Abstract—Frequent pattern mining (FPM) is important in data mining field with Apriori algorithm to be one of the commonly used approaches to solve it. However, Apriori algorithm encounters an issue that the computation time increases dramatically when data size increases and when the threshold is small. Many parallel algorithms have been proposed to speed up the computation using computer clusters or grid systems. GPUs have also been applied on FPM with only few adopting OpenCL although OpenCL has the advantage of being platform independent. Thus, the aim of this research is to develop efficient parallel Apriori strategy using GPU and OpenCL. Our novel method, Candidate Slicing Frequent Pattern Mining (CSFPM) algorithm, improves over the previous method by slicing candidate information to better balance the load between processing units. This strategy is proved to be more efficient according to our experiments. For example, CSFPM is at most 2.6 times faster than the previous method. Therefore, CSFPM is an efficient parallel Apriori algorithm which can reduce computation time and improve overall performance.

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

I. INTRODUCTION

Frequent Patterns Mining (FPM) extracts implicit patterns which show correlations between items from enormous transactional data sets. FPM has many applications such as the exploration of correlations between various diseases, medicine, and death. Another popular usage is, by mining the sales records of a store, to discover the correlations between customers and sale items and among sale items. FPM could be used to solve many problems, while different methods have been proposed.

Apriori algorithm [2] and Frequent-Pattern Tree (FP-tree) algorithm [3-4] are two commonly used methods for FPM. Apriori algorithm, commonly applied in Market Basket [1], is simple, easy to understand, and easy to implement although it requires repeatedly scanning of database. Apriori algorithm is more suitable for parallel processing than FP-tree method since balancing the load while parallelizing tree structures has been an issue of FP-tree [6]. However, Apriori algorithm experiences the fast computation acceleration problem when data set size increases and when the threshold is small. The maturity of data storage technology has tremendously increased the size of searched databases. While mining the database, the concern that some important information may be removed with high threshold value encourages low threshold pattern mining which costs much computing time.

The hardware development technology has been advanced so quickly that the CPU structure has moved from single-core to multi-core which can be applied in parallel processing. Another approach to enhance the performance is to make use of multiple hosts or clusters. Many parallel algorithms have been proposed to speed up the computation time of Apriori algorithm, but most of them are designed for computer clusters or grid systems. If the use of OpenMP can be changed to General-purpose computing on graphics processing units (GPGPU) [7-8] technology, it would achieve even better performance [9-10].

Compute Unified Device Architecture (CUDA) [11] is a GPGPU language which is used to solve FPM [12]. On the other side, OpenCL (Open Computing Language) [13-14] is a cross-platform and cross-operating system language which is more convenient than CUDA is. Nevertheless, OpenCL has not been as widely used as CUDA by developers.

In this study, Slicing Frequent Pattern Mining (CSFPM) algorithm was proposed to use GPU and OpenCL to develop efficient parallel Apriori algorithm for FPM problem. GPU is a computing device with many features different from CPU such as slower clock speed and limited memory size. Therefore, we need to fine tune the algorithm to get better performance. Since GPU clock is slower than CPU clock, assigning a thread to check one candidate is a good start to achieve parallel efficiency. Moreover, whether more speed up can be reached is interesting to explore. Thus, CSFPM tries to assign a thread to only checking one transaction in a candidate item. This strategy can reduce the processor waiting time since the load between processing units is more balanced. The results shows that, on the same platform, the CSFPM is faster than the previous method [16]—Parallel Frequent Patterns Mining Algorithm on GPU (GPU-FPM)—which parallelizes the algorithm using coarser approach.
The paper is organized as follows. Section 2 describes related work including Apriori Algorithm, Graphic Processing Unit, OpenCL and GPU-FPM method. Section 3 explains our method CSFPM Algorithm. Section 4 displays the experimental results and Section 5 concludes this paper.

II. RELATED WORK

A. Apriori Algorithm

Apriori Algorithm was proposed by R. Agrawal and R. Srikant in 1994 [2]. Its goal is to find those implicit and non-trivial frequent data patterns of interest from the data set or database. Whether a pattern is frequent or not is determined by a predefined threshold. A frequent pattern must occur more often in the database than, if not as often as, what the threshold defines. The Apriori method starts from finding frequent patterns with only one item. Two frequent patterns can be merged into a higher-level candidate pattern with more items. Then, each newly-generated candidate pattern is verified by checking whether its occurrence count exceeds the specified threshold or not. This step requires database scanning. The candidate generation and verification process repeats until no more new patterns can be generated.

For example, a transaction set in database D is \{T_1, T_2, T_3, T_4\}. The item information for each transaction is listed in Table 1. The number of occurrence for each item is displayed in Table 2. If the threshold is 50%, a frequent pattern needs to appear in at least two transactions. Among all the one-item candidate patterns, only \{A\}, \{B\}, and \{D\} are frequent. Then, the candidate set for 2-item frequent patterns is \{A, B\}, \{A, D\}, \{B, D\}\} and only \{A, B\} satisfies the frequent pattern condition as shown in Table 3. The process of generating frequent patterns stops here since no more new candidate can be generated.

<table>
<thead>
<tr>
<th>Transaction id</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>{A,B,C}</td>
</tr>
<tr>
<td>T2</td>
<td>{A,B}</td>
</tr>
<tr>
<td>T3</td>
<td>{A,D}</td>
</tr>
<tr>
<td>T4</td>
<td>{D,E}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Items</th>
<th>Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3: Only \{A, B\} appears at least two times

<table>
<thead>
<tr>
<th>Items</th>
<th>Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>A,B</td>
<td>2</td>
</tr>
<tr>
<td>A,D</td>
<td>1</td>
</tr>
<tr>
<td>B,D</td>
<td>0</td>
</tr>
</tbody>
</table>

The creation of Transaction Identification (Tid) set was proposed [2] to save the time of scanning database after the first pass. For example, if the transaction list for Item 1 is \{1, 2, 3, 4, 5\} which means Item 1 appears in transactions 1, 2, 3, 4, and 5. The transaction list for Item 2 is \{2, 4, 6\}, and the transaction list for Item 3 is \{1, 3, 5\}. The Tid tables—TidValue and TidIndex—can then be built as in Figure 1 so that the verification of candidate patterns can be done by reading them. These two tables become smaller and smaller when higher-level candidate pattern sets are generated.

![Figure 1: The Transaction Identification Set](image)

B. Graphic Processing Unit (GPU)

A GPU, a microprocessor for image processing, helps the CPU for graphics processing. There are hundreds or even thousands of computing units in a GPU. Each computing unit is like a simplified core of CPU. General-purpose computing on graphics processing units (GPGPU) was proposed to use a GPU as a CPU to provide non-graphic computing capabilities. The current GPGPU technologies include OpenCL and CUDA (Compute Unified Device Architecture). Since a GPU core is slower than a CPU core, it is normally assigned simple computing functions. The number of GPU cores is more than that of CPU cores, so GPU is suitable for parallel computing.

C. Open Computing Language (OpenCL)

The preliminary work of OpenCL was finished by AMD, IBM, Intel, and NVIDIA while it was initially developed by Apple Inc. The draft was submitted to the Khronos Group by the Apple Inc and then the GPGPU Khronos working group was established on June 16, 2008. Then OpenCL 1.1 was released on June 14, 2010.

OpenCL is a framework which allows C program development in heterogeneous platforms; that is, the framework can be applied in any system which is composed of different CPUs, GPUs, and other computing platforms. It is able to perform in different operating systems as long as the OpenCL library is installed. The CPU and GPU can then communicate with each other and work together by applying the appropriate C++ file for CPU and Kernel file for OpenCL on GPUs to perform parallel computation with GPU.

D. Parallel Frequent Patterns Mining Algorithm on GPU

In 2010, Zhou et al. [16] proposed an OpenCL GPGPU frequent pattern mining algorithm GPU-FPM. It is a coarse grain parallel Apriori algorithm using Tid tables. In the method, the Tid table entries of the items for each candidate pattern are assigned to one computing thread for comparison. For example, when the number of threads is 1024, it processes
1024 candidate patterns at the same time with each thread responsible for one candidate pattern verification. This method is suitable when using fast computation units. Since each candidate pattern may have different number of entries to process, some threads may need to wait for other threads to finish their tasks. Therefore, they are idle until all threads are done with the current processing.

III. CANDIDATE SLICING FREQUENT PATTERN MINING ALGORITHM (CSFPM)

Parallel Apriori Algorithm has been the focus of many researchers. Its issues are the massive amount of computation involved with large data set and the difficulty of load balancing between processing units. This work tries to efficiently parallelize the Apriori algorithm using GPU’s highly parallelized feature in a personal computer. Note that it can be more efficient if the Apriori Algorithm is parallelized in a GPU with further task decomposition, since each GPU core is slow although it has more cores than CPU.

The GPUs have highly parallel structure which makes them efficient in manipulating large set of data in parallel. Inspection of candidate patterns takes the most time in Apriori Algorithm and thus can be improved by applying GPU’s parallel computation capability. There are several basic steps when applying Apriori algorithm.

1. Scan the database and calculate the Tidset.
2. Find the first-level candidate set.
3. If the candidate set is not empty, find out whether each candidate pattern in the set is frequent or not. Otherwise, got to step 5.
4. Compose one level higher candidate set and go to step 3
5. All frequent patterns are found.

Tid set (Tidset) can be used to reduce the time of data access, compared with the database scanning for each level of candidate set processing. However, if Tidset in step 1 is generated by the GPU, it takes more time than that by only using CPU. Thus, data pre-treatment is not suitable for GPU in this case. It is better to use CPU to compute Tidset and to store it in GPU memory.

It’s natural to use one thread to verify whether a candidate pattern is frequent or not. There is a drawback in this type of task assignment; that is, the numbers of comparisons required in verifying each pattern is not the same and the variation can be very large. For example, if using a straightforward method, the amount of matches to check whether items 1 and 2 is a frequent pattern or not is 5*3 = 15 while that for checking items 2 and 3 is 3*3 = 9 as shown in Figure 1. Since only when all threads finishes their processing, the control come back to CPU, many computation units are in idle state which decreases the parallization efficiency and effectiveness. How to parallelize the frequent pattern matching more efficiently becomes an essential issue.

Consider three level-one frequent patterns: item 1, item 2, and item 3. Then, the next level candidate patterns are item 1-2, item 1-3, and item 2-3. When verifying each candidate pattern such as item 1-2, we call the first item (Item 1) compared item and the rest of the items in the same candidate pattern comparing item(s), i.e. item 2 in this example. If a candidate pattern owns three or more items, then there are two or more comparing items.

The idea of this slicing algorithm is to slice the compared item and dedicate one thread to each transaction of it. This transaction is called sliced transaction for this thread. For item 1-2 in the example, there are five threads assigned for five transactions of item 1. To facilitate the processing, two arrays of information are passed to the GPU. One array stores the candidate pattern pair (CPP) information such as [1, 2, 1, 3, 2, 3] meaning item 1-2, item 1-3, and item 2-3. Another array is the sliced thread assignment (STA) number [4, 9, 12] which indicates the thread numbers for the last transaction of all compared items in the CPP array. As Figure 2 shows, there are three compared items for three entries in STA. Five transactions—1, 2, 3, 4, and 5—are for item 1, so the first entry of array STA is 4 (0 to 4 for item 1). The second compared item is also item 1, so the thread numbers for it are from 5 to 9. The third compared item is item 2 with transactions 2, 4, and 6, so the third entry in STA is 12. Thus, for example, thread 10 is only responsible to verify whether transaction 2 is also a member of TidValue, [1, 3, 5], for item 3 or not.

A thread returns the sliced transaction number if there is a match; it returns 0 otherwise. Considering the result, threads 0 to 4 only check whether item 1 and item 2 share any transaction or not. If any one of these five threads returns the compared transaction number, we get one occurrence of this pattern item 1-2. The result of the threads 0 to 12 is shown in Figure 3.

From the result, item 1-2 has two occurrences whereas item 1-3 has three and item 2-3 has none. If the threshold is 40%, then both item 1-2 and item 1-3 are frequent patterns.

The algorithm of CSFPM is described as below.

**Algorithm CSFPM**

**Input:** a transaction database $D$ and a given minimum threshold.

**Output:** a complete set of frequent patterns.

Figure 2: The thread assignment for CPP [1,2,1,3,2,3].

<table>
<thead>
<tr>
<th>Thread</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tid</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>STA</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: The result from the thread.

<table>
<thead>
<tr>
<th>Thread</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Results</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Item</th>
<th>1-2</th>
<th>1-3</th>
<th>2-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item1-2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Item1-3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Item2-3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2346
CPU:
1. Start the CL program to be executed by the GPU.
2. Load D from disk.
3. Generate Tidset via scanning D and store it on CPU memory.
4. Generate and verify first level frequent patterns.
5. Transform the 2D Tidset table for first level frequent patterns into 1D arrays TidValue and TidIndex.
6. Allocate memory space in GPU for TidValue and TidIndex.
7. Stores arrays TidValue and TidIndex into GPU memory.
8. Generate enough candidate patterns (itemset)—arrays CPP and STA—for the GPU threads to process.
9. Allocate memory space in GPU for candidate patterns.
10. Store candidate patterns in GPU.
11. Allocate memory space in GPU to save the results.
12. Wait until GPU finishes its program execution.
13. Result manipulation:
   a. Retrieve the results from GPU and save them in CPU memory.
   b. Calculate the number of nonzero entries of each candidate itemset comparison.
   c. If this number is larger than or equal to what the threshold indicates, this pattern is frequent.
14. Repeat the process from step 8 to 13 until all the same level candidate patterns are done.
15. Move to next level candidate set generation and perform Steps 8-14 until all candidates are generated and verified.

GPU kernel thread function:
1. Retrieve the sliced transaction associated with this task.
2. Retrieve the comparing item transaction set(s) associated with this task.
3. Verify whether the sliced transaction exists in all comparing item transaction set(s) or not.
4. Skip the rest of verification step if no match, in any intermediate step, can be found.
5. Write down the result of this thread function.

IV. EXPERIMENTAL RESULTS
Experiments are conducted to verify the performance of the method CSFPM on GPU. The languages used are OpenCL and C++ on Visual Studio while the operating system is Microsoft Windows 7. Input data is from the IBM data generator [16]. Table 4 depicts the hardware and software configurations of the experiments. Table 5 is statistical characteristic of datasets. The time is averaged from 10 runs.

Table 4: Hardware and software configurations
<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>AMD Phenom II X4 965 3.4 GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>4G DDR3 memory</td>
</tr>
<tr>
<td>GPU</td>
<td>ATI Radeon HD 5850 with 1440 stream</td>
</tr>
</tbody>
</table>

A. Smaller data set and different number of threads.
The computation time is shown on Figure 3 when using the same data and threshold but different number of threads. The time used is the least when the number of threads is 16384. It only takes 3.7474 seconds. When the number of threads increases, large amount of data is fed to the GPU at all times which causes threads to be busy in swapping that wastes much time. Also, the amount of information internal to GPU also increases which consumes more time.

B. Larger data set and different number of threads
When increasing the amount of data, CSFPM also effectively reduces the computing time as in Figure 4. The computing time is the least when the number of threads is 65537. It only takes 38.8598 seconds. More time is needed when the number of threads is raised. When the number of thread is 2048 or 4096, the speedup is not that much until the number of threads goes to 8192.
C. CSFPM compared with CPU only

This experiment compares the computation time of CSFPM with that of using only CPU with data T10I4D100K and threshold 200. As in Figure 5, it takes 14.5326 seconds on CPU while CSFPM (CPU+GPU) only takes 3.7474 seconds. The speedup is 3.878. To use both GPU and CPU performs better than to only use CPU.

D. CSFPM Compared with GPU-FPM on the same platform

This experiment is conducted using ATI Radeon HD 5850, data T10I4D100K and threshold 200 on the same CPU and GPU, but with different methods. As in Figure 6, CSFPM is better than the previous method GPU-FPM on the same platform and number of data. This is true when applying different number of threads. For example, when the number of threads is 1024, it takes 13.756 seconds while GPU-FPM takes 35.787 seconds. The speedup is 2.6.

V. CONCLUSIONS

Frequent pattern mining (FPM) is an important problem and Apriori algorithm is a commonly used approach for it. However, the computation time suffers when the size of the data is increased or the threshold is very small.

GPUs are highly parallel devices with limited memory and slow input and output. GPGPU is a trend and most of the existing methods for FPM cannot be directly converted for GPGPU usage efficiently. Thus, there is big room for improvement as this study has shown.

We proposed a novel approach, CSFPM, to parallelize the candidate pattern matching in which each compared item is sliced into transaction granule for one GPU thread to process. CSFPM performs much better when compared with previous method GPU-FPM. The speedup is 2.6 when using ATI Radeon HD 5850, data T10I4D100K and threshold 200. Another experiment shows that if both GPU and CPU work together (CSFPM), the speedup is 3.878 compared with only using CPU. Thus, CSFPM is an efficient way to parallelize the FPM tasks in GPU.

REFERENCES


