Parallel Frequent Patterns Mining Algorithm on GPU

Jiayi Zhou
Department of Computer Science
National Tsing Hua University
Hsinchu 300, Taiwan
jyzhou@mx.nthu.edu.tw

Kun-Ming Yu[†]
Department of Computer Science
and Information Engineering
Chung Hua University
Hsinchu 300, Taiwan
yu@chu.edu.tw

Bin-Chang Wu
Department of Computer Science
and Information Engineering
Chung Hua University
Hsinchu 300, Taiwan
aquavit@pdlab.csie.chu.edu.tw

***Abstract*—Extraction of frequent patterns from a transactional database is a fundamental task in data mining. Its applications include association rules, time series, etc. The Apriori approach is a commonly used generate-and-test approach to obtain frequent patterns from a database with a given threshold. Many parallel and distributed methods have been proposed for frequent pattern mining (FPM) to reduce computation time. However, most of them require a Cluster system or Grid system. In this study, a graphic processing unit (GPU) was used to perform FPM with a GPU-FPM to speed-up the process. Because of GPU hardware delimitations, a compact data structure was designed to store an entire database on GPU. In addition, MemPack and CLProgram template classes were also designed. Two datasets with different conditions were used to verify the performance of GPU-FPM. The experimental results showed that the speed-up ratio of GPU-FPM can achieve 14.857 with 16 times of threads.**

Keywords—frequent pattern mining, parallel processing, graphic processing unit (GPU), OpenCL

I. INTRODUCTION

Both in business and scientific research, there has been a tremendous growth of data that needs to be processed. Extracting information from large amount of data is necessary in making correct and effective decisions. Different methods have been developed to determine the characteristics and interrelationships of data. Association rule learning, classification, clustering, and regression commonly need to mine data. Extraction of frequent patterns from a transaction-oriented database is one of the most important of these. Frequent patterns represent the number of times an itemset appears in a given database. Therefore, if an itemset is frequent it means there are strong relationship between items.

Most frequent patterns mining (FPM) either uses the generate-and-test (*Apriori*-like) [1] or the pattern growth approach (*FP-growth*) [2]. The core concept of the Apriori-like approach is that if the length k pattern is not frequent in the database, then the super-pattern (length $k+1$) will not be frequent. It uses a bottom-up approach, extending frequent subsets one item at a time. Since the generated candidates are independent, this approach is suitable for parallelization. Although many Apriori-like methods have been proposed [3-4], the computation time increases significantly when the database contains a large number of transactions. Some studies [5-7] apply parallel and distributed techniques to speed-up the mining processes. However, most of them require a high performance computing system, e.g., a Cluster or Grid System.

In recent years, the graphic processing unit (GPU) has developed from a 3D rendering device for games to a general-purpose computing device [8]. While the GPU can only execute some simple instructions and functions, compared with CPU, it has a large number of computing units. Moreover, using GPU as a high-performance computing device which does not only reduce the deployment cost but also saves on maintenance. GPU programming strategies can be classified according to either graphic APIs or GPU programming language. It is difficult for developers to use the graphic APIs since they need understand the graphic hardware and encode their data to graphic vectors. The most serious problem is the use of previously designed parallel algorithms. Therefore, GPU programming language is currently being used to develop GPU-enabled programs. NVIDIA and ATI have been proposed as GPU programming language by CUDA [9] and Stream [10] respectively. Programs can be written in C programming language (C99) to use the power of GPU. However, CUDA can only be used on NVIDIA's GPU and vice versa. Therefore, OpenCL [11] was proposed in 2009 to deal with this situation. The program design with OpenCL not only can be executed on different brand GPU devices, but also on multi-core CPUs.

In this study, the GPU-FPM algorithm was used to speed-up the mining processes for FPM when using GPU. Although GPU is a powerful computing device, there are limitations: like memory size, memory latency, etc. Therefore, the data structure has to be re-designed for the FPM algorithm on GPU. Since the verification time dominates computation time, the main goal of GPU-FPM is to use GPU to verify generated candidates in order to speed-up the FPM processes. Compact Data Structure, MemPack, and CLProgram class were used to achieve this. For verifying the GPU-FPM performance, it was implemented on Microsoft Windows with OpenCL 1.0, in addition, data generated by an IBM Quest Data Generator [12] was used. The proposed algorithm was tested under different conditions, including different transaction lengths, threads,

--
* This work is partially supported by National Science Council. (NSC 98-2221-E-216-023)
† The corresponding author.

block sizes, and thresholds. The experimental results showed that GPU-FPM significant reduced the computation time with increasing threads. The speed-up ratio achieved 14.857 with 16 times of threads (in case of the T40I10D100K threshold being 1900, and block size 10). Moreover, even in the worst case, GPU used 89.942% of the execution time. This means that GPU-FPM efficiently used the GPU computing power.

The rest of the paper is organized as follows: In section 2, the FPM, GPU, and OpenCL are described. The proposed GPU-FPM is introduced in section 3 and the experimental results are illustrated in section 4. Finally, the conclusions discussed in section 5.

II. PRELIMINARIES

A. Frequent Pattern Mining (FPM)

The main concept of FPM is to find the number of times a given pattern appears in a database. *FPM* is defined as follows:

Let D be a transactional database consisting of a set of transactions $T_1, T_2, \ldots, T_n$: $D = \{T_1, T_2, ..., T_n\}$. Let I be a set of items $i_1, i_2, \ldots, i_m$, a set $X = \{i_1, i_2, \ldots, i_k\} \subseteq I$ called an itemset or a k-itemset if it consists of k items. The support of an itemset X is the number of transactions containing X.

$$\text{support}(X, D) = \left|\{i \mid X \subseteq T_i, X \subseteq I\}\right| \text{ for } i = 1 \ldots n$$

An itemset is called frequent if the support is greater than or equal to the given absolute minimal threshold ξ. FPM is given a set of items I, a database D, and a minimal threshold ξ, then find $\text{FP}(D, \xi)$.

$$\text{FP}(D, \xi) = \{X \subseteq I \mid \text{support}(X, D) \geq \xi\}$$

B. Graphic Processing Unit (GPU)

GPU is a parallel-oriented computing device. It always consists of massive processing units to perform mathematical computing. It used to be used as a co-processor CPU for games and 3D design applications. The DirectX 9 proposed in 2005, has taken graphics cards to the next generation because of vertex and pixel shaders being integrated in general-purpose processing units—introducing the universal shader. The mainstream GPU has hundreds to thousands computing units. Each unit can be regarded as a simplified CPU. Compared with the multicore CPU, the number of processing units has also increased. Consequently, GPU also has a whole new application—general-purpose computing on graphics processing units (GPGPU).

C. OpenCL

The GPU programming language can be classified as graphic APIs (DirectX, OpenGL, etc.), GPU programming language (NVIDIA CUDA [9], ATI Stream [10], OpenCL [11], etc). Previously, GPU programming required developers with in-depth knowledge of graphics programming and hardware. In order to utilize the computation resources on GPU, developers had to encode data to a graphic vector, and then use the DirectX or OpenGL functions to perform rendering. After that, the rendered data had to be decoded. This procedure not only required graphic programming knowledge, but also depended on different GPUs. Recently, CUDA and Stream have been proposed by NVIDIA and ATI. Both of them provide C interface and allow developers to adapt the hardware, e.g., number of processing units, size of local and global memory. However, previous frameworks could only be used with the respective GPUs, e.g., CUDA could only be executed on NVIDIA's GPUs.

In order to solve this situation, the Khronnos Group and many industry-leading companies created the OpenCL. OpenCL is an open and cross-platform parallel heterogeneous programming system. It provides a uniform programming environment for developers to write efficient and portable codes using a diverse mix of multi-core CPUs, GPUs, and other processors.

III. GPU-FPM

The goal of GPU-FPM was using massive processing units on GPU to speed-up the FPM procedures. However, each processing unit on GPU can only perform simple instructions. Another important issue is memory size and access latency. Therefore, the algorithm and data structure had to be re-designed for GPUs to fully utilize its computation resources. ***Figure 1*** illustrates the architecture of GPU-FPM. GPU-FPM has the following features: (1) data handling between CPU and GPU, (2) compact data structure, and (3) highly parallel.

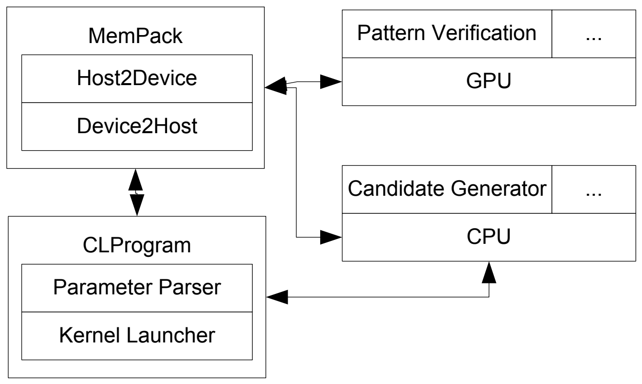


Figure 1. Architecture of GPU-FPM

A. Compact Data Structure

The memory access latency on GPU is very high, and it limits the computational speed-up ratio. Therefore, reducing the number of fetchings of memory improves the performance. It fetches the memory many times if used directly on a transaction-oriented database. This is because the entire transaction needs to be scanned for verifying each single itemset. Consequently, a transaction identification set (*Tidset*) was used to directly select transactions instead of scanning whole databases. Tid and Tidset were defined as follows:

$$\text{Tid}(i_j) = \{i_j \cap T_k \neq \varnothing\} \text{ for } k = 1 \ldots n$$

$$\text{Tidset} = \{\text{Tid}(i_j)\} \text{ for } j = 1 \ldots m$$

For example, if transactions 1 and 3 contain item i_1, $\text{Tid}(i_1) = \{1,3\}$, then a whole transaction-oriented database is represented by Tidset. In order to store the Tidset to memory on GPU, TidValue and TidIndex arrays were used to represent Tidset. ***Figure 2*** is an example of TidValue and TidIndex arrays. The TidValue array stored the Tid of each item, e.g., $\text{Tid}(i_1) = \{1,3\}$, $\text{Tid}(i_2) = \{1,2,5\}$, $\text{Tid}(i_3) = \{2\}$, etc. (***Figure 2*** (a)) The boundary of each item on the TidValue array was determined by the TidIndex array. The TidIndex stored each items start and end position, e.g., item i_4 ranging from 6 to 10 means six cells were used for i_4 in TidValue array and values were stored from TidValue[6] to TidValue[11]. Therefore, the information required for mining was transformed from database to two arrays.

1	3	1	2	5	2	4	5	7	10	15	18	…
Tid(i_1)		Tid(i_2)			Tid(i_3)	Tid(i_4)						

(a) TidValue

0	1	2	4	5	5	6	11	...
i_1		i_2		i_3		i_4		

(b) TidIndex

Figure 2. Example of TidValue and TidIndex

B. *GPU-FPM*

Compared with CPU, GPU is special hardware with massive processing units. GPU processing is in single instruction, multiple data (SIMD) and there is no support recursion on it. Therefore, a compact data structure was designed and implemented to store necessary data for mining on GPU. The FPM could be roughly summarized to the following steps: load database, generate candidate itemset, and verify the candidate itemset frequently or not. Candidate itemset verification usually dominates computing time. Therefore, in this study, GPU was used to reduce candidate verification time.

GPU-FPM was an Apriori-based mining algorithm and it generated and verified the itemset to produce frequent patterns. Since memory access between CPU and GPU is a common operation, MemPack was designed to lower GPU programming complexities. MemPack is C++ class template that provided abilities to store different types of data, e.g., int, float, customized structure, class, etc. Two transfer functions: *Host2Device* and *Device2Host* and two memory control functions: *ReleaseHost* and *ReleaseDevice* were also provided. Moreover, the *CLProgram* class was also designed to have the following abilities: allow arbitrary number of parameters, bind arbitrary of MemPack, launch with arbitrary number of threads, and launch with CPU. The GPU-FPM algorithm follows:

Algorithm GPU-FPM

```
Input: a transaction database $D$ and a given minimum threshold $\xi$.
Output: a complete set of frequent patterns $\mathrm{FP}(D,\xi)$.

1.  Load $D$ from disk.
2.  Generate Tidset via scanning the $D$ and store it on hash table.
3.  Transform hash table to compact array structure—TidValue and TidIndex.
4.  Create MemPacks mpTidValue and mpTidIndex to store TidValue and TidIndex.
5.  Perform Host2Device to copy mpTidValue and mpTidIndex to GPU.
6.  Use prefix tree data structure to generated candidate itemset.
7.  Create MemPack mpCandIS to store generated candidates.
8.  Perform Host2Device to copy mpCandIS to GPU.
9.  Create MemPack mpResults for storing results.
10. Create CLProgram clProg to store related parameters and bind the mpTidValue, mpTidIndex, mpCandIS, and mpResults.
11. Perform launch kernel of clProg (on GPU)
    a. Each processing unit (PU) allocated a set of candidate itemsets (CIs)
    b. for each CI in CIs
       i.  PU compute the support of CI according to mpTidValue and mpTidIndex
       ii. If support of CI greater than or equal to given threshold $\xi$ then set it is frequent on mpResults, else is not frequent.
12. Wait until kernel code executed.
13. Perform Device2Host of mpResults to store the results.
14. Perform Step 6 until all candidates generated and verified.
```

IV. Experimental Results

In order to evaluate the performance of the proposed algorithm, GPU-FPM was implemented along with OpenCL library and Visual C++ on Microsoft Windows. Synthesized datasets generated by IBM's Quest Synthetic Data Generator were used to verify the algorithm. The hardware and software configurations are given in ***Table 1***. The algorithm evaluated with different transaction lengths, different threads, different block sizes, and different thresholds. ***Table 2*** gives the details of the dataset testing.

Table 1. Hardware and Software configuration

Item	Description
CPU	AMD Phenom II X4 965 3.4 GHz
Memory	8G DDR3 memory
GPU	ATI Radeon HD 5850 with 1440 stream processing units and 1G DDR5 memory
OS	Microsoft Windows 7
Compiler	Microsoft Visual C++ 2008 w/ SP1
SDK	ATI Stream SDK 2.0 w/ OpenCL 1.0 support

Table 2. Statistical Characteristic of Datasets

Dataset	Avg Trans Len	Avg Len of Max Pattern	No of Trans
T10I4D100K	10	4	100
T40I10D100K	40	10	100

A. Various Thread NumberrsQuantities

In this section, two datasets with different threads and thresholds were used to verify the performance of GPU-FPM. ***Figure 3*** and ***Figure 4*** illustrate the computation time of various thresholds and threads. The computation time of the same threads was affected by the threshold. A smaller threshold refers to a smaller degree of support becoming a frequent pattern. There were 385 and 13,253 frequent itemsets with $\xi = 1000$ and $\xi = 200$, respectively. The speed-up ratio is depicted in ***Figure 5*** and ***Figure 6***. For 16 times of threads, the best speed-up ratio was 13.066. Even in the worst case, it was 11.037. The average speed-up ratio was 12.576.

B. Various Block Sizes

In this section, GPU-FPM used different block sizes to verify the performance. The block size is the number of candidates that each processing unit on GPU deals with at each kernel launch. A small block size implied that the kernel had to be launched more times. ***Figure 7*** and ***Figure 8*** show the computation time of various block sizes. The *Var* stands for the block size changing with the number of threads, and the block size ($blockSize$) and number of threads ($noOfThread$) had the following relationship.

$$blockSize * noOfThread = 1024$$

Although a larger block size saved a bit on computation time (block size from 2 to 10, saving 0.826 second in case of T10I4D100K with 1024 threads), it only had a small influence on computation time. In some cases, larger block size even caused more computation time. The speed-up ratio is shown in ***Figure 9*** and ***Figure 10***. The trend of the speed-up ratio could not be observed from the results. According to the experimental results, the block size had no significant effect on the computation time.

C. Computation Time used by CPU and GPU

Finally, the computation time used by CPU and GPU is depicted in ***Figure 11*** and ***Figure 12***. GPU occupied most of the computation time in all cases. This means that the GPU-FPM algorithm on GPU had the biggest workload. It also pointed out that pattern verification required much computing resources than pattern generation. For 1,024 threads, GPU occupied 93.837% computation time on average.

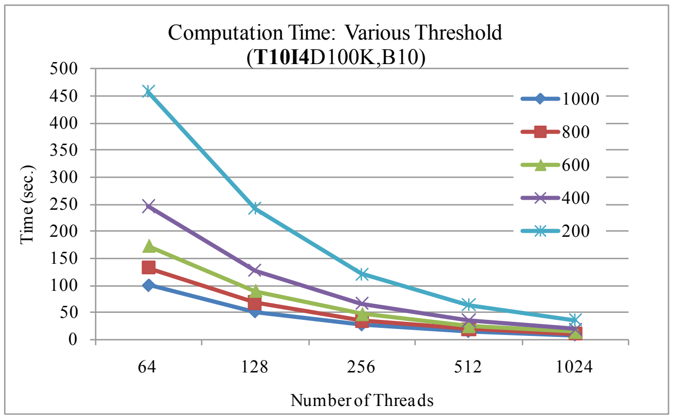


Figure 3. Computation Time of Various Thresholds (T10I4D100K, B10)

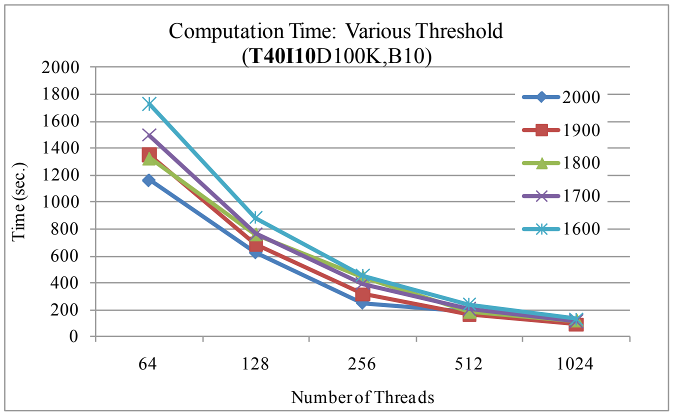


Figure 4. Computation Time of Various Thresholds (T40I10D100K, B10)

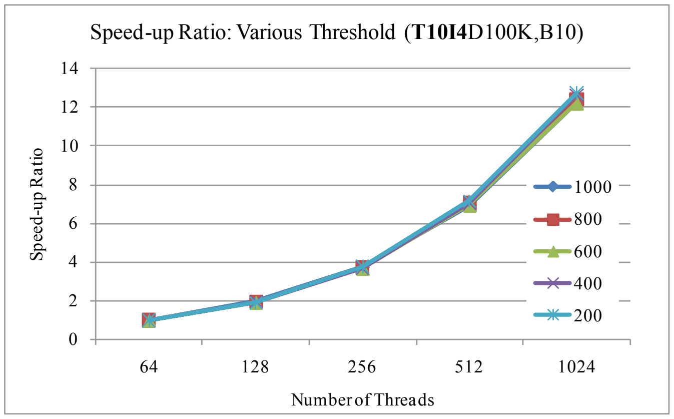


Figure 5. Speed-up Ratio of Various Thresholds (T10I4D100K, B10)

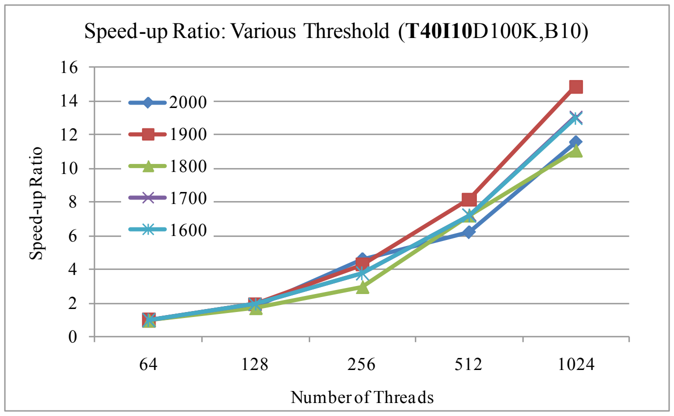


Figure 6. Speed-up Ratio of Various Thresholds (T40I10D100K, B10)

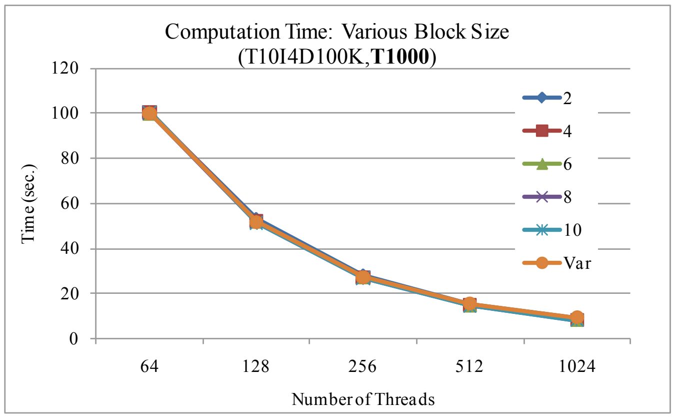


Figure 7. Computation Time of Various Block Sizes (T10I4D100K, T1000)

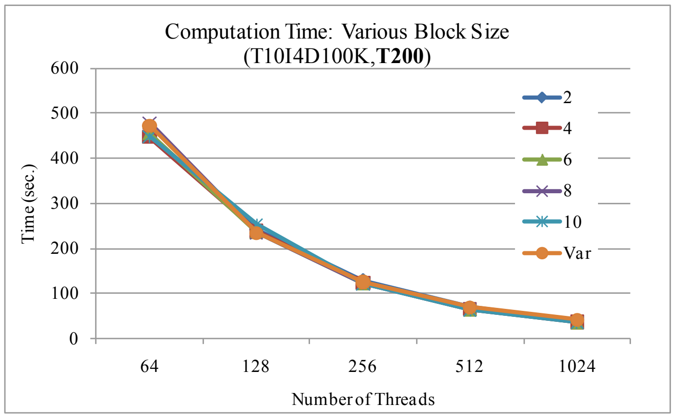


Figure 8. Computation Time of Various Block Sizes (T10I4D100K, T200)

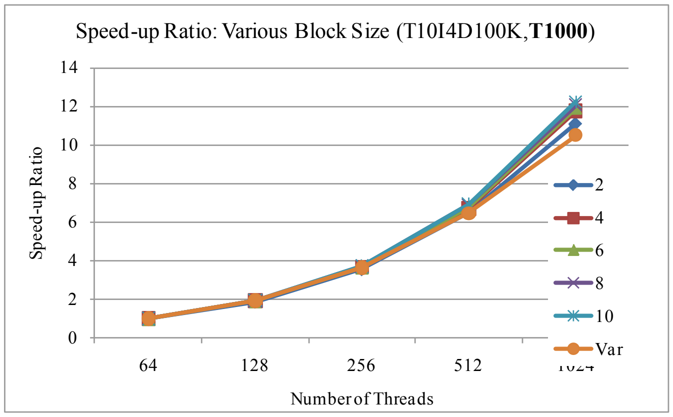


Figure 9. Speed-up Ratio of Various Block Sizes (T10I4D0100K, T1000)

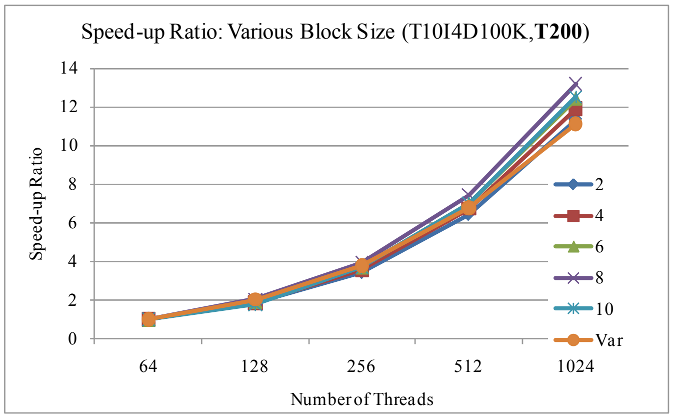
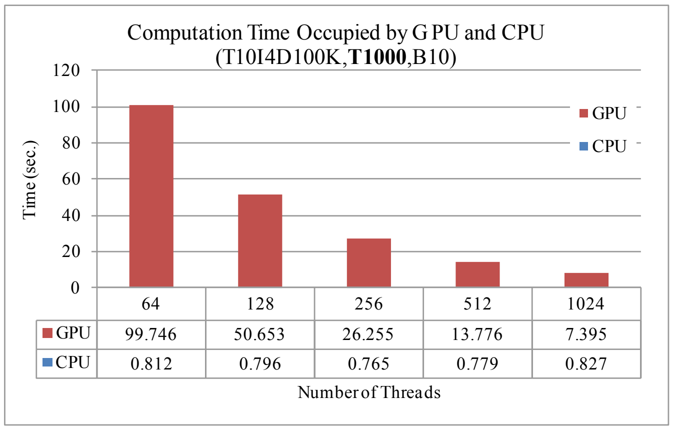
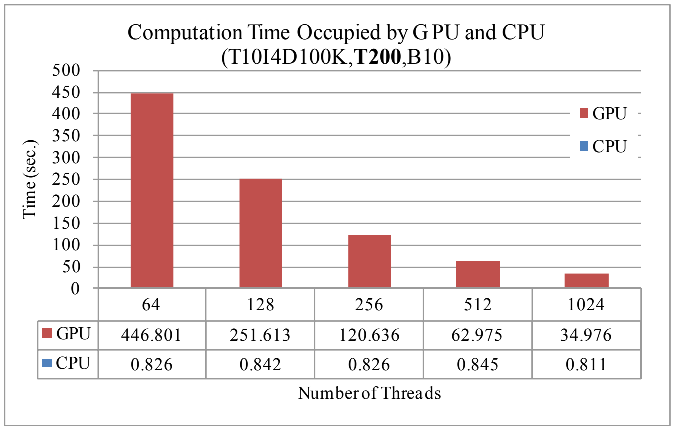


Figure 10. Speed-up Ratio of Various Block Sizes (T10I4D0100K, T200)



	64	128	256	512	1024
🟥 GPU	99.746	50.653	26.255	13.776	7.395
🟦 CPU	0.812	0.796	0.765	0.779	0.827

Figure 11. Computation Time Occupied by GPU and CPU (T10I4D100K, T1000, B10)



	64	128	256	512	1024
🟥 GPU	446.801	251.613	120.636	62.975	34.976
🟦 CPU	0.826	0.842	0.826	0.845	0.811

Figure 12. Computation Time Occupied by GPU and CPU (T10I4D100K, T200, B10)

V. CONCLUSIONS

Frequent pattern mining (FPM) is important and fundamental in data mining. Most FPM methods can be classified as Apriori-like or FP-growth-like. However, the computation time increased significantly when the number of transactions grew. In this study, a GPU based parallel algorithm—GPU-FPM was used to speed-up the mining processes. In order to conform to GPU hardware delimitation, a compact data structure was used to store entire database in GPU. Moreover, two template classes, MemPack and CLProgram were also used. Two datasets with different conditions were used to verify the performance of GPU-FPM. The speed-up ratio was 12.57 and 7.11 for 16 and 8 times of threads on average. In addition, most computation time was occupied by GPU because of all pattern verification processes being performed by it.

REFERENCES

[1] R. Agrawal, and R. Srikant, “Fast algorithms for mining association rules,” in International Conference on Very Large Data Bases, 1994, pp. 487-499.

[2] J. Han, J. Pei, Y. Yin *et al.*, “Mining frequent patterns without candidate generation: A frequent-pattern tree approach,” *Data Mining and Knowledge Discovery,* vol. 8, no. 1, pp. 53-87, 2004.

[3] E. Lazcorreta, F. Botella, and A. Fernández-Caballero, “Towards personalized recommendation by two-step modified Apriori data mining algorithm,” *Expert Systems with Applications,* vol. 35, no. 3, pp. 1422-1429, 2008.

[4] J. Park, M. Chen, and P. Yu, “An effective hash-based algorithm for mining association rules,” *ACM SIGMOD Record,* vol. 24, no. 2, pp. 175-186, 1995.

[5] A. Javed, and A. Khokhar, “Frequent pattern mining on message passing multiprocessor systems,” *Distributed and Parallel Databases,* vol. 16, no. 3, pp. 321-334, 2004.

[6] K.-M. Yu, J. Zhou, T.-P. Hong *et al.*, “A Load-Balanced Distributed Parallel Mining Algorithm,” *Expert Systems with Applications,* vol. 37, no. 3, pp. 2486-2494, 2009.

[7] M. Chen, C. Huang, K. Chen *et al.*, “Aggregation of orders in distribution centers using data mining,” *Expert Systems with Applications,* vol. 28, no. 3, pp. 453-460, 2005.

[8] D. Luebke, M. Harris, N. Govindaraju *et al.*, "GPGPU: general-purpose computation on graphics hardware." p. 208.

[9] NVIDIA. "Compute Unified Device Architecture (CUDA)," http://www.nvidia.com/object/cuda_home_new.html.

[10] ATI. "Stream SDK," http://developer.amd.com/gpu/ATIStreamSDK/.

[11] OpenCL. "OpenCL," http://www.khronos.org/opencl/.

[12] R. Agrawal, and R. Srikant, "Quest Synthetic Data Generator. IBM Almaden Research Center, San Jose, California," 2009.