gpuDCI: Exploiting GPUs in Frequent Itemset Mining

Claudio Silvestri, Salvatore Orlando
Università Ca' Foscari Venezia
silvestri@unive.it,orlando@unive.it

Abstract

Frequent itemset mining (FIM) algorithms extract subsets of items that occurs frequently in a collection of sets. FIM is a key analysis in several data mining applications, and the FIM tools are among the most computationally intensive data mining ones.

In this work we present a many-core parallel version of a state-of-the-art FIM algorithm, DCI, whose sequential version resulted, for most of the tested datasets, better than FP-Growth, one of the most efficient algorithms for FIM. We propose a couple of parallelization strategies for Graphics Processing Units (GPU) suitable for different resource availability, and we present the results of several experiments conducted on real-world and synthetic datasets.

1. Introduction

Frequent Itemset Mining (FIM) is one of the main and most demanding task in the data mining (DM) field. As an example of FIM analysis, consider a database where the data records are shop sales *transactions*, each composed of distinct bought *items*. The goal is to find the sets of items (*itemsets*) that are bought together in not less than a given number of transactions, namely the *minimum support*. FIM is not only an interesting problem by itself and a crucial part of Association Rules Mining (ARM)[16], but it also has a key role in the solution of several other DM problems. For this reason, even two decades after this problem was first introduced [3], it is still an active research topic. Han et al. in [9] provide a comprehensive survey of the state of the art in FIM and devise future research directions.

The challenges in FIM derive from the large size of its search space, which, in the worst case, corresponds to the power set of the set of items, and thus is exponential in the number of distinct items. Restricting as much as possible this space and efficiently performing computations on it are key issues for FIM algorithms. However, when the number of transactions/items are huge, or the minimum support is very small, analysts who need to quickly explore a large dataset through a FIM tool, ask for new techniques to improve algorithm performance, by taking advantage of the evolutions in distributed computing systems and parallel computing. Indeed, since FIM introduction, several works proposed distributed and parallel methods to deal with larger scale problems. A classical survey on parallel and distributed association-rule-mining algorithms was presented by Zaki a decade ago in [18].

In the last years, two factors stimulated a renewed interest respectively in distributed and parallel methods for data mining, and in particular for high performance FIM methods. The first is the wide availability of commercial, large-scale, distributed computing facilities, such as the Amazon Elastic Compute Cloud; the second is the increasing number of cores available in microprocessors, for example the recent NVIDIA GF110 microprocessor (the one used in Tesla M2090 and GeForce GTX 580 cards) features 512 core and the SPARC T3 microprocessor sports 16 CPU cores, with 8 hardware threads per core. These innovations open new scalability opportunities on the one hand and demand for additional care to handle their peculiarities on the other. Thus, to effectively exploit these opportunities, there is a need for a new generation of data mining algorithms, in particular able to exploit the General-Purpose computing paradigm on Graphics Processing Units (GP-GPU: http://gpgpu.org). This is due to the radically different architecture design and programming model of GPUs (many-core) with respect to traditional multi-core and multi-processor platforms.

In the case of FIM the optimizations that are conceived with multi-core CPU in mind and for a specific memory hierarchy, such as the ones described in [12], would be of little help in a completely different memory hierarchy and processor architecture. For example, the CUDA framework allows for, and in some cases forces to use, explicit movements of data between slower and faster memories and, at the same time, organizing threads in a hierarchy, posing strong constraints on the order in which data are accessed by concurrent threads. Whenever these constraint are not satisfied, partial serialization of the execution is forced, causing a large part of cores to remain idle. In these conditions it

is not uncommon to observe a GPU processor occupancy falling below 10% (the actual fraction depends on the device in use), thus obtaining a slowdown in the execution of the parallel algorithm.

In this work we present the GP-GPU version of a state-of-the-art FIM algorithm, DCI [15, 13], whose sequential version resulted orders of magnitude better than the well-known Apriori [3]. In addition, for most of the tested datasets, DCI resulted also better than FP-Growth [10], a famous divide&conquer FIM algorithm. While the parallelization of Apriori has been deeply studied [8], the effective parallelization of other more efficient algorithms that use dynamic data structures results much harder. Since DCI uses simple static data structures, and permits a lot of data parallelism (involving bitwise operations) to be exploited, it could be a good candidate for a GPU porting. However, many issues remain to be investigated, mainly concerning (i) the parallelizing strategies to adopt, (ii) the data access patterns, and (iii) the careful management of the GPU memory hierarchy. In this paper we focus on a pair of parallelization strategies. The former uses a simple map-reduce paradigm to realize in parallel collective logical operations between bitmaps with a final bitcount: we call this technique *transaction-wise*, since each bitmap records the presence/absence of a given item in all the transactions of the dataset. The latter adopts a nested data-parallelism, where blocks of (candidate) itemsets are assigned to distinct GPU's multiprocessors for computing their supports (number of occurrence of each candidate itemset in the dataset transactions), where each GPU's multiprocessor in turn adopts a map-reduce paradigm like in the previous approach. We call this strategy *candidate-wise*, and its exploitation affects a crucial feature of DCI: the management of the special cache used to store intermediate results and save work. We conducted several experiments on real-world and synthetic datasets. The GPU porting of DCI gives clear performance advantages over the CPU-based one, and the candidate-wise strategy unquestionably wins over the transaction-wise approach for most of the tested datasets, due to the better multiprocessor occupancy.

The rest of the paper is organized as follows. In Section 2 we describe the CUDA framework for GP-GPU and the DCI algorithm. In Section 3 we analyze the opportunities for parallelization in DCI and describe the implementation of two different strategies. Then, in Section 4 we assess the performances of the proposed methods. Finally, in Section 5, we survey some relevant related work and, in Section 6, we describe some possible extension of the proposed approaches.

2. Background

In this section we give an overview of two topics that are particular important to better understand the following parts of the paper. Specifically, the essentials of the CUDA framework for General Purpose GPU computing (GPGPU), which is used by the algorithm proposed in this paper, and a description of the DCI algorithm for Frequent Itemset Mining, which is the sequential algorithm that inspired our parallel algorithm.

2.1. GPGPU: the CUDA framework

The core of the CUDA parallel kit [1] are three abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization. These abstractions provide a mix of data parallelism and task parallelism at different granularities. Typically a problem is split into independent sub-problems that are mapped to blocks of threads. Each sub-problem is split again and the atomic pieces are assigned to individual threads of the same block that cooperate to solve the sub-problem.

A GPU usually is composed of several multiprocessors, sharing a common device memory (also called global memory). Each multiprocessor consists of several cores (8-32-48 depending on device kind) and a shared memory. Thread blocks are granted to be mapped to a specific multiprocessors, thus the threads in the same block can cooperate by sharing data in the same fast memory and synchronize by means of barriers. On the other hand, there is no kind of warranty on the multiprocessor assignement of different blocks or on their order of execution. Each block is further divided in several warps, that are groups of threads always scheduled at the same time and at the same program address.

2.2. The DCI algorithm

DCI is a multi-strategy algorithm for Frequent Itemset Mining (FIM), characterized by several phases, each exploiting a different strategy. The key idea, common to several other FIM algorithms, is to exploit the apriori principle: all the subsets of a frequent set must be frequent. Thus, the algorithm performs several iterations, starting with on item patterns and increasing the size of searched pattern at each iteration.

During the first phase, DCI adopt an Apriori-like strategy, by scanning the database by transaction, using specific direct-count data structure to update the counters of the itemsets, and operating a strong pruning of the on-disk dataset. This initial phase is named *Direct Count* (*DC*, which is the first part of the DCI acronym).

After the DC phase, when the number of surviving transactions and items does allow the vertical representation of the pruned dataset in memory, DCI switches to the *Intersection* phase (where I is the last part of the DCI acronym). In the vertical representation, for each item we have a *Tidset*, which is composed of the identifiers of all the transactions including the item itself. Indeed, DCI uses a bitwise data structure: each retained frequent item is associated with a *bitmap*, where the bit in the nth position is equal to 1 iif the nth transaction contains the item. The algorithm still processes candidates of increasing length, but now the support of a candidate can be immediately computed by a bitwise logical *And* operation on the bitmaps associated with the single items contained in the pattern, followed by a count of the bits set to 1 in the resulting bitmap (mapreduce).

The adoption in DCI of a suitable data structure for the management of the frequent patterns, featuring an iteration by common prefix in lexicographical order, gives two substantial benefits: an easy and efficient way to generate candidates (already in lexicographical order and, thus, ready to be stored in case they are frequent) by merging frequent patterns that share a common prefix, and a substantial item overlap between consecutive candidates. This last fact allows for the reuse of of intermediate results during the computation of bitwise ands. For example when computing the support of the following four candidate itemsets $\{2, 4, 7, 80\}$, $\{2, 4, 7, 81\}$, $\{2, 4, 7, 82\}$, and $\{2, 4, 82, 90\}$, where each item is represented as an integer, a straightforward approach would need $3 \times 4 = 12$ bitmap intersections (3 bitmap intersections for each 4-itemset), whereas, by caching and reusing the intermediate results, just $3 + 1 + 1 + 2 = 7$ are enough. Indeed, only the first candidate requires the intersection of all of the bitmaps, whereas the following ones have prefix overlaps with the preceding candidates and require a number of intersection which is equal to the number of different items.

3. FIM on GPUs: the gpuDCI algorithm

In this section we will describe gpuDCI, a parallel algorithm inspired by DCI that exploits GPUs to efficiently mine frequent itemsets from transactional datasets. Before discussing in depth the implementation details, in Section 3.5, we first address the opportunities for parallelization that the different phases of the DCI algorithm present and the parallelization strategies that are behind the two versions of the proposed algorithm.

3.1. GPU Parallelization opportunities

As we previously highlighted, DCI is a multi-strategy algorithm characterized by a Direct Count phase and an In-

tersection phase. According to our tests, the Direct Count phase accounts for a limited part of the running time of the algorithm, at least in the most computational intensive cases. Indeed, even if the later described Intersection phase is more efficient than the Direct Count phase, usually it has to deal with a much higher number of candidates and frequent patterns due to combinatorial explosion and, thus, account for the most significant part of the overall running time.
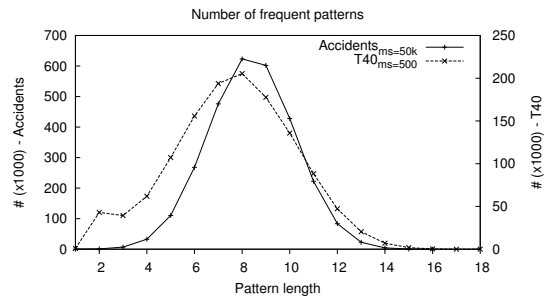

Figure 1. Number of frequent patterns for different pattern length

In support of this intuition, the reader can find in Figure 1 an example of the relation between the number of frequent patterns and their length for two different dataset (see the experimental result section for a description of the datasets). On the other hand, the peculiarities of the count phase would hinder the efficient exploitation of GPUs during this phase. To ensure an high efficiency, the GPU cores in the same GPU multiprocessor has to execute the very same code on data characterized by high spatial locality and accessed according to quite strict memory access pattern. This is hardly the case with a large number of counters. Moreover, if the number of counters is limited, for example during the item count phase for some problem setting, the running time is usually I/O bound and transfering the (still unpruned) dataset to the GPU memory could entail even worst performances.

The intersection phase, instead, accounts for a significant part of the running time, and the computation of the single intersection and set bits count operations require non trivial computational resources. Further, both bitmap intersection and count can be reasonably distributed among the cores of the same GPU and the bitmaps can be accessed in sequential strides, which is optimal to maximize GPU memory bandwidth.

Moving the candidate generation to GPU could give some advantage. However, according to our profiling tests, the candidate generation contribution to the overall running time is just a small fraction of the time due to bitmap intersection and count. Further, performing candidate gener-

ation in GPU would decrease the GPU memory available for bitmap intersection and count, due to the need to maintain frequent patterns of the current and previous iteration in GPU memory, possibly with negative effects on overall performances.

For these reasons, we have decided to focus our parallelization effort on the computation of candidate supports during the Intersection phase.

3.2. An overview of gpuDCI

The basic idea behind gpuDCI is to start computation on CPU, as in DCI, and move the pruned dataset to the GPU as soon as the bitwise vertical dataset fits into the GPU global memory. Afterward the support computation will be delegated to the GPU. However, after switching to GPU, the CPU still manages patterns, generates candidates and store patterns that are frequent according to the support computed by the GPU.

Also the cache used by DCI to store and reuse intermediate intersection results is moved to the GPU. Since the cache is only needed for support computation, and the CPU needs to know just the support of each pattern, this approach will limit the data transfer between system and GPU memories.

There are several conditions that have to be satisfied to ensure optimal GPU usage. Among them, three particularly important ones are related to processor utilization, blocking operations, and memory access patterns. The first is quite obvious, but not always easy to obtain: not only we have to provide enough workload for each core, but also to ensure that every core has the resources needed to execute the assigned computation. The second goal is to minimize the number of synchronizations, in particular operations causing global synchronization such as kernel launches and memory transfers, and other blocking operations. Finally, the last one subsumes several possible memory access optimizations. One of the most important is to ensure coalesced access to global memory by aligning memory accesses to avoid serializations.

In the following subsection we will describe two strategies that resulted suitable, thus producing good speedups, for different kinds of datasets.

3.3. Transaction-wise parallelization

In this approach all the GPU cores, independently of the GPU multiprocessor they belong to, work on the same intersection or count operation. Each thread is in charge of an interleaved portion of the bitmap, in such a way that threads having consecutive indexes work on consecutive parts of the bitmap (see Figure 2 left). Note that our bitmaps are actually vectors of integers (32 or 64 bit, depending on the architecture). The contiguous blocks of 32-64 bits, processed
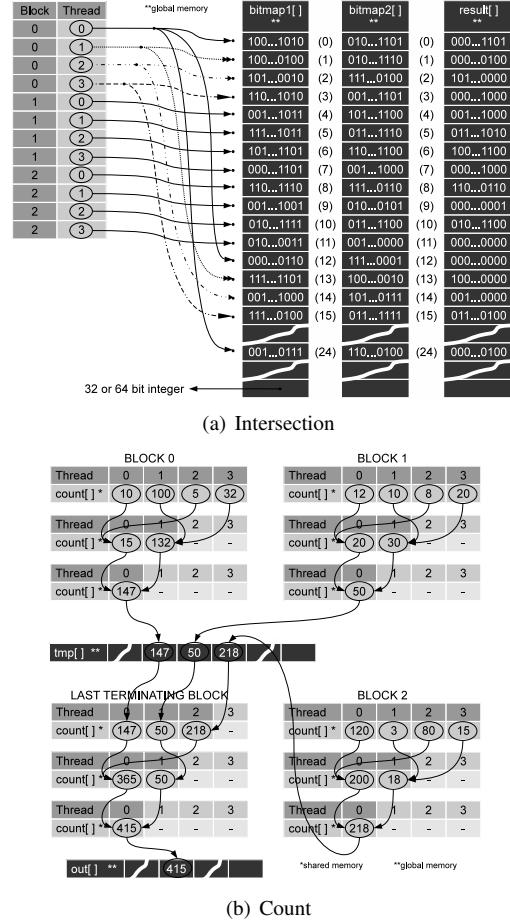

(a) Intersection

(b) Count

Figure 2. Transaction-wise parallelization

by the same thread, are thus separated by a fixed stride of $blocks \times threads$, where $blocks$ and $threads$ are respectively the number of thread blocks and thread per blocks decided at kernel launch time. In the case of intersection (bitwise and), this access pattern applies both to reading the bitmaps and writing the result of the operation. In the case of counting (popcount), this only applies to fetching the bitmaps from global memory, since the result is a single number. In both cases, the global memory accesses to bitmaps are coalesced.

The threads that are involved in the operations on a bitmap are not in the same multiprocessor, and thus do not have access to the same, fast, shared memory. For this reason, the reduction of the count of bits set to 1 (sum of the partial sums computed by each thread) has to be computed in two steps: after each thread has termined the count on its part of the bitmap, the counter of each thread is summed on a per multiprocessor base (local reduction), and then the partial sum for each multiprocessor are added to obtain the final result (global reduction). Both reduction are per-

formed by using a pair-wise, tree based, approach make use of the fast shared memory that is present on each multiprocessor (see Figure 2 right). The key difference, however, is that local reduction starts with data already stored in the shared memory, whereas in the global reduction the multiprocessor that is in charge for the reduction must fetch the counters that are relative to other multiprocessors from the GPU global memory.

To ensure that all cores are involved in the computation, the number of thread blocks must be at least equal to the number of GPU multiprocessors. Further, to ensure that the cores of each multiprocessor are active, global memory access should be overlapped with computation. On NVIDIA devices this usually happens when there are at least 200-300 threads per block (600-1200 for devices with computing capability 2.1). If we consider a top range NVIDIA device having computing capabilities 2.x (GTX580: 16 MP, 512 cores), this method can successfully hides the global memory latency when there are at least $16 \times 1200 \times 64 = 1228800$ transactions in the pruned dataset.

Due to the efficiency of the dataset pruning in DCI, quite often the intersection phase involves a significantly smaller number of transactions. In such a setting, we can chose either to pay the latency for the global memory access, due to an insufficient number of threads per block, or to leave some multiprocessor idle, due to an insufficient number of blocks.

The amount of GPU memory required by this strategy is mainly determined by the size of the pruned dataset ($\#items \times size(bitmap)$) plus the size of the intermediate result cache ($max\_pattern\_length \times size(bitmap)$). In case there is room for more than one cache, we can devise a different parallelization strategy, discussed in the next section, which entails a higher utilization of the cores even when bitmaps that are not huge.

3.4. Candidate-wise parallelization

In this approach each GPU multiprocessor works on the intersection and count operations related to a different candidate, whereas the cores of the same multiprocessor work on the same intersection or count operation, exactly as in the previous approach. At an abstract level, thread blocks are in charge of a block of candidates that are processed one by one; threads are in charge of the intersection or count on an interleaved portion of a bitmap. Note that in this case the threads that are involved in the operations on a bitmap are all in the same multiprocessor and have access to the same, fast, shared memory. The main issue to be tackled to implement this strategy is the management of the cache used to save intermediate intersection results. One of the advantages of the caching method adopted by DCI is its really simple policy. Candidates are examined in lexicographi-
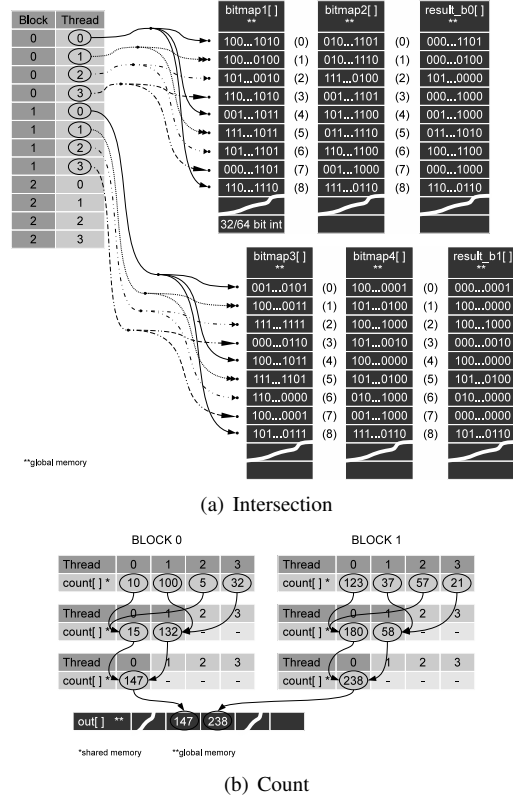

(a) Intersection

(b) Count

Figure 3. Candidate-wise parallelization

cal order, thus we just need a *stack* of intermediate results. When a new candidate is examined, the results that are no longer needed are popped out of the stack; that is, we retain the intermediate results corresponding to the common prefix of the current and last examined candidates.

Processing more than one candidate in parallel inevitably contrasts this simplicity. The solution is to assign collections of candidates to the same thread blocks (and thus multiprocessor), by setting an independent cache per each block. Further, to increase the chance of cache reuse, given the sequence of all lexicographically ordered candidates, we assign a contiguous sub-sequence of candidates to each thread block (see Figure 3).

The amount of GPU memory required is larger than the one required by the previous strategy. Indeed, in this case the size of the intermediate result cache is multiplied by the number of thread blocks used ($thread\_blocks \times max\_pattern\_length \times size(bitmap)$).

3.5. Implementation

In this section we describe the implementation of the two parallelization of the DCI algorithm introduced in the previous section. We will first describe how operations are struc-

tured in batches, then the building blocks that are common to the two approach, and finally how they are combined to obtain the final results.

3.5.1 Batches of operations

Operations on GPU that require memory transfer and kernel launches incur in fixed costs that can be amortized over multiple operations. A typical computation pattern is: move the relevant data to the GPU, invoke the kernel, move the result from the GPU to the CPU. However, for read-only data shared among many operations, it is better to move a large block of memory at once. For these reasons, we move the full dataset to the GPU global memory before computation, execute support count operations in batches and fetch blocks of results at the end of the batch. For efficiency reasons, what is actually passed to the GPU by the CPU is not a set of candidates, but instead a sequence of operation involving the dataset and the intermediate result cache. Such operations could be intersection of two bitmap or intersection and set bit count of two bitmap, either associated to items or to intermediate results. For example, the computation of the support of itemset $\{213, 345, 400\}$ followed by the computation of the support of itemset $\{213, 345, 430\}$ will be transformed into the sequence of commands (where `&` is the bitwise logical And):

- store in `cache[0]` the result of (`bitmap[213]` & `bitmap[345]`)

- store in `output[0]` the number of bits set to 1 in (`cache[0]` & `bitmap[400]`)

- store in `output[1]` the number of bits set to 1 in (`cache[0]` & `bitmap[430]`)

Note the reuse of the cache entry `cache[0]` to compute the itemset count of $\{213, 345, 430\}$.

In transaction-wise parallelization, all the threads execute the same operation at the same time, and that each operation is specified by a few parameters. In our tests, we verified that the kernel launch overhead is negligible wrt the cost of the intersection and count over a large bitmap. Further, one global synchronization is needed after each count operation. Thus, we decided to execute the different operations contained in a batch using one kernel launch for each. The parameters of the operations are specified by kernel parameters and results are fetched in block at the end of the batch.

In candidate-wise parallelism, there is a relevant difference: the threads belonging to distinct blocks are involved in different operations, hence there is an increased number of parameters to be specified. To cope with this issue, we decided to store the parameters of all the operations of a batch in a command buffer data structure, which is moved

---

Transaction-wise

```
i = blockIdx * threadsPerBlock + threadIdx
while (i < bitmapSize)
    out[i] = bmp1[i] & bmp2[i]
    i = i + threadsPerBlock*numBlocks
```

Candidate-wise

```
i = threadIdx
while (i < bitmapSize)
    out[i] = bmp1[i] & bmp2[i]
    i = i + threadsPerBlock
```

Figure 4. Intersection (kernel executed by each thread)

to the GPU "constant memory" when the processing of the batch starts, and is then repeatedly accessed by the kernel to determine the next operation to be performed. It is worth remarking that the constant memory is cached, thus accesses to the same location by several threads – e.g., all the threads working on the same operation – are particularly efficient. The batch is executed in several step, and in each step all the blocks are expected to execute one basic operation (intersection, or intersection and count). In an effort to maximize constant cache hits, we stored in close command buffer positions the parameters of the operations that are expected to be executed in the same step. Since each multiprocessor has its own constant cache, this approach gives benefits only when the kernel is launched with a number of thread blocks larger than the number of GPU multiprocessors (e.g., to hide latencies when the number of transactions is small).

3.5.2 Building blocks: basic operations on GPU

The pseudocode in Figure 4 describes the intersection operations, i.e. the most frequent one in DCI, carrier by the GPU threads. We highlight that the main difference between the implementations in the two approaches is the stride: in the Transaction-wise parallelization case it is determined by the block index and the thread index, whereas in the candidate-wise parallelization case it is determined only by the thread index, since different thread blocks are operating on different bitmaps, and `bmp1`, `bmp2`, and `out` refer to different GPU memory zones. It is worth recalling that such bitmaps are actually stored as vectors of `int` (32 or 64 bit, depending on the architecture), and thus the & operations actually perform 32 or 64 logical Ands.

The count of the cardinality of a Tidset intersection is the second most frequent operation in DCI, and is illustrated in Figure 5. In practice, due to the bitwise data representations, this corresponds to the count of the number of 1-bits in a bitmap. This bitwise operation is also known as `popcount` and has an hardware implementation on several modern hardware, including NVIDIA GPUs.

Transaction-wise

```
i = blockIdx * threadsPerBlock + threadIdx
count[threadIdx] = 0
while (i < bmpSize)
   count[threadIdx] += popcount(bmp1[i] & bmp2[i])
   i = i + threadsPerBlock*numBlocks

// reduce the count in shared memory
// and store the temporary result to global memory
blockCount[blockIdx] = localReduce(count)
// the last finished block loads temporary results
// to shared memory, reduce the count and stores
// the result
out = globalReduce(blockCount)
```

Candidate-wise

```
i = threadIdx
count[threadIdx] = 0
while (i < bmpSize)
   count[threadIdx] += popcount(bmp1[i] & bmp2[i])
   i = i + threadsPerBlock
// reduce the count in shared memory
// and store the result
out = localReduce(localCount)
```

Figure 5. Cardinality count (kernel executed by each thread)

Here we omit the implementation of the reductions operations (see `globalReduce()` and `localReduce()` in Figure 5). Refer to nvidia whitepapers (`http://www.nvidia.com/object/cuda_sample_data-parallel.html`) for the implementation and optimization details.

4. Performance evaluation

To assess the performance of the proposed methods and the convenience in exploiting GPU for frequent itemset mining with respect to traditional CPU only algorithms we run several tests. In the following subsections we will describe the test environment, the data used in our tests, and finally we will describe the experiments and the results obtained.

4.1. Test environment and datasets

The experiments where executed on a server equipped with an Intel Core2 Quad CPU @ 2.66GHz, 8 GB of RAM, and a NVIDIA GTX275 GPU featuring: 30 multiprocessors (240 cores) @ 1.4 GHz, 896MB device memory, and Cuda device capability 1.3.

In our experiments we used both real world datasets and generated ones, largely used in the FIM research community (`http://fimi.ua.ac.be/data/`).

Accidents A real world dataset provided by Karolien Geurts [6]. The data are related to 340.184 traffic ac-

cidents, each associated to a set of attributes selected among a set of 572 possible attributes (45 attributes per accident on average).

Kosarak Another real world dataset provided by Ferenc Bodon obtained from the click-stream of an on-line news portal. There are a total of 990k transactions, each containing an average of 8 items.

Pumsb and pumsb-star The Pumsb dataset contains census data. There are 49,046 records with 2,113 different items. Pumsb-star is the same dataset as Pumsb except all items of 80% support or more have been removed, making it less dense.

T40I10D100K A syntethical dataset used for the FIMI 03/04 competition, generated using the generator from the IBM Almaden Quest research group. There are 100k transactions with an average of 40 items per transaction selected among 10000 items.

T10I1D500k-12M A family of generated datasets obtained using the same IBM generator. We used this non-standard datasets with carefully tuned parameters to have datasets with a large number of transactions that survive the pruning of the DCI algorithm. The transactions contain, on average, 10 items per transaction selected among 1000 items. The number of transaction per dataset is in the range [500k, 12M].

4.2. Experimental results

The goal of the following tests is to assess the performance of the two parallelization strategies devised for gpuDCI, with respect to the execution of the sequential DCI algorithm running on the CPU. In the following these parallelization strategies are indicated as gpuDCI$_{TW}$ (transaction-wise parallelization, single global cache), and gpuDCI$_{CW}$ (candidate-wise parallelization, multiple block-based caches).

Different dataset. Our first test compares the running time of gpuDCI on the different datasets. The results of this experiment are reported in Figure 6: each group of bars show the execution times for each dataset.Close to the name of the datasets, we indicate the minimum support parameter used in each test (e.g.: ms=50k means that minimum support was set to 50,000 transactions).

We observe that there is a clear advantage for gpuDCI$_{CW}$ in most cases. On the other hand, the gpuDCI$_{TW}$ strategy is the worst choice in all the cases considered in this test. We will see later, however, that there are some particular conditions in which gpuDCI$_{TW}$ achieves better performances. A closer look at the case of dataset *kosarak*, in which
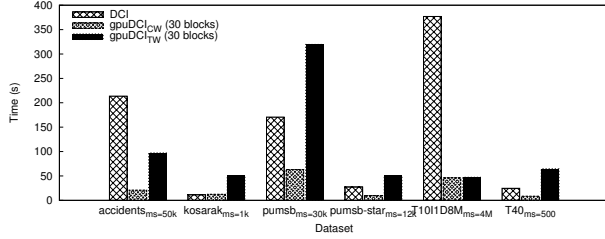
Figure 6. Running time for different datasets

Figure 7. Running time for different pattern length on two different datasets

gpuDCI$_{CW}$ and CPU perform similarly highlights that the comparison is biased in favor of the CPU algorithm. This is due to the fact that DCI checks for pruning opportunities also during the Intersection phase, whereas the current gpuDCI implementation use pruning just during the Direct Count phase. In the test case, after the 3 iteration (i.e., for patterns with more than 4 items) the dataset used by CPU is one tenth in size of the one used by GPU thanks to the more aggressive pruning strategy. If we limit our observation to the 3rd iteration, in which the datasets are of the same size, gpuDCI$_{CW}$ is more 6 times faster. The same bias in favour of CPU is present in other tests, however its effects are less evident.

Finally, we observe that in the case of the T10I1D8M dataset the two parallelization strategies perform almost similarly, and significantly better than the CPU version of the algorithm. Indeed, that dataset was built with the specific goal of involving all the cores in the computation.

Pattern length. In our second experiment we examined the running time of the single iterations of the algorithms, where each iteration produce all the frequent patterns of a given length. We focus on two dataset: Accidents and T40. The first is quite dense and a large number of frequent patterns and candidates is found, even for relatively high minimum supports (15% in this test). Figure 7 presents the results of this test. The chart plots the running time as a function of the iteration number (pattern length), starting from the third iteration, since the first two iteration are identical for all algorithms. In the same chart, also the number of candidates of the different lengths is reported (the corresponding scale is on the right ordinate axis).

We observe that gpuDCI$_{CW}$ has an advantage of nearly one order of magnitude wrt the CPU algorithm for all pattern lengths, and that the running time is roughly proportional to the number of examined candidates.

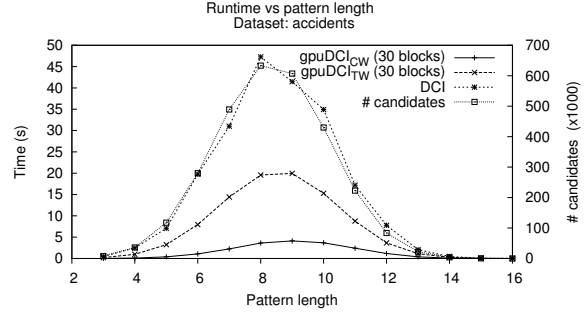**Dataset size.** In this experiment we compared the running time of the three algorithms on a homogeneous family of datasets of increasing size obtained by sampling the largest dataset. This one was built with the goal of having a large set of transactions surviving the pruning. In Figure 8(a) we can observe that the three algorithms running times are linear with respect to the dataset sizes. Further, comparing gpuDCI$_{TW}$ and gpuDCI$_{CW}$, we observe that the candidate-wise approach is not able to deal with the largest dataset. Indeed, in this case (10^7 transactions) gpuDCI$_{CW}$ needed to exploit many caches of larger size. Since the size of each cache is determined by the number of transactions and the size of the patterns under consideration, the device memory is not sufficient to host the dataset, long patterns and multiple cache instances.

Finally, we point out that gpuDCI$_{TW}$ and gpuDCI$_{CW}$ exhibit similar performances when the multiprocessors are completely scheduled. In some of our tests on different devices, we also reported marginally faster running times for gpuDCI$_{TW}$. Note, however, that these advantages are not relevant, and the driving factor for choosing the approach to use should be just the amount of available memory.

Number of multiprocessors. In the last test we evaluated the speed-up of the two parallelization methods as the number of used multiprocessors increases. Note that it is not possible to directly limit the number of multiprocessors used to run gpuDCI. However, since the threads in the same CUDA block are executed by the same multiprocessor, we can indirectly limit the number of multiprocessors used by reducing the number of thread blocks.

Figure 8(b) shows the running time of gpuDCI$_{CW}$ and gpuDCI$_{TW}$. We observe that the candidate-wise running time continues to decrease as the number of candidates processed in parallel (thread blocks) is increased, until the number of thread blocks becomes equal to the number of multiprocessors. This indi-
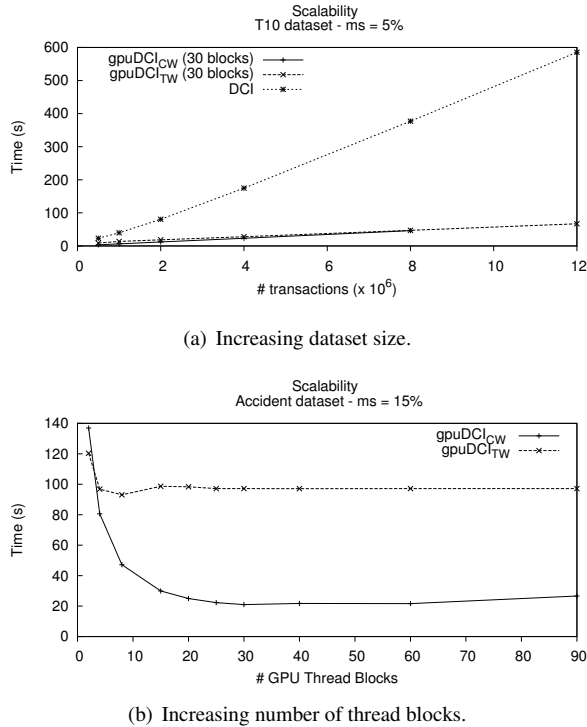
(a) Increasing dataset size.

(b) Increasing number of thread blocks.

Figure 8. Scalability

cates that multiprocessors are not under scheduled due to memory access latency, otherwise using a number of blocks larger than the number of multiprocessors would further reduce the running time.

The transaction wise approach, instead, benefits from the additional multiprocessors only up to 8 thread blocks, then the number of transactions is not sufficient to maintain busy all the multiprocessors.

5. Related works

GP-GPU for FIM was for the first time addressed in [5], where the authors presented a GPU-based implementation of the well-known Apriori algorithm. In their proposal, the dataset is represented as a binary matrix D, where $D_{t,i}$ is 1 *iff* the item i occurs in transaction t. Calculating the transactions that support a given item set just requires to intersect rows of the matrix D. 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. Nonetheless, this work presents several limits, in part inherited by the Apriori algorithm and in part by implementation choices. For example, the generation of the full set of candidates of a given length before computing their supports on GPU strongly limits its applicability due to memory usage issues.

Similar effects are caused by not pruning the dataset before switching to a bitmap representation. This cause a lot of unneeded computations due to useless data stored in the bitmap. As an example consider the size of the bitmap for the Retail dataset that, according to the authors, is 180 MB, whereas in more optimized bitmap based algorithms such as DCI [15], it depends on the minimum support threshold. For example, for the Retail dataset, 11MB at 1% minimum support and 74MB at 0.1% minimum support, after the pruning phase of DCI. Further, the use of a large size lookup table (64KB stored in constant memory, when the size of the constant cache is just 8KB) for the support count causes high memory read latency in 7 accesses over 8. Not surprisingly, the authors report that the proposed approach performs significantly worst than a state of the art serial algorithm running on CPU.

Another Apriori based FIM algorithm for GPU is presented in [19]. Contrary to [12], it uses inverted lists (stored as arrays) instead of bitmaps to represent the Tidsets associated with items. Similarly to [12], the work suffers from the choice of Apriori, a low performance algorithm, as baseline.

A different approach is used in [17], based on the TreeProjection algorithm described in [2]. This work represent a significant improvement with respect to the parallelizations of the Apriori algorithm. Nonetheless, TreeProjection is not a state of the art algorithm for FIM, as it is outperformed by FPGrowth [10] that, in turn, is slower than or comparable to DCI in most cases [7].

Finally, [11] presents a GPU implementation of the well known MAFIA [4] Maximal FIM algorithm that features significant performance gain with respect to its CPU version. We observe, however, that this work address a quite different problem, aiming to extract only the maximal frequent itemsets that are not set-included in any other frequent itemsets. Maximal frequent itemsets are less informative than frequent itemsets and the exact support of a large part of frequent itemsets can not be inferred from the maximal frequent itemsets.

6. Conclusions and future works

In this paper we introduced a parallel algorithm, gpuDCI, which exploits GPUs to compute frequent itemsets, that is to find the subsets of items that are contained in at least a given fraction of a collection of transactions (i.e., sets of items). We presented the rationale behind our design choices, focusing in particular on what to parallelize, and devised two parallelization strategies. Our experiments showed that, in general, using the GPU for computing the support of candidate patterns, gives clear advantages. Further, the candidate-wise approach unquestionably wins over the transaction-wise approach, with a tie in the cases in which there is a sufficiently large number of

transactions to allow full multiprocessor occupancy in both cases. The adoption of transaction-wise approach is advisable only when the other approach is not suitable due to an unusual large memory occupancy for storing caches to store and reuse intermediate results.

In the future we plan to improve the gpuDCI algorithm in several directions. The most recent version of DCI does not take advantage of the presence of additional CPU or multi-core CPU. We plan to extend DCI in order to exploit these additional resources, using a parallelization approach similar to the one we adopted for GPU. The same computer system can host more than one GPU. The algorithm we proposed in this paper makes use of a single GPU. However, the candidate-wise approach could be easily extended to a larger number of GPUs.

In some case it is possible to avoid the computation of the support of a pattern by making inferences on some property of its subsets [13]. This approach is particularly effective for dense datasets and can be applied to gpuDCI. Finally, frequent closed itemsets are a condensed representation of frequent itemsets that can be directly computed from the data. We plan to improve the efficiency of the DCI-based algorithm for extracting these closed patterns [14] by moving part of the computation to the GPU.

References

[1] NVIDIA CUDA Compute Unified Device Architecture. Progr. Guide. V2.0. Technical report, 2008.

[2] Ramesh C. Agarwal, Charu C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *J. Parallel Distrib. Comput.*, 61(3), March 2001.

[3] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, 1994.

[4] Douglas Burdick, Manuel Calimlim, Jason Flannick, Johannes Gehrke, and Tomi Yiu. Mafia: A maximal frequent itemset algorithm. *IEEE Trans. Knowl. Data Eng.*, 17(11):1490–1504, 2005.

[5] Wenbin Fang, Mian Lu, Xiangye Xiao, Bingsheng He, and Qiong Luo. Frequent itemset mining on graphics processors. In *Proc. 5th Int. Workshop on Data Management on New Hardware*, DaMoN 2009, 2009.

[6] Karolien Geurts, Geert Wets, Tom Brijs, and Koen Vanhoof. Profiling high frequency accident locations using association rules. In *Proc. 82nd Annual Transportation Research Board,Washington DC. (USA), January 12-16*, 2003.

[7] Bart Goethals and Mohammed J. Zaki. Advances in frequent itemset mining implementations: report on fimi'03. *SIGKDD Explor. Newsl.*, 6(1), June 2004.

[8] Eui-Hong Han, George Karypis, and Vipin Kumar. Scalable parallel data mining for association rules. *IEEE Trans. Knowl. Data Eng.*, 12(3), 2000.

[9] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. Frequent pattern mining: current status and future directions. *Data Min. Knowl. Discov.*, 15(1), August 2007.

[10] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Min. Knowl. Discov.*, 8(1), 2004.

[11] Haifeng Li and Ning Zhang. Mining maximal frequent itemsets on graphics processors. In Maozhen Li, Qilian Liang, Lipo Wang, and Yibin Song, editors, *FSKD*, pages 1461–1464. IEEE, 2010.

[12] Li Liu, Eric Li, Yimin Zhang, and Zhizhong Tang. Optimization of frequent itemset mining on multiple-core processor. In *Proc. 33rd int. conf. on Very large data bases*, VLDB '07. VLDB Endowment, 2007.

[13] Claudio Lucchese, Salvatore Orlando, and Raffaele Perego. kdci: a multi-strategy algorithm for mining frequent sets. In *FIMI Workshop*, 2003.

[14] Claudio Lucchese, Salvatore Orlando, and Raffaele Perego. Fast and memory efficient mining of frequent closed itemsets. *IEEE Trans. Knowl. Data Eng.*, 18(1), 2006.

[15] Salvatore Orlando, Paolo Palmerini, Raffaele Perego, and Fabrizio Silvestri. Adaptive and resource-aware mining of frequent sets. In *IEEE ICDM*, pages 338–345, 2002.

[16] A. Swami R. Agrawal T. Imielinski. Mining association rules between sets of items in large databases. In *SIGMOD*, 2003.

[17] George Teodoro, Nathan Mariano, Wagner Meira Jr., and Renato Ferreira. Tree projection-based frequent itemset mining on multicore cpus and gpus. In *Proc. 22nd Int. Symp. on Computer Architecture and HPC*, SBAC-PAD '10, Washington, DC, USA, 2010. IEEE Computer Society.

[18] Mohammed J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4), 1999.

[19] Jiayi Zhou, Kun-Ming Yu, and Bin-Chang Wu. Parallel frequent patterns mining algorithm on gpu. In *SMC*. IEEE, 2010.