GPU-Prepost: A GPU-Accelerated Frequent Pattern Mining Algorithm Based on PrePost

Tianyuan Jiang & Xin Lv, Zhihong Deng
School of Electrical Engineering and Computer Science
Peking University
China

***Abstract*—Frequent pattern mining has been an important and well-researched data mining task in recent years. In this paper, we propose a paralleled algorithm based on the Prepost algorithm, which obtains a better performance compared with other classical algorithms.**

Keywords—Frequent Pattern Mining; Prepost; Data Mining; GPU-Accelerated;

I. INTRODUCTION

Frequent Itemset Mining (FIM) algorithm as the basis of Frequent Pattern Mining, is generally utilized in large scale databases to find universal and potentially valuable patterns. In FIM algorithm, the data in databases are define as transactions, which are sets of times each with a unique ID. The purpose of FIM is to find subsets of transactions that "appears frequently". The criterion of frequency is defined by the "support ratio", which is the ratio of number of transactions containing a given itemset and the total number of transactions in the original database — only when the support ratio of a given itemset is not less than the predefined "minimum support threshold", can it be defined as "frequent". FIM algorithm returns all itemsets that satisfy the minimum support threshold.

Frequent Pattern Mining have a very wide utilization in fields of scientific research and commercials. A classic case is to analyze the sales records of a local super market. The result of the algorithm finds out some common patterns, for example, vegetables with fruits, bear, diaper with cigarettes, etc. And the analysis makes it possible for the arrangement of goods to follow a certain rule: putting the goods that are frequently purchased together accelerates the speed of customer flows; Or doing the opposite in smaller super markets to augment the chances that the customers purchase other goods when they are looking for what they first want. FIM algorithm are also applied to areas of database management system and information retrieval.

The research focus of this paper lays on the parallelization of PrePost [1] [2] algorithm, which is a state-of-the-art algorithm for mining frequent itemsets, in order to get a higher efficiency in larger and denser datasets.

II. RELATED WORK

Currently, the mainstream FIM algorithms are Apriori [3], Eclat [4], FP-Growth [5] and PrePost.

The processes of Apriori and Eclat are both generating the k+1-itemsets by k-itemsets iteratively, until all no more itemsets could be generated. Every step the algorithm combines two k-itemsets that have the same prefix to get the k+1 candidate itemsets with one more scan of the original database to filter out all the candidates that do not satisfy the minimum support threshold. Frequent databases scans are inevitable in this kind of algorithms.

FP-Growth, on the other hand, scans the original database only twice to get the preliminary frequent item head table and item prefix subtree. The head table and subtree together are defined as a FP-Tree. The head table comprises of the ID of an item, support ratio and a pointer to the corresponding node in the item prefix subtree, and is sorted in descending order of support ratio. The item prefix subtree is a prefix tree with a null-value root, that saves item ID and the support ratio of that item in the corresponding node. For a given node N in the FP-Tree, in the paths from the root node to node N, the partial paths that do not contain node N are called the prefix sub path of node N, while N is the postfix of the paths. All the prefix sub paths of node N in the FP-Tree are defined as the Conditional Pattern Bases of node N. The FP-Tree constructed from the Conditional Pattern Bases are called the Conditional Pattern Tree of N. The basic thought of FP-Growth algorithm is to start from length 1 frequent pattern, and construct its Conditional Pattern Bases, and thus its Conditional Pattern Tree. The process goes iteratively while the pattern grows by concatenating frequent patterns generated by the Conditional Pattern Tree. The advantage of FP-Growth is that it only scans the database twice, and there is no need for candidate generation, while the whole algorithm runs on the highly compressed data structure FP-Tree. Nonetheless, FP-Growth complicated the problem by constructing the FP-Tree when the minimum support threshold is relatively low or the dataset are especially sparse.

PrePost algorithm combines the upsides of the two aforementioned algorithms. PrePost algorithm first build a PPC-Tree that resembles FP-Tree, but with extra pre-order and post-order information stored in the node. The algorithm then extracts from the PPC-Tree to construct a data structure called N-List. The initial N-List contains 1-frequent itemsets as well as the pre-order, post-order and the corresponding support ratio for evey the PPC-Tree node. The process of mining is to combine the N-List nodes of two k-frequent itemsets, thus generating the k+1-frequent itemsets. The combination utilizes the pre-order and post-order number to determine the relationship of two nodes in

the original PPC-Tree, without maintain an actual tree data structure as in the FP-Growth algorithm. From the experiment results of paper [1], PrePost is the optimal algorithm in all circumstances, and the design of the algorithm satisfy the condition of parallelization. Therefore we decided to research on the parallelization of PrePost.

Apriori [6] [7], Eclat [8], FP-Growth [9] already have corresponding CPU parallelized implementation, while FP-Growth [10] and PrePost have implementation based on MapReduce. The basic thought of MapReduce PrePost is to first group the dataset with Round-Robin, and then generate the frequent itemsets as discussed above. Among all the algorithms aforementioned, only Apriori have GPU parallelized implementation [12] [13].

III. PREPOST OVERVIEW

Prepost is a frequent pattern mining algorithm published by Prof. Zh Deng in 2012. It uses a novel data structure "N-List" to represent the frequent patterns. In Prepost, all the frequent patterns are found by the intersection of several N-Lists. Given two n-length N-List, the intersection can be done in *O(n)* time complexity. The high-compressed data structure and the low time cost of merging make Prepost a very efficient algorithm for mining frequent patterns.

A. Data Structure

Prepost constructs a PPC-tree first to restore the data in the database. The initial N-Lists are created by traversing this PPC Tree.

1) PPC Tree

PPC-tree, the basis of N-list, is a tree structure similar to FP-tree in FP-Growth algorithm. It consists of one root labeled as "null", and a set of item prefix subtrees as the children of the root. Unlike FP-tree, each node in PPC-tree contains five fields: item-name, count, children-list, pre-order, post-order.

2) PP-Code

For each node N in a given PPC-tree, we call

<(N.pre,N.post):count>

the PP-code of N.

3) N-List

Given a PPC-tree, the N-list of a frequent item is a sequence of all the PP-codes of nodes registering the item in the PPC-tree. The PP-codes are arranged in an ascending order of their pre-order values.

4) Example

For a clear understand of the PPC-tree and N-List, see the example below. Figure 1 is an example database, including 5 transactions. To construct a PPC-tree, we first sort each transaction by the item support and then remove the infrequent item (support ≤ min support) from it as well. Then we insert each transaction into the tree. The completed PPC-tree is shown in Figure 2. After the PPC-tree is built, we traverse it to get the PP-Code of each node. Finally we collect the PP-Code of every item to get the 1st N-Lists (N-Lists of 1-frequent patterns).

ID	Items	Ordered frequent items
1	a, c, g, f	c, f, a
2	e, a, c, b	b, c, e, a
3	e, c, b, i	b, c, e
4	b, f, h	b, f
5	b, f, e, c, d	b, c, e, f

Figure 1: A example database

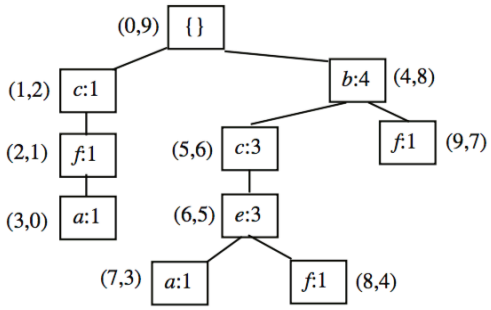

Figure 2: The PPC-tree built from the database in Figure 1

$b \longrightarrow \langle(4,8): 4\rangle$

$c \longrightarrow \langle(1,2): 1\rangle \text{ - - - } \langle(5,6): 3\rangle$

$e \longrightarrow \langle(6,5): 3\rangle$

$f \longrightarrow \langle(2,1): 1\rangle \text{ - - - } \langle(8,4): 1\rangle \text{ - - - } \langle(9,7): 1\rangle$

$a \longrightarrow \langle(3,0): 1\rangle \text{ - - - } \langle(7,3): 1\rangle$

Figure 3: The initial 1st N-Lists of Figure 1

B. Process

From the definition of N-List, we can see that N-List represents the structure of the tree, and thus represents the whole database. Therefore, the tree structure is no longer useful and all the mining process can be done only on the N-List.
Quite like Apriori, Prepost generates (n+1)th N-Lists from the nth N-Lists. We intersect the N-Lists of two patterns with the same postfix to get a new N-List of a longer pattern. For example, by intersecting N-Lists of pattern "ab" and "cb", we can get the N-List of pattern "acb". Then we check if the support of "acb" is less than minimum support. If yes, we do not proceed with it. Otherwise we keep the N-List and continue the mining task. Prepost repeats such process until no N-List of frequent patterns are available for further mining.
For details and correctness of this algorithm, you may refer to [1], the original paper of this algorithm.

The following is the pseudo-code of algorithm:

Function: $\boldsymbol{mining(L_k, NL_k)}$
Input: k-frequent pattern $\boldsymbol{L_k}$, and their N-list $\boldsymbol{NL_k}$

1: $\boldsymbol{for\ i \leftarrow L_k.size() - 1\ to\ 1\ do}$

```
    L^i_{k+1} ← ∅
    NL^i_{k+1} ← ∅
    for j = i − 1 to 0 do
        merge L_k[i], L_k[j] to get new pattern l
        l.Nlist = NL_intersection(NL_k[i], NL_k[j])
        if l.count > |D|×ξ then
            L^i_{k+1} ← L^i_{k+1} ∪ {l}
            F ← F ∪ {l}
            NL^i_{k+1} ← NL^i_{k+1} ∪ {l.Nlist}
        end if
    end for
    if L^i_{k+1} ≠ ∅ then
        if NL_k[i].length() = 1 then
        Assume that L^i_{k+1} = {P_1, ... P_n},  P_i = y_i x_1 x_2 .. x_k
        for any p = y_{v1} y_{v2} .. y_{vu} x_1 x_2 .. x_k do
            p.count ← NL_k[i].count
                F ← F ∪ {p}
            end for
        else
            mining(L^i_{k+1}, NL^i_{k+1})
        end if
    end if
end for
```

IV. Design and Implementation of GPU-Prepost

A. Introduction to CUDA

CUDA, which stands for Compute Unified Device Architecture, is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing – an approach known as GPGPU. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements.

B. Parallelism of Prepost

To get a better understanding of the process of Prepost, we draw a Set Enumeration Tree in Figure 4. In this tree, each node represents a frequent pattern, containing all the elements on the path from this node to root. Besides, a $(n+1)^{th}$ level node is generated by two n^{th} level nodes in Prepost. This tree clearly indicates the whole process of the mining process of Prepost.

From the structure of tree, we can find that the process of mining nodes under different branches is independent. In the mining process of a node *N*, only *N*'s siblings are useful, the other nodes will never interfere with the mining of *N* and *N*'s children. Note that the nodes under different branches have different postfix. We then come to a conclusion: The mining process of patterns with different postfix can be paralleled.

Based on this idea, we can easily design a paralleled algorithm for Prepost. We do the mining of nodes with different postfix at the same time.

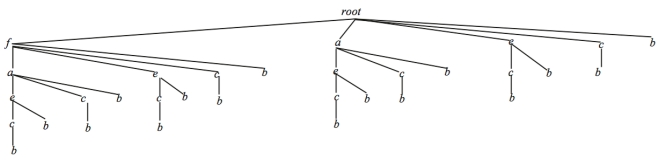

Figure 4: The Set Enumeration Tree

C. Implementation of GPU-Prepost

Considering the strong computing ability of Graphical Processing Unit (GPU), we decide to use CUDA, a GPU-based parallel computing architecture to implement our algorithm.

The process of GPU-Prepost algorithm can be divided into three following parts.

1) PPC-tree Construction

Since this part requires lots of IO operations, we do this on CPU, just like the original Prepost algorithm.

2) Paralleled Mining

In this part, we utilize the parallelism of GPU to our mining process. We use breadth first search (BFS) to iteratively do our mining task. $(n+1)^{th}$ frequent patterns are generated after **all** the n^{th} frequent patterns are generated.

To make our algorithm paralleled, we dispatch one thread for each node to do the mining task. Note that each thread in a block can access a shared memory for this block in CUDA. In order to let a node and its siblings share info with each other, we put the nodes with the same postfix into the same block in CUDA.

After the PPC-tree construction, we already have the initial 1^{st} N-Lists in CPU. We then pass them from CPU to GPU by copying them to GPU memory. The GPU then generates the 2^{nd} N-Lists by intersecting the 1^{st} N-Lists simultaneously. When all the 2^{nd} N-Lists are generated, we return them from GPU to CPU in order to save the results. We repeat the process until no further N-Lists are available: CPU prepares n^{th} N-Lists → GPU mines n^{th} N-Lists to generate $(n+1)^{th}$ N-Lists → CPU saves $(n+1)^{th}$ N-Lists. When the algorithm terminates, we get all the frequent patterns in CPU.

The pseudo-code of the kernel function in CUDA is as follows:

```
Function: generate_next_level(NL_k)
Input: k^th N-Lists NL_k
Output: (k+1)^th N-Lists NL_{k+1}

NL_{k+1} ← ∅
i = threadIdx.x + blockDim.x * blockIdx.x
for j = i + 1 to NL_k.size() do
    l_i = NL_k[i]
    l_j = NL_k[j]
    if l_i.label and l_j.label shares the same postfix then
        l = NL_intersection(l_i, l_j)
        if l.count ≥ min_support then
            NL_{k+1} ← NL_{k+1} ∪ {l}
        end if
    end if
end for
```

3) Details of GPU-Prepost

Due the the limitation of CUDA, we must pass a fixed array from CPU to GPU. However, both the number of frequent patterns and the length of N-Lists vary from level to level. A naïve idea is to set the size of the array to a large number. But it is definitely a waste to use a very large array to restore our N-Lists, since lots of redundant data structures are passed to GPU. So instead of using a big array for always, we count the upper

bound of the length of the N-Lists of the next level in the mining process.

Actually, we pass a 3-d array $NL_k[n][m][3]$ as our N-List structure to GPU. n represents the number of frequent patterns at this level; m represents the maximum length of N-List at this level; 3 represents the three fields of PP-Code: pre, post and count. We use the following two rules to determine the size of n and m:

- Avoid d Assume n_k is the number of frequent patterns in level k, then we have $n_{k+1} \leq \frac{1}{2} n_k(n_k - 1)$. This can be easily proved by showing that $\frac{1}{2} n_k(n_k - 1)$ is the maximum number of the pairs of n_{k+1} N-List.
- Assume n_k is the number of frequent patterns in level k, then we have $n_{k+1} \leq \frac{1}{2} n_k(n_k - 1)$. This can be easily proved by showing that $\frac{1}{2} n_k(n_k - 1)$ is the maximum number of the pairs of n_{k+1} N-List.

V. RESULT

A. Experiment Setup

Table 1 contains the experiment environment, in which All codes are compiled and experimented. Table 2 is the datasets chosen for the experiments (source: http://fimi.ua.ac.be/data/).

TABLE I. EXPERIMENT ENVIRONMENT

Operating System	Ubuntu 12.04
Memory	64 GB
CPU	Xeon E5-2620
GPU	NVIDIA Tesla K20

TABLE II. EXPERIMENT DATASETS

Dataset	Parameters		
	Avg.length	*#Items*	*#Trans*
mushroom	23	119	8,124
T10I4D100K	10	949	98,487
T40I10D100K	40	942	92,113

Table 3 contains the minimum support threshold chosen according to the datasets and number of corresponding frequent items.

TABLE III. EXPERIMENT DATASETS

Dataset	Parameters		
	Avg.length	*#Items*	*#Trans*
mushroom	23	119	8,124
T10I4D100K	10	949	98,487
T40I10D100K	40	942	92,113

a.

B. Experiment Results

The experiments are conducted by the comparison of Apriori [3], GPApriori [13] [14], PrePost [1] algorithms and GPU-PrePost algorithm proposed in this paper from two aspects.

1) Runing Time

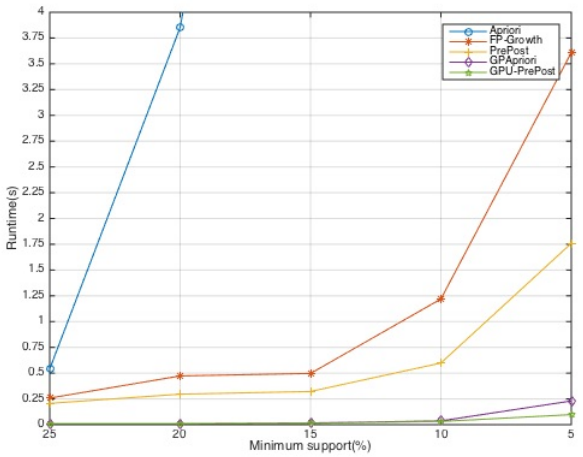

Figure 5: Running time on Mushroom

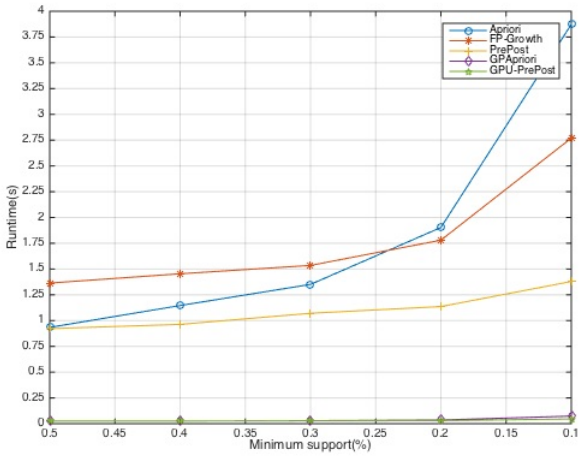

Figure 6: Running time on T10I4D100K

Figure 5-7 are the running time comparison of the aforesaid algorithms. X-axis stands for the minimum support threshold, Y-axis stands for the running time, excluding time for input and output.

Figure 5 is the running time comparison on Mushroom dataset. It is clear that the two parallelized algorithm out performs the others. And GPU-PrePost wins in larger datasets.

Figure 6 is the running time comparison on T10I4D100K dataset, which is a dataset of great sparsity, even when the minimum support threshold is very low, there are still only a few frequent itemsets. It is clear that the two parallelized algorithm out performs the others, but there is no big difference between parallelized ones.

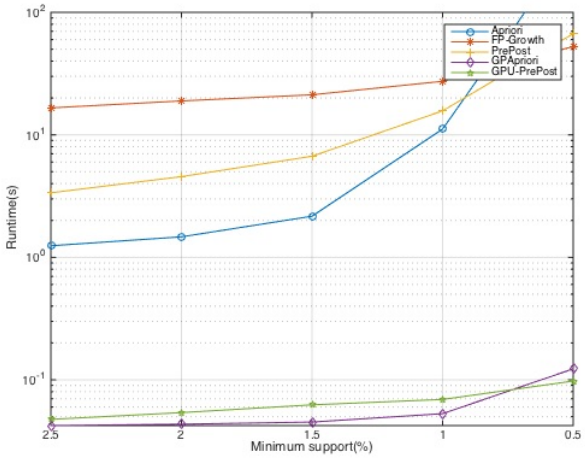

Figure 7: Running time on T40I4D100K

Figure 7 is the running time comparison on T40I4D100K dataset, which is also a dataset of great sparsity, but when the minimum support threshold is very low, the density grows. still only a few frequent itemsets. It is clear that the two parallelized algorithm out performs the others, but there is no big difference between parallelized ones. Upon observation, GPU-PrePost takes more time than GPApriori, this is because the initialization of N-List gives more overhead in GPU-PrePost, while the simply structured GPApriori only need to finish the easy mining task.

2) Speedup Ratio

From the speedup ratio experiments, GPU-PrePost algorithms shows 10-100 times acceleration compared to traditional algorithms, and, though performs nearly the same when datasets are sparse, still out performs the GPApriori algorithm by 2.5 times when the datasets are dense.

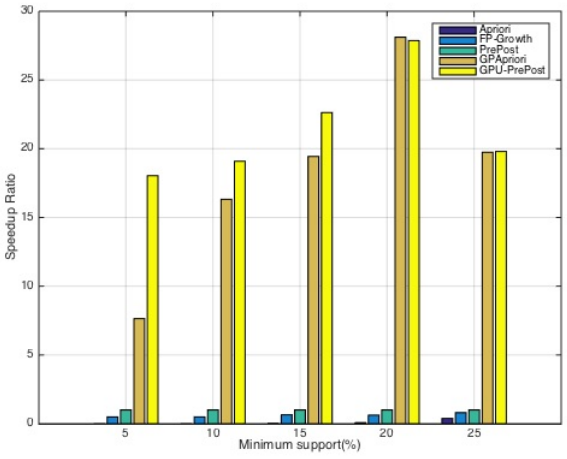

Figure 8: Speedup Ratio on Mushroom. PrePost as baseline.

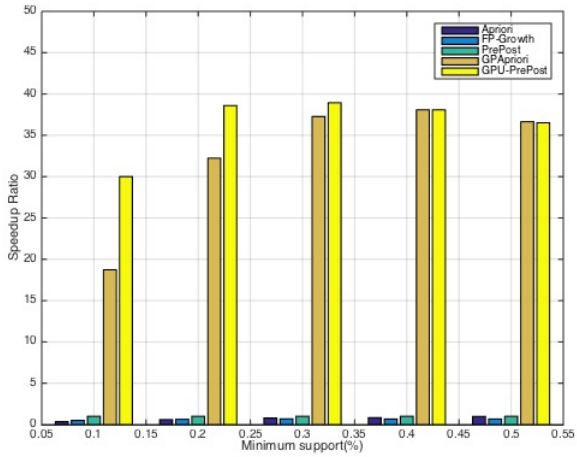

Figure 9: Speedup Ratio on T10I4D100K. PrePost as baseline.

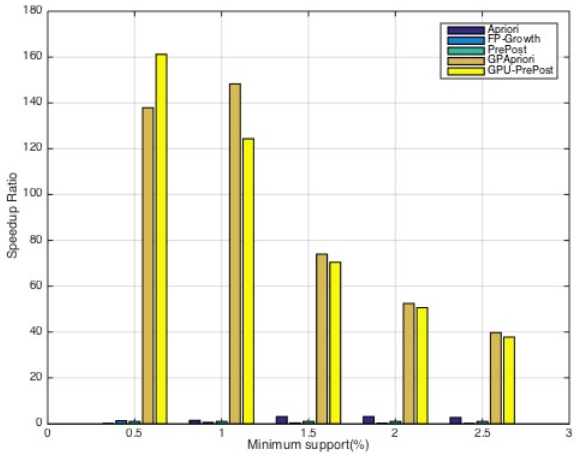

Figure 10: Speedup Ratio on T40I4D100K. PrePost as baseline.

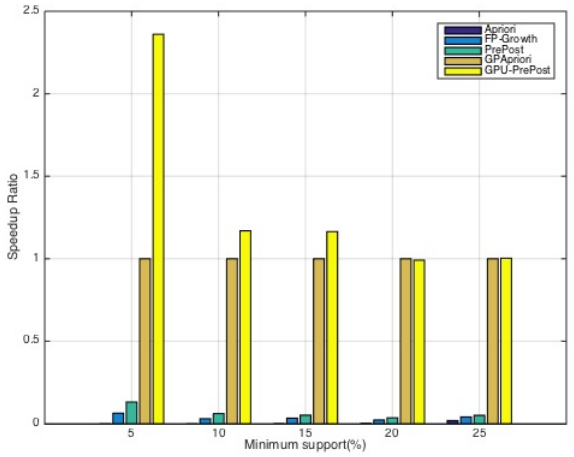

Figure 11: Speedup Ratio on Mushroom. GPApriori as baseline.

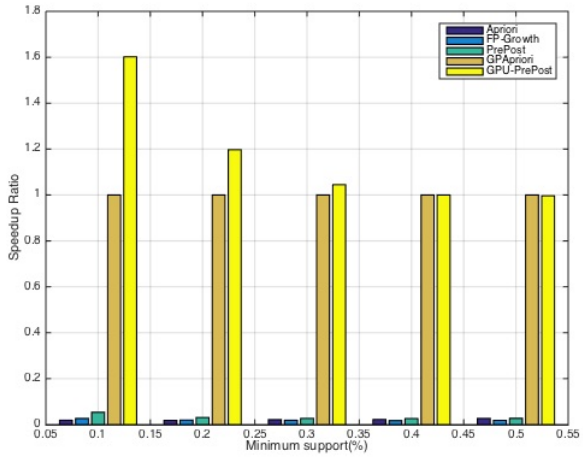

Figure 12: Speedup Ratio on T10I4D100K. GPApriori as baseline.

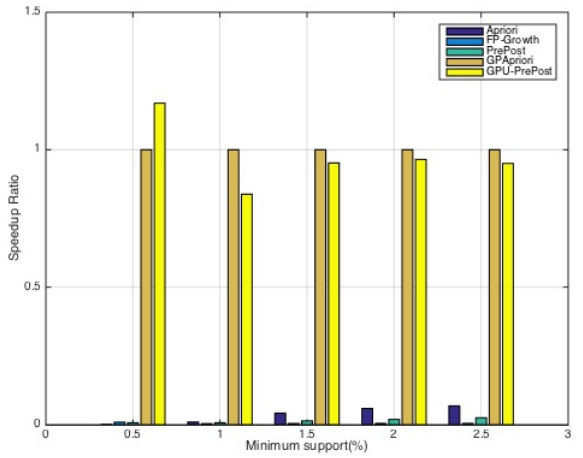

Figure 13: Speedup Ratio on T10I4D100K. GPApriori as baseline.

VI. CONCLUSION

This work researches and analyzes the state-of-art Frequent Pattern Mining algorithm PrePost, and discusses the possibility for parallelization. Considering the independence of N-List nodes with difference postfix, we proved that it is possible to parallelize the process of PrePost and thus augment its efficiency.

According to the basic idea mentioned above, we proposed GPU-PrePost algorithm based on CUDA, and implemented this algorithm. The experiments have proved the GPU-PrePost algorithm out performs other algorithms in most circumstances. Meanwhile, this result shows that GPU model are compatible for generic data mining tasks. By utilizing the high calculation capability of GPU, it is possible to accelerate traditional data mining algorithms significantly. This would be of great value for future data mining algorithm researches.

ACKNOWLEDGMENT

This work is support by the Peking University Principal Undergraduate Research Foundation (URTP2014PKU004). (校长基金)

REFERENCES

[1] Deng, Z., Wang, Z., & Jiang, J. (2012). A new algorithm for fast mining frequent itemsets using N-lists. *Science China Information Sciences*, *55*(9), 2008-2030.

[2] Deng, Z. H., & Lv, S. L. (2014). Fast mining frequent itemsets using Nodesets. *Expert Systems with Applications*, *41*(10), 4505-4512.

[3] Agrawal, R., & Srikant, R. (1994, September). Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB* (Vol. 1215, pp. 487-499).

[4] Zaki, M. J., Parthasarathy, S., Ogihara, M., & Li, W. (1997, August). New Algorithms for Fast Discovery of Association Rules. In *KDD* (Vol. 97, pp. 283-286).

[5] Han, J., Pei, J., & Yin, Y. (2000, May). Mining frequent patterns without candidate generation. In *ACM SIGMOD Record* (Vol. 29, No. 2, pp. 1-12). ACM.

[6] Agrawal, R., & Shafer, J. C. (1996). Parallel mining of association rules. IEEE Transactions on Knowledge & Data Engineering, (6), 962-969.

[7] Zaki, M. J., Ogihara, M., Parthasarathy, S., & Li, W. (1996). Parallel data mining for association rules on shared-memory multi-processors. In Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on (pp. 43-43). IEEE.

[8] Zaki, M. J., Parthasarathy, S., & Li, W. (1997, June). A localized algorithm for parallel association mining. In Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures (pp. 321-330). ACM.

[9] Zaïane, O. R., El-Hajj, M., & Lu, P. (2001). Fast parallel association rule mining without candidacy generation. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on* (pp. 665-668). IEEE.

[10] Zhou, L., Zhong, Z., Chang, J., Li, J., Huang, J. Z., & Feng, S. (2010, November). Balanced parallel fp-growth with mapreduce. In *Information Computing and Telecommunications (YC-ICT), 2010 IEEE Youth Conference on* (pp. 243-246). IEEE.

[11] Liao, J., Zhao, Y., & Long, S. (2014, May). MRPrePost—A parallel algorithm adapted for mining big data. In *Electronics, Computer and Applications, 2014 IEEE Workshop on* (pp. 564-568). IEEE.

[12] Zhang, F., Zhang, Y., & Bakos, J. (2011, September). Gpapriori: Gpu-accelerated frequent itemset mining. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on* (pp. 590-594). IEEE.

[13] Zhang, F., Zhang, Y., & Bakos, J. D. (2013). Accelerating frequent itemset mining on graphics processing units. *The Journal of Supercomputing*, *66*(1), 94-117.