Journal of Computing and Information Technology - CIT 22, 2014, 3, 159–169
doi:10.2498/cit.1002361

Frequent Pattern-growth Algorithm on Multi-core CPU and GPU Processors

Khedija Arour[1] and Amani Belkahla[2]

[1] Computer Science Department, National Institute of Applied Sciences and Technology, Tunis, Tunisia
[2] Computer Science Department, Faculty of Sciences of Tunis, Tunisia

Discovering association rules that identify relationships among sets of items is an important problem in data mining. It's a two steps process, the first step finds all frequent itemsets and the second one constructs association rules from these frequent sets. Finding frequent itemsets is computationally the most expensive step in association rules discovery algorithms. Utilizing parallel architectures has been a viable means for improving FIM algorithms performance. We present two FP-growth implementations that take advantage of multi-core processors and utilize new generation Graphic Processing Units (GPU).

Keywords: association rule mining, frequent itemset mining, GPU computing, parallel computing, GPGPU

1. Introduction

Frequent Itemset Mining (FIM) also known as Frequent Pattern Mining is one of the most popular problems in data mining, which consists of discovering frequently co-occurred itemsets and then generating rules used in many decision support applications. Frequent sets play an essential role in many data mining tasks that try to find interesting patterns from databases.

A FIM algorithm scans the database and finds itemsets that occur in the transaction frequently then a user-specified threshold. The challenges in frequent itemsets mining derive from a large size of a search space, which corresponds, in the worst case, to the power of the set of items. To quickly explore a large dataset through a FIM tool, analysts ask for new techniques to improve algorithm performance, by taking advantage of the evolutions in parallel computing.

Thus, to exploit this opportunities, there is a need for a new generation of FIM algorithms, able to exploit the parallelism on multi and many-core architectures by exploiting the performance of microprocessors, that has been steadily improved and the General-Purpose computing generation paradigm on Graphic Processing Units (GPGPU).

With the emergency of the GPU as a hardware accelerator for various non-graphics applications and GPGPU programming frameworks that greatly reduce the complexity of GPGPU computing, developers must carefully design their algorithms in order to take full advantages of the GPU's massively multi-threaded SIMD (Single Instruction, Multiple Data) architecture. Previous works have taken advantage of the massive computation power of the new architecture by accelerating database operations [30, 31, 32], approximate stream mining of quantiles and frequencies [29], MapReduce [33] and k-means clustering [34].

In the best knowledge, there has been little work that focuses on accelerating FIM algorithms on the GPU, even though parallel FIM has been studied on simultaneous multi threading (SMT) processors [36], shared memory systems [37], and, most recently, multi-core CPUs [8].

The key issue for studying FIM problems on modern processors is how to fully exploit the performance of the multi-core CPU and fully utilize the GPU architectural features.

To improve the performance of the FIM algorithm, we take one of the most known algorithms, FP-growth [1], as our baseline and identify the optimizations to make FIM efficient on modern processors. However, previous study shows that the pointer-based nature of the FP-tree is not cache friendly and requires costly

dereferences, which prevents it from achieving satisfying performance on a modern processor. This structure still underutilizes the modern system due to poor data locality and insufficient parallelism expression. To alleviate these problems, we improve the cache performance through the FP-array (Frequent Pattern array) structure [8], proposed by Li and *al*. This structure not only reduces the cache misses for single-core processor, but alleviates also the off-chip memory accesses and improves the scalability of the multi-core processor.

The remainder of the paper is organized as follows. Section 2 introduces the related works. We present FP-growth in Section 3. Section 4 develops the insufficiencies of pointer based structure on modern processors. We present FP-array structure and details of our two implementations in Section 5. Detailed experimental evaluations are given in Section 6. Finally, we offer conclusion and future works in Section 7.

2. Related Works

Several algorithms have been proposed in the literature to overcome the scalability (size of databases) and run-time performance of sequential algorithms. Apriori [10] is one of the most popular applications for enumerating frequent itemsets.

Many other FIM algorithms are proposed such as DHP [9], DIC [19], Eclat [18] and Partition [20] but they are I/O insufficient and suffer from multi-scan problem. Salvatore et *al.* proposed kDCI [21] (Direct Count and Intersect) and ParDCI [22] but it requires at least 3 datasets scan. FP-growth [1] proposed by Han et *al.* consists of creating a compressed FP-tree structure for mining a complete set of frequent itemsets without candidate generation. This structure is significantly smaller than the original database, but its pointer based nature requires costly dereferences and is not cache friendly, which prevents it from achieving a good performance on modern processors.

For more improving the cache performance, Ghoting and *al.* [23] proposed a cache conscious FP-tree (CC-tree), a reorganization of the original FP-tree by allocating the nodes in sequential memory space and a tiling strategy for temporal locality. This structure yields a better cache performance than FP-growth, but it still experiences cache misses when traversing the CC-tree due to its tree structure. Razs presented nonordfp [24] which implements FP-growth without rebuilding the projected FP-tree recursively to improve the cache performance. Li et *al.* proposed an FP-growth implementation [8] based on two techniques: a cache conscious FP-array and a lock free parallelization enhancement to improve data locality performance and make use of the benefits from hardware and software prefetching.

For performance improvement of this algorithm, most researchers [2, 5, 7, 8, 3, 4] have been made for parallelizing FP-growth. Pramudiono and Kitsuregawa [2] reported results for parallel FP-growth algorithm on shared nothing cluster environment. In [7], Li et *al.* proposed a Parallel FP-growth that shard a large-scale mining task into independent parallel tasks. Osmar et *al.* presented a MLFPT (Multiple Local FPtree) approach [3] that consists of two main stages: the first stage is the construction of a parallel FP-tree for each processor and the second stage is mining these data structures much like the FP-growth algorithm.

In [5], Manaskasemsak et *al.* presented a parallel version of FI-growth algorithm [6] that parallelizes the association rule mining process by employing a data parallelism technique on a PC cluster.

Researchers have also studied FIM algorithms on new-generation graphics processing units (GPUs), regarded as massively multi-threaded many-core processors. Different from multi-core CPUs, the cores on the GPU are virtualized, and GPU threads are executed in SIMD (Single Instruction, Multiple Data) and managed by the hardware. Such a design simplifies GPU programming and improves program scalability and portability. Nevertheless, it makes the implementation of algorithms with complex control flows a challenging task on the GPU, even though the GPU has an order of magnitude computation capability as well as memory bandwidth higher than a multi-core CPU.

Taking advantage of the massive computation power and the high memory bandwidth of the GPU, there have been some studies that focus on studying the GPU acceleration for FIM algorithms.

GPGPU for FIM algorithms was for the first time addressed in [25], where Luo et *al.* presented two GPU-based implementations of the well known Apriori algorithm, that takes advantage of the GPU's massively multi-threaded SIMD (Single instruction, multiple data) architecture. Both implementations employ a bitmap data structure to exploit the GPU's SIMD parallelism. One implementation runs entirely on the GPU, since the other employs both the CPU and the GPU for processing. Another Apriori based FIM algorithm for GPU is presented in [26], GPApriori, which includes a set of fine-grained parallel data structures and algorithms design to achieve premising degree of speed up on modern GPU.

A different approach is proposed in [28] by Teodoro and *al.* based on the Tree Projection algorithm described in [35]. Nonetheless, Tree Projection is not a state of the art algorithm for FIM, as it is outperformed by FP-growth [1]. In [27], Orlando and Silvestri proposed gpuDCI, a parallel algorithm inspired by DCI [19] that exploits GPUs to efficiently mine frequent itemsets. In [39], Zhang and *al.* proposed an improved data-parallel algorithm derived from the Equivalent Class Expansion (Eclat) method. After candidate generation on the CPU, the algorithm counts the support values of each candidate on GPU by vertical list intersection.

As our optimizations are presented in the context of FP-growth algorithm, we will give, in the next section, a description of this algorithm. We will then illustrate the optimization introduced in this algorithm for better implementation on modern processors. As current FP-tree based algorithms still under-utilize these systems due to poor data locality and insufficient parallelism expression, we propose a compact data structure (FP-array: Frequent pattern array) to make use of the benefits from multi-core and graphic architectures. Our contributions will be described in the following sections.

3. FP-growth Algorithm

FP-growth method proposed by Han et *al.* is a depth-first algorithm based on data structure called FP-tree (Frequent Pattern tree). The FP-tree is a projected dataset, which provides a compact representation of the original database. Each node of the tree stores an item label and a count, representing the number of the transactions which contain all the items in the path from the root node to the current node.

For constructing the FP-tree, FP-growth requires two database scans when mining all frequent itemsets. The first scan of the database derives a list of the 1-itemsets, denoted as F, sorted by item's support in descending order. In the second scan, the FP-tree is constructed, a compressed representation of the original database. For each transaction, its frequent items were inserted into the FP-tree with item's support descending order. A new order is generated when the node with the appropriate label is not found; otherwise, the count of the existing nodes is increased.

Once the FP-tree has been constructed, the frequent itemsets mining can be performed. For each item in the FP-tree, FP-growth finds all frequent items in the conditional pattern base, it recursively constructs a new FP-tree for this conditional pattern base when it has at least two frequent items. By concatenating each 1-item with the frequent itemsets generated from conditional FP-trees, all of the frequent itemsets are discovered within the FP-tree.

4. Synthesis

Current pointer-based algorithms still under-utilize modern processors due to the irregular nature of this structure, its poor data locality and insufficient parallelism expression. To alleviate this problem, exploit the thread level parallelism and take advantage from the advanced characteristics of the graphic processor, we propose FP-array structure proposed by Li et *al.* [8], which efficiently improves the data locality performance.

FP-array is a data reorganization of the pointer based tree, FP-tree, which has poor cache utilization, mainly for these reasons. First, each node in FP-tree has a total of 5 elements: an item label, a count, a nodelink pointer, a parent pointer and a list of child pointers. Thus, for traversing FP-tree, only two fields in FP-tree node are required; the item label and the parent pointer. This significantly degrades cache line utilization. Second, a node and each associated child node may not be contiguous in memory

due to the way that FP-tree is built. Third, two nodes with the same item label cannot reside in the same cache line due to the pointer based data structure in nodelink. The next node with the same item label can be represented in other cache line due to the lack of temporal locality. Transforming FP-tree into a compact and contiguous data structure can efficiently improve data locality performance.

5. Design and Implementations

In this section, we present our CPU and GPU implementations based on FP-array structure, presented in [8], which significantly improves the cache performance. This structure uses two simple static data structures; *item array* and *node array*, which can efficiently improve data locality performance and permit to exploit a lot of data parallelism.

The two tables are allocated in contiguous memory space. The *item array* works essentially as a replication of the FP-tree. The *node array* permits to record the occurrences of frequent items figured in *item array*. This structure is organized as an array list. Each list is associated with one frequent item and each element in *node array* corresponds to a node in FP-tree, which has three fields; the begin position of an item in the *item array*, reference count and transaction size.

5.1. Construction of FP-tree and FP-array

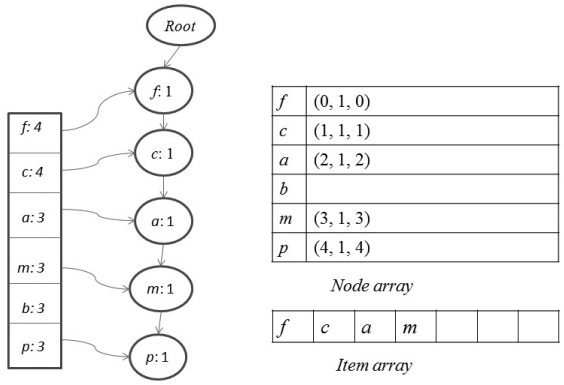
To construct FP-array, we have made some changes to the original process proposed by Li and *al*. In [8], to build FP-array, authors traverse FP-tree in depth-first order, copy the item in each node to the *item array* sequentially and then create array lists which record the occurrences of the frequent items in the *item array*. In our approach called FP-growth+, FP-array is created in parallel while building FP-tree as described in algorithm 1.

The root of the tree is created and labelled as null. The scan of the first transaction in the database leads to create the first branch in the tree and the corresponding nodes in *node array*. While inserting a node *n* in FP-tree, a label item of *n* is inserted in *item array* and a new node is inserted in *node array*. For the second transaction, if its item list shares a common prefix with the existing path in FP-tree, the count of each node along the prefix is incremented by 1 and a new node is created and linked as a child of the last node in the prefix. For each item in the common prefix, its label item is inserted in *item array* if the transaction doesn't appear in this structure. The count of the corresponding items, which appear in the common prefix, is incremented by 1 and a new node is created for the rest of items in the transaction. Figure 1 illustrates the difference between the two methods.

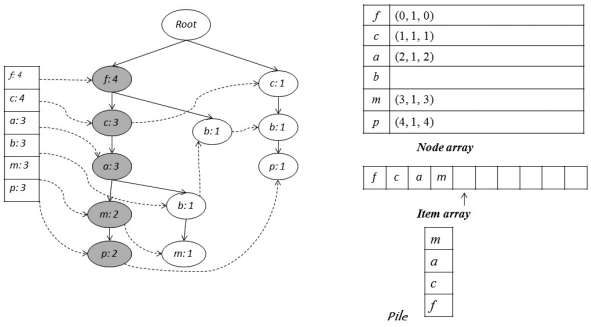
Algorithm 1

```
Input:  D: Dataset
        Min_sup: Minimum threshold
Output: FP-tree
        IA: item array
        NA: node array

Begin
  FI ←{} // List of the frequent itemsets
  H ←{} // Header Table
  // Count frequent items
  for each (item i∈D) do
    Insert the item i in H
    H ← H ⋃{i}
  for each (item i∈H) do
    if Support(i) < Min_sup
    Eliminate the infrequent item i from H
  Initialize it to 0
// it : the position of the label item i of the
   node N in the item array
  for each (transaction t∈D) do
    for each (item i∈t) do
      if The item list in which appears i
      shares a common prefix with the
      existing path in FP-tree then
        The count of the node N in
        FP-tree is incremented by 1
        The count of the node N in node
        array is incremented by 1
      else
        Insert a new node N in FP-tree
        Insert the label item i of the node
        N in the item array IA
        Insert a new node N in the node
        array NA
        Increment it
  Delete (FP-tree)
End
```



a) The proposed method



b) Li and *al.*'s method

Figure 1. The difference between Li and *al.* method and our proposed method to construct FP-array.

The scan of the first transaction (*f,c,a,m,p*) in the database leads to the construction of the first node branch in FP-tree and its corresponding nodes in FP-array: the process starts from node *f*, the first child node of the root. The label item of the node *f*, created and labeled into FP-tree, is inserted in *item array* and its corresponding node $(0, 1, 0)$ is inserted then in *node array*. The rest of the first branch in FP-tree (c:1), (a:1), (m:1), (p:1) and its corresponding members $(1, 1, 1)$, $(2, 1, 2)$, $(3, 1, 3)$, $(4, 1, 4)$ are then inserted in *node array*. For the second transaction (*f,c,a,b,m*), since its frequent items share a common prefix (*f,c,a*) with the existing path, the count of each item in FP-tree, which appears in the common prefix, is incremented by 1 and the corresponding *node array* members are updated. A new node (*b:1*) is then created and linked as a child node of (*a:2*), its label item is inserted in *item array* and a new node $(7, 1, 3)$ in *node array*. Another node (*m:1*), new child node of (*b:1*), is also created in FP-tree, its label item and corresponding node $(8, 1, 4)$ are then inserted respectively in *item array* and *node array*.

This method eliminates redundancy in *item array* construction, which can decrease the memory consumption and improve the cache performance.

5.2. CPU Parallel Implementation

We describe in this section the FP-growth implementation for multi-core processor.

The proposed solution employs transaction processing parallelism, which divides the transactions among different CPU threads to be inserted in FP-tree, then in FP-array structure. Processing the transactions in parallel may need to update the same node in FP-tree and the same cell in *node array*.

This problem may be solved by creating critical section to avoid threads to update nodes concurrently. Our solution is to create a critical section associated with a level tree, thus only a thread can update a node in FP-tree and *node array*. This approach is simple and inexpensive in terms of overhead. It imposes barriers among threads accessing different nodes.

5.3. GPU Frequent Itemsets Discovering

The FP-growth is one of the most important FIM algorithms, which makes it an interesting candidate for GPU acceleration.

In this section, we describe gpuFP-growth+, a parallel algorithm based on FP-growth that exploits the performance of the graphic processor to mine frequent itemsets. The basic idea behind our algorithm is to start computation on CPU, count frequent 1-itemsets and build *item array* and *node array*, which records the occurrences of frequent 1-itemsets in the *item array*. The array lists, which are associated with frequent 1-itemsets, are maintained by the host, however *item array* is moved to the GPU global memory. Figure 2 shows how the FP-growth+ is executed on CPU and GPU.

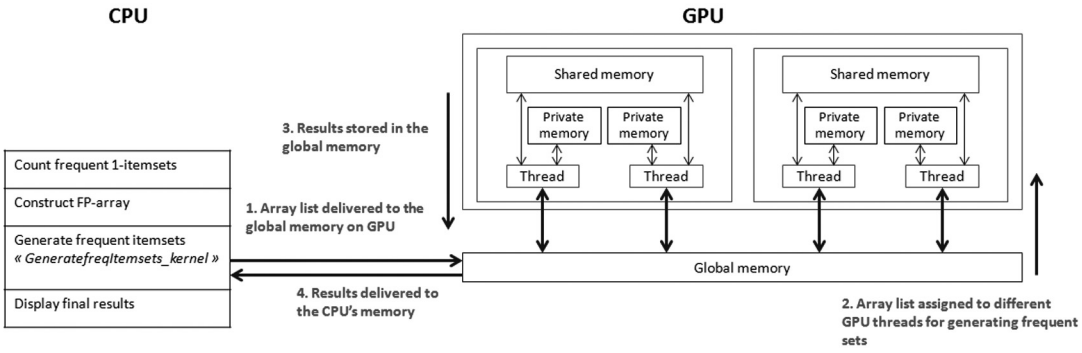


Figure 2. Process of FP-growth+ on GPU.

The proposed parallelization of the Generate-FreqItemsets procedure is implemented as a GPU kernel, that processes the array lists in the FP-array, using OpenCL [40]. Since array lists are organized in lines, one dimension index space is enough. Before launching the GPU *GenerateFreqItemsets_kernel*, the host program defines the kernel's context and manages its execution. The largest possible contiguous block of free memory on the GPU is allocated to store generated new structures and results. When the kernel is submitted for execution, the CPU host defines the computation domain, while each independent element of execution in this domain is a work-item (or thread). During the procedure, each array list in *node array* is delivered to global memory on GPU and is stored using a vector as shown in Figure 3. The work-items are organized into independent work-groups (or blocks), where each work-item executes concurrently within a single compute unit. Work-items into a work-group work on the same array list to generate frequent sets corresponding to a giving item. Each work-item is in charge of a portion of the array list, in such way that threads having the consecutive indexes work on consecutive parts of the array list. New frequent sets are generated by intersecting the old sets with the common prefix in the top of the stack. New projected *node array*, represented by a set of array lists, and *item array* corresponding to a given array list are then generated and stored using a vector. An auxiliary index is created to identify the beginning of each array list into the new projected *node array* where it is stored. The algorithm expands repeatedly the allocated index space by traversing the array list and constructing the new projected *node array* and *item array* structures. The process is repeated until the projected *node array* created is empty.

Afterwards, the generated sets are written back to global memory and then copied back to CPU memory. The array lists are deallocated by the

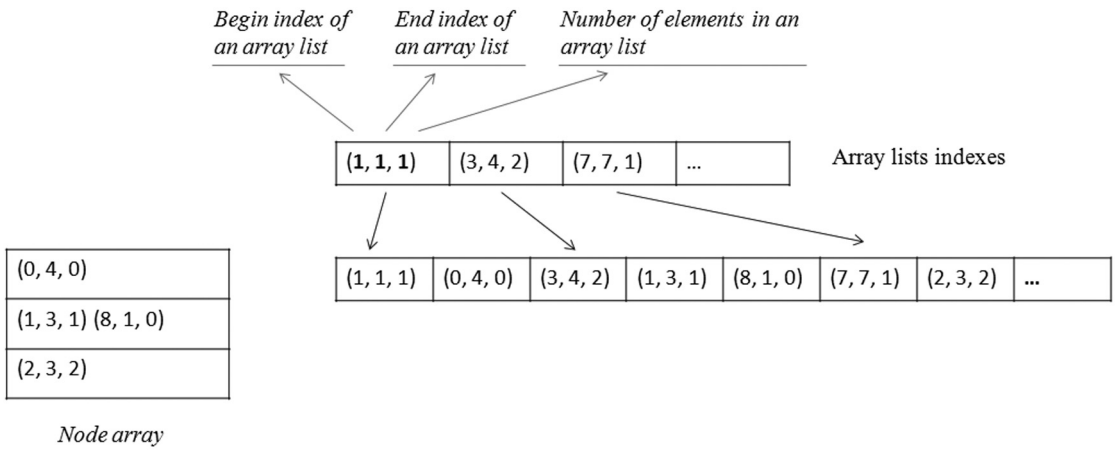


Figure 3. Array lists representation on GPU.

program in order to exploit the memory space to store the new projected *node array* and *item array* structures corresponding to the next array list in the original *node array* maintained by the host.

6. Experimental Evaluation

In this section, we evaluate the performance of our implementations. We used several real and synthetic data to compare the execution time of these implementations based on FP-growth algorithm with various thresholds. The datasets are downloaded from the Frequent Itemset Mining Implementation repository. Table 1 summarizes the characteristics of these datasets with different number of items, the number of transactions and the average transaction length in each dataset.

Dataset	#Transactions	#Items	Average transaction length
T40I10D100K	100000	1000	40
RETAIL	88162	16470	13
KOSARAK	990000	1530	5
CONNECT	67775	130	43
CHESS	3196	76	37

Table 1. Database characteristics.

All of the following tests were run on a dual 2.1 GHz Intel Core i3 processor with ATI Radeon GPU and 4 gigabytes of main memory, running on Windows7. The source code was written and compiled using the Visual Studio 2008. The version of OpenCL is 1.1. The running times of different implementations on the test datasets are displayed in Figures 4 and 5.

6.1. FP-growth+ on Multi-core Processor

This section analyzes the FP-growth+ on multi-core system. For more improving the performance of this implementation, we used FP-array data structure.

The code is analyzed using the Intel Vtune Amplifier[1], which collects profiling information with hardware-based event sampling. In our case, we consider the CPI ratio (Clock Per Instruction), used as a cumulative performance metric to determine if the execution on a multi-core processor is efficient and how it can be improved further. High values of CPI indicate that the code performs sub-optimally, which can be caused by execution stalls due to branch mispredictions, cache misses, and other undesirable effects.

The result of analysis of our code is showed in Table 2. We compared the performance characteristics of the pointer based FP-growth and our implementation based on FP-array data structure in terms of CPI.

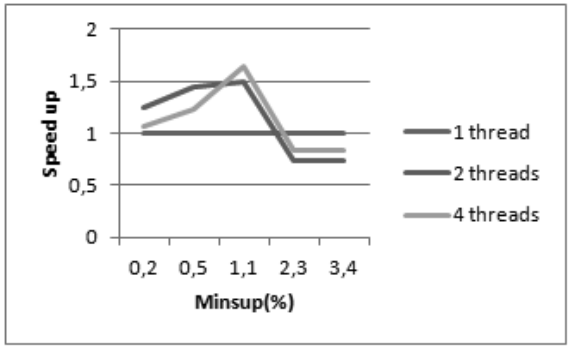
Dataset	FP-growth	FP-growth+
T40I10D100K	3	1.5
CONNECT	3	1
KOSARAK	2	1.5
CHESS	1.5	1
RETAIL	2	1.5

Table 2. Cache performance: CPI metric.

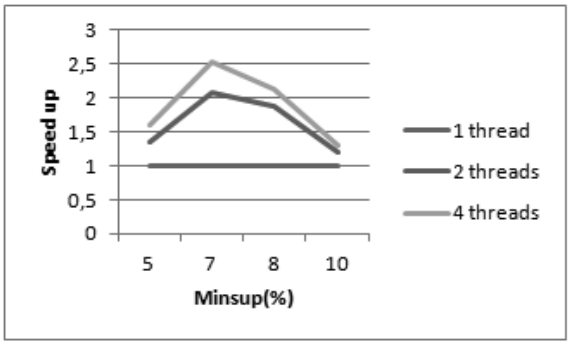
From Table 2, it is evident that we achieve a significant performance improvement due to transforming FP-tree to the compact data structure FP-array. This result shows that FP-array is a cache friendly structure. It removes pointers and stores all data sequentially in memory, which improves the cache locality performance. On the other hand, FP-tree based FP-growth employs a pointer based tree structure. These irregular data references will break the data spatial locality.

In this section, we compare also the parallel execution time between our proposed FP-array based FP-growth, FP-growth+, and the original implementation of the same algorithm. In Figure 4, we present the multi-core CPU implementation speedup over the sequential CPU version for the datasets Retail, *T40I10D100K*, *Kosarak* and *Chess* as the support is varied. For the datasets, the proposed solution achieved good scalability while increasing the number of threads.

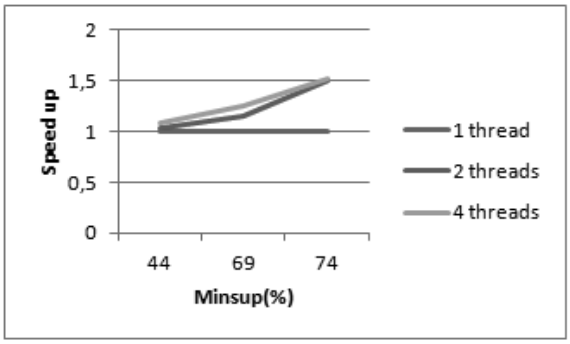
[1] `http://software.intel.com/en-us/intel-vtune-amplifier-xe`



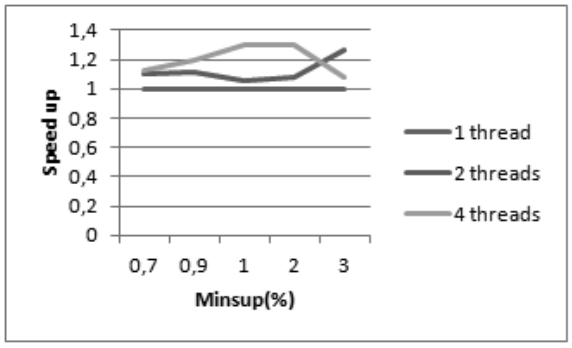
a) Retail



b) T40I10D100K



c) Chess



d) Kosarak

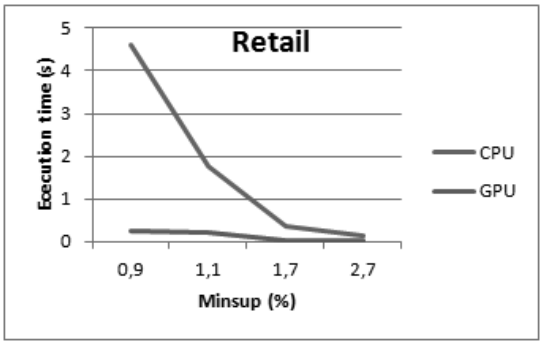
Figure 4. FP-growth+ on multi-core processor.

6.2. FP-growth+ on GPU

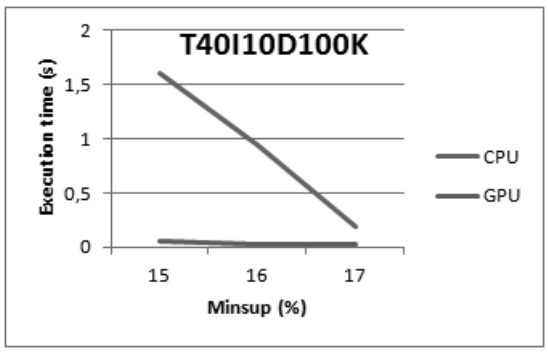
The FP-growth implementation to GPU is evaluated in this section. Firstly, Table 3 and Figure 5 present the proposed GPU based FP-growth speedup over the sequential CPU version as the support is varied. Table 3 and Figure 5 show the absolute execution times for each processor and the datasets *Retail* and *T40I10D100K*.

Dataset	Minsup (%)	Time (CPU)	Time (GPU)	Speed up
RETAIL	0.9	4.6	0.26	17.7
	1.1	1.75	0.2	8.75
	1.7	0.35	0.04	8.75
	2.7	0.15	0.03	5
T40I10D100K	15	1.6	0.05	32
	16	0.94	0.03	31.3
	18	0.19	0.02	9.5

Table 3. Speed up on multi-core processor vs. GPU.



a) Retail



b) T40I10D100K

Figure 5. FP-growth+ on GPU.

These results show that our proposed GPU-based FP-growth is much more efficient when compared to the CPU-based version, and is also capable of scaling up the support decreases. Our experiments show that the GPU achieved speedup over the sequential FP-growth of 17x and 32x respectively for *Retail* and *T40I10D100K* datasets.

Although gains of the GPU version of FP-growth over the CPU-based version still high as the support decreases. Figure 5 shows that the GPU relative performance is high as the support decreases for *Retail* and *T40I10D100K* datasets. This behavior is due to the increasing of occupancy on GPU.

7. Conclusion and Future Works

In this paper, we show that the existing frequent itemset mining implementations like FP-growth are still under-utilized on modern processors due to poor data locality performance and low thread level parallelism. We used FP-array structure to resolve this problem and improve the data locality performance. We presented also a multi-core and a GPU-based implementation of FP-growth algorithm, and also a detailed analysis of algorithm's performance on this modern processor. The experimentations showed that our proposed solution achieved good scalability while increasing the number of threads. Our GPU-based implementation of the FP-growth algorithm, on the other hand, outperformed the CPU implementation in up to 32x when compared to the sequential version. As future work, we intend to evaluate the performance of our proposed GPU implementation of FP-growth with less support threshold and to compare the results with the results of other authors.

References

[1] H. JIAWEI, P. JIAN, Y. YIWEN, Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, (2000) Dallas, Texas, USA, pp. 1–12.

[2] I. PRAMUDIONO, M. KITSUREGAWA, Parallel FP-growth on PC Cluster. In *Proceedings of the 7th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, PAKDD'03*, (2003) Seoul, Korea, pp. 467–473. `http://dl.acm.org/citation.cfm?id=1760894.1760956`

[3] O. R. ZAIANE, M. EL-HAJJ, P. LU, Fast parallel association rules mining without candidacy generation. *Proceedings of the 2001 IEEE International Conference on Data Mining*, (2001).

[4] A. JAVED, A. KHOKHAR, Frequent pattern mining on message passing multiprocessor systems. *Distributed and Parallel Databases*, **16**(3) (2004), 321–334.,

[5] B. MANASKASEMSAK, N. BENJAMAS, A. RUNGSAWANG, A. SURARERKS, P. UTHAYOPAS, Parallel association rule mining based on FI-growth algorithm. *Parallel and Distributed Systems, International Conference on*, **1** (2007), 1–8. `http://doi.ieeecomputersociety.org/10.1109/ICPADS.2007.4447743`.

[6] K. AMPHAWAN, A. SURARERKS, An approach of frequent item tree for association generation. In *Proceedings of Artificial Intelligence and Soft Computing*, (2005).

[7] H. LI, Y. WANG, D. ZHANG, M. ZHANG, E. Y. CHANG, Parallel Fp-growth for Query Recommendation. *Proceedings of the 2008 ACM Conference on Recommender Systems*, (2008) Lausanne, Switzerland, pp. 107–114.

[8] E. LI, L. LIU, Optimization of Frequent Itemset Mining on Multiple-Core Processor. *Proceedings of the 33rd International Conference on Very Large Data Bases*, (2007) University of Vienna, Austria, pp. 23–27.

[9] J. S. PARK, M.-S. CHEN, P. S. YU, An effective hash-based algorithm for mining association rules. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, (1995), pp. 175–186.

[10] R. AGRAWAL, R. SRIKANT, Fast algorithms for mining association rules. *Proceedings of the 20th International Conference on Very Large Data Bases*, (1994).

[11] R. AGRAWAL, T. IMIELIŃSKI, A. SWAMI, Mining Association Rules Between Sets of Items in Large Databases. *SIGMOD Rec.*, **22**(2) (1993), 207–216. `http://doi.acm.org/10.1145/170036.170072`

[12] S. BRIN, R. MOTWANI, C. SILVERSTEIN, Generalizing association rules to correlations. *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, (1997), pp. 265–276.

[13] C. SILVERSTEIN, S. BRIN, R. MOTWANI, J. ULLMAN, Scalable Techniques for Mining Causal Structures. *Data Min. Knowl. Discov.*, **4**(2-3) (2000), 163–192. `http://dx.doi.org/10.1023/A:1009891813863`

[14] R. AGRAWAL, R. SRIKANT, Mining Sequential Patterns. *Proceedings of the Eleventh International Conference on Data Engineering, ICDE '95*, (1995), pp. 3–14. `http://dl.acm.org/citation.cfm?id=645480.655281`

[15] H. MANNILA, H. TOIVONEN, A. INKERI VERKAMO, Discovery of Frequent Episodes in Event Sequences. *Data Min. Knowl. Discov.*, **1**(3) (1997), 259–289. http://dx.doi.org/10.1023/A:1009748302351

[16] J. HAN, Efficient Mining of Partial Periodic Patterns in Time Series Database. *Proceedings of the 15th International Conference on Data Engineering, ICDE '99*, (1999), pp. 106. http://dl.acm.org/citation.cfm?id=846218.847205

[17] G. DONG, J. LI, Efficient Mining of Emerging Patterns: Discovering Trends and Differences. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '99*, (1999) San Diego, California, USA, pp. 43–52. http://doi.acm.org/10.1145/312129.312191

[18] M. J. ZAKI, S. PARTHASARATHY, M. OGIHARA, W. LI, New Algorithms for Fast Discovery of Association Rules. In *Proceedings of the 3rd Intl. Conf. on Knowledge Discovery and Data Mining*, (1997), pp. 283–286. http://www.aaai.org/Library/KDD/1997/kdd97-060.php

[19] S. BRIN, R. MOTWANI, C. SILVERSTEIN, Beyond Market Baskets: Generalizing Association Rules to Correlations. *SIGMOD Rec.*, **26**(2) (1997), 265–276. http://doi.acm.org/10.1145/253262.253327

[20] A. SAVASERE, E. ASHOKANSKI, S. B. NAVATHE, An Efficient Algorithm for Mining Association Rules in Large Databases. In *Proceedings of the 21th International Conference on Very Large Data Bases, VLDB '95*, (1995), pp. 432–444. http://dl.acm.org/citation.cfm?id=645921.673300

[21] S. ORLANDO, C. LUCCHESE, P. PALMERINI, R. PEREGO, F. SILVESTRI, kDCI: a Multi-Strategy Algorithm for Mining Frequent Sets. In *Proceedings of the Workshop on Frequent Itemset Mining Implementations, ICDM 2003*, (2003), Melbourne, Florida, USA. http://SunSITE.Informatik.RWTH-Aachen.de/Publications/CEUR-WS//Vol-90/palmerini.pdf

[22] S. ORLANDO, P. PALMERINI, R. PEREGO, F. SILVESTRI, An efficient parallel and distributed algorithm for counting frequent sets. In *Proceedings of VECPAR*, (2002).

[23] A. GHOTING, G. BUEHRER, S. PARTHASARATHY, D. KIM, A. D. NGUYEN, Y.-K. CHEN, P. DUBEY, Cache-conscious Frequent Pattern Mining on a Modern Processor. In *VLDB* (K. BÖHM, C. S. JENSEN, L. M. HAAS, M. L. KERSTEN, P.-A. LARSON, B. C. OOI, Eds.), (2005) pp. 577–588. ACM.

[24] B. RÁCZ, nonordfp: An FP-growth variation without rebuilding the FP-tree. In *FIMI, CEUR Workshop Proceedings*, (R. J. BAYARDO, B. GOETHALS, M. J. ZAKI, Eds.), (2004). http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-126/racz.pdf

[25] W. FANG, M. LU, X. XIAO, B. HE, Q. LUO, Frequent itemset mining on graphics processors. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware, DaMoN 2009*, (2009) Providence, Rhode Island, USA, pp. 34–42.

[26] F. ZHANG, Y. ZHANG, J. D. BAKOS, GPApriori: GPU-Accelerated Frequent Itemset Mining. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing (CLUSTER)*, (2011) Austin, TX, USA, pp. 590–594.

[27] C. SILVESTRI, S. ORLANDO, gpuDCI: Exploiting GPUs in Frequent Itemset Mining. In *Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2012*, (2012) Munich, Germany pp. 416–425.

[28] G. TEODORO, N. MARIANO, W. MEIRA JR., R. FERREIRA, Tree Projection-Based Frequent Itemset Mining on Multicore CPUs and GPUs. *22nd International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD*, (2010) Petropolis, Brazil, pp. 47–54.

[29] N. K. GOVINDARAJU, N. RAGHUVANSHI, D. MANOCHA, Fast and Approximate Stream Mining of Quantiles and Frequencies Using Graphics Processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (2005) Baltimore, Maryland, USA, pp. 611–622.

[30] N. GOVINDARAJU, J. GRAY, R. KUMAR, D. MANOCHA, GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data, SIGMOD '06*, (2006) Chicago, IL, USA, pp. 325–336.

[31] N. K. GOVINDARAJU, B. LLOYD, W. WANG, M. C. LIN, D. MANOCHA, Fast Computation of Database Operations using Graphics Processors. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (2004) Paris, France, pp. 215–226.

[32] B. HE, K. YANG, R. FANG, M. LU, N. K. GOVINDARAJU, Q. LUO, P. V. SANDER, Relational joins on graphics processors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008*, (2008) Vancouver, BC, Canada, pp. 511–524.

[33] B. HE, W. FANG, Q. LUO, N. K. GOVINDARAJU, T. WANG, Mars: a MapReduce framework on graphics processors. *17th International Conference on Parallel Architecture and Compilation Techniques (PACT 2008)*, (2008) Toronto, Ontario, Canada, pp. 260–269.

[34] S. CHE, M. BOYER, J. MENG, D. TARJAN, J. W. SHEAFFER, K. SKADRON, A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel Distrib. Comput.*, **68**(10) (2008), 11.

[35] R. C. AGARWAL, C. C. AGGARWAL, V. V. V. PRASAD, A Tree Projection Algorithm for Generation of Frequent Item Sets. *J. Parallel Distrib. Comput.*, **6K1**(3) (2001), 350–371.

[36] A. GHOTING, G. BUEHRER, S. PARTHASARATHY, D. KIM, A. D. NGUYEN, Y.-K. CHEN, P. DUBEY, Cache-conscious Frequent Pattern Mining on a Modern Processor. *Proceedings of the 31st International Conference on Very Large Data Bases*, (2005) Trondheim, Norway, pp. 577–588.

[37] M. J. ZAKI, Y. GUO, R. GROSSMAN, Parallel Data Mining for Association Rules on Shared-memory Systems. *Knowledge and Information Systems*, (1998).

[38] J. TOMPSON, K. SCHLACHTER, An Introduction to the OpenCL Programming Model, 2012.

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

[40] J. TOMPSON, K. SCHLACHTER, An Introduction to the OpenCL Programming Model, 2012.

Received: February, 2014
Revised: June, 2014
Accepted: July, 2014

Contact addresses:
Khedija Arour
Computer Science Department
National Institute of Applied Sciences and Technology
1080 Tunis
Tunisia
e-mail: khedija.arour@issatm.rnu.tn

Amani Belkahla
Computer Science Department
Faculty of Sciences of Tunis
1060 Tunis
Tunisia
e-mail: amani.belkahla@gmail.com

KHEDIJA AROUR received his Engineering diploma and Ph.D. degree from the Department of Computer Science of the Science Faculty of Tunis, Tunisia in 1992 and 1996, respectively. She is currently an assistant professor in the Department of Computer Science and Mathematics at National Institute of Science and Applied Technology of Tunis, Tunisia, Carthage University. Dr. Arour's research interests are mainly in haute performance data mining and large scale information retrieval systems.

AMANI BELKAHLA received his Master in Applied Informatics from the Faculty of Economics and Management Management of Nabeul and Research Master in Computer Science from the Department of Computer Science of the Science Faculty of Tunis, Tunisia in 2014. Belkahla's research interests are mainly in haute performance data mining under multicores and GPU processors.