2018 Tenth International Conference on Advanced Computational Intelligence (ICACI)
March 29–31, 2018, Xiamen, China

D2P-Apriori: A deep parallel frequent itemset mining algorithm with dynamic queue

Yuxin Wang, Tongkun Xu, Shiqing Xue, and Yanming Shen
School of Computer Science and Technology
Dalian University Of Technology
Dalian, China
Email: wyx@dlut.edu.cn; tongkunxu@mail.dlut.edu.cn; xueshiqing@mail.dlut.edu.cn; shen@dlut.edu.cn

***Abstract*—As the core methodology of implementing association rules mining, frequent itemsets mining is used to extract frequent itemsets from items in a large database of transactions. In this paper, we proposed Dynamic queue & Deep Parallel Apriori (D2P-Apriori), a parallel frequent itemset mining algorithm on GPU to satisfy the high-performance requirement. The contributions of D2P-Apriori include as follows. The dynamic queue with bitmap is improved upon the vertical data structure to deal with the problem that the required memory for sparse data may exceed the size of GPU global memory. The Graph-join way is devised to adapt GPU architecture for candidate generation method. And the improved data structure also contributes to the significantly accelerated performance in support counting method on GPU. The implementation of this algorithm achieves the deep and comprehensive parallelization. Our parallel implementation on GeForce GTX 1080 graphic processor outperforms several state-of-the-art frequent itemset mining algorithms on CPU, up to 63x speedup ratio can be obtained on large dataset.**

Keywords*— *Frequent itemset mining; Association rule mining; GPU Computing; Apriori; CUDA; Parallel Algorithms

I. Introduction

Data mining is the process of turning raw data into useful knowledge from the massive and irregular data [1]. One of the most popular data mining algorithm is association rules mining (ARM) which is widely used in commercial recommendation, medical diagnostic, financial investment and many more. The frequent itemsets generated by Apriori can be used to generate all association rules. The original algorithm Apriori for mining frequent itemset, which was published in 1994 by Agrawal, is still frequently used [2]. Frequent itemset mining (FIM) finds potentially interesting patterns which called frequent itemset in large transactions dataset. The measure of frequent itemset is minimal support which represents the frequency threshold of this itemset occurrence.

The original Apriori suffers from scanning the database repeatedly and a large number of candidate items. With the growing of data size, the algorithm performance becomes insufficient. Most of literature was ommitted to optimizing the algorithm in a serial execution manner. Luckily, the high-performance computing device GPU can be easily accessed and using GPU Computing to speed up this algorithm is a feasible solution. In this paper we propose a new parallel D2P-Apriori that is optimized to meet the limitation of GPU global memory and suit the Compute Unified Device Architecture (CUDA) programming model. D2P-Apriori is an abbreviation of *Dynamic queue & Deep Parallel Apriori*. The main contributions in our implementation include a dynamic bitmap queue data structure to avoid starting CUDA kernel for redundant times and the Graph-join way is designed in parallel to generate candidates to realize a better performance than state-of-the-art serial implementations. Our results have shown that our optimization is feasible and promising for frequent itemset mining applications.

II. Background and related work

Three most popular algorithms of frequent itemset mining are Apriori, Eclat [3], and FP-growth [4]. Apriori works on horizontal layout based database and uses breadth first search order to generate candidate subsets. Apriori generates higher order itemset of k+1 items based on k itemsets and this step is called candidate generation. The process to get the occurrence number of each candidate set is called support counting. During the support counting step, the computation can be parallelized due to the independence of candidate itemset. Apriori has a high potential for parallelization. Eclat is a vertical database layout algorithm in which the data is represented in a bit matrix format. It uses depth first search and prefix tree which is memory efficient with backtracking. FP-growth is a tree based algorithm and no candidate frequent itemset is needed. FP-growth takes the least memory because common prefixes can be shared to create a fp-tree. For serial implementations and optimizations, Bart Goethals implemented Apriori based on Agrawals algorithm [2] and Ferec Bodon investigated a trie-based Apriori algorithm with omitting the prefix-equisupport extensions [5].

Although the performance comparison analyses [6] show that FP-growth outperforms Apriori and Eclat in serial algorithms, the data structure and task order make it less suitable for parallelization than Apriori. Most of related literature parallelizes FIM algorithm based on Apriori or ECLAT. These two algorithms both contain two independence tasks because of using generate-and-count methodology [7].

Since most serial algorithms cannot take advantage of the high-performance computing platform, R. Agrawal et al. [8] proposed the first parallel version. S. Rustogi et al. [9] demonstrated an improved parallel Apriori that applies

978-1-5386-4362-4/18/$31.00 © 2018 IEEE

data parallelism in multi-core environment. Wenbin Fang [10] group of authors developed GPUMiner-Apriori that adapts the trie-based bitmap Apriori implementation on the GPU and Syed et al. [11] introduced a pure-based bitmap Apriori implementation. The two different approaches have the same way of support counting step on GPU, the performance differences are mainly in candidate generation. But both of their work does not implement candidate generation on GPU which is also an important part to improve the performance. An Apriori-Like algorithm called GPU-FPM is presented in [12] which realized 15x speedup ratio because of the limitation of GPU global memory size.

III. Apriori Algorithm Description

Based on the property of FIM, The ultimate goal of Arpiori is to find all the itemset whose support ratio is bigger than min-support threshold. The algorithm is described as follow:

$I = \{i_1, i_2, \ldots, i_m\}$ represents an itemset and $D = \{T_1, T_2, \ldots, T_n\}$ is a dataset, that contains n transactions. K-itemset is an itemset that contains k items. If $X \subseteq T$, T includes X subset of I. Association rule mining is an implication form $X \Rightarrow Y$, with $X \subseteq I, Y \subseteq I$ and $X \cap Y = \emptyset$. Apriori uses the anti-monotonicity of itemset which is also called Downward Closure Property, if an itemset is infrequent, all the superset of this itemset is also infrequent, for instance, if itemset {A,B} is infrequent, the superset {A,B,C} is inevitable infrequent. This algorithm of finding the frequent itemset can be described step by step as follows:

1. Scan all the transaction database and get the frequency of each unique item C_1 and get L_1 by pruning the infrequent item by min-support threshold.

2. In the next k-1 turn, the frequent itemset L_{k-1} will generate candidate C_k.

3. Get the frequency of each candidate using support count function and prune to get L_k. L_k will work as a seed to generate candidate until no itemset can meet the min-support.

Two main steps are candidate generation and support counting which need numerous computing resources and are the main performance bottleneck. Depending on downward closure property the candidate generation step will generate redundant itemsets. To optimize this process and avoid the $O(n^2)$ complete join the concept of Equivalent Class [13] is needed, which is defined by the itemset cluster which has k size of itemset and has the same k-1 prefix.

IV. Proposed Algorithm and Implementation

This section describes D2P-Apriori algorithm, which includes two innovations in our implementation. The first innovation is the finely parallelizing candidate generation for GPU using the Graph-join way. The second innovation is the design of a dynamic bitmap queue data structure to avoid starting the redundant times of CUDA kernel in support counting for a large dataset.

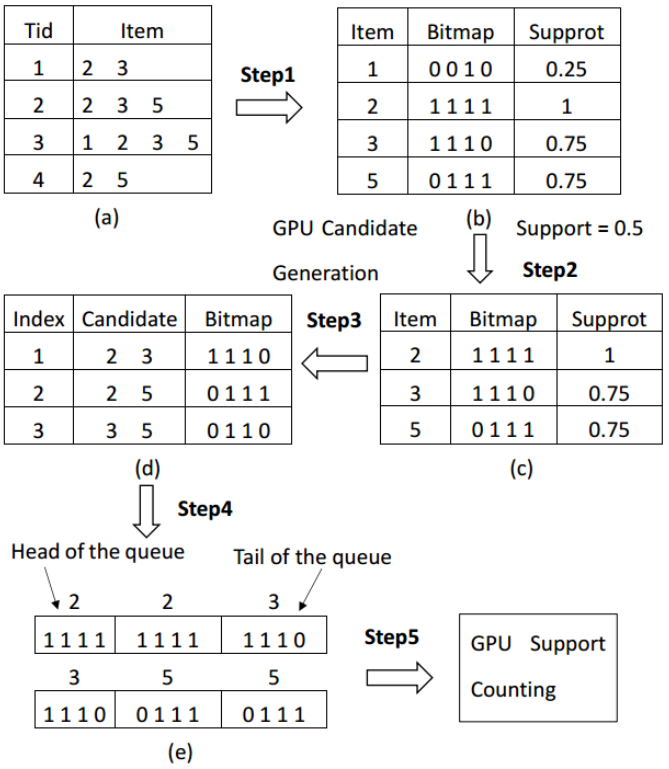

Fig. 1. The steps in data preprocessing

A. Data Structure Improvement

Concerning the SIMD system and CUDA programming model, the data representation needs to be suitable for parallel implementations especially in support counting function. The traditional way is to represent the transactions with Horizontal structure, and each line from the database is a transaction and the line index is Tid. We call this horizontal representation shown in Figure 1(a). The major performance impacts of Apriori have yet to be addressed. Support counting needs to multiple-scan the database. Using the horizontal representation cannot avoid this and in the step of getting the C_k Frequency the comparison with transactions is also discontinuous.

When using the vertical representation shown in Figure 1(b), each item has an Item Id, and each Item Id reflects to the Tid set that means this item contains in these transactions. In this way the candidate generation can use the Item index directly and we used a bit reflecting to the item location in support counting step which can make full use of GPU parallel computing ability and reduce logic operation. However, for sparse data the algorithm using vertical bitmap data structure may exceed the global memory on the GPU.

We proposed the bitmap queue in Figure 1(e) which is organized by Item Id of candidates itemsets. Bitmap representation contains the same information that is optimized on vertical representation. The initial contents of the queue are empty. During the candidate generation step, the bitmap of generated candidates is pushed into the tail of the queue until length of

this queue is bigger than M. We chose the value of M equal to the maximum length of bitmap that can be accommodated on GPU. The full queue is transferred to GPU to count the support of each candidate in this queue. The insert operations are repeated until the candidate array is empty.

At first it sorts all the unique items from low to high, then it transfers the transactions to vertical like format. If this item is included in this transaction the corresponding position will be set as 1, if not, it will be set as 0. In essence, a binary representation of each Item is corresponding to the Tid set. The data transfer step is the preprocessing step before the algorithm starts. Figure 1 demonstrate the steps of the processing that can be outlined as follows:

1. Transfer the dataset which is format as horizontal data structure to vertical format which can use $\langle item, bitmap \rangle$ for representation.

2. The infrequent itemset will not participate in the next turn calculation. Remove the infrequent itemset according to the property of Apriori. After step 2, the infrequency item will already be removed, then the dense matrix can contribute to a better speedup performance.

3. Use the Graph-join way on GPU to generate candidates.

4. The bitmap corresponding to the candidates is pushed into the queue in lexicographic order.

5. Transfer the full bitmap queue into GPU to avoid starting GPU Kernel for the redundant times.

B. GPU Accelerated Candidate Generation

The Equivalent Class is contributed to candidate generation process. Although using tree structure can achieve data compactness and fast data traversal, the irregular data structure and Depth-first search order are inappropriate to distribute data into SIMD threads for parallelization.

We proposed a Graph-join way which uses the adjacency matrix to realize the candidate generation. The L_{k-1} itemset is the vertex set V, the join matrix is a square $|V| \times |V|$ matrix S such that its element S_{IJ} is one when there is a combination between itemset I and itemset J, and zero when there is no combination. The diagonal and lower triangular elements of the matrix are all zeros, since the connection from an itemset to itself and the combination between itemsets are undirected. Before the join step, it is required to preprocess L_{k-1} itemset to adapt to the generation. Itemset I needs to connect to one number with the help of P value.

P is defined as the magnitude of the unique items number in the dataset. For example the max unique items number in this dataset is 2048, the P is equal to 4 that represents to its magnitude. The item in the itemset $I = \{i_1, i_2, \ldots, i_{k-1}\}$ are connected to one number N with (1)

$$N = i_{k-1} + \sum_{x=k-1}^{2} i_x 10^{p+x-1} (k \geq 2) \qquad (1)$$

After this $L_{k-1} = \{N_1, N_2, \ldots, N_m\}$ m is the number of rank k-1 candidate itemsets. Considering the massive data of candidate itemsets, L_{k-1} are passed to GPU and

Algorithm: Graph-join
Input: L_{k-1} itemsets
Output: matrix S

```
1.  initialize matrix S
2.  i = blockDim.x × blockId.x + threadId.x
3.  if Lk-1 number n < blockDim.x;
4.      put Lk-1 into __shared__ side1[MAX_SIDE_LEN]
5.      for each Lk-1 item Ni in side1
6.          j = i +1
7.          if side1[Ni] / P = side1[Nj] / P   //have the same prefix
8.              S[i *n + j ]=0    //generated candidate itemset
9.  if Lk-1 number n > blockDim.x;
10.     put Lk-1 into side1[MAX_SIDE_LEN] and side2[MAX_SIDE_LEN] in shared memory
11.     for each Lk-1 item Ni in side1
12.         for each Lk-1 item Nj in side2
13.             if side1[Ni] / P = side2[Nj] / P and i<j   //have the same prefix
14.                 S[i *n + j ]=0    //generated candidate itemset
```

Fig. 2. Pseudo code of the Graph-join algorithm with no branch divergence

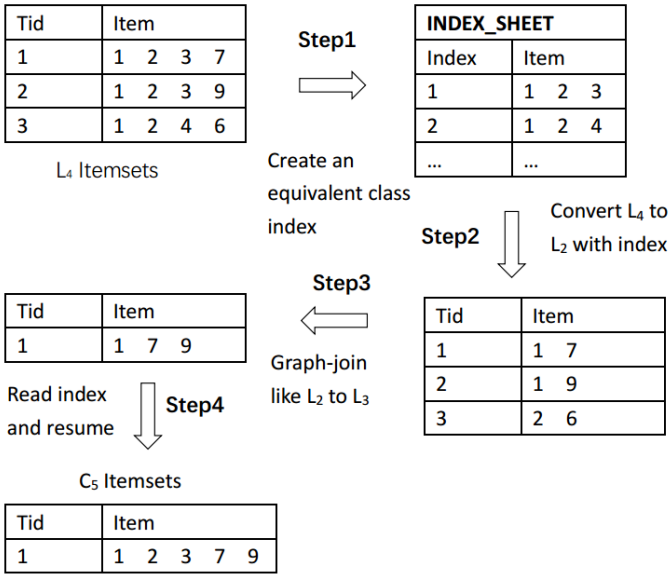

Fig. 3. Pretreatment progress in the Graph-join iterative implementation with an index of equivalent class

using shared memory in batches to reduce global memory access. In this kernel function, the algorithm puts all the L_{k-1} itemsets into shared memory. To avoid starting unless blocks, if m is less than block_dim only start 1st block and put L_{k-1} into __shared__ side1[MAX_SIDE_LEN]. If m is bigger than blockDim.x starting 2nd block is required. Each two itemsets in side1[MAX_SIDE_LEN] or both side1 [MAX_SIDE_LEN] and side2[MAX_SIDE_LEN] are allotted to one thread. Pseudo code of the Graph-join algorithm is shown in Figure 2. Because the itemset is dictionary sorting, the equivalent class can be continuously processed which makes the join step more efficient. B y r emoving *e lse* path to avoid branch divergence and make the implementation for one cycle.

The upper part of Figure 4 shows the Graph-join process, A,B,C,D k-1-itemsets in dark green are equivalent class, E,F,G

itemsets in green are equivalent class and H,I itemsets in light green are equivalent class. During the graph-join, all the itemsets in $L_{k-1} = \{A, B, C, D, E, F, G, H, I\}$ are vertex set, and it compose the Graph-join matrix. The thread blocks inside the red line are the sub-matrix for each equivalent class and the yellow blocks represent the k-candidate itemsets.

As the size of N increases, a 32-bit integer sometimes cannot contain it. We use an index sheet to replace an equivalent class before we do Graph-join. From the perspective of CUDA kernel, the itemset are now equivalent to an L_2 itemset. The introduction of equivalent class index helps reduce the memory usage greatly. Figure 3 shows the L_4 itemsets pretreatment progress. An index sheet is built in every iteration in step1. With the sheet we convert the L_4 to L_2 itemsets in step2. The third step realizes the reuse of the Graph-join method which is used to find C_3. After that, the C_5 itemsets is resumed in step4.

C. GPU Accelerated Support counting

The most arduous computation in this algorithm is to calculate the support count that needs to scan the database frequently and get the frequency of each item. To take full advantage of GPU parallel mathematical computing ability, we realized this step using CUDA.

To adapt to the GPU computation modal, we chose to use the vertical bitmap structure that performs better in the intersection step. The vertical bitmap structure can make sure the intersection step process is continuous without uncoalesced items.

After candidate generation step, the candidate_list is created based on candidate itemset that can be passed to kernel function to realize the support counting. The candidate_list which consists of the candidate itemset bitmap list is an $(bitmap\ length \times number\ of\ candidate\ itemsets)$ array. As shown in Figure 5 the kernel has two candidate_lists (Vertical list 1, Vertical list 2) in blue and one result_list in green which represents the source bitmap of candidate itemset and the result list after list intersection. Each bitmap of candidate itemset will be calculated by one block and each thread is responsible for two bitmaps intersection of 32-bit shown in the lower part of Figure 4. Intersection step uses bitwise operators (&)which does not require short circuit evaluation and has a better performance than logical operators (&&).

The intersection result of each thread is a 32-bit integer, the total sum of 1 bit in one block is the frequency of one itemset. NVDIA provides the function __popc( ) [14] to count the number of bits in a 32-bit integer.

The result counted by the __popc( ) is stored in support[blockDim.x]. In order to reduce the access latency for parallel reductions, the support array is stored in shared memory of per thread block. The thread in the same block can access the support array. Using __syncthreads(); The parallel reduction optimization [15] strategy is used to sum support[threadIdx.x] in one block, adding all the values recursively into support[0]. At the end of the reduction, thread 0 holds the sum of the support values in one block. This

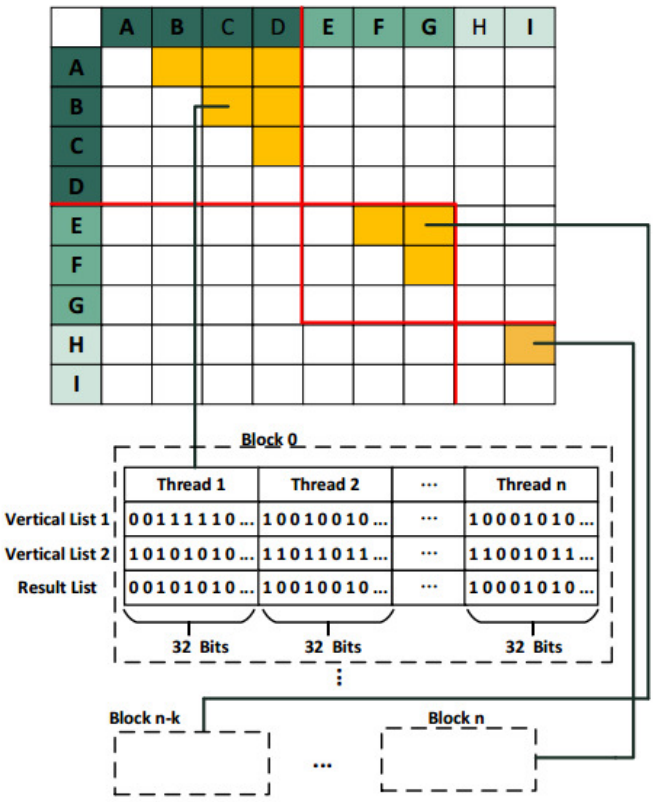

Fig. 4. Demonstration of the Candidate generation using Equivalent Class and support counting process

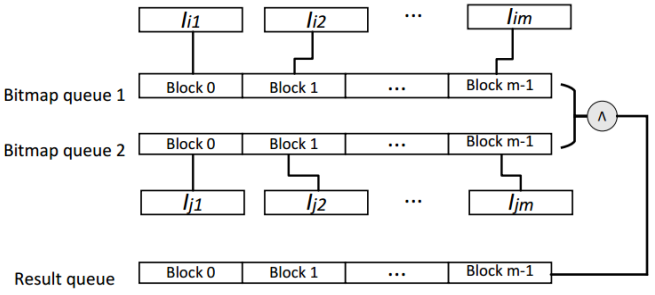

Fig. 5. Illustration of GPU support counting

optimization strategy keeps all multiprocessors on the GPU busy and each thread block reduces a portion of the array. We replace interleaved addressing with sequential addressing to avoid shared memory bank conflicts and use reversed loop.

V. Experimental Results

To evaluate the performance of our Graph-join based Apriori algorithm, our experiments were performed using the synthetic datasets and the real-world datasets provided by FIMI repository. For the experiments we used Intel(R) Core(TM) i7-6800K CPU @3.40GHz and GeForce GTX 1080 and 8G DDR5X memory, running on the ubuntu 16.04.4. The tradition CPU program is compiled by gcc 5.4.0 and NVIDIA CUDA driver 8.0 is installed for CUDA programs. T10I4D100K,

TABLE I.
EXECUTION TIME (IN SEC) WITH VARIATION ON SUPPORT THRESHOLD

Min-Support	Goethals	Bodon	CPU-Test	D2P-Apriori
1.25%	17.898	4.25257	23.290426	1.34694
1.5%	12.8151	2.8877	15.529775	1.16524
1.75%	9.53665	1.92446	12.762612	1.00094
2%	8.56893	1.64511	10.687833	0.953761

T20I4D100K, T30I4D100K and T40I10D100K are generated by IBM Market-Basket Synthetic Data Generator [16]. Mushroom dataset is prepared by Roberto Bayardo from the UCI datasets and PUMSB [17]. Accidents dataset is provided by Karolien Geurts and contains anonymized traffic accident data [17].

We compared the performance of D2P-Apriori with Bodon Apriori, Goethals Apriori and the CUP tradition test version of Apriori with bitmap based implementation in Figure 6 Figure 7 Figure 8 and Table 1. Bodon Apriori and Goethals Apriori are state-of-art implementations of CPU based Apriori. We did not find the Apriori implementation code based on GPU in the public domain. To present the small changes of data better, Table 1 shows the execution time with variation on support threshold on T40I10D100K. We conducted each experiment ten times and plotted the chart with arithmetic mean.

From the listed execution time it can be observed that speedup ratio scales with the size of dataset. For the same dataset processing D2P-Apriori requires less time than Bodon, Goethals and CPU_Test_Apriori. On the Mushroom dataset with less transactions and items, D2P-Apriori can achieve 5x speedup compared with Goethals. Since large computation workload can hide the global memory access latency and reduce thread idle time. On the larger Accidents dataset D2P-Apriori can achieve 63x speedup compared with Goethals, 19x speedup to Bodon.

Figure 9 demonstrates the speedup ratio performance respecting transaction average length variation. The synthetic dataset with average length varies from 10 to 40 were used in these experiments. The transaction number is 10000 and minimum support is 1%. The comparison demonstrated in Figure 9 shows that D2P-Apriori performs better on the dataset with longer transactions. When the transaction is long, the computation workload is high. We can investigate that the stream processors of GPU can be fully utilized with workload increase.

VI. CONCLUSION AND FUTURE WORK

This paper proposed a GPU based Apriori algorithm called D2P-Apriori. The main idea of D2P-Apriori is to accelerate Aprori by paralleling the candidate generation using the Graph-join way and dynamic bitmap queue based on vertical bitmap structure with low-latency memory on the GPU. The above exploration shows D2P-Apriori can obtain high speedup ratio compared to the state-of-the-art CPU-only serial especially on large datasets. Future work will focus on the following two points: extend this implementation to multi-

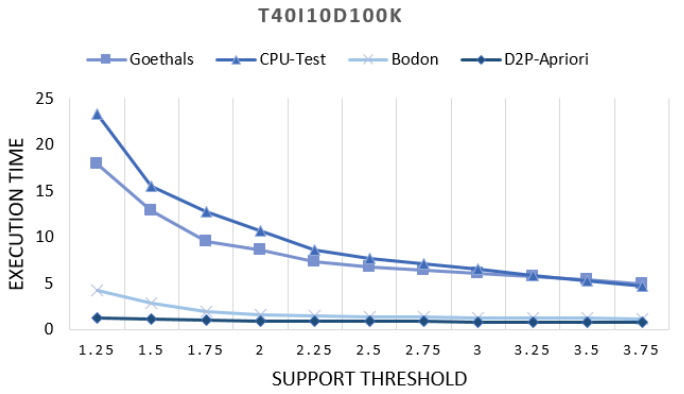

Fig. 6. Performance comparison between Goethals, CPU-Test, Bodon and D2P-Apriori on dataset T40I10D100K with variation on support threshold. The characteristics of this dataset are: Average length of transaction = 40, unique item number = 1000, transaction number = 100000

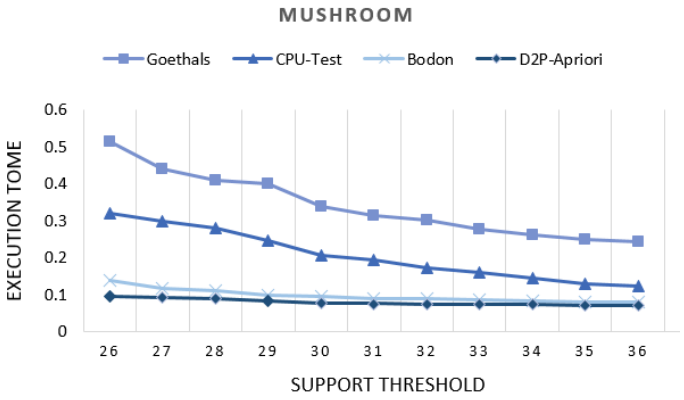

Fig. 7. Performance comparison between Goethals, CPU-Test, Bodon and D2P-Apriori on dataset Mushroom with variation on support threshold. The characteristics of this dataset are: Average length of transaction = 23, unique item number = 120, transaction number = 8124

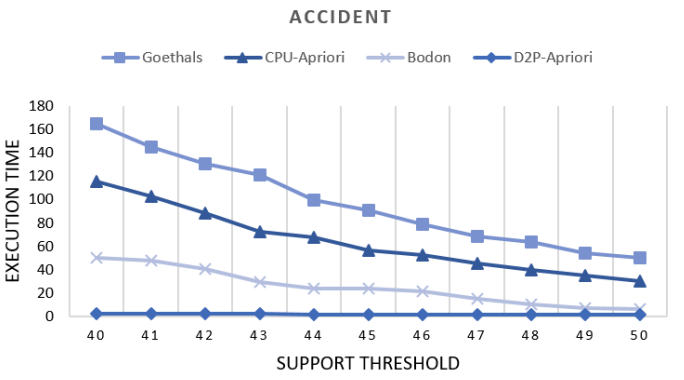

Fig. 8. Performance comparison between Goethals, CPU-Test, Bodon and D2P-Apriori on dataset Accidents with variation on support threshold. The characteristics of this dataset are: Average length of transaction = 34, unique item number = 468, transaction number = 340183

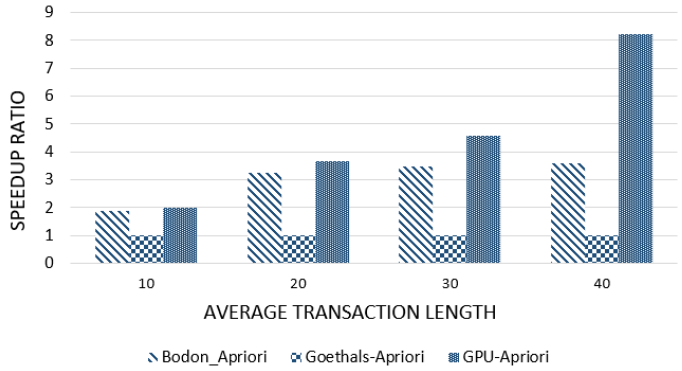

Fig. 9. Performance comparison between Goethals, CPU-Test, Bodon and D2P-Apriori on synthetic datasets T10I4D100K, T20I4D100K, T30I4D100K and T40I10D100K with variation on transaction length. The characteristics of these datasets are: Unique item number = 1000, transaction number = 100000, minimum support threshold = 1%

GPUs cluster and devise a parallel computing model to process other Apriori-like algorithms.

ACKNOWLEDGMENT

This work was partially supported by National Natural Science Foundation of China Grants (NSFC under grant No. 11372067 and 61173160).

REFERENCES

[1] S. Chakrabarti, M. Ester, U. Fayyad, J. Gehrke, J. Han, S. Morishita, G. Piatetsky-Shapiro, and W. Wang, Data mining curriculum: A proposal (Version 1.0), 2006.

[2] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in Proc. 20th int. conf. very large data bases, VLDB, vol. 1215, 1994, pp. 487–499.

[3] M. J. Zaki and K. Gouda, "Fast vertical mining using diffsets," in Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2003, pp. 326–335.

[4] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach," Data mining and knowledge discovery, vol. 8, no. 1, pp. 53–87, 2004.

[5] F. Bodon, "A trie-based apriori implementation for mining frequent item sequences," in Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations. ACM, 2005, pp. 56–65.

[6] K. Garg and D. Kumar, "Comparing the performance of frequent pattern mining algorithms," International Journal of Computer Applications, vol. 69, no. 25, 2013.

[7] R. L. Uy and M. T. C. Suarez, "Survey on the current status of serial and parallel algorithms of frequent itemset mining," Manila Journal of Science, 2016.

[8] R. Agrawal and J. C. Shafer, "Parallel mining of association rules," IEEE Transactions on knowledge and Data Engineering, vol. 8, no. 6, pp. 962–969, 1996.

[9] S. Rustogi, M. Sharma, and S. Morwal, "Improved parallel apriori algorithm for multi-cores," International Journal of Information Technology and Computer Science (IJITCS), vol. 9, no. 4, p. 18, 2017.

[10] W. Fang, K. K. Lau, M. Lu, X. Xiao, C. K. Lam, P. Y. Yang, B. He, Q. Luo, P. V. Sander, and K. Yang, "Parallel data mining on graphics processors," Hong Kong Univ. Sci. and Technology, Hong Kong, China, Tech. Rep. HKUST-CS08-07, 2008.

[11] S. H. Adil and S. Qamar, "Implementation of association rule mining using cuda," in Emerging Technologies, 2009. ICET 2009. International Conference on. IEEE, 2009, pp. 332–336.

[12] J. Zhou, K.-M. Yu, and B.-C. Wu, "Parallel frequent patterns mining algorithm on gpu," pp. 435–440, 2010.

[13] M. J. Zaki, S. Parthasarathy, M. Ogihara, W. Li *et al.*, "New algorithms for fast discovery of association rules." in KDD, vol. 97, 1997, pp. 283–286.

[14] "CUDA toolkit documentation," http://docs.nvidia.com/cuda/cuda-math-api/ , accessed November 27, 2017.

[15] H. Mark, "Optimizing parallel reduction in cuda," NVIDIA CUDA SDK, vol. 2, 2008.

[16] "IBM guest synthetic data generator," http://www.philippe-fournier-viger.com/spmf/datasets/ , accessed November 27, 2017.

[17] "Frequent itemset mining dataset repository," http://fimi.ua.ac.be/data/ , accessed November 27, 2017.