.22) of Borgelt's FP-Growth. To the best of our knowledge, LCM 3.0 [Uno et al. 2005] has never been used as reference

Table VI. Execution time compared to software tools (in seconds)

Dataset	Support	Borgelt Apriori	Borgelt Eclat	Borgelt FP-Growth	LCM 3.0	VC709
T10I4D100K	0.15%	0.433	**0.192**	0.300	0.200	0.235
T40I10D100K	1%	4.46	1.53	4.18	**1.50**	1.28
pumsb	52%	537	1.24	1.27	**0.837**	31.8
accidents	10%	10480	5.66	4.22	**2.90**	45.8
webdocs	7%	18500	191	*swap*	**84.3**	286
webdocs5x	7%	18710	414	*swap*	**280**	322
T40I10D03N500K	1%	292	91.5	234	**72.9**	**51.1**
T40I10D03N1000K	2%	101	86.6	*swap*	**62.8**	**16.4**
T60I20D05N500K	2%	197	103	*swap*	**79.4**	**45.6**
T90I20D05N500K	5%	1428	419	*swap*	**249**	**120**

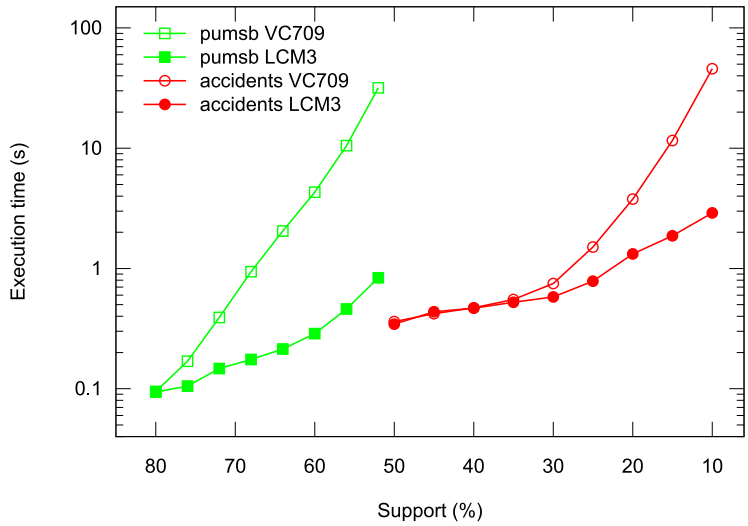

Fig. 11. VC709 vs LCM 3.0 - Datasets pumsb and accidents

for hardware-accelerated FIM works, despite it being known for its efficient implementation of the LCM algorithm.

In Table VI, the time of the fastest software tool is highlighted for each dataset. We indicate *swap* when the memory requirements exceed the 12 GB RAM of our workstation. LCM 3.0 largely outperforms all other tools on most datasets and this trend can be confirmed on much broader ranges of support values. Only for the tiny datasets *T10I4D100K* and *T40I10D100K* Borgelt's Eclat tool is very close to LCM 3.0. For this reason, we perform a more detailed comparison of our VC709 platform against LCM 3.0. The time of our VC709 board is highlighted when it outperforms the software tools.

In Figures 11 and 12 the overall execution times of LCM 3.0 and of our FPGA-accelerated platform are shown. The 4 datasets from [Wang et al. 2015] are used with the same support ranges: *pumsb*, *accidents*, *webdocs* and *webdocs5x*.

LCM 3.0 is still notably faster than our platform for dataset *pumsb*. This dataset is known to be exceptionally dense: a huge number of itemsets is produced from a very small dataset. The filtered dataset is actually small enough to entirely fit in the processor cache, which makes the depth-first, memory-efficient LCM algorithm almost unbeatable. Similarly, LCM 3.0 clearly leads for dataset *accidents* for support values below 35%. For higher support values, mining time is negligible compared to dataset loading and filtering time, for both tools.

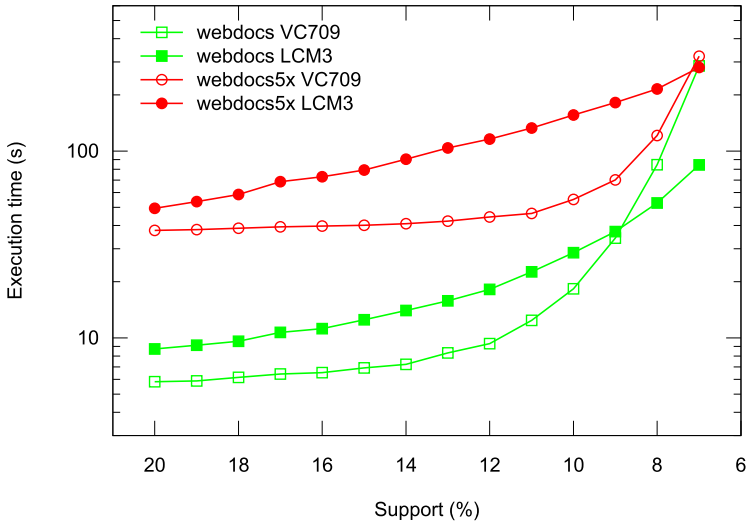

Fig. 12. VC709 vs LCM 3.0 - Datasets webdocs and webdocs5x

Table VII. Distribution of execution time

Dataset	pumsb		accidents		webdocs	
Support	80%	52%	20%	10%	10%	7%
Dataset loading + filtering	65.3%	0.17%	7.58%	0.58%	31.3%	2.07%
FPGA receives candidates	5.33%	13.2%	1.64%	0.84%	0.04%	0.06%
FPGA receives dataset	2.11%	37.8%	88.3%	95.7%	67.4%	97.2%
FPGA sends last counters	0.17%	0%	0%	0%	0%	0%
PCIe latency	14.4%	33.4%	1.63%	1.39%	0.34%	0.43%
Software between iterations	4.91%	10.9%	0.79%	0.89%	0.04%	0.08%
Other	7.78%	4.53%	0.06%	0.60%	0.88%	0.16%
Total	100%	100%	100%	100%	100%	100%
Concurrent CPU usage	40.5%	44.7%	2.86%	3.58%	2.45%	1.96%

Nevertheless, our VC709 platform is faster than LCM 3.0 on a large range of support values for the large and sparse datasets *webdocs* and *webdocs5x*, with a maximum speedup of 1.95x for *webdocs* (support 12%) and 2.87x for *webdocs5x* (support 11%). Our platform also outperforms LCM 3.0 on the 4 datasets from [Zhang et al. 2013b], with speedups ranging from 1.11x for dataset *T40I10D03N500K* to 3.51x for dataset *T40I10D03N1000K*.

However according to our figures, LCM 3.0 still presents a better scalability for low support values. For the 7% support in Figure 12, there are more than 5 million generated itemsets, which makes it questionable whether such a high number is of any practical interest. This is even worse with datasets *pumsb*, support 52%, and *accidents*, support 10%, for which the number of frequent itemsets is more than 98 million and 10 million, respectively.

We highlight that, with the linear scalability of our FPGA design, using larger FPGAs like the Virtex-7 2000T would directly make our platform 2.8x faster. This confirms that, even though the Apriori algorithm seems to perform very poor as pure software, FPGA acceleration techniques that exploit this algorithm are still pertinent, particularly for large and/or sparse datasets.

6.8. Efficiency of FPGA usage

In order to analyze how efficiently our framework uses the FPGA accelerator, Table VII gives the distribution of the time spent inside the main processing steps, for several representative datasets and support values.

It shows the time related to loading the dataset (all in software), the ideal time for FPGA operations (calculated from the theoretical number of clock cycles), the real PCIe latency overhead, and the software time spent between Apriori iterations (updating the in-memory representation of itemsets). The rest includes potential software time that was not masked by FPGA operation, creation of threads to launch Riffa PCIe transfers, miscellaneous PCIe transfers due to setting configuration registers in the FPGA design, and miscellaneous prints of execution details.

The worst results are for dataset *pumsb*, this was expected. When filtered, this dataset is so small that the actual support counting operation represents a relatively small part of the overall processing time. The main cause is the PCIe latency to initiate transfers, which is high compared to the theoretical time needed to transfer the data. We highlight that for a support of 80%, the overall processing time is so small (around 90 ms) that the time to read the counters of the last batch of candidates is noticeable (not masked by transfer of a batch of candidates).

Our experiments show that any filtered dataset substantially larger than *pumsb* will lead to a relatively good FPGA utilization efficiency. This is illustrated with datasets *accidents* and *webdocs*: apart from the uncompressible loading time, support counting and more generally FPGA utilization represents most of the overall processing time.

Table VII also confirms that, in the proposed pipelined processing of batches of candidates (see section 5.2), the candidate generation done in software is well masked by FPGA operations. Indeed any unmasked CPU time would appear in row *Other* and this remains relatively low. As can be seen on the last line of Table VII, the average CPU usage during FPGA operations is no more than a few percents of one CPU core (at least for datasets larger than *pumsb*), which confirms that the proposed hardware acceleration solution actually relieves the CPU.

7. CONCLUSION

Our works bring notable improvements for Apriori acceleration solutions, whether based on FPGA or Micron Automata technologies. It also competes well with alternative FPGA-accelerated FIM algorithms. Our implementation is designed for maximum scalability and versatility, which additionally makes it appropriate for other *generate-and-test* mining algorithms, including sequence mining in real time from uninterruptible data streams.

This paper describes all details that we believe are missing in most previous acceleration papers, with the notable exception of Micron AP works [Wang et al. 2015]. We study the impact of counter size, item encoding and maximum itemset size on the total number of support counting units that can fit a given FPGA. This is important because it has a direct correlation on total mining time with Apriori.

We provide extensive and fair comparisons with several previous works, including many types of hardware accelerators and pure software solutions. We discovered that FIM hardware acceleration is only useful for large and low-density datasets. Besides, our Apriori accelerator is well suited for ultra-low latency sequence recognition and signaling from uninterruptible data streams. Using one common FPGA prototyping board, we achieved up to 1 GB/s throughput while monitoring up to 5818 patterns, or 250 MB/s with up to 11636 patterns.

As a perspective, when only pattern recognition and signaling on streams are needed, our IP could be further shrunk by removing the support counters. This could bring up to a 100% increase in the number of patterns that can be simultaneously monitored.

Acknowledgments

The authors would like to thank Olivier Menut from ST Microelectronics for his valuable inputs and continuous support.

References

Rakesh Agrawal, Ramakrishnan Srikant, and others. 1994. Fast algorithms for mining association rules. In *The International Conference on Very Large Databases, VLDB*, Vol. 1215. 487–499.

Chris Anderson. 2006. *The Long Tail: Why the Future of Business Is Selling Less of More*. Hyperion.

Z.K. Baker and V.K. Prasanna. 2005. Eff cient hardware data mining with the Apriori algorithm on FPGAs. In 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. 3–12.

Z.K. Baker and V.K. Prasanna. 2006. An Architecture for Eff cient Hardware Data Mining using Reconf gurable Computing Systems. In 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. 67–75.

Christian Borgelt. 2003. Eff cient implementations of apriori and eclat. In Proceedings of the IEEE ICDM workshop on frequent itemset mining implementations.

Lázaro Bustio, René Cumplido, Raudel Hernández, José M. Bande, and Claudia Feregrino. 2016. New Frontiers in Mining Complex Patterns: 4th International Workshop, NFMCP 2015. Springer International Publishing, Chapter Frequent Itemsets Mining in Data Streams Using Reconf gurable Hardware, 32–45.

Octavian Cret, Zsolt Mathe, Paul Ciobanu, Sonia Marginean, and Adrian Darabant. 2009. A hardware algorithm for the exact subsequence matching problem in DNA strings. Romanian Journal of Information Science and Technology 12, 1 (2009), 51–67.

FIMI Repository. 2003. Frequent Itemset Mining Dataset Repository. (2003). http://f mi.ua.ac.be/data/

Xiaoqi Gu, Yongxin Zhu, Shengyan Zhou, Chaojun Wang, Meikang Qiu, and Guoxing Wang. 2016. A Real-Time FPGA-Based Accelerator for ECG Analysis and Diagnosis Using Association-Rule Mining. ACM Transactions on Embedded Computing Systems 15, 2 (2016), 25.

Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining frequent patterns without candidate generation. In ACM Sigmod Record, Vol. 29. ACM, 1–12.

IBM. 2012. IBM Quest Synthetic Data Generator. (2012). http://sourceforge.net/projects/ibmquestdatagen/

M. Jacobsen, D. Richmond, M. Hogains, and R Kastner. 2015. RIFFA 2.1: A reusable integration framework for FPGA accelerators. ACM Transactions on Reconf gurable Technology and Systems 8, 4 (Sept. 2015).

Micron, Inc. 2016. Micron Automata Developer Portal - Hardware. (2016). http://www.micronautomata.com/hardware

Micron Technology, Inc. 2013. Micron Automata Processor - A Brief Introduction. (dec. 2013).

V.B. Nikam and B.B. Meshram. 2014. Scalable Frequent Itemset Mining using Heterogeneous Computing: ParApriori Algorithm. International Journal of Distributed and Parallel Systems 5, 5 (2014), 13.

Shaobo Shi, Yue Qi, and Qin Wang. 2013. FPGA Acceleration for Intersection Computation in Frequent Itemset Mining. In 2013 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery. 514–519.

Song Sun, M. Steffen, and J. Zambreno. 2008. A Reconf gurable Platform for Frequent Pattern Mining. In International Conference on Reconf gurable Computing and FPGAs. 55–60.

S. Sun and J. Zambreno. 2011. Design and Analysis of a Reconf gurable Platform for Frequent Pattern Mining. IEEE Transactions on Parallel and Distributed Systems 22, 9 (Sept 2011), 1497–1505.

DW Thoni and Alfred Strey. 2009. Novel strategies for hardware acceleration of frequent itemset mining with the apriori algorithm. In 2009 International Conference on Field Programmable Logic and Applications.

Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. 2003. LCM: An Eff cient Algorithm for Enumerating Frequent Closed Item Sets. In Proceedings of the IEEE ICDM workshop on frequent itemset mining implementations.

Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. 2005. LCM Ver.3: Collaboration of Array, Bitmap and Pref x Tree for Frequent Itemset Mining. In Proceedings of the 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations (OSDM '05). ACM, 77–86.

Ke Wang, Yanjun Qi, J.J. Fox, M.R. Stan, and K. Skadron. 2015. Association Rule Mining with the Micron Automata Processor. In IEEE International Parallel and Distributed Processing Symposium. 689–699.

Ying-Hsiang Wen, Jen-Wei Huang, and Ming-Syan Chen. 2008. Hardware-Enhanced Association Rule Mining with Hashing and Pipelining. IEEE Transactions on Knowledge and Data Engineering 20, 6 (June 2008), 784–795.

Xilinx Inc. 2015. Device Reliability Report - First Half 2015. Technical Report.

Osmar Zaiane. 2014. Rich Data: Risks, Issues, Controversies & Hype. (dec. 2014). Keynote speech at the International Conference on Advanced Data Mining and Applications .

Mohammed J. Zaki. 2000. Scalable Algorithms for Association Mining. IEEE Transactions on Knowledge and Data Engineering 12, 3 (May/Jun 2000), 372–390.

M. J. Zaki, Parthasarathy, M. S., Ogihara, and W. Li. 1997. New Algorithms for Fast Discovery of Association Rules. In 3rd International Conference on Knowledge Discovery and Data Mining. pp. 283–286.

Fan Zhang, Yan Zhang, and Jason D. Bakos. 2013a. Accelerating Frequent Itemset Mining on Graphics Processing Units. J. Supercomput. 66, 1 (Oct. 2013), 94–117.

Yan Zhang, Fan Zhang, Zheming Jin, and Jason D. Bakos. 2013b. An FPGA-Based Accelerator for Frequent Itemset Mining. ACM Transactions on Reconf gurable Technology and Systems 6, 1, Article 2 (May 2013), 17 pages.