Parallel pattern mining - application of GSP algorithm for Graphics Processing Units

Krzysztof Hryniów
Institute of Control and Industrial Electronics
Warsaw University of Technology
Warsaw, Poland
e-mail: hryniowk@ee.pw.edu.pl

Abstract—**Frequent pattern mining is the field with many practical applications, where large computational power and speed are needed. Many solutions, both software and hardware, are proposed for those applications, but specialised solutions in form of embedded systems are not so common as one could imagine. This is especially true when we consider problems that can be paralleled. Many of the state-of-the-art frequent pattern mining applications are inefficient when used on shared memory systems or multiprocessor systems. To solve this problem both hardware and software solutions are proposed - remapping system architecture, improving memory performance, modifying task allocation.**

This article proposes modification of classical frequent pattern mining algorithms from Apriori family, illustrated with the example of popular GSP algorithm. Addition of GPU (Graphics Processing Unit) or multiple GPUs to embedded system is proposed and algorithm is modified in such a way, that it is best suited for solving GPGPU (general-purpose computation on graphics hardware) problems. Both theoretical and experimental evaluation of modifications are made, the latter with use of setup consisting of NVIDIA Tesla card and CUDA parallel computing platform.

It is shown in the article, that for tested data sets modified GSP algorithm finishes finding frequent sequences 50-100 times faster and with the same accuracy. Such speed-up allows the use of classical pattern mining algorithms for real-time solutions. This also permits broader scope of algorithms to be used in embedded systems with real-time constraints.

Index Terms—**embedded systems, sequential pattern mining, CUDA, GPGPU, GSP.**

I. Introduction

Sequential pattern mining was first defined by Agrawal and Srikant in [1] as

> Given a set of sequences, where each sequence consists of a list of itemsets, and given a user-specified minimum support threshold (min_support), sequential pattern mining is to find all frequent subsequences whose frequency is no less than min_support.

The GSP algorithm is an algorithm from the Apriori family presented first in [2]. Algorithm makes multiple passes over the data set, the first one determining the support for each item. At the end of the first pass, the algorithm knows which items are frequent (have at least minimum support) and for each such item creates a 1-element frequent sequence including it. Each subsequent pass starts with a seed set - the set of frequent sequences found in the previous pass. The seed set is used to generate new potentially frequent sequences, called candidate sequences. Each candidate sequence has one more item than a seed sequence; so all the candidate sequences in a given pass will have the same length (if the seed sequence was consisting of elements of length of 3, candidates will all have 4 characters each). The support for these sequences is found during the pass over the data - at the end of the pass, the algorithm determines which of the candidate sequences are actually frequent. These frequent candidates create the seed for the next pass. The algorithm terminates when there are no frequent sequences at the end of given pass [2].

TABLE I
WORKING TIMES OF GSP ALGORITHM FOR LARGE DATA SETS

Data set name	Data set size [thousands]	Number of characters	GSP working time [s]
King James Bible	4834,8	81	5618
Pan Tadeusz	461,4	78	428

GSP algorithm is proven to work well and with satisfying speed for different types of data sets [3]–[5], but as shown in Table I for large data sets the classical GSP algorithm is too slow to be used for real-time solutions. Analogous case is for other sequential pattern mining algorithms [6]. In addition, existing algorithms are unable to solve all sequential pattern mining problems in real-time, therefore there is a need for other solutions. This article proposes modification of existing algorithms by treating sequential mining problems as GPGPU (general-purpose computation on graphics hardware) problems and accelerating algorithms with use of graphics processing units.

In [7], [8] Apriori algorithm was modified to work on GPU and results were obtained faster then with use of CPU-based FP-growth algorithm. Still in the field of data mining such modifications are made mainly in the field of frequent itemset mining, and more complicated problems, like pattern mining, still need fast parallel solutions.

Research was done as part of N N514 671140 research grant "Pattern mining in real-time rule-based systems."

GPU (graphics processing unit) is a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second[1]. It is dedicated to high-performance 3D graphics, but can be used for GPGPU problems, as it is suited for parallel programming with coherent and predictable memory access. In such problems GPU and CPU co-processing (sequential parts of algorithms are run on CPU and parallel parts are run on GPU) outstrippes traditional CPU-based algorithms due to its high computational capacity [9]. GPUs architecture is optimized for executing many concurrent threads slowly, rather than executing a single thread quickly. In this case Tesla C2050 GPU was used along with CUDA platform.

Tesla C2050 is graphics processing unit with 448 CUDA cores, 4 GB DRR5 RAM ECC (Error Correction Code memory) and asynchronous data transfer, allowing for peak efficiency of 515 GFLOPS for double precision computation and 1 TFLOPS single precision computation.

CUDA is a parallel computing platform and programming model developed by NVIDIA Corporation, increasing computing performance for parallel problems with use of GPUs. It provides both low and high level APIs (application programming interfaces) with debuggers, shared memory access, scattered reads and fast read backs. CUDA platforms comes with CUDA C programming language, which is an extension of C++ for parallel programming with NVIDIA GPUs.

II. Algorithm

Principle of operation of GSP algorithm is presented in Introduction and in short on diagram (Figure 1).

Modified version of GSP algorithm has three CUDA kernels added, for operations that can be paralleled with substantial gain: creation of 1-element frequent sequence seed set, creation of n-element candidate sequence sets and determination of support for each candidate in candidate set.

Parallel algorithm

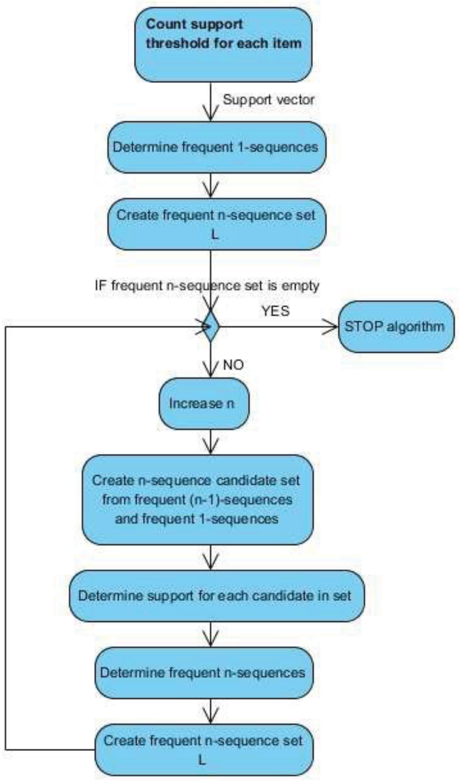
Step 1 First pass over sequence S - support for each ASCII sign is counted and elements with support greater or equal to minimum support threshold are put into 1-element frequent set F. Seed set L is created - for first step it is identical to set F;

Step 2 2-element candidate set A is generated - each element from set L is ended with element from set F:
$a_{i,j} = l_i + f_j$, $\forall l_i \in L = \{l_1, l_2, ..., l_n\}$, $\forall f_j \in F = \{f_1, f_2, ..., f_m\}$; $i = 1, ..., n; j = 1, ..., m$;
thus $|A| = |L| * |F|$;

Step 3 In loop, for n = 2 and increasing by 1 in each iteration:

a) n-th pass over sequence S - support for each element from A is calculated and elements with support greater or equal to minimum support threshold are put into new n-element seed set L;
b) if $|L| = 0$ algorithm terminates, otherwise it prepares for next iteration;
c) new candidate set A is generated, its elements of size n+1 are created by merging elements from set L (elements of size n) with elements from set F (elements of size 1);

[1]Technical definition of GPU introduced by NVIDIA Corporation, 31 august 1999.


Count support threshold for each item
Support vector
Determine frequent 1-sequences
Create frequent n-sequence set L
IF frequent n-sequence set is empty
YES
STOP algorithm
NO
Increase n
Create n-sequence candidate set from frequent (n-1)-sequences and frequent 1-sequences
Determine support for each candidate in set
Determine frequent n-sequences
Create frequent n-sequence set L
Fig. 1. Diagram of how GSP algorithm works

Most of operations is performed by CUDA kernels on Tesla C2050 card and therefore paralleled, ensuring high computational speed that is needed.

III. Evaluation

Parallel version of GSP algorithm was evaluated both theoretically (with computational complexity analysis) and experimentally - by running it on large real-life and artificial data sets on experimental setup consisting of Intel i7 960@3.20 GHz CPU, ASUS P6T7WS motherboard, 24 GB DDR3 RAM, Seagate ST31000524AS hard drive and Tesla C2050 GPU.

A. Theoretical evaluation

Theoretical evaluation of parallel version of the algorithm is done by calculating computational complexity for each step.

- Step 1 Computational complexity of this step is $O(n)$ - but note that operation is performed with use of kernel and therefore paralleled with up to 1 million threads (cube consisting of 1024 blocks with 1024 threads in each).
- Step 2 Computational complexity of this step is $O(|L| * |F|)$, but as $n >> |L|$ and $n >> |F|$ it can be presented as $O(1)$ - for example for $|A| \leq 1024^2$ step can be performed in just one operation.
- Step 3a Computational complexity of this step is $O(n)$ - but as operation is paralleled note that it comes with large divisor.
- Step 3b Computational complexity of this step is negligible.
- Step 3c Computational complexity of this step is the same as for step 2.

Modified version of algorithm has the same computational complexity as original algorithm, $O(n)$, but as most of time-consuming operations are paralleled and can be reduced towards complexity of $O(1)$, it's time complexity is much lower - it can be expected to work from 2 to 3 degrees of magnitude faster, depending on size of sequence and minimum support threshold (and thus number of frequent sequences found in each step of algorithm).

B. Experimental evaluation

Algorithm was evaluated on experimental setup and working times and results were checked for two real-life large data sets and benchmark sets from [10], with results shown in tables II and III.

CPU (original) version of GSP algorithm was run with use of single processor (100% processor load during whole working time of algorithm). GPU version was run on single Tesla C2050 graphic card (53-96% load during algorithm's run) with use of CUDA 4.0, without any code optimisation and using only global memory. According to [11] after memory optimisation algorithm should speed up 3-10 times.

For each tested value of minimum support threshold parameter (27 for King James Bible [12] data set, 15 for Pan Tadeusz data set, 3-6 for benchmark datasets) algorithm was run from 5 to 10 times (both CPU version and GPU version) to calculate differences between running times. Values shown in tables II and III are averages. Average deviation from shown values was below 1% for both versions of algorithm and also for data set loading step. Maximum deviation was 4% for CPU version, 3% for GPU version and 7% for data set loading.

For longest frequent sequence length and number of frequent n-sequences for each step (the results of algorithm), there was no difference between algorithm runs, both for CPU and GPU version.

TABLE II
WORKING TIMES OF GSP ALGORITHM

Data set name	Minimum support threshold	Working time CPU [s]	GPU [s]	Longest freq. sequence length
King	50000	5618	44	7
James	15000	6492	45	7
Bible	8000	6520	57	9
	1500	6741	131	13
Pan	10000	428	2	2
Tadeusz	1000	451	2	4
	250	634	4	5
	100	670	10	7
Chess	1500	31,593	9,571	24
Connect	100000	4724,109	34,954	3
Mushroom	10000	74,737	2,329	3
Pumsb	100000	8392,599	77,294	4
Pumsb*	10000	$>$ 24h	2759,130	147

TABLE III
RESULTS FOR BENCHMARK DATA SETS

Data set name	Minimum support threshold	Working time [s]	Longest freq. sequence length
Chess	0,15	42,724	65
	0,5	20,881	44
	1,5	9,571	24
	2,5	5,409	14
Connect	10	780,520	60
	50	225,772	22
	100	34,954	3
Mushroom	0,5	22,510	32
	1	17,230	26
	3	11,338	17
	10	2,329	3
Pumsb	3	12399,900	252
	5	10036,169	238
	10	8999,172	222
	100	77,294	4
Pumsb*	3	3717,003	167
	5	2955,191	150
	10	2759,130	147
	100	51,891	4

It can be seen, that with raise of accuracy GPU version has sharper increase of working time then CPU-based version (although it is still lower in seconds). It is caused by lack of optimisation of CUDA device code and lack of memory optimisation presented in [11]. Rising number of n-sequences created and size of calculated matrices causes device part of algorithm to slow down parallel processing that was not prepared for handling such large numbers of data simultaneously and due to this synchronisation times are getting larger and algorithm slows down. This problem can be solved by optimising device code and memory operations according to guidelines in [11], but it was beyond scope of this article.

Despite this that GPU-based version of GSP algorithm, even when run with very high accuracy (minimum support threshold below 0,03%) for both data sets finished its run successfully in time below 3 minutes. For 0,1% minimum support algorithm finishes below 1 minute for larger of data sets. It is clearly seen, that GPU-based version of algorithm is capable of handing large data sets in real-time allowing real-time pattern mining and preprocessing for rule based systems, that was impossible with CPU version.

IV. Conclusions

Results for tested data sets prove that paralleled GSP algorithm run on setup consisting of GPU under CUDA can be successfully applied as embedded system with real-time constraints as it possesses guaranteed response time (due to CUDA synchronisation mechanism) and is fast - two orders of magnitude faster then original, CPU-based version (with even some place for additional speed up, . This modifications allow the use of broader scope of algorithms normally not feasible for embedded systems to be used in embedded systems in real-time constraints.

References

[1] R. Agrawal and R. Srikant, “Mining sequential patterns,” in *Proc. ICDE-95*, 1995, pp. 3–14.

[2] ——, “Mining sequential patterns: Generalizations and performance improvements,” in *Proc. EDBT-96*, 1996, pp. 3–17.

[3] K. Hryniów, “Probabilistic sequence mining - evaluation and extension of ProMFS algorithm,” presented at the IIPhDW2009, 2009.

[4] F. Djahantighi, M. Feizi-Derakhshi, M. Pedram, and Z. Alavi, “An effective algorithm for mining users behaviour in time-periods,” *European Journal of Scientific Research*, vol. 40, pp. 81–90, 2010.

[5] B. Goethals, “Survey on frequent pattern mining,” HIIT Basic Research Unit, Department of Computer Science, University of Helsinki, Tech. Rep., 2003.

[6] J. Huang, C. Tseng, J. Ou, and M. Chen, “General model for sequential pattern mining with a progressive database,” *IEEE Transactions on knowledge and data engineering*, vol. 20, no. 9, 2008.

[7] W. Fang, M. Lu, X. Xiao, B. He, and Q. Luo, “Frequent itemset mining on graphics processors,” in *Proc. Workshop on Data Management On New Hardware (DaMoN 2009)*, 2009.

[8] F. Zhang, Y. Zhang, and J. Bakos, “GPApriori: GPU-accelerated frequent itemset mining,” in *Proc. IEEE International Conference on Cluster Computing*, vol. 0, 2011, pp. 590–594.

[9] D. Luebke and G. Humphreys, “How GPUs work,” *IEEE Computer*, 2007.

[10] B. Goethals, “Frequent itemset mining implementations repository,” 2003-2004. [Online]. Available: http://fimi.cs.helsinki.fi/

[11] *CUDA C Best Practices Guide*, nVidia Corporation, 2011, version 4.0. [Online]. Available: http://developer.nvidia.com/nvidia-gpu-computing-documentation

[12] “King james bible,” 1611 Authorized Version, 1769 Revised Edition. [Online]. Available: http://printkjv.ifbweb.com/AV_txt.zip