Chapter 27
Research on Parallel Association Rules Mining on GPU

Qingmin Cui and Xiaobo Guo

Abstract In this paper, general-purpose computation on graphics processing unit (GPGPU) is playing an important role in super-computing. We proposed a parallel association mining solution based on graphics processing unit (GPU) using Compute Unified Device Architecture (CUDA)–Cuda Apriori. The support-counting step for candidate frequent itemsets is off-loaded from CPU to GPU. First, candidate frequent item sets and transactions are partitioned in the pattern of thread block and grid of thread blocks of GPU. Second, the task of support counting is performed in parallel by massive threads with the simple matching computation, suitable to stream access model of GPU. In our experimental work, we simulated transactions on both our Cuda Apriori and the standard Apriori.

Keywords Association rule · Data mining · GPU · Support counting

27.1 Introduction

The rapid increase in the performance of graphics processing unit (GPU), coupled with recent improvements in its programmability, have made it a compelling platform for computationally demanding tasks in a wide variety of application domains [1] (called General Purpose GPU, GPGPU). Current-generation GPUs are designed to act as high performance stream-processors. Their performance is derived from parallelism at both the data and instruction levels. Additionally,

Q. Cui (✉) · X. Guo
School of Computer Science and Engineering, Henan Institute of Engineering,
Zhengzhou, China
e-mail: eielw@sina.com

current-generation GPUs are architected to take advantage of the independence of the data-elements in traditional graphics scenarios. This data-independence enables a memory-model that is simple and fast. Although these characteristics allow for high levels of performance, they pose challenges when conventional (i.e., non-graphical) applications are ported to the GPU. The streaming architecture can be a difficult programming environment for developers accustomed to working with traditional CPUs. In particular, the simplified memory model is restrictive when it comes to implementing well-known data structures such as linked-lists, trees, etc. Thus, although the GPU provides a new opportunity for optimizing the runtime performance of conventional algorithms, these algorithms must be redesigned into a form that is appropriate for the parallel, stream-architecture of GPUs.

In this paper we use the GPU to our advantage by demonstrating that offloading support counting computation to the GPU allows Apriori to function effectively at significantly higher frequent item sets-processing. To this end, we ported the Apriori [2], the newest generation GPU. In the experiment we simulated transactions on both our GPU-based version of Apriori (called compute unified device architecture, Cuda Apriori) and the standard Apriori. The results show that Cuda Apriori produces a 10-fold performance enhancement of frequent k-item sets mining ($k > 2$) step to Apriori and outperforms it by up to 80 % on the whole.

27.2 Cuda Apriori Algorithm

Apriori is a heavy computational task. Let $|I| = m$, there are 2^m subsets to be the candidate frequent itemsets at worst and selecting all frequent k-itemsets need to scan the whole transaction once. Agrawal and Srikant [2, 3] etc., are a class of Apriori-like algorithms with characteristics of large candidate frequent itemsets generation and transactions scan multi-times. Another is Fp-growth [4] algorithm, which adopts frequent-pattern growth to overcome the fault of Apriori. Fp-growth has better behavior at dense transactions and is opposite to the sparse. Especially, it is unreal to construct FP tree in main memory for large database (Table 27.1).

The parallel algorithms based on Apriori-like or Fp-growth usually run on shared-memory systems or clusters platform, whose bus and message transfer delay cannot be neglected [4–6].

In this section, we give a new association rules mining algorithm employing G80 and the Compute Unified Device Architecture (CUDA): The transactions and candidate frequent itemsets are partitioned, and support counting task suited the stream model is parallelized using massive threads on multi-processors of G80.

27.2.1 Data Partition and Task Parallelization

The G80 contains a set of multiprocessors, each of which contains a set of stream processors that operate on single instruction multiple data (SIMD) programs.

Table 27.1 Notations

k-itemset	An itemset having k-items
L_k	Set of frequent k-itemsets (those with minimum support) Each member of this set has two fields: (1) itemset and (2) support count
C_k	Set of candidate k-itemsets (potentially frequent timesheets) Each member of this set has two fields: (1) itemset and (2) support count
B^i	Block of G80 with id i
$T^{i,j}$	Thread of G80 with id j in block B^i
D^i	The dataset local to the block B^i
C_k^i	The candidate set maintained with the block B^i during the kth pass (there are k items in each candidate)
$C_k^{i,j}$	The candidate set maintained with the block B^i and thread T^j during the kth pass (there are k items in each candidate)

The stream processors in the G80 are general purpose. They are quite different from earlier GPU design, which had fixed numbers of special-purpose processors (e.g., vertex and fragment shades), very limited support for arbitrary memory accesses (scatter/gather), and little or no support for integer data types.

Figure 27.1 illustrates the Programming Model of CUDA. A unit of work issued by the host (equal to CPU) to the G80 is called a kernel and defines the computation to be performed by a large number of threads, organized in blocks. Each multiprocessor executes one or more blocks. Blocks are organized in grid. Each kernel is executed as a batch of threads organized as a grid of blocks.

Each thread is with thread ID, which is the thread number within the block. To help with complex addressing based on the thread ID, an application can also specify a block as a two- or three-dimensional array of arbitrary size and identify each thread using a 2- or 3-component index instead. For a two-dimensional block of size (B_x, B_y), the thread ID of a thread of index (x, y) is (x, B_y) and so is a three-dimensional block. Each block is identified with block ID, which is the block number within the grid. An application can also specify a grid as a two-dimensional array of arbitrary size and identify each block using a 2-component index instead. For a two-dimensional grid of size (G_x, G_y), the block ID of a block of index (x, y) is (x, yG_y).

The strategy of partitioning the transactions and candidate frequent itemsets is: let the total number of threads in one block be t, and the total number of blocks in one grid be b, parallel computation model based on G80 is shown in Fig. 27.2. Candidate k-itemsets C_k^i are averaged to $C_k^{i,j}$ ($i = 0, 1,\ldots,b - 1; j = 0, 1,\ldots,t - 1$) in block B^i, while transactions set D is averaged to D^i in whole grid. Every thread $T^{i,j}$ counts the support of every itemset in $C_k^{i,j}$ based on D^i, then every block B^i can complete the C_k^i and a grid does C_k.

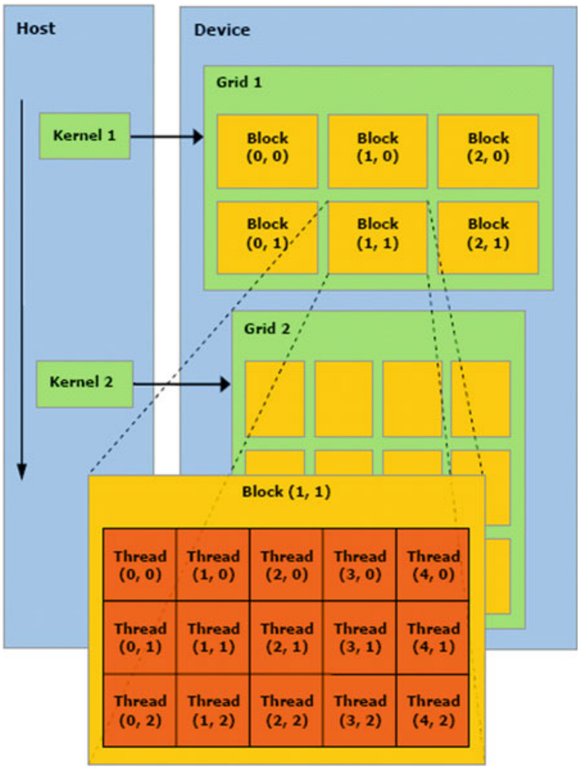


Fig. 27.1 Programming model of CUDA

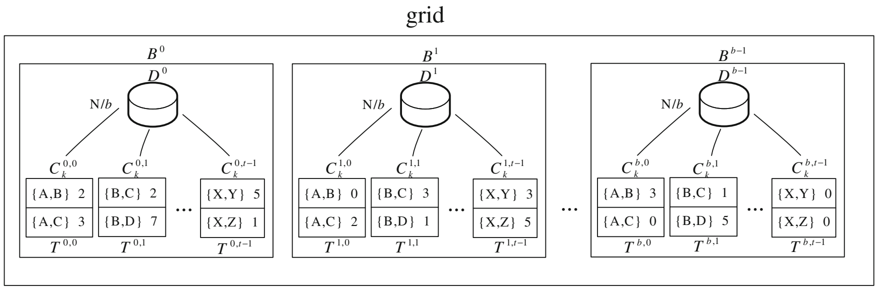

grid
B^0 D^0 N/b
$C_k^{0,0}$ {A,B} 2 {A,C} 3 $T^{0,0}$
$C_k^{0,1}$ {B,C} 2 {B,D} 7 $T^{0,1}$
$C_k^{0,t-1}$ {X,Y} 5 {X,Z} 1 $T^{0,t-1}$
B^1 D^1 N/b
$C_k^{1,0}$ {A,B} 0 {A,C} 2 $T^{1,0}$
$C_k^{1,1}$ {B,C} 3 {B,D} 1 $T^{1,1}$
$C_k^{1,t-1}$ {X,Y} 3 {X,Z} 5 $T^{1,t-1}$
B^{b-1} D^{b-1} N/b
$C_k^{b,0}$ {A,B} 3 {A,C} 0 $T^{b,0}$
$C_k^{b,1}$ {B,C} 1 {B,D} 5 $T^{b,1}$
$C_k^{b,t-1}$ {X,Y} 0 {X,Z} 0 $T^{b,t-1}$

Fig. 27.2 The transactions and candidate itemsets partition

27.2.2 Support Counting on G80

Apriori implemented by CPU widely uses a hash-tree for determining the candidates in C_k contained in a given transaction d, while such a data-structure would be difficult on GPU given the simple memory model of GPU. Thus we opted instead

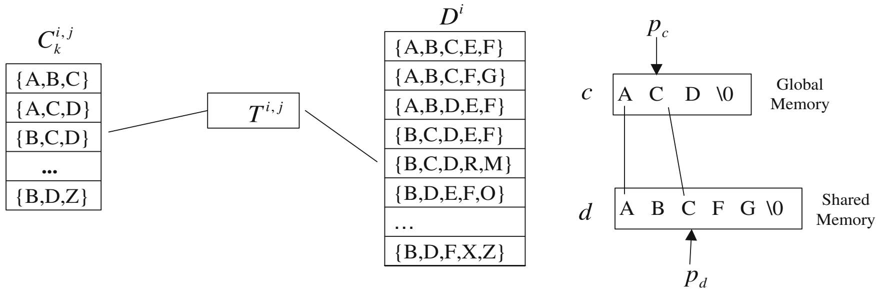


Fig. 27.3 Support counting on G80

to use the parallel rendering pipelines of G80 to build a string match method as Fig. 27.3 shows:

To process support counting, the following steps are required:

(1) To sort every item in $C_k^{i,j}$ and D^i in lexicographic order. It can be done by the host before the invocation of kernel. It does not product extra spending compared to the sequential Apriori algorithm.

(2) Thread $T^{i,j}$ continuously reads a transaction d (such as {A B C F G}) of D^i from Global Memory to Shared Memory shared by all threads in one block. This memory is on-chip and can be accessed much quicker than Global Memory. We retain candidate k-itemsets in Global Memory because the length of every k-itemset is k, which is very small and only scans once relative to single d.

(3) Thread $T^{i,j}$ increases support counts of every element c in $C_k^{i,j}$, if c is contained in d. The procedure of judging whether $c \subseteq d$ or not follows:

Step 1: Data points p_c and p_d respectively reside in the first element of c and d.

Step 2: If p_c points to '\0', then $c \subseteq d$ is correct and the procedure is over, otherwise, if two elements pointed to by p_c and p_d are equal, both p_c and p_d move next and the current step should be repeated. If not, go to Step 3.

Step 3: P_d moves next. If p_d points to '\0', $c \not\subseteq d$ and the procedure is over, otherwise go to Step 2.

Support counting on G80 adopts the sequential access model on the GPUs memory, which is very fit to the stream model of GPU.

27.3 Experiments

We compared Cuda Apriori with Apriori. All experiments were performed on a Dell Compatible PC with an Intel Pentium D CPU 3.7 GHz, 1G-byte main memory and Geforce 8800GTX graphic card. 768 MB of RAM, and 128 stream processors, organized into 16 multiprocessors. Each stream processor executes at 1.35 GHz. The raw (theoretical) compute power of the 8800GTX is approximately

350 GFLOPS. Apriori is implemented using Microsoft Visual.net 2003 and Cuda Apriori using CUDA.

We use synthetic 10 datasets, ranging from 100 K to 1 M, to find out the relationship between the performance of Cuda Apriori and the size of datasets. Average size of frequent itemsets is comparatively small as Apriori algorithm has a good exhibition under short pattern.

The efficiency estimation includes the support-counting price and total price. The support counting price of Apriori notated *CAsc* is (the time consumed by CPU at supports counting/($k-1$)); corresponding to that, CudaAsc is (the time consumed by G80 at supports counting/($k-1$)). k is the length of max frequent itemsets. Apriori's total time notated as CAac is measured as the time elapsed from the initiation of the execution to the end time of the association rule generation completed. So is the Cuda Apriori, and notated as CudaAac.

27.3.1 CAsc Versus CudaAsc and CAac Versus CudaAc

First of all, we compare CAsc to CudaAsc in Fig. 27.4 (*T* is average transaction length and *I* represents average size of frequent itemsets). The results are very encouraging: we get a 10-fold performance enhancement. On the other hand, Fig. 27.5 shows that CudaAac is able to outperform CAac by up to 80 %. CAac/CudaAac is far smaller than CAsc/CudaAsc, because the time of frequent k-itemsets mining ($k > 1$) only occupies half of the whole cost which also includes data input, initial frequent 1-itemsets and the finial rules generation etc. In spite of L_1 mining also can adapt our method, we use the traditional implementing fashion that single item's support counting is carrying out at the same time of reading transactions as data is seldom distributed on disk.

27.3.2 The Cost of Data Transfer

The extra cost of Cuda Apriori is data transfer. Candidate k-itemsets should be copied from host to G80 before support counting, and counting results from G80 to

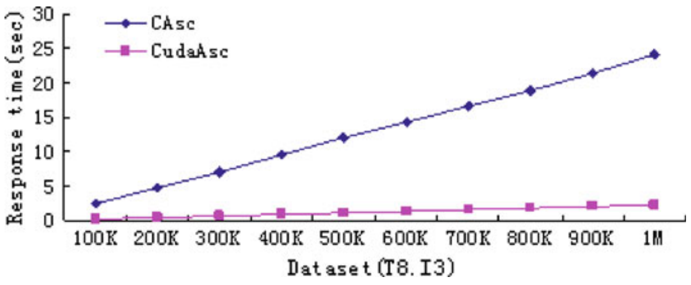


Fig. 27.4 CAsc vs. CudaAsc

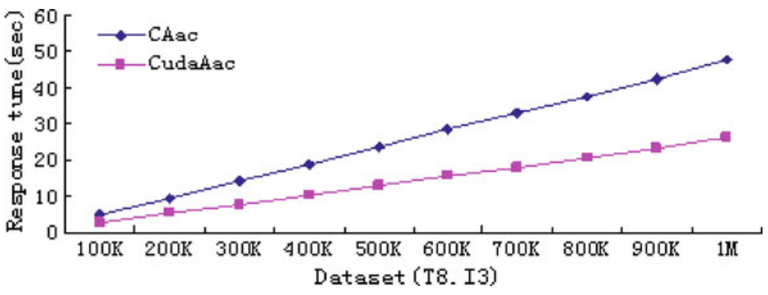
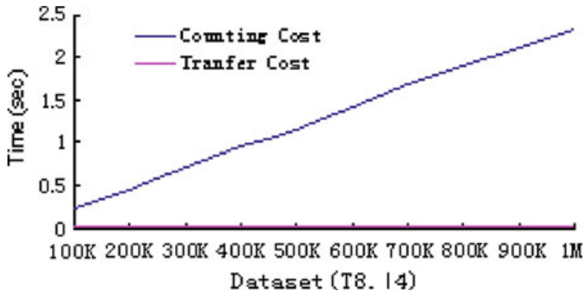


Fig. 27.5 CAac vs. CudaAac

Fig. 27.6 The cost of data transfer



host after that operation. The data transfer cost and support counting cost of G80 containing the data transfer cost is given as follows.

Figure 27.6 shows the cost doesn't vary under the all datasets. The transfer rate between the G80 and host, under the CUDA, is about 2 GB/s so that it hardly brings influence on the total executing time of Cuda Apriori.

27.4 Conclusion

The result shows that Cuda Apriori produces a 10-fold performance enhancement of frequent *k*-itemsets ($k > 2$) mining phase to Apriori and outperforms it by up to 80 % on the whole. In spite of the data, transmission of Cuda Apriori between GPU and CPU is the extra cost to Apriori, its performance reducing can be neglected with the high 2 GB/s speed road.

With the rapid progress of semiconductor technology and graphic chip, the company is taking GPGPU in market seriously. GPUs are developing towards lower latency of memory, with more multi-processor and higher transmission rate. Schemes including neural networks, concept lattice, and other data mining algorithms are waiting for migrating to GPU.

References

1. John OD, David L, Naga G, Mark H (2007) A survey of general-purpose computation on graphics hardware. Comput Graph Form 26(1):80–113
2. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules. In: Proceedings of international conference on very large databases 4(2):43–46
3. Park JS, Chen MS, Yu PS (1995) An effective hash-based algorithm for mining association rules. In: Proceedings of ACM SIGMOD on management of data 45(38):35–43
4. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Proceedings of ACM SIGMOD on management of data 17(12):63–68
5. Agrawal R, Shafer JC (1996) Parallel mining of association rules. IEEE Trans Knowl Data Eng 8(6):962–969
6. Aflori C, Craus M (2007) Grid implementation of the Apriori algorithm. Adv Eng Softw 38(25):295–300