Sorting using BItonic netwoRk wIth CUDA

Ranieri Baraglia
ISTI - CNR
Via G. Moruzzi, 1
56124 Pisa, Italy
r.baraglia@isti.cnr.it

Gabriele Capannini
ISTI - CNR
Via G. Moruzzi, 1
56124 Pisa, Italy
g.capannini@isti.cnr.it

Franco Maria Nardini
ISTI - CNR
Via G. Moruzzi, 1
56124 Pisa, Italy
f.nardini@isti.cnr.it

Fabrizio Silvestri
ISTI - CNR
Via G. Moruzzi, 1
56124 Pisa, Italy
f.silvestri@isti.cnr.it

ABSTRACT

Novel "manycore" architectures, such as graphics processors, are high-parallel and high-performance shared-memory architectures [7] born to solve specific problems such as the graphical ones. Those architectures can be exploited to solve a wider range of problems by designing the related algorithm for such architectures. We present a fast sorting algorithm implementing an efficient bitonic sorting network. This algorithm is highly suitable for information retrieval applications. Sorting is a fundamental and universal problem in computer science. Even if sort has been extensively addressed by many research works, it still remains an interesting challenge to make it faster by exploiting novel technologies. In this light, this paper shows how to use graphics processors as coprocessors to speed up sorting while allowing CPU to perform other tasks. Our new algorithm exploits a memory-efficient data access pattern maintaining the minimum number of accesses to the memory out of the chip. We introduce an efficient instruction dispatch mechanism to improve the overall sorting performance. We also present a cache-based computational model for graphics processors. Experimental results highlight up to an order of magnitude improvements over prior CPU-based sorting methods, and a remarkable improvement over previous GPU-based sorting algorithms.

1. INTRODUCTION

Every day people use Web Search Engines as a tool for accessing information, sites, and services on the Web. Information retrieval has to face those issues due to the growing amount of information on the web, as well as the number of new users. Creating a Web Search Engine which scales even to today's Web contents presents many challenges. A fast crawling technology is needed to gather web documents and keep them up to date. Storage space must be used efficiently to store indexes, and documents. The indexing system must process hundreds of gigabytes of data efficiently. Queries must be handled quickly, at a rate of hundreds to thousands per second. All these services run on clusters of homogeneous PCs. PCs in these clusters depends upon price, CPU speed, memory and disk size, heat output, and physical size [3]. Nowadays these characteristics can be find also in other commodity hardware originally designed for specific graphics computations. Many special processor architectures have been proposed to exploit data parallelism for data intensive computations and graphics processors (GPUs) are one of those. For example, the scientific community uses GPUs for general purpose computation. The result obtained, in term of computational latency, outperform the time charge requested on classical processors. Unfortunately, such programs must rely on APIs to access the hardware, for example OpenGL or DirectX. These APIs are simultaneously overspecified, forcing programmer to manipulate data that is not directly relevant, and drivers. These APIs make critical policy decisions, such as deciding where data resides in memory and when they are copied.

In last years, due to the growing trend of media market, the request of rendering algorithms is rapidly evolving. For those companies producing hardware, it means to design every time new hardware both able to run novel algorithms and able to provide higher rate of computations per second. Such processors require significant design effort and are thus difficult to change as applications and algorithms evolve. The request for flexibility in media processing motivates the use of programmable processors, and the existing need for non-graphical APIs pushed the same companies into creating new abstractions designed to last.

Finally, according to what Moore's law foresee, the number of transistor density doubles every two years. Furthermore, mainly due to power-related issues, new generation processors (such as traditional CPUs) tend to incorporate an ever-increasing number of processors (also called cores) on the same chip [9]. The result is that nowadays the market proposes low-cost commodity hardware that is able to execute heavy loads of computations. In order to enable developers to leverage the power of such architectures, they usually make available ad-hoc programming tools for.

For the reasons listed so far, these architectures are ideal

candidates for the efficient implementation of those component of a Large-Scale Search Engine that are eager of computational power.

This paper focuses on using a GPU as a co-processor for sorting. Sorting is a core problem in computer science that has been extensively researched over the last five decades and still remains a bottleneck in many applications involving large volumes of data. One could argue why efficient sorting is related with LSDS-IR. First of all, sorting is a basic application for indexing. We will show in Section 3 how many indexing algorithms are basically a sorting operation over integer sequences. Large scale indexing, thus, required scalable sorting. Second, the technique we are introducing here is of crucial importance for Distributed Systems for IR since it is designed to run on GPUs that are considered by many as a basic building block for future generation data-centers [4]. Our bitonic sorting network can be seen as a viable alternative for sorting large quantities of data on graphics processors. In the last years general purpose processors have been specialized adopting mechanisms to make more flexible their work. Such facilities (i.e. more levels of caches, out-of-order execution paradigm, and branch prediction techniques) leads to make the theoretical performance of CPUs closer to the real achievable one. From the other side specialized processors, like GPUs, expose lower flexibility at design phase, but are able to reach higher computational power providing more computational cores with respect to other the class of processors. We map a bitonic sorting network on GPU exploiting the its high bandwidth memory interface. Our novel data partitioning improves GPU cache efficiency and minimizes data transfers between on-chip and out-chip memories.

This paper is organized as follows. Section 2 discusses related works, while Sections 3 introduces some relevant characteristics about the applicability of GPU-based sorting in Web Search Engines. Section 4 presents some issues arising from the stream programming model and the single-instruction multiple-data (SIMD) architecture. The central part is devoted to expose our solution, and the computational model used to formalize Graphics Processing Units. Section 6 presents some results obtained in a preliminary evaluation phase. Section 7 discuss hot to evolve and improve this research activity.

2. RELATED WORK

Since most sorting algorithms are memory bandwidth bound, there is no surprise that there is currently a big interest in sorting on the high bandwidth GPUs.

Purcell *et al.* [24] presented an implementation of bitonic merge sort on GPUs based on an implementation by Kapasi *et al.* [17]. Author used that approach to sort photons into a spatial data structure providing an efficient search mechanism for GPU-based photon mapping. Comparator stages were entirely realized in the fragment units[1], including arithmetic, logical and texture operations. Authors reported their implementation to be compute-bound rather than bandwidth-bound, and they achieve a throughput far below the theoretical optimum of the target architecture.

Kipfer *et al.* [19, 20] showed an improved version of the bitonic sort as well as an odd-even merge sort. They presented an improved bitonic sort routine that achieves a performance gain by minimizing both the number of instructions to be executed in the fragment program and the number of texture operations.

Zachmann *et al.* [14] presented a novel approach for parallel sorting on stream processing architectures based on an adaptive bitonic sorting [6]. They presented an implementation based on modern programmable graphics hardware showing that they approach is competitive with common sequential sorting algorithms not only from a theoretical viewpoint, but also from a practical one. Good results are achieved by using efficient linear stream memory accesses and by combining the optimal time approach with algorithms.

Govindaraju *et al.* [13] implemented sorting as the main computational component for histogram approximation. This solution is based on the periodic balanced sorting network method by Dowd *et al.* [10]. In order to achieve high computational performance on the GPUs, they used a sorting network based algorithm and each stage is computed using rasterization. They later presented a hybrid bitonic-radix sort that is able to sort vast quantities of data [12], called GPUTeraSort. This algorithm was proposed to sort record contained in databases using a GPU. This approach uses the data and task parallelism on the GPU to perform memory-intensive and compute-intensive tasks while the CPU is used to perform I/O and resource management.

Sengupta *et al.* [25] presented a radix-sort and a Quicksort implementation based on segmented *scan* primitives. Authors presented new approaches of implementing several classic applications using this primitives and shows that this primitives are an excellent match for a broad set of problems on parallel hardware.

Recently, Sintorn *et al.* [28] presented a sorting algorithm that combines bucket sort with merge sort. In addition, authors show this new GPU sorting method sorts on *nlog(n)* time.

Cederman *et al.* [8] showed that their GPU-Quicksort is a viable sorting alternative. The algorithm recursively partition the sequence to be sorted with respect to a pivot. This is done in parallel by each GPU-thread until the entire sequence has been sorted. In each partition iteration a new pivot value is picked and as a result two new subsequences are created that can be sorted independently by each thread block can be assigned one of them. Finally, experiments pointed out that GPU-Quicksort can outperform other GPU-based sorting algorithms.

3. APPLICATIONS TO INDEXING

A modern search engine must scale even with the growing of today's Web contents. Large-scale and distributed applications in Information Retrieval such as crawling, indexing, and query processing can exploit the computational power of new GPU architectures to keep up with this exponential grow.

We consider here one of the core-component of a large-scale search engine: *the indexer*. In the indexing phase, each crawled document is converted into a set of word occurrences called *hits*. For each word the hits record: frequency, position in document, and some other information. Indexing, then, can be considered as a "sort" operation on a set of records representing term occurrences [2]. Records repre-

[1]In addition to computational functionality, fragment units also provide an efficient memory interface to server-side data, i.e. texture maps and frame buffer objects.

sent distinct occurrences of each term in each distinct document. Sorting efficiently these records using a good balance of memory and disk usage, is a very challenging operation.

In the last years it has been shown that sort-based approaches [29], or single-pass algorithms [21], are efficient in several scenarios, where indexing of a large amount of data is performed with limited resources.

A sort-based approach first makes a pass through the collection assembling all term-docID pairs. Then it sorts the pairs with the term as the dominant key and docID as the secondary key. Finally, it organizes the docIDs for each term into a postings list (it also computes statistics like term and document frequency). For small collections, all this can be done in memory.

When memory is not sufficient, we need to use an external sorting algorithm [22]. The main requirement of such algorithm is that it minimizes the number of random disk seeks during sorting. One solution is the Blocked Sort-Based Indexing algorithm (BSBI). BSBI segments the collection into parts of equal size, then it sorts the termID-docID pairs of each part in memory, finally stores intermediate sorted results on disk. When all the segments are sorted, it merges all intermediate results into the final index.

A more scalable alternative is Single-Pass In-Memory Indexing (SPIMI). SPIMI uses terms instead of termIDs, writes each block's dictionary to disk, and then starts a new dictionary for the next block. SPIMI can index collections of any size as long as there is enough disk space available. The algorithm parses documents and turns them into a stream of term-docID pairs, called tokens. Tokens are then processed one by one. For each token, SPIMI adds a posting directly to its postings list. Instead of first collecting all termID-docID pairs and then sorting them (as BSBI does), each postings list is dynamic. This means that its size is adjusted as it grows. This has two advantages: it is faster because there is no sorting required, and it saves memory because it keeps track of the term a postings list belongs to, so the termIDs of postings need not be stored.

When memory finished, SPIMI writes the index of the block (which consists of the dictionary and the postings lists) to disk. Before doing this, SPIMI sorts the terms to facilitate the final merging step: if each block's postings lists were written in unsorted order, merging blocks could not be accomplished by a simple linear scan through each block. The last step of SPIMI is then to merge the blocks into the final inverted index.

SPIMI, which time complexity is lower because no sorting of tokens is required, is usually preferred with respect to BSBI that presents an higher time complexity.

In both the methods presented for indexing, sorting is involved: BSBI sorts the termID-docID pairs of all parts in memory, SPIMI sorts the terms to facilitate the final merging step [22].

In order to efficiently evaluate these two approaches on a heterogeneous cluster we have to compare "standard" SPIMI performances with the performances of a BSBI-based sorter implemented by us. Moreover, to fully analyze the indexing phase, we need a GPU-based string sorted able to outperform CPUs as well as our sorter for integers does. In this way we have the possibility to compare both solutions, on all architectures, then to choose the best combination. Having all possible implementations available, a *flexible* execution of indexing running on various hardware can be imagined. This option is even more important if the allocation of the task is scheduled dynamically, as it can be done depending of the workload of the single resources.

The-state-of-art in string sort lacks of solution for GPU architectures: nowadays we are not aware of solutions for parallel SIMD processors. In literature, this problem is efficiently solved by using different approaches. The most interesting and suitable for us seems to be Burstsort [27]. It is a technique that combines the burst trie [15] to distribute string-items into small buckets whose contents are then sorted with standard (string) sorting algorithms. Successively, Sinha *et al.* [26] introduced improvements that reduce by a significant margin the memory requirements of Burstsort. This aspect is even more relevant for GPU architectures having small-sized on-chip memories.

4. SORTING WITH GPUS

This section gives a brief overview of GPUs highlighting features that make them useful for sorting. GPUs are designed to execute geometric transformations that generate a data stream of display-pixels. A data stream is processed by a program running on multiple SIMD processors, which are capable for data-intensive computations. The output, then, is written back to the memory.

4.1 SIMD Architecture

SIMD machines are classified as processor-array machines: a SIMD machine basically consists of an array of computational units connected together by a simple network topology [23]. This processor array is connected to a control processor, which is responsible for fetching and interpreting instructions. The control processor issues arithmetic and data processing instructions to the processor array, and handles any control flow or serial computation that cannot be parallelized. Processing elements can be individually disabled for conditional execution: this option give more flexibility during the design of an algorithm.

Although SIMD machines are very effective for certain classes of problems, the architecture is specifically tailored for data computation-intensive work, and it results to be quite "inflexible" on some classes of problems[2].

4.2 Stream Programming Model

A stream program [18] organizes data as *streams* and expresses all computation as *kernels*. A stream is a sequence of similar data elements, that are defined by a regular access pattern. A kernel typically loops through all the input stream elements, performing a sequence of operations on each element, and appending results to an output stream. These operations exhibits an high instruction level parallelism. Moreover, these operations cannot access to arbitrary memory locations but they keep all the intermediate values locally, into kernels. Since each element of the input stream can be processed simultaneously, kernels also expose large amounts of data-level parallelism. Furthermore, stream memory transfers can be executed concurrently with kernels, thereby exposing task-level parallelism in the stream program. Some other important characteristics common to all stream-processing applications are: (i) elements are read once from memory, (ii) elements are not visited twice, and

[2] For example, these architectures cannot efficiently run the control-flow dominated code.

(iii) the application requires high level of arithmetic operations per memory reference, i.e. computationally intensive.

4.3 Nvidia's CUDA

CUDA [1], acronym of Compute Unified Device Architecture, is defined as an architecture built around a scalable array of multithreaded streaming multiprocessors (MPs). Each MP is defined as a unit comprising one instruction unit, a collection of eight single precision pipelines also called cores, a functional units for special arithmetical operations and a 16 KB local store also called shared memory. In practice, this means that each MP is a *SIMD* processor, whose cores form an arithmetic pipeline that executes scalar instructions. All MPs create, manage, and execute concurrent threads in hardware with zero scheduling overhead, and implements a barrier for threads synchronization. Nevertheless, only threads concurrently running on the same MP can be synchronized.

CUDA model also introduces the entity *warp* as a group of 32 parallel scalar threads, and reports that each warp executes one common instruction at a time. This is another way of saying that warp is a stream of vector instructions: scalar threads are then vector elements. But, unlike others SIMD instruction set, such as Intel's SSE, a particular value of the vector length is not specified. A *thread block* is defined as a collection of warps that run on the same MP and share a partition of local store. The number of warps in a thread block is configurable.

4.3.1 CUDA SDK

The SDK provided by Nvidia for its GPUs consist of a large, collection of code examples, compilers and run-time libraries. Clearly the CUDA model is "restricted" to Nvidia products, mainly for efficiency reasons, and it is conform to the stream programming model. Threads and thread blocks can be created only by invoking a parallel kernel, not from within a parallel kernel. Task parallelism can be expressed at the thread-block level, but block-wide barriers are not well suited for supporting task parallelism among threads in a block. To enable CUDA programs to run on any number of processors, communication between different blocks of threads, is not allowed, so they must execute independently. Since CUDA requires that thread blocks are independent and allows blocks to be executed in any order. Combining results generated by multiple blocks must in general be done by launching a second kernel. However, multiple thread blocks can coordinate their work using atomic operations on the external (out-chip) memory.

Recursive function calls are not allowed in CUDA kernels. Recursion is, in fact, unattractive in a massively parallel kernel. Providing stack space for all the active threads it would require substantial amounts of memory. To support an heterogeneous system architecture combining a CPU and a GPU, each with its own memory system, CUDA programs must copy data and results between host memory and device memory. The overhead of CPU/GPU interaction and data transfers is minimized by using DMA block-transfer engines and fast interconnects.

4.4 Cache-oblivious algorithm

The cost of communication can be larger up to an order of magnitude than the cost of the pure computation on such architectures. Our idea is to model the proposed solution as cache-oblivious algorithm. The model underlying this type of algorithm is not directly applicable on GPU's parallel architecture, which is equipped with local memory instead of cache. Adopting local memory approach, the programmer has to bear the effort of synchronizing, sizing, and scheduling the computation of data and its movement through the out-chip memory and the in-chip one. On the other hand in cache-based architectures this aspect is automatically managed by the underlying support. A local memory approach permits to move data located in different addresses composing a specific access pattern. This capability is impossible to realize with caches, where the hardware hides this operation by automatically replacing missed cache lines.

Prokop *et al.* [11] presents cache-oblivious algorithms that use both asymptotically optimal amounts of work, and asymptotically optimal number of transfers among multiple levels of cache. An algorithm is cache oblivious if no program variables dependent on hardware configuration parameters, such as cache size and cache-line length need to be tuned to minimize the number of cache misses. Authors introduce the "Z, L" ideal-cache model to study the cache complexity of algorithms. This model describes a computer with a two-level memory hierarchy consisting of an ideal data cache of Z words of constant size, and an arbitrarily large main memory. The cache is partitioned into cache lines, each consisting of L consecutive words that are always moved together between cache and main memory.

The processor can only reference words that reside in the cache. If the referenced word belongs to a line already in cache, a cache hit occurs, and the word is delivered to the processor. Otherwise, a cache-miss occurs, and the line is fetched into the cache. If the cache is full, a cache line must be evicted. An algorithm with an input of size n is measured in the ideal-cache model in terms of its work complexity $W(n)$ and its cache complexity $Q(n, Z, L)$, that is the number of cache misses it incurs as a function of the size Z and line length L of the ideal cache.

The metrics used to measure cache-oblivious algorithms need to be reconsidered in order to be used with GPUs that are parallel architectures. To do that $W()$ has to be defined taking care of the level of parallelism exposed by GPUs. Evaluating $Q()$, we must translate the concept of Z and L that refer to cache characteristics. More precisely, a GPUs is provided of p MPs each one with a local memory. We can abstract such architectural organization by considering each local memory as one cache-line, and the union of all local memories as the entire cache, taking no care of which processor is using data. In this point of view, if the shared memory of each MP is 16KB, we obtain $L = 16$ KB and $Z = 16 \cdot p$ KB.

5. BITONIC SORT

To design our sorting algorithm in the stream programming model, we started from the popular Bitonic Sort (BS) network and we extend it to adapt to our specific architecture. Specifically, BS is one of the fastest sorting networks [5]. A sorting network is a special kind of sorting algorithm, where the sequence of comparisons do not depend on the order with which the data is presented, see Figure 1. This makes sorting networks suitable for implementation on GPUs. In particular, the regularity of the schema used to compare the elements to sort, makes this kind of sorting network particularly suitable for partitioning the elements in

the stream programming fashion, as GPUs require. Finally, we compared theoretical results with the ones resulting from the tests, in order to say if the adapted ideal-cache model is useful to abstract GPUs.

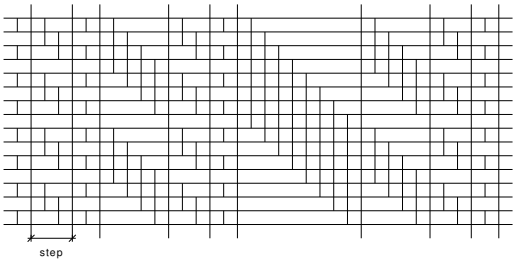

Figure 1: Bitonic sorting networks for 16 elements. Each step is completed when all comparisons involved are computed. In the figure each comparison is represented with a vertical line that link two elements, which are represented with horizontal lines.

The main issue to address is to define an efficient schema to map all comparisons involved in the BS on the elements composing the streams invoked.

The first constraint is that the elements composing each stream must be "distinct". This means that each item in the input has to be included in exactly one element of the stream. From this point of view, each stream of elements defines a partition of the input array to be sorted. This characteristic is necessary due to the runtime support, because it does not permit any type of data-exchange, or synchronization, between two elements (Figure 2).

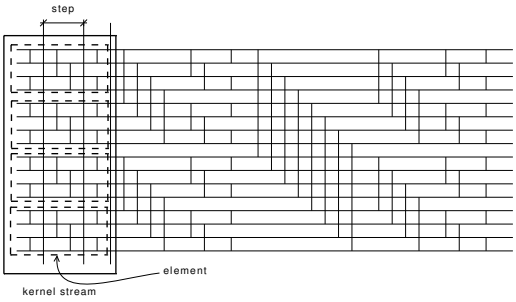

Figure 2: Example of a kernel stream comprising more sorting network steps. The subset of items composing each element must perform comparison only inside itself.

The second aspect to optimize is the partitioning of the steps composing the sorting network. Since a BS is composed by several steps (Figure 1), we have to map the execution of all steps into a sequence of independent runs, each of them corresponding to the invocation of a kernel. Since each kernel invocation implies a communication phase, such mapping should be done in order to reduce the number of these invocations, thus the communication overhead. Specifically, this overhead is generated whenever the SIMD processor begins the execution of a new stream element. In that case, the processor needs to flush the results contained in the proper shared memory, then to fetch the new data from the out-chip memory. In the ideal-cache computational model, it corresponds to a cache-miss event, which wastes the performance of the algorithm.

Resuming, performing several network steps in the same kernel has the double effect to reduce the number of cache-misses, i.e. improving $Q()$ metric, and to augment the level of arithmetic operations per memory reference. The unique constraint is that the computation of an element has to be independent from the one of another element in the same stream.

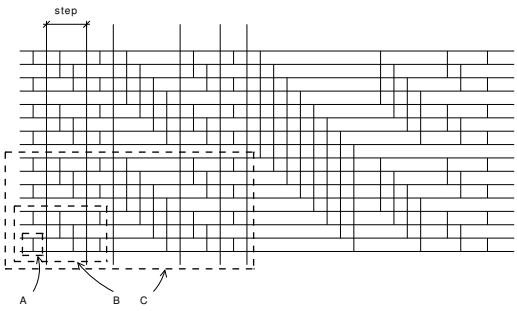

Figure 3: Increasing the number of steps covered by the partition, the number of items included doubles. A, B and C are partitions respectively for local memory of 2, 4 and 8 locations.

Let us introduce our solution. First of all, we need to establish the number of consecutive steps to be executed by one kernel. We must consider that for each step assigned to a kernel, in order to maintain the all stream elements independent, the number of memory location needed by the relative partition doubles, see Figure 3. So, the number of steps a kernel can cover is bounded by the number of items that it is possible to include into the stream element. Furthermore, the number of items is bounded by the size of the shared memory available for each SIMD processor.

Algorithm 1 Bitonic Sort algorithm.

```
a ← array to sort
for s = 1 to log2 |a| do
  for c = s − 1 down to 0 do
    for r = 0 to |a| − 1 do
      if r/2^c ≡ r/2^s (mod 2) ∧ a[r] > a[r ⊕ 2^c] then
        a[r] ⇆ a[r ⊕ 2^c]
      end if
    end for
  end for
end for
```

More precisely, to know how many steps can be included in one partition, we have to count how many distinct values c assumes, see Algorithm 1. Due to the fixed size of memory locations, i.e. 16 KB, we can specifies partition of SH = 4 K items, for items of 32 bits. Moreover such partition is able to cover "at least" $sh = log(SH) = 12$ steps. From this evaluation it is also possible to estimate the size of the kernel stream: if a partition representing an element of the stream contains SH items, and the array a to sort contains $N = 2^n$ items, then the stream contains $b = N/SH = 2^{n-sh}$ elements.

A important consideration must be done for the first kernel invoked by the algorithm: until the variable s in the Algorithm 1 is not greater than sh the computation of the several first steps can be done with this first kernel. This because the values assumed by c remain in a range of sh distinct values. More precisely the first kernel computes the first $\frac{sh\cdot(sh+1)}{2}$ steps (Figure 3).

This access pattern schema can be traduced in the function $\ell_c(i)$ that for the current kernel stream, given the current value of c, is able to define the subset of a to be assigned to the i-th stream element. In other words, ℓ describes a method to enumerate the set of indexes in a that the i-th element of the kernel stream has to perform. Formally, it is defined as:

$$\ell_c : i \in [0, b-1] \rightarrow \Pi \subseteq \pi_{sh}(a)$$

where $\pi_{sh}(a)$ is a partition of a, namely a set of nonempty subsets of a such that every element in a is in exactly one of these subsets, and each subset contains exactly 2^{sh} elements of a.

Let us assume $n = 32$ and the size of the shared memory can contains 4 K items, so we have $|a| = 2^{32}$ and $b = 2^{n-sh} = 2^{32-12} = 2^{20}$ elements for each stream. Basically, we need 32 bits to point an element of a, and $\log(b) = 20$ bits to identify the i-th partition among the b existing. The i value is used to build a bit-mask that is equal for each address produced by $\ell_c(i)$. Such mask sets $\log(b)$ bits of the 32 bits composing an index for a. The missing sh bits are generated by using a variable x to enumerate all values in the range $\mathbb{X} = [0, 2^{sh}-1]$ and by inserting each of them in c-th position of the i mask. This composition leads to a set of addresses of n bits whose relative items compose the b-th partition. Formally, we obtain:

$$\ell_c(i) = \{x \in \mathbb{X} \ : \ i_{[31\ldots c]} \circ x \circ i_{[c+1\ldots 0]}\}$$

where $i_{[31\ldots c]}$ notation identifies the leftmost bits, namely from the 31th bit down to the c-th one, and "$\circ$" is the concatenation operator.

The rule to compose the elements in $\ell_c(i)$ is easy, but in some case it leads to some exception. When $c < sh$, then x is divided in two parts, that are x_L and x_R, and they are inserted in c-th position and in the c'-th position of i respectively. In particular, the statement $c < sh$ occurs whenever the middle loop in the Algorithm 1 ends and c start a new loop getting the new value of s, denoted with c'. Specifically, it happens that, depending on the current value of c, the algorithm needs to make two insertions: to add x_R at position c, and to add x_L at position c'. Formally, when $c < sh$, we have to define a second function $\ell_{c,c'}()$ as in the following:

$$\ell_{c,c'}(i) = \{x \in \mathbb{X} \ : \ i_{[31\ldots\bar{c}]} \circ x_L \circ i_{[c'+sh-c\ldots c]} \circ x_R\}$$

where $x_L = x_{[sh-1\ldots c]}$, $x_R = x_{[c-1\ldots 0]}$, and $c' = s + 1$.

5.1 Evaluation

For our solution we obtain that $W(n,p)$, where p indicates the number of MPs, and n the size of a, is equal to the computational cost of the sequential BS divided p, specifically $W(n,p) = O\big(\frac{n}{p} \cdot \log^2 n\big)$. To know $Q(n,Z,L)$ we must estimate the number of cache-misses. Assuming $|a| = n$, we obtain that the sorting network is made of $\sigma = \frac{1}{2}\big(\log^2(n) + \log(n)\big)$ steps. Furthermore, let us assume $L = SH$, $Z = SH \cdot p$, and each kernel covers sh steps, except the first kernel that covers $\sigma' = \frac{1}{2}\big(sh^2 + sh\big)$. Then the number of cache-misses is $\lceil(\sigma - \sigma')/sh\rceil$.

The last consideration regards $W(n,p)$ measure, that should be estimated considering that each MP is a SIMD processor. In fact, each MP reaches its maximum performance whenever the data-flow permits to the control unit to issue the same instruction for all cores. In this way such instruction is executed in parallel on different data in a SIMD fashion.

In order to compare our bitonic sort with the quick sort proposed by Cederman *et al.*, we tried to extend the analysis of ideal-cache model metrics to their solution. Unfortunately their solution does not permit to be accurately measured like ours. In particular, it is possible to estimate the number of transfers among multiple levels of cache, but quick sort uses out-chip memory also to implement prefix-sum for each stream element ran. In particular quick sort splits input array in two parts with respect to a pivot by invoking a procedure on GPU, and recursively repeats this operation until each part can be entirely contained in the shared memory. In the optimistic case, assuming $|a| = n$ and a shared memory equal to $L = SH$, this operation is invoked $\log(n) - \log(L)$ times, that is also the number of cache-misses for quick sort, namely $Q()$. This value is sensibly lower to the $Q()$ measured for bitonic sort, but the evaluation of the number of such cache-misses should be proportionally augmented due to the prefix-sum procedure. Regarding $W()$, the authors report a computational complexity equal to the one obtained for sequential case, i.e. $O\big(n \cdot \log(n)\big)$, without referring the parallelism of the underlying hardware. However, optimistic evaluation of such parallel, version should be, also in this case, lower than $W()$ computed for bitonic sort.

6. RESULTS

The experiments have been conducted on an Ubuntu Linux Desktop with an Nvidia 8800GT, namely a GPU provided with 14 SIMD processors and CUDA SDK 2.1. To generate the arrays for these preliminary tests we used uniform, gaussian and zipfian distributions. Specifically, for each distribution was generated 20 different arrays. Figure 4 shows: the means, the standard deviation, the maximum and the minimum for the times elapsed for all runs of each distribution. The tests involved CPU-based quick sort provided with C standard library, our solution and the one proposed by Cederman *et al.* [8]. In that paper, the GPU-based quick sort resulted the most performing algorithm in literature, so our preliminary tests take into consideration only such GPU-based algorithm.

Figure 4 shows that GPU-based solutions are able to outperform CPU's performance for the sorting problem. The same figure also shows that our solution is not competitive with respect to the one of Cederman until the number on items to sort reaches 8 Mega. One more consideration is that GPU-based quick sort is not able to successfully conclude the tests for arrays greater than 16 Mega. Further analysis pointed out that bitonic algorithm spends the main part of the time elapsed for the data-transfer phase of some specific kernel instance. Since element streams were always of the same size, we deduced that the number of transfers is not the only aspect to take into consideration to minimize the communication overhead, as metrics of ideal-cache models suggests.

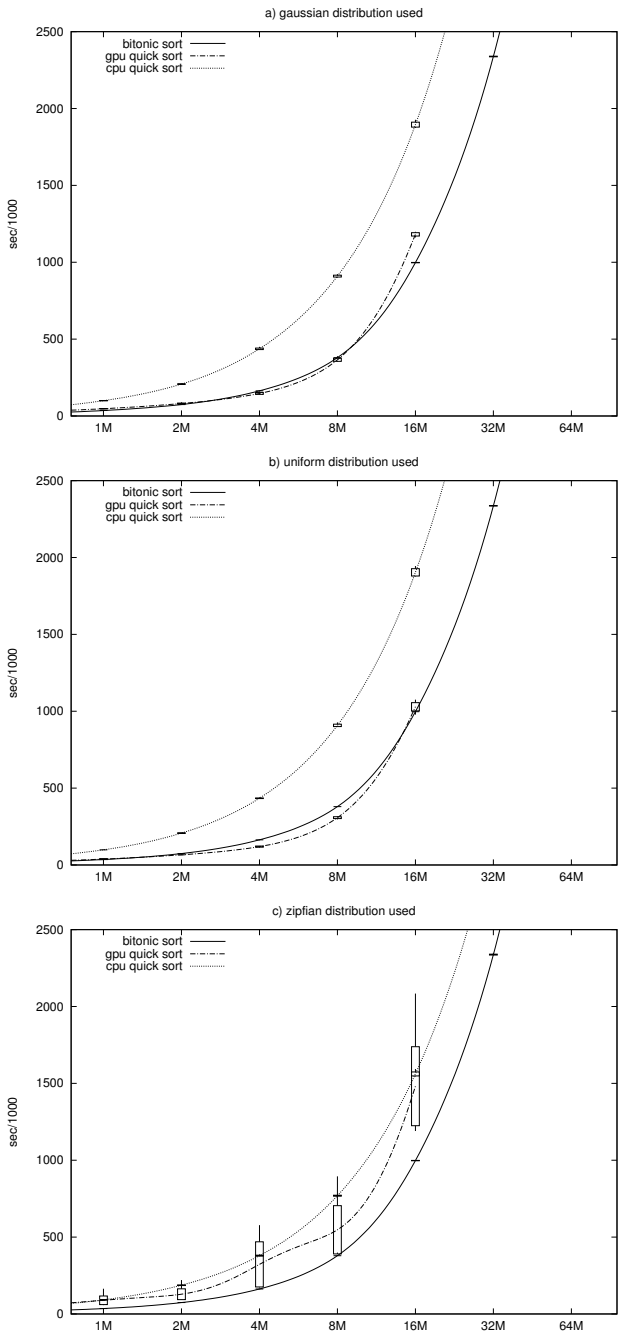

Figure 4: Squared white areas represent the variance obtained for several running for each size of the problem. Vertical lines point out the maximum and the minimum value obtained.

As suggest by Helman *et al.* [16] a deeper evaluation of the algorithm can be conducted by using arrays generated by different distributions. This type of test puts in evidence the behavior of the algorithms regarding to the variance of the times obtained in different contexts. For the two distribution tested, bitonic sort algorithm has a better behavior with respect to variance. Obviously, this result is caused by the type of algorithm used. Quick sort is a data-dependent approach whereas sorting network are based on a fixed schema of comparisons that does not vary with respect to data-distribution.

Consideration on the results obtained from these preliminary test suggest us that ideal-cache model does not seem sufficiently accurate to abstract GPU's architecture. If theoretical results lead to better performance for GPU-based quick sort, from the tests conducted, it arises that bitonic sort has a better performance-trend by increasing the size of the problem. This consideration is enforced by the analysis of the data-transfer: we strongly believe that by improving the data-transfer bandwidth, bitonic sort can reach better results without increasing theoretical $W()$ and $Q()$ metrics.

7. FUTURE WORKS

Preliminary results show that the number of transfers is not the only aspect to take into consideration for minimizing communication overhead. Another important factor is the *transfer bandwidth* that is relevant to achieve better results. Results show that the ideal-cache model is not able to fully describe and capture all the aspects determining the performance for such kind of architectures. Probably different kinds of performance metrics are needed to evaluate an algorithms on these novel hardware resources.

Furthermore, since indexing is a tuple-sorting operation we will extend our solution to include the sorting of tuples of integers. In this paper, in fact, we assume the tuples are sorted by using multiple passes on the dataset. We reserve to future work the extension to tuples.

Of course, we have to run more tests to enforce the results obtained and to analyze more in deep the causes of the waste of performance that affect our algorithm.

8. REFERENCES

[1] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[2] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed web retrieval. In *ICDE*. IEEE, 2007.

[3] L. A. Barroso, J. Dean, and Holzle. Web search for a planet: The google cluster architecture. *Micro, IEEE*, 23(2):22–28, 2003.

[4] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, San Rafael, CA, USA, 2009.

[5] K. E. Batcher. Sorting networks and their applications. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, New York, NY, USA, 1968. ACM.

[6] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. Technical report, Ithaca, NY, USA, 1986.

[7] J. Bovay, B. H. Brent, H. Lin, and K. Wadleigh. Accelerators for high performance computing investigation. White paper, High Performance Computing Division - Hewlett-Packard Company, 2007.

[8] D. Cederman and P. Tsigas. A practical quicksort algorithm for graphics processors. In *ESA '08: Proceedings of the 16th annual European symposium*

on Algorithms, pages 246–258, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] J. D. Davis, J. Laudon, and K. Olukotun. Maximizing cmp throughput with mediocre cores. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 51–62, Washington, DC, USA, 2005. IEEE Computer Society.

[10] M. Dowd, Y. Perl, L. Rudolph, and M. Saks. The periodic balanced sorting network. *J. ACM*, 36(4):738–757, 1989.

[11] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, Washington, DC, USA, 1999. IEEE Computer Society.

[12] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: High performance graphics coprocessor sorting for large database management. In *ACM SIGMOD International Conference on Management of Data*, Chicago, United States, June 2006.

[13] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 611–622, New York, NY, USA, 2005. ACM.

[14] A. Greß and G. Zachmann. Gpu-abisort: Optimal parallel sorting on stream architectures. In *The 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS '06)*, page 45, Apr. 2006.

[15] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20(2):192–223, 2002.

[16] D. R. Helman, D. A. B. Y, and J. J. Z. A randomized parallel sorting algorithm with an experimental study. Technical report, Journal of Parallel and Distributed Computing, 1995.

[17] U. J. Kapasi, W. J. Dally, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany. Efficient conditional operations for data-parallel architectures. In *In Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 159–170. ACM Press, 2000.

[18] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, 2001.

[19] P. Kipfer, M. Segal, and R. Westermann. Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM Press.

[20] P. Kipfer and R. Westermann. Improved GPU sorting. In M. Pharr, editor, *GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 733–746. Addison-Wesley, 2005.

[21] N. Lester. Fast on-line index construction by geometric partitioning. In *In Proceedings of the 14th ACM Conference on Information and Knowledge Management (CIKM 2005*, pages 776–783. ACM Press, 2005.

[22] C. D. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval*. 2008.

[23] M. A. Nichols, H. J. Siegel, H. G. Dietz, R. W. Quong, and W. G. Nation. Eliminating memory for fragmentation within partitionable simd/spmd machines. *IEEE Trans. Parallel Distrib. Syst.*, 2(3):290–303, 1991.

[24] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[25] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[26] R. Sinha and A. Wirth. Engineering burstsort: Towards fast in-place string sorting. In *WEA*, pages 14–27, 2008.

[27] R. Sinha, J. Zobel, and D. Ring. Cache-efficient string sorting using copying. *ACM Journal of Experimental Algorithmics*, 2006.

[28] E. Sintorn and U. Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, In Press, Corrected Proof.

[29] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.