Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 86 – 95

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

Frequent Items Mining Acceleration Exploiting Fast Parallel Sorting on the GPU

Ugo Erra[a], Bernardino Frola[b]

[a]*Dipartimento di Matematica e Informatica, Università della Basilicata, Viale Dell'Ateneo, Macchia Romana, 85100, Potenza, Italy*
[b]*Dipartimento di Informatica, Università di Salerno, Via Ponte don Melillo, 84084, Fisciano(SA), Italy*

Abstract

In this paper, we show how to employ Graphics Processing Units (GPUs) to provide an efficient and high-performance solution for finding frequent items in data streams. We discuss several design alternatives and present an implementation that exploits the great capability of graphics processors in parallel sorting. We provide an exhaustive evaluation of performances, quality results and several design trade-offs. On an off-the-shelf GPU, the fastest of our implementations can process over 200 million items per second, which is better than the best known solution based on Field Programmable Gate Arrays (FPGAs) and CPUs. Moreover, in previous approaches, performances are directly related to the skewness of the input data distribution, while in our approach, the high throughput is independent from this factor.

Keywords: Frequent Items, Data Stream Mining, GPU.

1. Introduction

Data-intensive science consists of the analysis of scientific vast volumes of data captured by instruments or generated by simulations. Due to the amount of data and the rate at which they are generated several problems occur in processing such information named data streams. In general, there is not enough space to store all the data streams for online processing and also there is not enough time to rescan the whole dataset or perform a rescan whenever an update occurs. Streaming data processing is a new computing paradigm that tackles the problems occur in data streams and then goes beyond the traditional store and process approach. In streaming data processing, we have two interesting aspects: the first one is parallelism, which enables the same function to be applied to all records of an input stream simultaneously without waiting for results. The second one is locality, which means that data is produced, consumed, and never used again. This paradigm has received considerable attention in the recent years thanks also new programmable processors such as Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) which enable more easily to exploit the characteristics of this paradigm by using low cost parallel architectures.

Finding frequent items in data streams is an important problem that has attracted significant attention in research. Informally, the problem is simply to find those items which occur most frequently in a given data stream and in particular those that exceed a specified percentage of the total number. Despite its simplicity, the problem has numerous

Email addresses: `ugo.erra@unibas.it` (Ugo Erra), `frola@dia.unisa.it` (Bernardino Frola)

doi:10.1016/j.procs.2012.04.010

practical important applications. In networking several applications need to monitor the frequency of occurrence of packets come from the network [1]. In marketing businesses are interested in identify the most selling products in order to launch special promotions [2]. Recently, items are used to model queries made to an Internet search engine, and frequent items approach is used to count the popular terms [3].

In this paper, we tackle the calculation of frequent items in a data stream, and show how it can be implemented using GPUs. We achieve throughput rates up to 207 million items per second, a rate 2.6 times higher than the best FPGA acceleration [4], from 3.8 to 9.5 times higher than the best CPU published implementation [5] and 1.5 times higher than the best published result on parallel CPU [6]. Our paper discusses a sort-based approach to solve the frequent items problem on GPU, and illustrates the design considerations that we faced. We also give guidance on how to find the right balance between resource availability and performance using our approach. As a main reference for the quality of existing solutions, we use the comprehensive study of the frequent items mining by Cormode and Hadjieleftheriou [5].

The rest of the paper is organized as follows. The upcoming Section 2 introduces formally the frequent items problem, and illustrates the Frequent and the SpaceSaving algorithms. Section 3 discusses two straightforward solutions to compute frequent items on GPU. Section 4 describes our efficient approach based on sorting. Section 5 reports results of experiments and assesses resource and performances trade-offs. Section 6 relates similar works on mining frequent items in parallel. Finally, Section 7 presents our conclusions and future directions.

2. Background and Related Works

2.1. The Frequent Items Problem

Given a stream of n items $t_1 \dots t_n$, the frequency of an item i is $f_i = |\{t_j = i\}|$. The *exact ϕ-frequent* items comprise the set $\{i | f_i > \phi n\}$, where the parameter ϕ is called *frequency threshold*. As an example, given a stream $\{w, x, w, u, y, w, x, u\}$, we have $f_w = 3$, $f_x = 2$, $f_y = 1$, and $f_u = 2$. If we set $\phi = 0.2$, the exact ϕ-frequent items are w, x, and u. Since the frequent items problem require a space proportional to the length of the stream [5], an approximate version is defined based on a tolerance for an error ϵ. The ϵ-approximate problem returns a set of F items so that $\forall i \in F, f_i > (\phi - \epsilon)n$ and there is no $i \notin F$ such that $f_i > \phi n$. As consequence, this version allows false positives but no false negative.

2.2. Counter-based Algorithms

Counter-based algorithms track a subset of items from the inputs, and monitor counts associated with these items. For each new arrival, the algorithms decide whether to store this item or not, and if so, what counts to associate with it. The Majority[7] represents the first counter-based algorithm to the frequent items problem. In this algorithm, if the same item occurs in the stream the counter is incremented by 1, while if a new item occurs, the counter is decremented by 1. Each time the counter is zero, and a new item arrive the counter is set to 1 and the new item is stored. At the end of the stream the stored item is the majority item.

Figure 1 lists two counter-based algorithms that we used as a baseline for our work: the Frequent [5, 8] algorithm, which includes essentially the same generalization of the Majority algorithm to solve the problem, and the SpaceSaving [5, 9] algorithm. Given a data stream of n items, a set T stores $k - 1$ (item, counter) pairs in Frequent and k (item, counter) pairs in SpaceSaving while processing all items. Setting $k = 1/\epsilon$ ensures that the error in any approximate count is at most ϵn. At runtime, a new item is compared against the stored items T. If the item exists, the corresponding counter is incremented by 1. Otherwise, the new item is allocated and the corresponding counter is set to 1. If all counters are allocated the two algorithms follow two different strategies. In Frequent, all counters are decremented by 1, while in SpaceSaving, the (item, count) pair with the smallest count has its item value replaced with the new item, and the counter incremented.

2.3. GPU and Parallel Sorting

We briefly review the salient details of NVIDIAs current GPU architecture[10] with its parallel programming model CUDA[11] and how sorting on the GPUs is highly competitive with CPU implementations. Modern NVIDIA GPUs are fully programmable many-core chips called CUDA processors. In detail, the GPU consists of an array of streaming multiprocessors (SM), each of which contains 32 CUDA processors. The number of SMs ranges from 1

Algorithm 1: FREQUENT(k)

```
T ⟵ ∅;
n ⟵ 0;
foreach i do
    n ⟵ n + 1;
    if i ∈ T then c_i ⟵ c_i + 1;
    else if |T| < k − 1 then
        T ⟵ T ∪ {i};
        c_i ⟵ 1;
    else forall the j ∈ T do
        c_j ⟵ c_j − 1;
        if c_j = 0 then T ⟵ T \ {j};
```

Algorithm 2: SPACESAVING(k)

```
T ⟵ ∅, n ⟵ 0;
foreach i do
    n ⟵ n + 1;
    if i ∈ T then c_i ⟵ c_i + 1;
    else if |T| < k then
        T ⟵ T ∪ {i};
        c_i ⟵ 1;
    else
        j ⟵ arg min_{j∈T} c_j;
        c_i ⟵ c_j + 1;
        T ⟵ T ∪ {i} \ {j};
```

Figure 1: Pseudocode for counter-based algorithms FREQUENT and SPACESAVING.

SM at the low end to 16 SMs at the high end. Each SM is capable of supporting up to 1536 threads. Then, current NVDIA GPUs managed up to 24576 threads in realtime. All thread management, including creation, scheduling, and barrier synchronization is performed entirely in hardware by the SM with essentially zero overhead.

From the point of view of software model, CUDA provides the means for developers to execute parallel programs on the GPU. In CUDA a program called kernel executes in parallel across a set of parallel threads. The programmer organizes these threads into a hierarchy of thread blocks and grids. A thread block is a set of concurrent threads that can cooperate among themselves through barrier synchronization and have access to the shared memory with latency comparable to registers. The grid is a set of thread blocks that may each be executed independently. All threads have access to the same global or constant memory. Each thread block is mapped to one SM and are executed concurrently. SM resources (registers and shared memory) are split among the mapped thread block. As a consequence, this limits the number of thread blocks that can be mapped onto the same SM.

Since the introduction of programmable GPUs and NVIDIAs CUDA framework, many sorting algorithms have been successfully implemented on the GPU in order to exploit its computational power. Before the advent of scatter functionality and local stores on GPUs, bitonic[12] or similar sorting networks[13] were well suited for GPU implementations but capable of achieve only non-optimal time complexity $O(n \log^2 n)$. After improvements in GPU technology, other comparison sorts with lower algorithmic complexity of $O(n \log n)$ such as merge sort and radix sort have become viable. Actually, radix sorting is currently the fastest approach for sorting 32- and 64-bit keys on both CPU and GPU processors [14]. The radix sort is based on a positional representation for keys where each key is an ordered sequence of digits. For a given input sequence of keys, this method produces a lexicographic ordering of those keys iterating over the digit-places from least-significant to most significant. Given an n-element sequence the entire radix sorting process its algorithmic complexity is $O(n)$. In [15], authors demonstrate a radix sorting approach which is capable to exceed 1 billion 32-bit keys/sec on a single GPU microprocessor. This approach has been incorporated into the Thrust Library [16] used in our work.

3. Straightforward Approaches

A GPU implementation of the counter-based algorithms described in this paper requires to break down the input stream into *strips*. Each strip is buffered, copied to the GPU memory and processed in parallel by one or more kernels. In order to minimize the overhead due to memory transfers and exploit the computational resources of the GPU, the size of the strip is usually required to be at least of the order of thousands of items. Before starting the design of a sort-based algorithm, we considered different straightforward approaches to parallelize the algorithms described before.

Our first idea was to launch a GPU thread for each item of the strip, each thread running the FREQUENT or SPACESAVING algorithm. We found several difficulties to adapt any of the available CPU implementations of these algorithms to

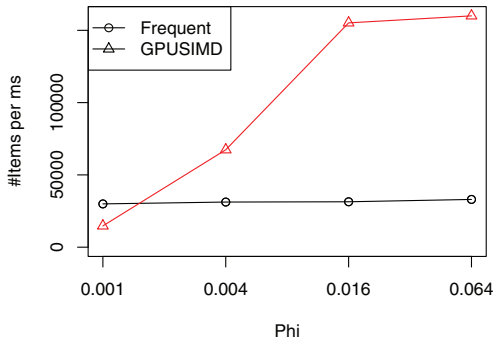


Figure 2: Preliminary results from our first attempt to parallelize Frequent. This plot shows the speed in terms of the number of items/ms vs. frequency threshold ϕ, and compares the most efficient implementation of Frequent with our GPU-based SIMD implementation of Frequent (GPUSIMD). The throughput of GPUSIMD is deeply influenced by ϕ, that is the inverse of the summary size k. This implementation limits k to the maximum size of the CUDA thread block. Test configuration is described in detail in Section 5.

the GPU processing model. Based on multiple linked-lists [8, 9] or hashing [17], these implementations are extremely efficient, and a high effort in designing new data structures and applying low level optimizations is required to create an efficient GPU counterpart. Even with a very fast GPU-based implementation, the problem of merging the outputs generated by threads still remains. In [6], this problem is solved for multiple outputs of the Frequent algorithm. The time required to merge outputs increases linearly with $k \log p$, where p is the number of outputs. This is an unpractical solution in our case, where there are thousands of outputs to merge.

Another approach we considered was to associate a thread to each item of the output T, and at the same time process several sub-strips concurrently. We developed a SIMD (Single Instruction, Multiple Data) algorithm resembling the behavior of Frequent, that executes in parallel by k threads the `forall` block of the serial algorithm [Figure 1 (left)]. This schema allowed us to develop a very simple algorithm using simple data structures, and reduce the number of outputs to the order of hundreds without reducing the number of concurrent threads. However, this solution has two strong limitations. First, we used the CUDA shared memory to allow communication among threads. Thus, the maximum number of threads that can communicate, that is equal to the maximum value of k, is limited by he maximum size of the CUDA thread block[1]. The second limitation is related to the scalability of performances. Figure 2 shows the throughput of the SIMD algorithm compared to Frequent as a function of increasing frequency threshold (we set $\phi = \epsilon = 1/k$, as explained in Section 5). Results of this test showed that with a small value of ϕ, the throughput of the SIMD algorithm is lower than the serial version of Frequent. The number of concurrent threads per block is exactly $1/\phi$, and performances decrease as the number of threads increases because of the synchronization overhead and the serialization of diverging paths.

4. Sort-Based Approach

After we declared unfeasible our first straightforward solutions, we focused our attention on sorting, and decided to design a new solution. In this section, we introduce NaiveSB, a naive sort-based algorithm for frequent items mining, and AccurateSB, an high accuracy algorithm that solves the problems of the naive version.

4.1. Naive Sort-Based Algorithm

Given a data stream of n items, a set B stores $b = k + s$ (item, counter) pairs, while processing all strips of size s. Setting $k = 1/\epsilon$ ensures that the error in any approximate count is at most ϵn. The output T is represented by

[1]The maximum block size is 1024 on most recent GPU architectures

Algorithm 3: NaiveSB(k, s)

```
B ⟵ ∅, n ⟵ 0;
foreach strip S do
    n ⟵ n + s;
    B_k, B_{k+1}, ..., B_{k+s} ⟵ S;
    Sort (B.items);
    Reduce (B.items);
    Sort (B.counters);
```

Algorithm 4: AccurateSB(k, s)

```
B ⟵ ∅, n ⟵ 0;
foreach strip S do
    n ⟵ n + s;
    B_k, B_{k+1}, ..., B_{k+s} ⟵ S;
    m ⟵ c_{k-1};
    for j ← 0 to k − 1 do
        c_j ⟵ c_j − m;
    Sort (B.items);
    Reduce (B.items);
    Sort (B.counters);
    for j ← 0 to k − 1 do
        c_j ⟵ c_j + m;
```

Figure 3: Pseudocode for NaiveSB and AccurateSB.

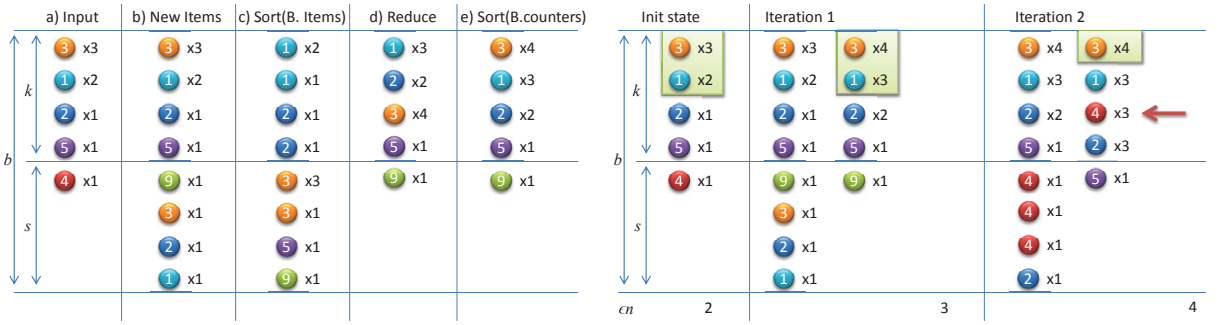


Figure 4: On the left. Single iteration of NaiveSB showing the state of B at each instruction. The strip size s is 4, and k is 4 ($\epsilon = 0.25$). This figure shows items inside circles and associated counters. a) Initial state of B. b) Items of the incoming strip of data substitutes the last 4 elements of B. c) The first `Sort` operation gathers repeated items in groups. d) The `Reduce` operation deletes repetitions and sums the counters. e) The second `Sort` operation moves the most frequent items to the top of the list. On the right. Two iterations of NaiveSB. Each iteration shows the state of B after new items substituted items in the second part of B (on the left) and after items have been sorted by counters (right). An item is frequent if its counter is greater or equal to ϵn (bottom of the figure). A rectangle highlights frequent items at each iteration. At iteration 2, there is a burst of items 4. Since item 4 was substituted by item 9 at iteration 1 (because it was in the second part of B), the counter of item 4 is equal to 3 while the real frequency is 4, and thus NaiveSB does not report it as frequent item.

the first k pairs of B. The NaiveSB algorithm is listed in Figure 3 (left). B is filled with items of S starting from the k-th position, substituting old items of B in positions between k and $k + s$ and keeping untouched those in the first k positions. Counters of new items are set to 1. The first `Sort` operation gathers repetitions in B in groups of consecutive items. The `Reduce` operation compacts repetitions and removes all but the first item of each group. The counter of the fist item of each group is set to the sum of counters associated with the items of the group. The second `Sort` operation moves most frequent items (those with highest values of counters) to initial positions of B. This allows to identify frequent items and makes B ready for the next strip of data. Only items of T with values of counters equal or greater than the maximum approximation error, ϵn, are reported as frequent. Figure 4 (left) shows an example of execution of this algorithm.

The main limitation of NaiveSB is that it does not ensure to correctly report all frequent items. Items in B whose positions start from the k-th are simply substituted by items of the new strip of data. As a consequence, non-frequent items that become suddenly frequent could not be recognized as frequent because, in past iterations, they were substituted by other items and their counters reset. Figure 4 (right) shows an example in which this problem causes a wrong output.

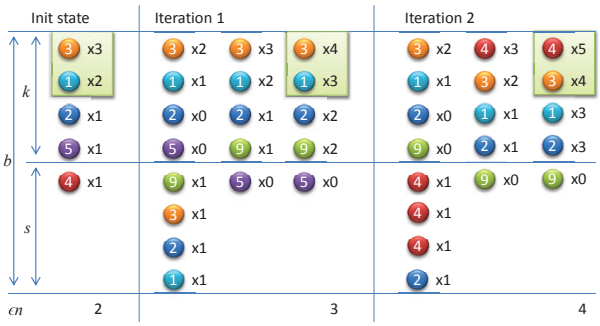


Figure 5: Two iterations of AccurateSB. Each iteration shows the state of B after new items substituted the second part of B and counters are decremented by m (on the left), after items are sorted by counters (middle), and after counters are incremented by m (right). A rectangle highlights frequent items at each iteration. Differently from NaiveSB, AccurateSB gives an advantage to new entries. The counter of item 4 is set to 5, that is the sum of 3 (number of occurrences at iteration 2) and $m = 2$ (the minimum value among counters of the previous iteration). The estimation of frequency of item 4 is bigger than ϵn, thus is reported as frequent.

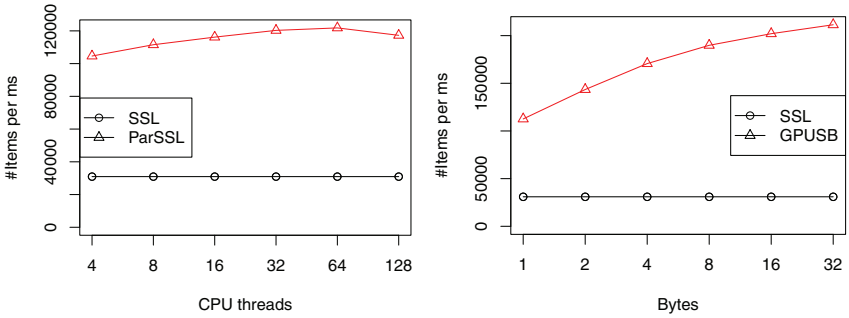
4.2. Accurate Sort-Based Algorithm

NaiveSB produces incorrect results because of the underestimation of frequencies of new items. Our solution to this problem is inspired by the SpaceSaving algorithm [9]. The frequency of new items is overestimated to the minimum value m among all counters of items in the first k positions, i. e. those are not substituted by incoming new items. Each new item could have occurred in the past between 0 and m times. This is true because if one of them occurred more than m times, then it is placed by the second `Sort` operation in a position such that is not substituted by new incoming items, and thus this cannot be a new item. We do not know the exact number of occurrences in range $[0, m]$, thus we overestimate the frequency by choosing the maximum value m. By overestimating frequencies, real frequent items satisfy the condition $counter > \epsilon n$. As we show in our tests, the error on frequency estimation is negligible. The algorithm is illustrated in Figure 3 (right). At the end of each iteration items in B are sorted by counters in a descending order, thus m is equal the counter associated with the item at position $k-1$. The two `for` loops allow to both (i) keep intact the frequency of items already in T, whose counters are decremented and incremented by the same value, and (ii) increase counters of new items by m. Figure 5 shows an example of the execution of two iterations of AccurateSB using the same input of the example shown in Figure 4 (right).

5. Benchmarks and Results

We analyzed performances and quality results of three different implementations:

- SSL (SpaceSaving Linked-list): the fastest available CPU sequential implementation [5, 9]. We do not take into account the Frequent algorithm in our tests since its best implementation is slower and less accurate than SSL [5].
- ParSSL (Parallel SSL): a CPU parallel implementation of SpaseSaving. The Posix Threads API [18] was used to run multiple parallel instances of the serial algorithm, one for each concurrent thread. Note that we introduce ParSSL only for performances comparison. The analysis of quality results of this implementation is not necessary in this work.
- GPUSB: A GPU-based implementation of the AccurateSB algorithm. Its implementation is based on a library which resembles the C++ Standard Template Library, named Thrust[16], of parallel algorithms based on CUDA. The concept of structure-of-array has been used to handle the buffer B as two arrays: *B.items* and *B.counters*. We implemented the two `Sort` operations using the function `sort_by_key`, the `Reduce` by means of an adapted version of the 2D bucket sort algorithm [19] based on binary search and implemented in the Thrust library, and the two `for` loops are implemented using the function `transform`.



(a) ParSSL: Speed (items/ms) vs. CPU threads (b) GPUSB: Speed (items/ms) vs. b (size of B)

Figure 6: CPU and GPU parameters. In our tests default parameters are CPU thread equal to 32 and b equal to 16 millions.

We generated data (320 million items) from a skewed distribution (Zipf), varying the skew from 0.8 to 2 (in order to obtain meaningful distributions that produce at least one heavy hitter per run). Finally, we also varied the frequency threshold ϕ, from 0.00001 to 0.01. In our experiments, we set the error guarantee $\epsilon = \phi$, since our results showed that this was sufficient to give high accuracy in practice. Consequently, $k = 1/\phi$ in all our experiments. The default skew parameter, unless otherwise noted, is $z = 1.0$, and the default frequency threshold is $\phi = 0.001$. In [5] the trends observed in real network data (HTTP and UDP traffic) are similar to the ones for generated data, hence we used only generated data. For all of the above, we perform 20 runs per experiment (by dividing the input data into 20 chunks of 16 million items each and querying the algorithms once at the end of each run). Furthermore, we ran each algorithm independently from the others to take advantage of possible caching effects. We report averages on all graphs, along with the 5th and 95th percentiles as error bars.

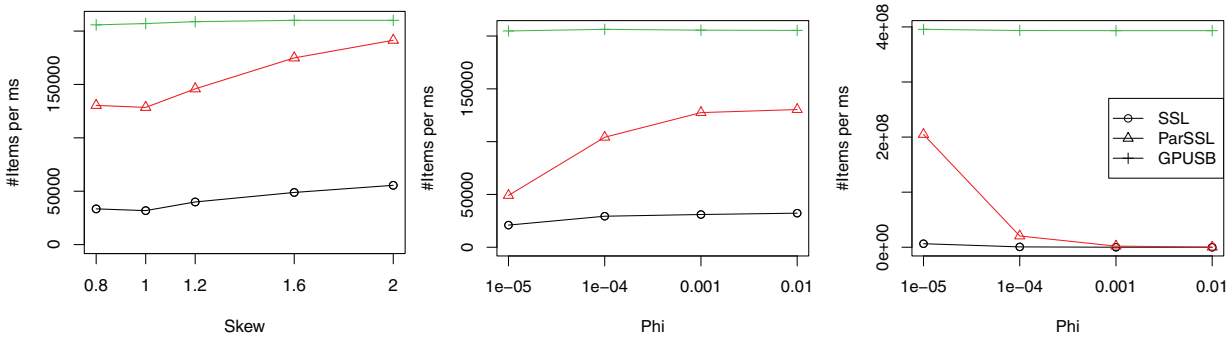
We compared the efficiency of the algorithms with respect to: (i) Update throughput, measured in the number of updates per millisecond. (ii) Space consumed, measured in bytes. (iii) Recall, measured in the total number of true heavy hitters reported over the number of true heavy hitters given by an exact algorithm. (iv) Precision, measured in the total number of true heavy hitters reported over the total number of answers reported. (v) Average relative error of the reported frequencies.

The hardware configuration is based on a CPU Intel Core i7-2600@3.4Ghz (quad-core HT) with 8GB of RAM and a GPU NVIDIA GeForce GTX 480 (480 CUDA cores) with 1.5GB of RAM running Microsoft Windows 7. The code was compiled using Microsofts Visual C++ 2010 and Nvidia CUDA 4.1 compiler.

5.1. Tuning ParSSL and GPUSB parameters

Figure 6a shows the throughput of ParSSL as function of the size of the number of CPU threads. Speed test results of ParSSL does not take into account the time required to generate the final output, that consist of merging as many outputs as the number of threads. The output time is proportional to the number of thread and inversely proportional to the frequency threshold. Performances of ParSSL depends on the number of threads running concurrent instances of SSL. The update throughput increases with the number of threads as the CPU Intel Core i7 processor is able to run 8 threads in parallel. Furthermore, a higher number of threads allows a better load balancing. With more than 64 threads, the overhead is too high and overall performances decrease. The required memory of ParSSL is equal to the product of the number of running threads and the required memory of SSL. SSL allocates, as result of our tests, roughly $64k$ bytes. With frequency threshold $\phi = 0.00001$ ($k = 100000$), SSL allocates 6.4Mbytes and ParSSL 204Mbytes with 32 threads and 409Mbytes with 64 threads.

Figure 6b shows the throughput of GPUSB as function of b, the size of the buffer B. Performances increases with b because host-device memory transfers, `Sort` operations, and `Reduce` operations are more efficient if executed on a big amount of data. GPUSB works with very simple data structures, and the summary memory size is equal to the number of bytes required to store two arrays (items and counters) of size b, and allocates 4 temporary arrays used by the binary search implementation of the `Reduce` operation. The total amount of memory is roughly $6b$ bytes. GPUSB allocates on the GPU memory 384Mbytes with b equal to 16 millions and 768Mbytes with b equal to 32 millions.



(a) Speed (items/ms) vs. skew (b) Speed (items/ms) vs. ϕ (c) Size (bytes) vs. ϕ

Figure 7: Performances comparison: processing speed and required memory

For both GPUSB and ParSSL, we chose parameters representing a thread-off between throughput and memory usage. The default number of threads in ParSSL is 32 and the default size of B in GPUSB is 16 millions.

5.2. Performances Results

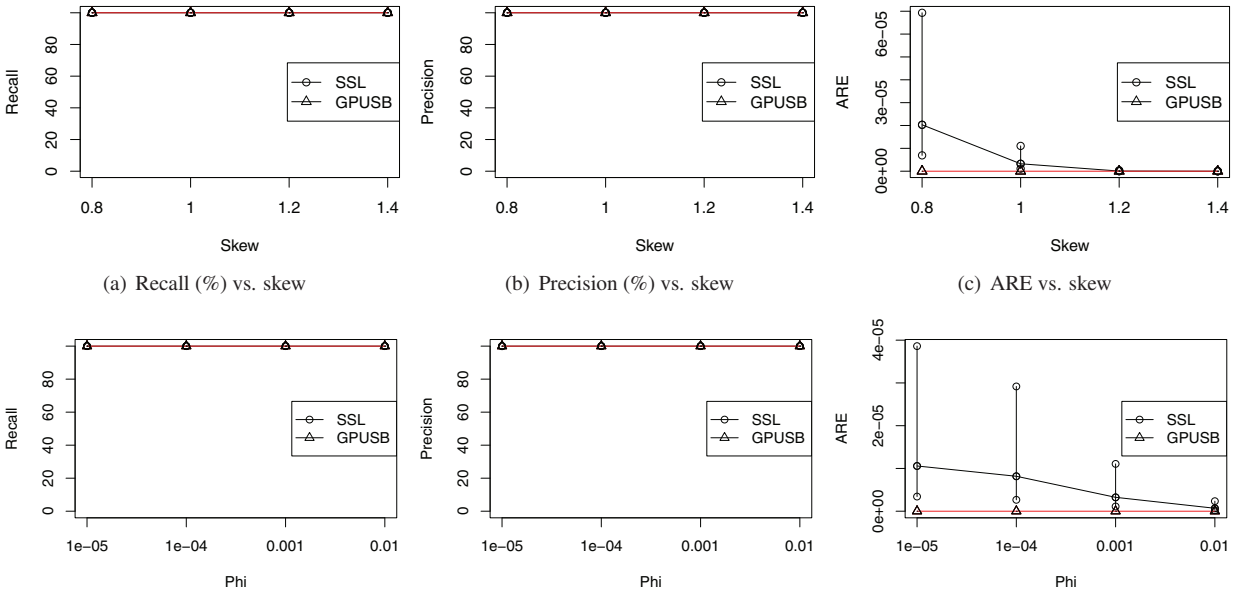
Figure 7a shows the update throughput of the algorithms as a function of data skew. We can see that update throughput of SSL, and ParSSL increases significantly for highly skewed data. This is expected, since a high skew translates to a very small number of truly frequent items, simplifying the problem. SSL is very fast and is able to process between 32K and 55K items per millisecond (respectively for skew equal to 1.0 and 2.0). ParSSL is roughly 4 times faster than SSL. GPUSB is barely influenced by the skewness of the data distribution, because with a high skewness only the `Reduce` operation is slightly simplified, while the two `Sort` operations are not influenced by this parameter. With skew = 2.0, GPUSB processes 210K items/ms and is 3.8 times faster than SSL and 1.1 times faster than ParSSL, while with skew = 1.0, GPUSB processes 207K items/ms and is 6.5 times faster than SSL and 1.5 times than parallel SSL.

Figures 7b and 7c shows, respectively, the update throughput of the algorithms and the used memory as a function of increasing frequency threshold (ϕ). Performances of SSL and ParSSL are competitive if the size of the summary fits the second and third level cache of the CPU. Performances of SSL decrease with $\phi = 0.00001$ as the summary size is bigger than the L2 cache. ParSSL always uses a summary bigger than the L2 cache but only for the larger values of ϕ the summary fits the L3 cache. The update throughput of GPUSB is only slightly influenced by the frequency threshold. This is reasonable, because the size of the buffer B is fixed to $s + k$, and s (the size of the strip) is always at least 16 times bigger than k. With the lowest value of frequency threshold GPUSB is approximately 9.5 times faster than SSL and 4 times faster than ParSSL.

5.3. Quality Results

Figures 8a and 8d plot recall, computed as the total number of true frequent items returned over the exact number of frequent items. Figures 8b and 8e plot precision, an indication of the number of false positives returned. Higher precision means a smaller number of false positive answers. In all cases, both SSL and GPUSB yeld 100% precision and recall.

Figures 8c and 8f plot the average relative error (ARE) in the frequency estimation of the truly frequent items. The graph also plots the 5th and 95th percentiles as error bars. Quality results of the two algorithm are slightly different in this case. SSL yields very low relative error (note the y-axis scale) and GPUSB yields an even smaller relative error, close to zero. Even though both algorithms rely on the same assumption of overestimating new incoming items, GPUSB computes in parallel s items of the input stream, and the frequency estimation is based on more information than SSL.



(a) Recall (%) vs. skew (b) Precision (%) vs. skew (c) ARE vs. skew

(d) Recall (%) vs. ϕ (e) Precision (%) vs. ϕ (f) ARE vs. ϕ

Figure 8: Quality comparison: precision, recall and average relative error (ARE).

6. Discussion and Related Works

Govindaraju et al. presented in [13] an algorithm for fast frequency estimation in large data streams using GPUs. Similarly to our work, they used sorting on GPU to compute frequent items, but differently from us, they used as baseline of their work the LossyCounting algorithm of Manku and Motwani [20]. LossyCounting produces results showing a better precision than Frequent but lower than SpaceSaving on which we implemented our sort-based solution. In [6], Cafaro and Tempesta dealt with the strictly related parallel case of the Frequent algorithm and showed how to merge in parallel multiple counter-based summaries. Our sort-based solution returns a single output and thus is not required to merge multiple summaries. Their best result in terms of performances[2] is 133 million items per second while our sort-based implementation achieves 207 million items per second in similar conditions. The Frequent algorithm generates low precision results in frequency estimation, roughly 20% according to quality results reported in [5], while our sort-based algorithm yields results showing 100% precision. Finally, the fastest FPGA-based implementation of SpaceSaving is presented by Teubner et al. in [4], and achieves a throughput of 80 million items per second. Our implementation is roughly 2.6 times faster.

7. Conclusions

We implemented on GPU an algorithm for finding frequent items in data streams and gave an experimental comparison of its behavior with respect to SpaceSaving algorithm. We observed that our implementation GPUSB offers in general better performances. In particular, as the number of items is large enough to saturate the GPU resources, our approach has a clear speed-up over a parallel implementation of the SpaceSaving. The trade-off in using large amount of GPU memory is that the result is not affected by the skewness of the data distribution as in the case of SpaceSaving which in the case of low skew demonstrated the greater gap. Furthermore, the obtained result is valid without take into account the time to join the output of each concurrent thread in the parallel implementation of SpaceSaving. From the point of view of quality results, our approach is comparable with SpaceSaving and in same case, the relative error is slightly better.

[2] C/MPI implementation running on an in IBM cluster of 30 p575 nodes. In the experiment #14, they computed frequent items on $n = 8$ billion input items in approximately 60 seconds, with $k = 90$ (ϵ is therefore roughly 0.01), and data distribution skewness equal to 0.8.

As future works, we are going to investigate further the GPU for sorting in order to obtain better results. The choice about how estimate new items is fundamental to improve the quality of results as we observed from the naive to accurate implementation. We think that other strategies about how estimate new items could be careful studied. Finally, a further extension of this GPU work could be to other streaming problem, such as finding quantiles, frequency moments, and counting distinct elements.

Acknowledgements

We greatly acknowledge NVIDIA for providing us the hardware used during the experiments.

References

[1] C. Estan, G. Varghese, New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice, ACM Trans. Comput. Syst. 21 (2003) 270–313.
[2] R. Kohavi, F. Provost, Applications of data mining to electronic commerce, Data Min. Knowl. Discov. 5 (2001) 5–10.
[3] R. Pike, S. Dorward, R. Griesemer, S. Quinlan, Interpreting the data: Parallel analysis with sawzall, Sci. Program. 13 (2005) 277–298.
[4] J. Teubner, R. Mller, G. Alonso, FPGA acceleration for the frequent item problem, in: F. Li, M. M. Moro, S. Ghandeharizadeh, J. R. Haritsa, G. Weikum, M. J. Carey, F. Casati, E. Y. Chang, I. Manolescu, S. Mehrotra, U. Dayal, V. J. Tsotras (Eds.), Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA, IEEE, 2010, pp. 669–680.
[5] G. Cormode, M. Hadjieleftheriou, Finding the frequent items in streams of data, Commun. ACM 52 (2009) 97–105.
[6] M. Cafaro, P. Tempesta, Finding frequent items in parallel, Concurr. Comput. : Pract. Exper. 23 (2011) 1774–1788.
[7] J. S. Moore, A fast majority vote algorithm, Tech. rep., Automated Reasoning: Essays in Honor of Woody Bledsoe (1981).
[8] E. D. Demaine, A. López-Ortiz, J. I. Munro, Frequency estimation of internet packet streams with limited space, in: Proceedings of the 10th Annual European Symposium on Algorithms, ESA '02, Springer-Verlag, London, UK, UK, 2002, pp. 348–360.
[9] A. Metwally, D. Agrawal, A. El Abbadi, Efficient computation of frequent and top-k elements in data streams, in: T. Eiter, L. Libkin (Eds.), Database Theory - ICDT 2005, Vol. 3363 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2005, pp. 398–412.
[10] NVIDIA Corporation, NVIDIA's Next Generation CUDA Compute Architecture: Fermi, Tech. rep., Nvidia Corporation (2009).
[11] NVIDIA Corporation, NVIDIA CUDA C Programming Guide 4.0, NVIDIA Corporation, 2010.
[12] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, P. Hanrahan, Photon mapping on programmable graphics hardware, in: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, HWWS '03, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2003, pp. 41–50.
[13] N. K. Govindaraju, N. Raghuvanshi, D. Manocha, Fast and approximate stream mining of quantiles and frequencies using graphics processors, in: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, SIGMOD '05, ACM, New York, NY, USA, 2005, pp. 611–622.
[14] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, P. Dubey, Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort, in: Proceedings of the 2010 international conference on Management of data, SIGMOD '10, ACM, New York, NY, USA, 2010, pp. 351–362.
[15] D. Merrill, A. Grimshaw, High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing, Parallel Processing Letters 21 (02) (2011) 245–272.
[16] N. Bell, J. Hoberock, Thrust: A productivity-oriented library for CUDA, in GPU Computing Gems Jade Edition 2 (2011) 359–371.
[17] R. M. Karp, S. Shenker, C. H. Papadimitriou, A simple algorithm for finding frequent elements in streams and bags, ACM Trans. Database Syst. 28 (2003) 51–55.
[18] D. R. Butenhof, Programming with POSIX threads, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
[19] H. Jared, Thrust by Example: Advanced Features and Techniques, GPU Technology Conference 2010.
[20] G. S. Manku, R. Motwani, Approximate frequency counts over data streams, in: Proceedings of the 28th international conference on Very Large Data Bases, VLDB '02, VLDB Endowment, 2002, pp. 346–357.