Mining Frequent Patterns with Optimized Candidate Representation on Graphics Processor

Dharmesh Bhalodia
RK University
Rajkot, Gujarat
India

Chhaya Patel
RK University
Rajkot, Gujarat
India

ABSTRACT

Frequent itemset mining algorithms mine subsets of items that appear frequently in a collection of sets.FIM is a key investigation in numerous data mining applications, and the FIM tools are among the most computationally demanding in data mining. In this research paper we present a new approach to represent candidate in parallel Frequent Itemset Mining algorithm. Our new approach is extension of GPApriori, a GP-GPU version of FIM. This implementation is optimized to achieve high performance on a heterogeneous platform consisting of a shared memory multiprocessor and multiple cores NVIDIA based Graphics Processing Unit (GPU) coprocessor. An experiments compared with the GPApriori on NVIDIA Kepler GPUs and observed 1.5X to 2X required less memory and significant improvements in time relative to GPApriori.

Keywords:

Association rule mining, CUDA, GPU computing, Frequent itemset mining, Parallel computing

1. INTRODUCTION

Continuous and considerable growth of the digital data and size of database make a persistent challenge for developing new data mining techniques. New problems appear every day and due to growing amount of data to be mined, emphasis on real-time constraints and changes in memory and computational efficiency, proven solutions tend to be insufficient and are in need of modification. One of data mining problems that always requires newer version which is fast and efficient mining of frequent patterns, especially under real-time constraints.frequent pattern mining was first defined by Agrawal and Srikant in [1] as Defined by, Given a set of transactions, where each transaction consists of a list of itemsets, and given a user-supplied minimum support threshold (min support), frequent pattern mining is to find all frequent subsequences whose frequency/occurrence is no less than min support.

The goal of Frequent Itemset Mining (FIM) is to find frequently occurs subsets within a database of sets. Many scientific and industrial applications, including those in machine learning, intrusion detection from fraud transaction detection, computational biology from particular pattern matching, web mining from web server log data mining, and e-business benefit from the use of frequent itemset mining. Much of the literature in frequent itemset mining highlights the development of algorithmic improvements as contrasting to parallelizing existing algorithms. As such, state-of-the-art FIM implementations are generally sequential and there is relatively little effort devoted to mapping these algorithms to high-performance platforms.

Frequent Itemset Mining is common in many commercial applications. A model can be shown in the sales data analysis of supermarkets .Take an example of super market. In that, each transaction is collected and after getting large amount of data. Then apply market basket analysis to find out the common patterns. The discovered patterns are set of items that are frequently repeated in database. Like ”Bread, Butter and Egg are most frequent items”. Decision making person use this detail for identify the customer buying habits. Like ”people who buy the Bread and Butter often also buy the Egg”. Analysis of these patterns can be useful for designing the layout of the super market goods, products usually sold together can be placed near each other. So it will be the higher probability of increase in sales.

The current advances architecture are transforming from traditional high performance distributed platforms into multi-core and many-core hierarchical environments. Thus, creating efficient FIM algorithms that fully exploit this parallelism is a very important contribution.

In addition, existing algorithms are unable to solve all frequent pattern mining problems in real-time, therefore there is a need for additional solutions. One of possible solutions is modification of existing algorithms by treating sequential mining problems as GPGPU (general-purpose computation on graphics hardware) problems and accelerating existing algorithms with use of Graphics Processing Units (GPUs).

1.1 General Purpose Computing with Graphics Processing Unit

In the last five years, two factors stimulated a renewed attention respectively in distributed and parallel methods for data mining, and in particular for high performance Frequent Pattern Mining methods. The first is the wide availability of commercial, Elastic and Distributed computing facilities, such as the Amazon Elastic Compute Cloud; the second is the Microprocessors with increasing number of cores availability, for example the recent NVIDIA Kepler microprocessor, the one used in GeForce GTX 770 and GeForce GTX 780 cards. Features 2880 core with 6GB

total Global Memory[2]. These innovations open new scalable opportunities on the one hand and demand for further care to handle their peculiarities on the other. Thus, to effectively utilize these opportunities, there is a need for a new generation and modification of the existing data mining algorithms, it is suited for coherent and predictable memory access, in able to exploit this particular General- Purpose computing paradigm on Graphics Processing Units (GP-GPU: http://gpgpu.org). This is due to the fundamentally different architecture programming model and design of many core GPUs with respect to traditional multi-core CPUs.

CUDA is a massively parallel computing platform and as well as programming model developed by NVIDIA Corporation, increasing computing performance for parallel problems with use of Graphics Processing Units. It also provides low and high level APIs (application programming interfaces) with debuggers, shared memory access, scattered reads and fast read backs. NVIDIA CUDA platform come with C programming extension and C++ integration too. Recently NVIDIA CUDA releases CUDA 6.0 which enable unified memory, in other word developers are free from manualy memory copy(*CudaMemcpy*) from host to device and visa versa. Now CUDA support such complex data structure like link list and tree.[3]

In this paper, we propose a new approach to represent candidate itemset in FIM algorithm that is optimized for a heterogeneous platforms consisting of GPUs. This allows for efficient utilization of the computational and memory resources of the GPU coprocessors. We have also redesigned and optimized the memory allocation strategy to improve the memory utilization of the algorithm. The experimental results demonstrate the performance benefit of our technique as compared to well known GPApriori a GP-GPU version of FIM

2. RELATED WORK

The research activity in the domain of association rule mining has been primarily focused on defining efficient algorithms to perform the frequent itemset mining task. This problem can be defined as follows. Let I be a set of items where $I = \{i_1, i_2, ..., i_n\}$. A database D is a collection of transactions, where each transaction t is a set of items in I. An itemset I is a set of items, characterized by its frequency of occurrence into D, that frequency called as support. Given a minimum support threshold as *minsup*, frequent itemset is the extracted from D of the complete set of itemsets which having larger support or equal to *minsup*.

While itemset mining is computationally concentrated, a number of articles have been proposed to parallelizing frequent itemset mining task. The earliest work on parallelizing itemset mining was based on Apriori like algorithms [4]. One step towards to more efficient parallel itemset mining algorithms has been proposed based on FP-Tree with prefix-tree-like structures [5]. Even multi tree algorithm has been proposed in [6] where Each thread analyzes a horizontal segment of the database then builds its own FP-tree and performs the mining process on individual segment. At final step a merge operation is required to join together the individual candidate pattern base. This approach is seems to be efficient and may achieve good scalability in performance. On the other hand a large amount of main memory and large number of threads when traversal of redundant node could be required. The work done on multi-core processor architecture to improve the frequent itemset mining task [7] to utilize the vacant hardware resources. Further enhanced in FP-Tree based method by path tilling technique [8]. This technique is retrieve the data from higher memory level by improving temporal locality access.In our previous work we presented AprioriDP [9] algorithm with Dynamic programming approach which shows 100x speed up for large itemset of size 1 and 2 compare to traditional Apriori.

Recent trend for frequent itemset mining toward to parallelize the algorithm with massive power of General Purpose Graphical Processing Unit (GP-GPU). We known Apriori algorithm implementation on GPU, was first time addressed in [10]. In this case, two kinds of their GPGPU implementation, one based on the ”pure bitmap” representation and another based on the ”trie-based bitmap” representation were proposed. These approach, the candidates and vertical transactions are coded into bitmaps and handled by GPU. Their dataset is represented as a binary matrix of $M \times N$ where M is number of items present in database and N is total number of transaction in database divide by size of integer. Calculating the transactions that support a given item set just requires to intersect rows of the matrix . The great advantage given by the adoption of a vertical bitmap representation, is that the expensive support counting is achieved with fast bitwise intersection and population count of bit-vectors.

Another Apriori based FIM algorithm for GPU is presented In GPApriori [11] and achieved higher magnitude in speed up ration with compare to CPU base algorithm. They also presented ”Frontier Expansion” Eclat [12] and FP-Growth [13] based implementation on GPU in [14], experimental results shows that they got sufficient improvement.

The TreeProjection approach is used by [15], based on the algorithm described in [16]. This work signify a significant improvement with respect to the parallel version of the Apriori algorithm. However, TreeProjection is not a state of the art algorithm for FIM, as it is outperformed by FP-Growth. Another state of art algorithm for FIM is Dynamic itemset counting [17] was implemented by [18] on GPU, they apply various techniques like transaction and candidate wise parallelism to improve the performance. Recently we conducted survey on recognized GP-GPU versions of FIM [19] , which thoroughly described comparative analysis on predefine parameters.

3. APRIORI ALGORITHM INTRODUCTION

In this section, we introduce the fundamental concepts of FIM algorithms. We discuss basic concepts in FIM algorithm design.

The initial solution to find frequent itemsets would be to generate all the k, $k \in \{1, ...N\}$ subsets of the universe of m items, count their support by scanning the database, and export candidate those meeting minimum support condition. The initial method exhibits exponential Complexity , because it requires the computation of the power set of m items, for example $\sum_{i=1}^{n} = 2^n - 1$, is impossible. The earliest solution, as formulated by the Apriori algorithm, is based on the rule support monotonicity that an item set is frequent if all its sub itemsets are frequent. Using this rule, the search space can be reduced by joining iteratively from smaller itemsets to larger ones of frequent itemsets and pruning candidates with infrequent subsets. The Apriori algorithm intensively study after first being published. Improvements are made in critical candidate generation and support counting steps of FIM. Candidate generation is used to generate $k + 1$ candidates from k frequent itemsets. Assume that the number of k itemsets are N, a complete join from the N itemsets expands candidate set size by $O(N^2)$. clustering itemsets can be use to decrease the cost of complete join operation by Appling Equivalent Class Clustering (ECC) [12] , which prevents the creation of redundant candidates in each new generation. ECC is able to generate candidate in $O(\delta N)$ instead of quadratic time, where δ is the expectation of the equivalent class size.

Support counting step is performed after the new candidates are generated. The minimum threshold value that is support value of the candidates, decides which of them frequent itemsets and removes infrequent candidates are. There are two ways to represent transactions in support counting step, those are horizontal representation and vertical representation. The most straightforward way is the horizontal representation in that each transaction ID is associated with a list of item IDs. Instead, the vertical representation associates each item ID with a list of transaction IDs. When using horizontal representation, in Apriori, support counting is performed by matching each candidate itemset against the sorted transactional database using a binary search. While using the vertical representation, the support of new candidates is computed by intersecting the vertical list of the previous generation with the vertical list of the item that has been added to form the new candidate. The vertical representation speeds up support counting by saving the occurrence information for the counted candidates but in the other hand consumes more memory. Figure 1 shows the horizontal and vertical transaction representation. Figure 1 (b) shows two forms of the corresponding vertical transaction lists: tidset and bitset. A tidset records itemset's occurrence information as an array of the transaction IDs, and a bitset represents the same information with a bit mask.

T_{ID}	Item
1	1,2,3,4
2	2,3,4
3	3,4
4	1,3,4

(a) Horizontal Transactions.

Candidate	Tidset	Bitset
{1}	1,4	1001
{2}	1,2	1100
{3}	1,2,3,4	1111
{4}	1,2,3.4	1111

(b) Vertical Transactions.

Fig. 1. Comparison of (a) horizontal representation and (b) vertical representation of transactions

4. ALGORITHM AND IMPLEMENTAION

In this section we describe our new approach to represent candidate itemset on GPU. The novelty of our approach is compact candidate size and fine-grain parallelization of the support counting algorithm.

4.1 Dataset Representation

Accelerating FIM with a GPU comprises careful attention of the vertical transaction representation. Whereas, Tidsets are linear ordered arrays, and to traverse them during the support counting operation it leads to poor performance due to unpredictable instruction branching behavior. The tidset representation is compact nevertheless join operations on Tidsets are highly dependent and difficult to parallelize. On the other hand, the Bitset representation is more suitable for designing a parallel set join operation but it requires more memory space compare to Tidsets. Bitsets is better suited for GPU, Joining two bitset transactions can be performed by a *bitwise AND* operation between the two bit rows. Below figure 2 shows memory access pattern difference between Tidsets and Bitsets representation Figure 2 (b) represents that Bitset memory access pattern is coalescing, which means all threads in a warp execute a load instruction, the hardware detects whether the threads access consecutive memory locations. The most favorable global memory access is achieved when the same instruction for all threads in a warp accesses global memory locations. In this

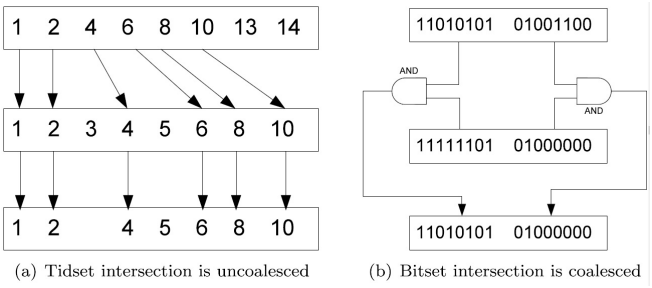

Fig. 2. Comparison of Tidsets and Bitsets intersection

favorable case, the hardware coalesces all memory accesses into a consolidated access to consecutive DRAM locations. If thread 0 accesses location n, thread 1 accesses location $n+1$, ... thread 31 accesses location $n+31$, then all these accesses are coalesced, that is combined into one single access. The CUDA C Best Practices Guide gives a high priority recommendation to coalesced access to global memory [20]

4.2 Candidate Representation

In order to explain our new candidate set representation, we first introduce the concept of "Equivalent Class", which can be defined by a set of candidates which share the common $k+1$ prefix. For example, (1,2,3), (1,2,4) and (1,2,5) are in the same equivalent class (1,2, x). But (1,2,3) and (1,3,4) are not in the same class because they have the different $k-1$ prefix (1,2,x) and (1,3,x).
In GPApriori, candidate representation is based on Traditional Apriori, where candidates with size k and total number of candidate l consume $k \times l$ memory. Instead we can represents candidate set using equivalent class. For that we require two lists, one list will store equivalent class candidate, in other word we store those candidate of size $k-1$ which shared by k^{th} candidates. And in second list we store k^{th} candidates from equivalent class. i.e. if we take 50% support count on dataset shown in figure 1 we get 5 candidate of size 2 shown in figure 3, it demonstrate the two ways to represent the candidate sets in cuda. In figure 3(a) one to one

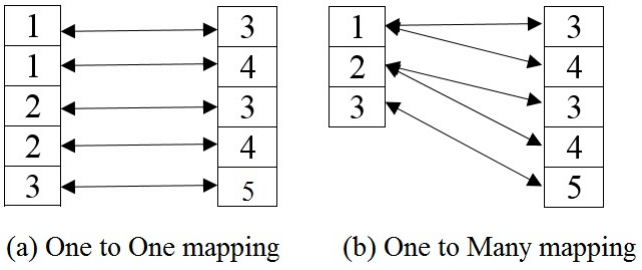

Fig. 3. Candidate Representations

mapping of $k-1$ candidate to k^{th} candidate. Alternatively we can represent those candidates by one to many relation, as shown in figure 3 (b). in one to one mapping $k-1$ candidates are repeated until next equivalent class. In contrast we represent $k-1$ candidate once for the same equivalent class. In figure there are 3 equivalent classes so only three $k-1$ candidates are required. But in traditional GPApriori require 5 candidates of size $k-1$.
Here we address the critical problem when there are large number

of candidates with size k require redundant $k-1$ candidates, it requires more GPU global memory, which is limited in size. Another problem is $k-1$ bitsets intersections. Those intersection performs on GPU global memory so it leads to higher *global read* penalty.

4.3 Support Counting

Once candidate generation step completes, two lists of candidates are transfer to the GPU global memory. The CUDA kernel structure consists of threads, blocks and grids. Where group of Threads called Block and group of Blocks called Grid. In each Thread, inherited parameter blockIdx and threadIdx for identify unique Block and Thread respectively. 32 threads within the same block called warp will be grouped on to the same SIMD stream processor, each stream processor is assigned to a set of warps, only one or more warps will be active at a time depend on compute capability, and the stream multiprocessor rotates between warps [21].

Kernel 1: Bitset Pattern Cache

Input: $Candidate$ array of $K-1$ items
Output: Bitset pattern of $K-1$ item in $BitsetCache$
tid=$ThreadID.X \times BlockID.X + BlockDim.X$;
if *($tid \succ Bitset\ width$)* **then**
    Break;
end
for *j = 1 to $candidate\ width$ - 1* **do**
    $BitsetCache[tid]$ = $BitsetCache[tid]$ &
    $Bitset[Candidate[j] \times width + tid]$;
end

Our GPU support counting kernels computes the Bitset intersection and calculate the support value for each candidate. kernel 1 passed with first equivalent class candidate of size k-1, i.e. "1" in figure 3, which perform the intersection or *AND* operation with Bitset and store in *BitsetCache*. *BitsetCache* is temporary array with length equal to Bitset width. Now Kernel 2 performs intersection between Bitset of kth candidates and *BitsetCache*.

Kernel 2: Support Counting

Input: $BitsetCache$ and $Candidate$ array of K^{th} item
Output: Support Count of each candidate in $support$
can = $Candidate[BlockID.X]$;
for tid = *1 to* $Bitset\ width$ **do**
    $temp$ = $BitsetCache[tid]$ & $Bitset[can \times width + tid]$;
    $sup[tid]$+= $popc(temp)$;
end
$support[BlockID.X]$ = $Parallel\ Reducation()$;

Here *BitsetCache* array act as caching mechanism for $k-1$ candidate and hide computation overhead and redundancy. Another advantage is only one intersection, of candidate size of $k-1$, required within an equivalent class. Where as in GPApriori does same work very time it intersects for candidate size of $k-1$.
The Bitset representation is better suited for GPU-based support counting than the tidset representation. The width of the Bitset equals the number of the transaction in the database divide by 32. Because each thread will read 32 bit data. Each 0 or 1 represents whether the corresponding transaction contains the candidate or not. Intersection of two bit-represented transaction lists can be performed by a "bitwise AND" operation between *BitsetCache* and Bitset.
Once we get local count of "1" for each 32 bit data in *sup*, we need global count for entire transaction of candidates. Figure 4 demonstrates how Kernel 2 support counting is computed on the GPU, same process done by Fang et al.[14] . Each candidate intersections will be count by one Thread Block. Threads in the Block will process intersections of a word-length subset(32 or 64 bit). The width of Bitset is rounded to be the multiple of 64 bytes to ensure coalesced memory reads. The intersection results of each thread are stored in a 32-bit integer, and the number of "1" present s in the integer is counted by CUDA in-built *popc* (Population Count) function and stored in shared memory. The parallel summation reduction algorithm [21] is used to add all the values in shared memory recursively into its first position of array *sup[0]*. The support count for the candidate is then written back to an output buffer on GPU memory and transferred back to host memory or CPU memory. The upper half of the figure 4 intersect and give the local count of candidates and lower half give global count by parallel reduction algorithm

5. EXPERIMENTS

In this section, we describe the performance of our new approach, and we analyze the experimental results. Our results are tested and collected on a Intel core 2 dual PC with GeForce GTX 680. The code was written in C++ and compiled with CUDA 5.0. In this section when we use the term "GPU" as Graphics device itself, which in our case contains 8 "streaming multiprocessor" cores, with each of these containing 192 scalar cores, for a total of 1536 GPU cores.
In our results we compare the performance of our new approach with GPApriori [11] algorithms code collected from Zhang et al., repository. According to their experimental result it clearly proven that GPApriori outperform Borgelt Apriori and achieve 4X - 10X speed up on most modern datasets. So in this experiment we analyze the performance between our new approach and GPApriori. The synthetic datasets , T10I40D100k and T40I10D100k, are generated by IBM Market-Basket Synthetic Data Generator from Paolo Palmerini's DCI website [22]. The real datasets, Retail, Chess and Mushroom, are collected from UCI Data Mining Repository. To generate large amount of candidates we set lower support count ratio to demonstrate the memory requirement of algorithms. The characteristic of the datasets found in Table 1.
We executed each experiment five times and collect the mean results. The whole GPApriori computation is comprised of three parts: initialization on GPU, candidate generation, and support counting on GPU. We do not analyze the initialization and candidate generation part because we follow the GPApriori method, in that approach initialization of the GPU memory with the input dataset and this dataset remains in the GPU global memory until entire execution is completed. In our experiment we analyze the global memory occupied by total number candidates transfer to GPU Global memory and Bitset.

5.1 Performance analysis on Real datasets

Figure 5 demonstrate the performance result with respect to minimum support count ratio on Retail dataset. To obtain clear understanding on how memory occupancy requirement affected by both the algorithm we set lower minimum support ratio. As shown in figure, proposed method require less memory compare

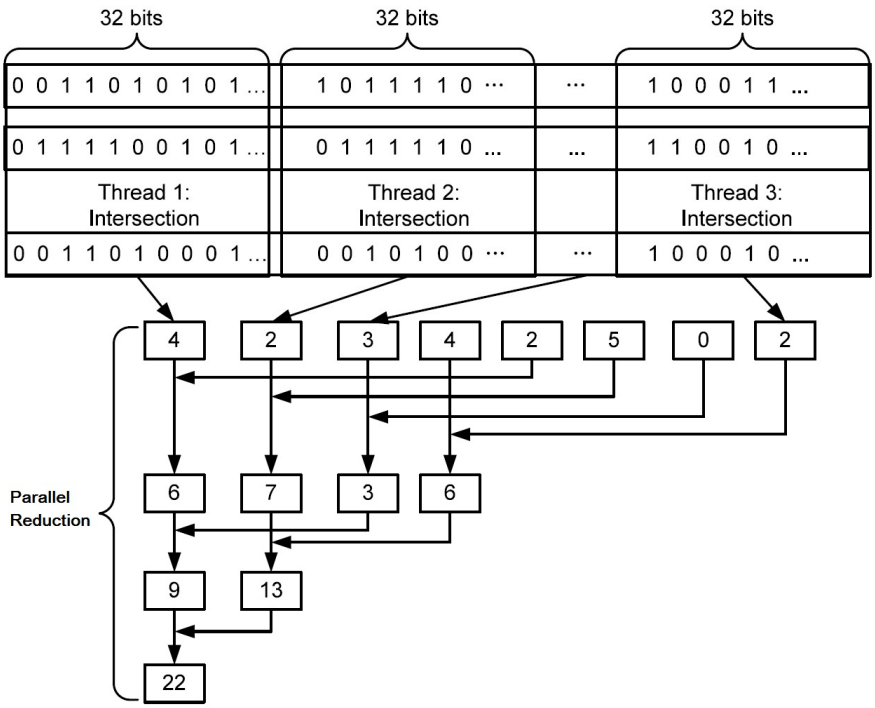

Fig. 4. Support counting process on GPU [14]

Table 1. Synthetic and Real dataset characteristic

Name	Total No. of Transactions	Item	Avg. Trans. Len	Density	Type
Chess	3196	75	37	0.37	Real
Retail	88163	16470	10	0.1	Real
Mushroom	8125	119	25	0.25	Real
T10I4D100K	100000	1000	10	0.10	Synthetic
T40I10D100K	92113	943	40	0.40	Synthetic

to GPApriori. At 5% and 4% support count we get difference of more than 100 MB memory requirement.

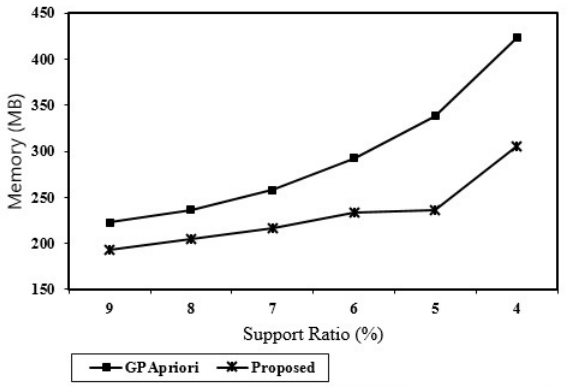

Fig. 5. Performance comparison between our new proposed approach and GPApriori with varying minimum support on Retail dataset .

Figure 6 demonstrate the performance result of time require to complete the support counting step on GPU. We used two different Graphics Processors for portability purpose. In figure (a) execution time of support counting step on GTX 680 Card, and in (b) GTX 420 Card. As shown in both the figure we achieved 2X speed up ratio. Figure 7 demonstrate the performance result of achieved memory transfer bandwidth throughput with respect to minimum support count ratio. As shown in figure, throughput is inversely correlated to minimum support count ratio. At higher threshold count there is a less memory requirement and higher throughput. For this performance also we used two different Graphics Processors, same used in previous one. On both the results we achieved higher throughput in bandwidth. Our proposed algorithm maximum achieved bandwidth is more than 1 GB/s, where as in GPApriori maximum bandwidth is 800 MB/s, on both the Graphics Processors. These Bandwidth calculated by NVIDIA Visual Profiler Tool.

Figure 8 demonstrate the performance result of memory occupancy requirement with respect to minimum support count ratio, on Chess and Mushroom datasets. As shown in figure, on both real datasets our new approach require less amount of memory with compare

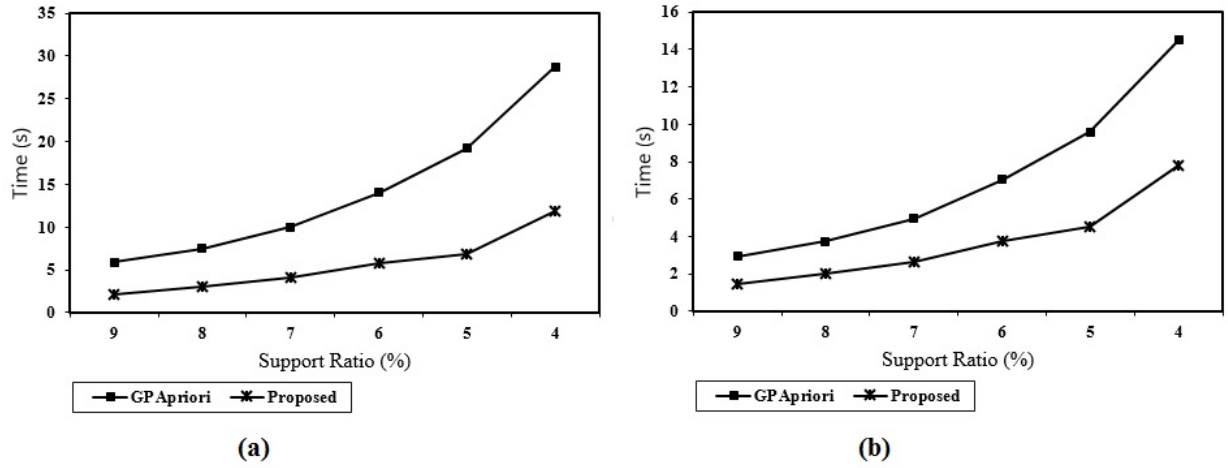

Fig. 6. Performance comparison between our new proposed approach and GPApriori with varying minimum support on Retail dataset (a) GeForce GTX 480 and (b) GeForce GTX 680.

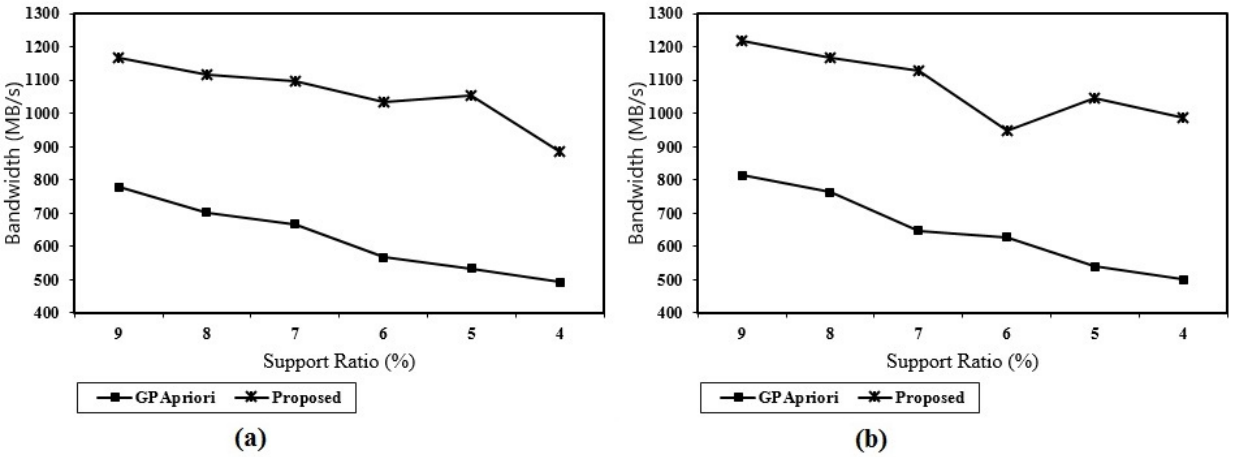

Fig. 7. Performance comparison between our new proposed approach and GPApriori with varying minimum support on Retail dataset (a) GeForce GTX 480 and (b) GeForce GTX 680.

to GPApriori. Chess dataset is dance compare to Mushroom, So Chess dataset will generate large number of candidates at lower threshold count and some time it is limitation of hardware to store all candidate in main memory. to demonstrate performance analysis on this dataset we kept minimum threshold or support count ratio as we can.

5.2 Performance analysis on Synthetic Datasets

Figure 9 show the performance study of our proposed algorithm on two different Synthetic datasets. In figure (a), after 35% support count, our proposed algorithm requires 2X less memory compare as to GPApriori. And in figure (b), which is dense dataset compare to (a), after 5% support count proposed algorithm require 1.5X less memory as compare to GPApriori.
And now, some more datasets generated by IBM Market-Basket Synthetic Data Generator from Paolo Palmerini's DCI website [22]. The characteristic of these datasets are listed in Table 2, where average Transaction size is very. we taken four different datasets to analyze the performance with respect to density of datasets.

Table 2. Synthetic Datasets characteristic

Name	# Trans.	# Items.	Avg. Trans	Density
T15I10D100K	1,00,000	1,000	15	15
T20I10D100K	1,00,000	1,000	20	20
T25I10D100K	1,00,000	1,000	25	25
T30I10D100K	1,00,000	1,000	30	30

Finally, figure 10 demonstrate the Performance comparison between proposed approach and GPApriori with varying **Density** of the Synthetic datasets. We observe that at higher Density Memory Consumption reduce by 50% to 75%. Here we kept higher support count (60%), but at even lower support count, we may observe large different between both implementations.

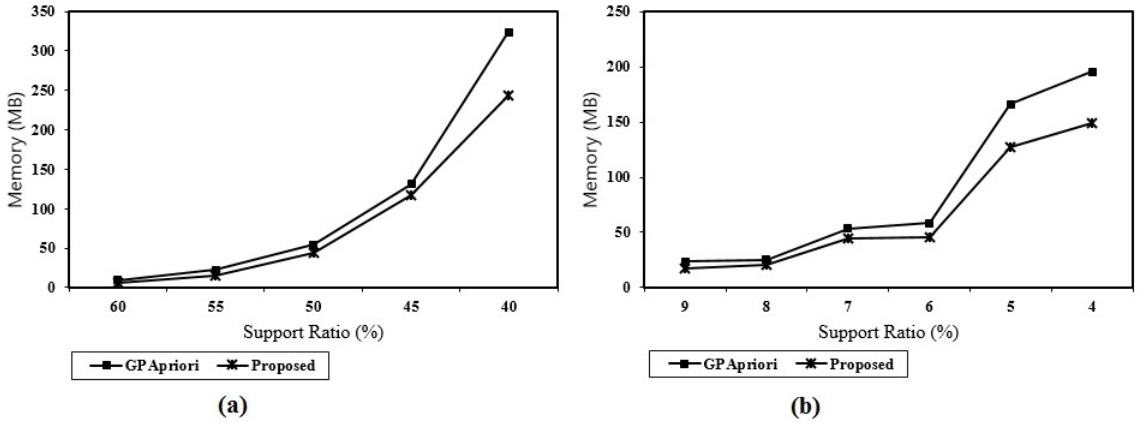

Fig. 8. Performance comparison between our new proposed approach and GPApriori with varying minimum support on (a) Chess Dataset and (b) Mashroom Dataset.

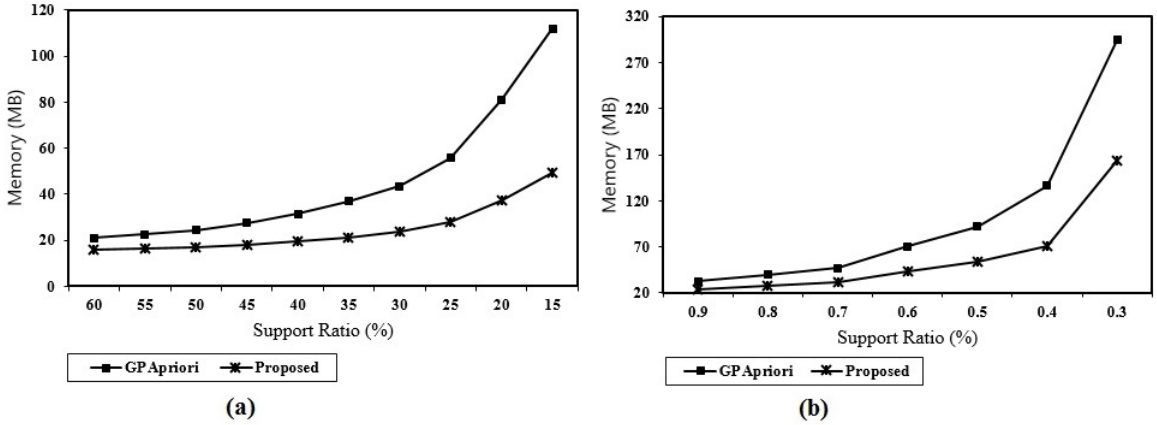

Fig. 9. Performance comparison between our new proposed approach and GPApriori with varying minimum support on (a) T10I4D100K dataset and (b) T40I10D100k Dataset.

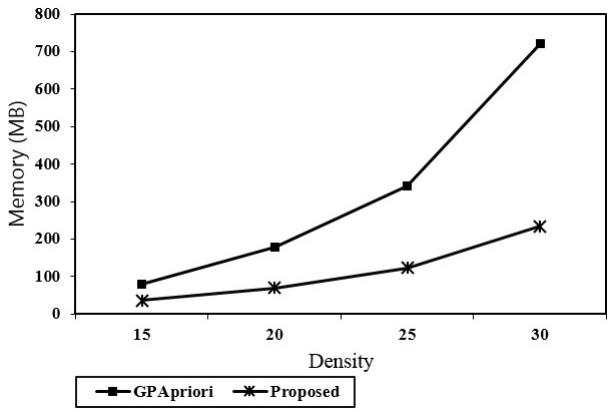

Fig. 10. Performance comparison between our proposed approach and GPApriori with varying Density and Fixed Support Count Ratio 60 %

6. CONCLUSIONS

In this research paper we described our new approach to represent the candidate itemsets, consisting of GPU kernels for FIM support counting based on Bitset representation. We also conducted a detailed analysis of the algorithm's performance on two different Graphics Processors. And compare with well known GPApriori GP-GPU version of FIM. We evaluated this approach for real datasets and synthetic datasets by varying the target support threshold. Our results indicate that our new approach require less memory compare to state-of-the-art GPU implementation. And, in general, using the GPU for computing the support of candidate patterns, gives clear advantages.

In the future we plan to improve the our new proposed algorithm in several directions. The most recent version of GPU based Apriori does not take advantage of available resources like multi-core CPU. We plan to extend proposed algorithm in order to exploit these additional resources, using a parallelization approach similar to candidate wise and transaction wise parallelism. The single host can have more than one device (GPU). The algorithm we proposed

in this research paper makes use of a only GPU. However, we can exploit more parallelism with more than one GPU.

Acknowledgment

We are very thankful to Prof. Amit M. Lathigra, HOD, School of Engineering, RK University, Rajkot, Gujarat, for proving base resources and grant for experimental work under CUDA Teaching Center and moral supports as and when required while research work. Also we would like to express deep gratitude to staff members of M.Tech for giving their continuous courage and motivation to publish research work.

7. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between set of items in large databases," *ACM SIGMOD International conference on management of data*, pp. 207–216, May 1993.

[2] *NVIDIA Kepler GK110 Architecture Whitepaper(White Paper)*, january 2013. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[3] *NVIDIA CUDA TOOLKIT VERSION 6.0 RELEASE NOTE*, Feb. 2014. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/6_0/rc/docs/CUDA_Toolkit_Release_Notes.pdf

[4] M. J. Zaki, "Parallel and distributed association mining: A survey," *IEEE Concurrency*, vol. 7, no. 4, pp. 14–25, October 1999.

[5] J. P. J. Han and Y. Yin, "Mining frequent patterns without candidate generation," *in SIGMOD*, pp. 1–12, 2000.

[6] M. E.-H. O. R. Zaiane and P. Lu, "Fast parallel association rule mining without candidacy generation," no. 1, 2001, pp. 665–668.

[7] Y. Z. L. Liu, E. Li and Z. Tang, "Optimization of frequent itemset mining on multiple-core processor," 2007, pp. 1275–1285.

[8] S. P. D. K. A. N. Y. K. C. A. Ghoting, G. Buehrer and P. Dubey, "Cache-conscious frequent pattern mining on modern and emerging processors," *The VLDB Journal*, vol. 16, no. 1, pp. 77–96, 2007.

[9] P. K. M. Bhalodiya Dharmesh and P. C., "An efficient way to find frequent pattern with dynamic programming approach," in *Engineering (NUiCONE), 2013 Nirma University International Conference on*, Nov 2013, pp. 1–5.

[10] X. X. B. H. Q. L. Wenbin Fang, Mian Lu, "Frequent itemset mining on graphics processors," *Proceedings of the Fifth International Workshop on Data Management on New Hardware DaMoN2009*, 2009.

[11] Y. Z. Fan Zhang and J. Bakos, "Gpapriori: Gpu-accelerated frequent itemset mining." *In 2011 IEEE International Conference on Cluster Computing (CLUSTER)*, no. 3, pp. 590–594, March 2011.

[12] P. S. Zaki MJ, "New algorithms for fast discovery of association rules." Menlo Park: AAAI Press, 1997, pp. 283–286.

[13] P. J. Han J, "Mining frequent patterns without candidate generation: a frequent-pattern tree approach," *Data Mining Knowl Discov*, no. 8, pp. 53–87, 2004.

[14] Y. Z. Fan Zhang and J. Bakos, "Accelerating frequent itemset mining on graphics processing units," *Journal of Supercomput*, no. 66, pp. 94–117, February 2013.

[15] W. M. J. George Teodoro, Nathan Mariano and R. Ferreira, "Tree projection-based frequent itemset mining on multicore cpus and gpus." Washington, DC, USA: IEEE Computer Society, March 2010., no. 3.

[16] C. C. A. Ramesh C. Agarwal and V. V. V. Prasad, "A tree projection algorithm for generation of frequent item sets," *Journal of Parallel Distributed Computing*, no. 3, March 2001.

[17] R. Agrawal and J. Shafer, "Parallel mining of association rules," *In IEEE Trans. on Knowledge and Data Engg*, no. 8, pp. 962–969, 1996.

[18] F. V. C. S. Salvatore O., Universit a Ca, "Exploiting gpus in frequent itemset mining," *20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pp. 416–425, 2012.

[19] B. Dharmesh and P. Chhaya, "Comparative study of frequent itemset mining techniques on graphics processor," *International Journal of Engineering Research and Applications*, vol. 4, no. 1, pp. 159–163, April 2014.

[20] *CUDA C BEST PRACTICES GUIDE*, July 2013. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf

[21] *NVIDIA CUDA compute unified device architecture programming guide*, February 2014. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[22] B. G, *Frequent itemset mining dataset repository*, 2004. [Online]. Available: http://fimi.ua.ac.be/data/