Contents lists available at ScienceDirect

Information Processing and Management

journal homepage: www.elsevier.com/locate/infoproman

Sorting on GPUs for large scale datasets: A thorough comparison

Gabriele Capannini, Fabrizio Silvestri, Ranieri Baraglia *

Information Science and Technology Inst., via G. Moruzzi 1, 56100 Pisa, Italy

ARTICLE INFO

Article history:
Available online xxxx

Keywords:
Stream programming
Graphical processor unit
Bitonic sorting network
Computational model

ABSTRACT

Although sort has been extensively studied in many research works, it still remains a challenge in particular if we consider the implications of novel processor technologies such as manycores (i.e. GPUs, Cell/BE, multicore, etc.). In this paper, we compare different algorithms for sorting integers on stream multiprocessors and we discuss their viability on large datasets (such as those managed by search engines). In order to fully exploit the potentiality of the underlying architecture, we designed an optimized version of sorting network in the K-model, a novel computational model designed to consider all the important features of many-core architectures. According to K-model, our bitonic sorting network mapping improves the three main aspects of many-core architectures, i.e. the processors exploitation, and the on-chip/off-chip memory bandwidth utilization. Furthermore we are able to attain a space complexity of $\Theta(1)$. We experimentally compare our solution with state-of-the-art ones (namely, Quicksort and Radixsort) on GPUs. We also compute the complexity in the K-model for such algorithms. The conducted evaluation highlight that our bitonic sorting network is faster than Quicksort and slightly slower than radix, yet being an in-place solution it consumes less memory than both algorithms.

1. Introduction

Given the demand of massive computing power in modern video game applications, GPUs (as well as the Cell/BE) are designed to be extremely fast at rendering large graphics data sets (e.g. polygons and pixels). Indeed, inspired by the attractive performance/cost ratio, several studies adopt such type of processors also for carrying out data-intensive and general-purpose tasks. For some problems, such as Web information retrieval, the results obtained, in term of computational latency, outperform those obtained using classical processors. Recently, there have been some efforts aimed at developing basic programming models like Map-Reduce. For example Bingsheng, Wenbin, Qiong, Naga, and Tuyong (2008) designed Mars, a Map-Reduce framework, on graphics processors, and Kruijf and Sankaralingam (2007) presented an implementation of Map-Reduce for the Cell architecture.

The main idea in modern processors like GPUs, Sony's Cell/BE, and multi-core CPUs, is stuffing several computing cores (from a few to a few hundreds) on a single chip. In current CPUs a common design practice is to devote a huge percentage of the chip-area to cache mechanism and memory management, in general. Differently from standard CPUs, the majority of the chip-area in modern manycores is devoted to implement computational units. For this reason, they are able to perform *specialized* computations over massive streams of data in a fraction of the time needed by traditional CPUs. The counterpart, though, is that it is difficult for the programmer to write applications able to reach the maximum level of performance they support.

In this paper we digress from the motivation of sorting efficiently a large amount of data on modern GPUs to propose a novel sorting solution that is able to sort in-place an array of integers. In fact, sorting is a core problem in computer science

* Corresponding author.
E-mail address: r.baraglia@isti.cnr.it (R. Baraglia).

0306-4573/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.ipm.2010.11.010

that has been extensively researched over the last five decades, yet it still remains a bottleneck in many applications involving large volumes of data. Furthermore. sorting constitutes a basic building block for large-scale distributed systems for IR. First of all, as we show in Section 3, sorting is the basic operation for indexing. Large scale indexing, thus, required scalable sorting. Second, the technique we are introducing here is viable for distributed systems for IR since it is designed to run on GPUs that are considered as a basic building block for future generation data-centers (Barroso & Hölzle, 2009). Our bitonic sorting network can be seen as a viable alternative for sorting large amounts of data on GPUs.

During our research we studied a new function to map bitonic sorting network (BSN) on GPU exploiting its high bandwidth memory interface. We also present this novel data partitioning schema that improves GPU exploitation and maximizes the bandwidth with which the data is transferred between on-chip and off-chip memories. It is worth noticing that being an in-place sorting based on bitonic networks our solution uses less memory than non in-place ones (e.g. (Cederman & Tsigas, 2008; Sengupta, Harris, Zhang, & Owens, 2007)), and allows larger datasets to be processed. Space complexity is an important aspect when sorting large volume of data, as it is required by large-scale distributed system for information retrieval (LSDS-IR).

To design our sorting algorithm in the stream programming model, we started from the popular BSN, and we extend it to adapt to our target architecture. Bitonic sort is one of the fastest sorting networks (Batcher, 1968). Due to its large exploitation bitonic sorting is one of the earliest parallel sorting algorithms proposed in literature (Batcher, 1968). It has been used in many applications. Examples are the divide-and-conquer strategy used in the computation of the Fast Fourier Transform (Govindaraju & Manocha, 2007), Web information retrieval (Capannini, Silvestri, Baraglia, & Nardini, 2009), and some new multicasting network (Al-Hajery & Batcher, 1993).

The main contributions of this paper are the following:

- We perform a detailed experimental evaluation of state-of-the-art techniques on GPU sorting and we compare them on different datasets of different size and we show the benefits of adopting in-place sorting solutions on large datasets.
- By using the performance constraints of a novel computational model we introduce in Capannini, Silvestri, and Baraglia (2010), we design a method to improve the performance (both theoretical and empirical) of sorting using butterfly networks (like bitonic sorting). Our theoretical evaluation, and the experiments conducted, show that following the guidelines of the method proposed improve the performance of bitonic sorting also outperforming other algorithms.

This paper is organized as follows. Section 2 discusses related works. Section 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. Section 5 describes the new function to map BSN on GPU we propose. Sections 6 and 7 presents the results obtained in testing the different solutions on synthetic and real dataset. Section 8 presents the conclusions and discusses how to evolve in this research activity.

2. Related work

In the past, many authors presented bitonic sorting networks on GPUs (e.g., Govindaraju, Gray, Kumar, & Manocha, 2006), but the hardware they use belongs to previous generations of GPUs, which does not offer the same level of programmability of the current ones.

Since most sorting algorithms are memory-bound, it is still a challenge to design efficient sorting methods on GPUs.

Purcell, Donner, Cammarano, Jensen, and Hanrahan (2003) present an implementation of bitonic merge sort on GPUs based on an original idea presented by Kapasi et al. (2000). Authors apply their approach to sort photons into a spatial data structure providing an efficient search mechanism for GPU-based photon mapping. Comparator stages are entirely realized in the fragment units,[1] including arithmetic, logical and texture operations. Authors report their implementation to be compute-bound rather than bandwidth-bound, and they achieve a throughput far below the theoretical optimum of the target architecture.

In (Kipfer, Segal, & Westermann, 2004, 2005) it is shown an improved version of the bitonic sort as well as an odd-even merge sort. They present an improved bitonic sort routine that achieves a performance gain by minimizing both the number of instructions executed in the fragment program and the number of texture operations.

Greß and Zachmann (2006) present an approach to parallel sort on stream processing architectures based on an adaptive bitonic sorting (Bilardi & Nicolau, 1986). They present 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, Raghuvanshi, and Manocha (2005) implement sorting as the main computational component for histogram approximation. This solution is based on the periodic balanced sorting network method by Dowd, Perl, Rudolph, and Saks (1989). In order to achieve high computational performance on the GPUs, they used a sorting network based algorithm,

[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.

and each stage is computed using rasterization. Later, they presented a hybrid bitonic–radix sort that is able to sort vast quantities of data, called GPUTeraSort (Govindaraju et al., 2006). This algorithm was proposed to sort record contained in databases using a GPU. This approach uses the data and task parallelism to perform memory-intensive and compute-intensive tasks on GPU, while the CPU is used to perform I/O and resource management.

Cederman and Tsigas (2008) show that 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 up and as a result two new subsequences are created that can be sorted independently by each thread block. The conducted experimental evaluation point out the superiority of GPU–Quicksort over other GPU-based sorting algorithms.

Recently, Sengupta et al. (2007) present a Radixsort and a Quicksort implementation based on segmented scan primitives. Authors presented new approaches to implement several classic applications using this primitives, and show that this primitives are an excellent match for a broad set of problems on parallel hardware.

3. Application to data indexing

Large-scale and distributed applications in information retrieval such as crawling, indexing, and query processing have to exploit the computational power of novel computing architectures to keep up with the exponential growth in Web content. In this paper, we focus our attention on a core phase of one of the main components 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 (Baeza-Yates, Castillo, Junqueira, Plachouras, & Silvestri, 2007). Records represent distinct occurrences of each term in each distinct document. Sorting efficiently these records using a good balance of memory and disk exploitation, is very challenging. In the last years it has been shown that sort-based approaches (Witten, Moffat, & Bell, 1999), or single-pass algorithms (Lester, 2005), are efficient in several scenarios, and in particular where indexing of a large amount of data has to be performed with limited resources. A sort-based approach first makes a pass through the collection assembling all termID–docID pairs. Then, it sorts the pairs with termID as primary key and docID as the secondary key. Finally, it organizes the docIDs for each termID 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 resort to use an external sorting algorithm (Manning, Raghavan, & Schütze, 2008). The main requirement of such algorithm is the minimization of the number of random disk seeks during sorting. A possible approach is Blocked Sort-Based Indexing (BSBI). BSBI works by segmenting a 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 blocks 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. Differently from BSBI where all termID–docID pairs are collected and then sorted, in SPIMI each postings list grows dynamically. 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 blocks 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. All in all, in both indexing methods we mention, sorting is a core step: BSBI sorts the termID–docID pairs of all parts in memory, SPIMI sorts the terms to facilitate the final merging step (Manning et al., 2008).

4. Key aspects of manycore program design

GPUs have been originally designed to execute geometric transformations that generate a data stream of pixels to be displayed.

In general, a program is processed by a GPU by taking as input a stream of data to be distributed among different threads running on multiple SIMD processors. Each thread processes its own portion of data stream, generates the results and writes them back to the main memory.

4.1. The SIMD architecture

SIMD machines, also knows as processor-array machines, basically consists of an array of execution units (EUs) connected together by a network (Kumar, 2002). 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

Please cite this article in press as: Capannini, G., et al. Sorting on GPUs for large scale datasets: A thorough comparison. *Information Processing and Management* (2011), doi:10.1016/j.ipm.2010.11.010

array and handles any control flow or serial computation that cannot be parallelized. Execution units, thus, can be individually disabled to allow conditional branches to be executed. Although SIMD machines are very effective for certain classes of problems, the architecture is specifically tailored for compute-intensive jobs. For this reason it is usually quite "inflexible" on some classes of problems.

4.2. The stream programming model

A stream program (Khailany, Dally, Kapasi, & Mattson, 2001) organizes data as streams and expresses all computation as kernels. A stream is a sequence of homogeneous 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 should usually be arranged in a way to increase the amount of parallel instructions executed by the operation itself. Moreover, they should not access arbitrary memory locations. To avoid expensive concurrent memory access operations they should work on each stream element independently. If all the above constraints are satisfied, each element of the input stream can be processed simultaneously allowing kernels to exhibit a large amount of data parallelism. Task parallelism, instead, can be obtained by allowing kernels to simultaneously access elements from the stream, this can only be accomplished if elements are opportunely arranged in main memory to disallow concurrent accesses to overlapping memory areas. Furthermore, other important features shared by all efficient stream-based applications are: elements are read and computed once from memory, and applications perform a high number of arithmetic operations per memory reference, i.e. applications are compute-intensive.

4.3. K-model: a many-core stream-based computational model

K-model has been designed to model all the main peculiarities of the novel generation of stream multiprocessors (Capannini et al., 2010). It is not the main goal of this paper to present in details *K*-model. Instead, we report, briefly, the main peculiarities we need in order to understand, discuss, and compare our approaches.

K-model consists of a computer with an array of k scalar execution units linked to a single instruction unit. The memory hierarchy consists of an external memory and a local memory made of a set of private registers, and a shared memory of σ locations equally divided into k parallel modules.

According to *K*-model, each kernel of the algorithm is evaluated by measuring the *time complexity*, the *computational complexity*, and the *number of memory transactions*, related to the computation of its stream elements. In order to compute the overall algorithm's complexity the three complexities are managed separately and, the complexity of a kernel is obtained by multiplying the complexity of a stream element by the total number of elements in a stream.

The *time complexity* is defined as the sum of the latencies of each instruction an algorithm performs. It can be thought of as the parallel complexity of the algorithm assuming a collection of k scalar processors. *K*-model evaluates the latency of a data-access instruction proportionally to the level of contention it generates. Whenever an instruction addresses a shared memory location with no bank conflict or a register, the latency of the instruction is 1. Otherwise, the latency of a data-access instruction corresponds to the highest number of requests involving one of the k memory banks. Regarding arithmetic instructions, their latency has unitary cost.

The *computational complexity* is defined as the classical sequential complexity assuming we are simulating the execution of the algorithm on a serial RAM. If the result obtained by dividing the computational complexity by the time complexity, is close to k, it means that the majority of the k computational elements are simultaneously working, and the designed algorithm is efficient.

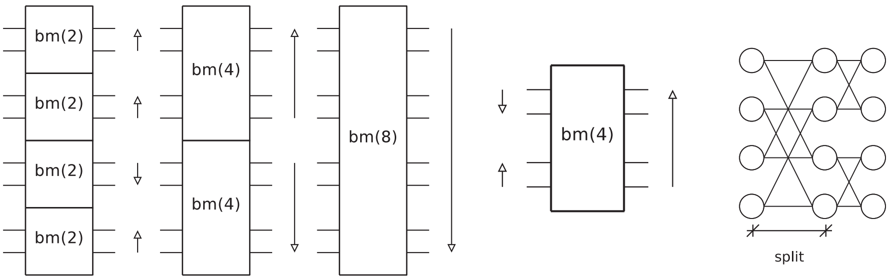
To evaluate the *number of memory transactions*, we need to take into account the data-transfers from/to the off-chip memory. To minimize this quantity, off-chip memory accesses are to be "*coalesced*" into a unique memory transaction. In other words accesses have to be constrained to refer to locations lying in the same size-k segment for each memory transaction.

5. *K*-Model-based bitonic sorting network

A sorting network is a mathematical model of a sorting algorithm that is made up of a network of wires and comparator modules. The sequence of comparisons thus does not depend on the order with which the data is presented. 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 *K*-model requires.

In particular, BSN is based on repeatedly merging two bitonic sequences[2] to form a larger bitonic sequence (Knuth, 1973). On each bitonic sequence the bitonic split operation is applied. After the split operation, the input sequence is divided into two bitonic sequences such that all the elements of one sequence are smaller than all the elements of the second one. Each item on the first half of the sequence, and the item in the same relative position in the second half are compared and exchanged if needed. Shorter bitonic sequences are obtained by recursively applying a binary merge operation to the given bitonic sequence

[2] A bitonic sequence is composed of two sub-sequences, one monotonically non-decreasing and the other monotonically non-increasing.



(a) A BSN made up of Bitonic Merging Network (b) Bitonic Merging Network

Fig. 1. (a) Structure of a BSN of size $n = 8$. With bm(x) we denote bitonic merging networks of size x. The arrows indicate the monotonic ordered sequence. (b) Butterfly structure of a bitonic merge network of size $n = 4$.

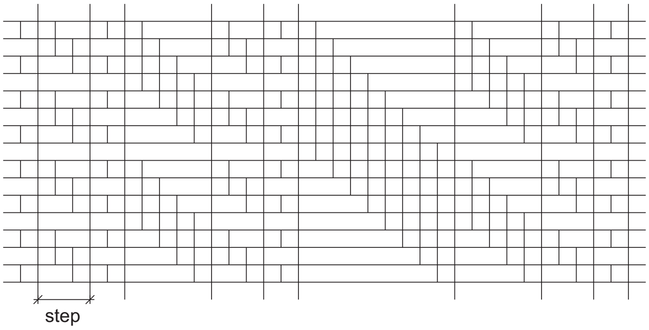


Fig. 2. Example of BSN for 16 elements. Each comparison is represented with a vertical line that link two elements, which are represented with horizontal lines. Each step of the sort process is completed when all comparisons involved are computed.

(Batcher, 1968). The recursion ends and the sequence is sorted when the input of the merge operation is reduced to singleton sequences. Fig. 1 shows graphically the various stages described above.

The pseudo-code of a sequential BSN algorithm is shown in Algorithm 1. Fig. 2, instead, shows an instantiation of a fan-in 16 BSN.

Algorithm 1. BitonicSort (A)

```
$n \leftarrow |A|$
for $s = 1$ to $\log n$ do
  for $c = s - 1$ to $0$ step $-1$ do
    for $r = 0$ to $n - 1$ do
      If $\frac{r}{2^c} \equiv \frac{r}{2^s} (mod2) \wedge A[r] > A[r \oplus 2^c]$ then SWAP $(A[r], A[r \oplus 2^c])$
```

To design our sorting algorithm in the stream programming model, we start from the original parallel BSN formulation (Batcher, 1968) and we extend it to follow the K-model guidelines. In particular, the main aspect to consider is to define an efficient schema for mapping items into stream elements. Such mapping should be done in order to perform all the comparisons involved in the BSN within a kernel. The structure of the network, and the constraint of the programming model, indeed, disallow the entire computation to be performed within one stream. Firstly, the number of elements to process by the merging step increases constantly (as it is shown in Fig. 1). On the other hand, due to the unpredictability of their execution order, the stream programming model requires the stream elements to be "independently computable". In other words, each item has to be included into at most one stream element, see Fig. 3. Following these constraints, the set of items would then be partitioned (and successively mapped) into fewer but bigger parts (Fig. 4). For example, referring to the last merging step of a BSN all the items would be mapped into a unique part. This is clearly non admissible since the architectural constraints limit the number of items that can be stored locally (i.e. the size of a stream element). In particular in the K-model, such limit is fixed by the σ parameter, i.e. the amount of memory available for each stream element.

In our solution we define different partition depending on which step of the BSN we are. Each partitioning induces a different stream. Each stream, in turn, needs to be computed by a specific kernel that efficiently exploits the characteristic of the stream processor.

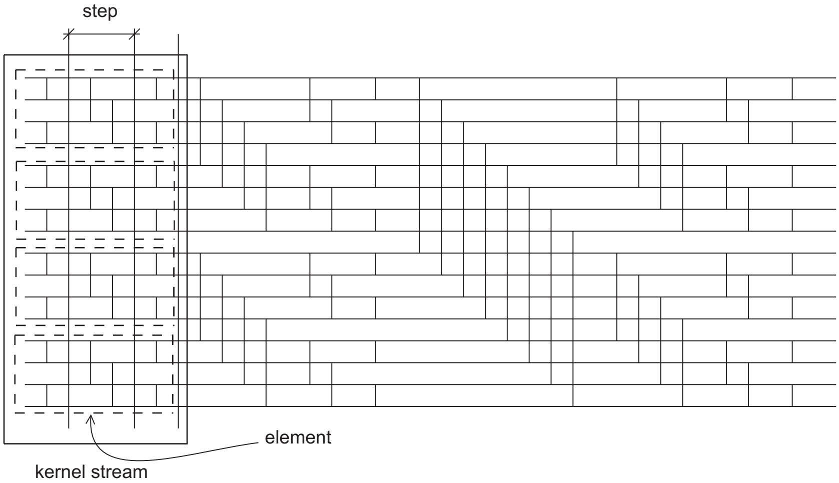


Fig. 3. Example of a kernel stream comprising more steps of a BSN. The subset of items composing each element must perform comparison only inside itself.

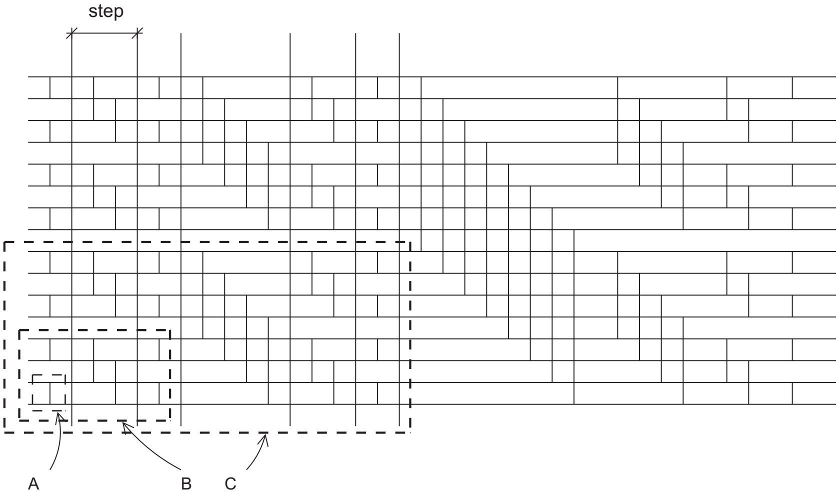


Fig. 4. Increasing the number of steps covered by a partition, the number of items included doubles. A, B and C are partitions respectively for local memory of 2, 4 and 8 locations.

Since each kernel invocation implies a communication phase, such mapping should be done in order to reduce the communication overhead. Specifically, this overhead is generated whenever a processor begins or ends the execution of a new stream element. In those cases, the processor needs to flush the results of the previous computation stored in the local memory, and then to fetch the new data from the off-chip memory. Taking into account the *K*-model rule, depending on the pattern used to access the off-chip memory, the "latency" of such transfer can increase up to *k* times translating in an increase of up to one order of magnitude when measured on the real architecture.

Resuming, in order to maintain the communication overhead as small as possible, our goals are: (i) to minimize the number of communications between the on-chip memory and the off-chip one, (ii) to maximize the bandwidth with which such communications are done. Interestingly, the sequential version of the bitonic network algorithm exposes a pattern made up of repeated comparisons. It turns out that this core set of operations can be then optimally reorganized in order to meet the two goals above described.

Let us describe how a generic bitonic network sorting designed for an array A of $n = 2^i$ items, with $i \in \mathbb{N}^+$, can be realized in *K*-model.

In order to avoid any synchronization, we segment the n items in such a way each part contains all the items to perform some steps without accessing the items in other parts. Since the items associated with each stream element have to be temporarily stored in the on-chip memory, the number of items per part is bounded by the size of such memory. In the follow, we show the relation between the number of items per part, and the number of steps each kernel can perform. This relation emerges from the analysis of Algorithm 1.

Briefly, to know how many steps can be included in the run of a partition, we have to count how many distinct values the variable c can assume. First of all, by the term *step* we refer to the comparisons performed in the loop at line 4 of Algorithm 1. Furthermore, let c and s be the variables specified in Algorithm 1, the notation $step_{\bar{s},\bar{c}}$ represents the step performed when $c = \bar{c}$ and $s = \bar{s}$. At each step, the indexes of the two items involved in the comparison operation are expressed as a function of the variable c.

Claim 1. *Within a $step_{s,c}$ two elements are compared, if and only if, the binary representation of their relative indexes differ only by the cth bit.*

Proof. By definition of bitwise $\oplus$ the operation $r \oplus 2^c$, invoked at line 5, corresponds to flipping the cth bit of r, in its binary representation. □

The claim above gives a condition on the elements of the array A involved in each comparison of a step. Given an element $A[r]$ at position r this is compared with the one whose position is obtained by fixing all the bits in the binary representation of r but the cth one which is, instead, negated. The previous claim can be extended to define what are the bits flipped to perform the comparisons done within a generic number of consecutive steps, namely k, called *k-superstep*.[3] This definition straightforwardly follows from Algorithm 1, and it is divided in two cases, specifically for $k \leqslant s$ and $k > s$.

Definition 1. (Γ-**sequence**) Within the k-superstep starting at $step_{s,c}$, with $1 \leqslant k \leqslant s$, the sequence Γ of bit positions that Algorithm 1 flips when it performs the comparisons is defined as follows:

$$\Gamma = \begin{cases} \Gamma_{\geqslant} = [c, (c-k) & \text{if } c > k \\ \Gamma_{<} = [s, (s-k+c+1) \cup [c, 0] & \text{otherwise} \end{cases}$$

The sequence basically consists of the enumeration of the different values taken by c in the k-superstep considered. It is worth being noted that the values assigned to c in the k steps are distinct because of the initial condition $k \leqslant s$. Now, let us consider the behavior of Algorithm 1 when $s < k$. In particular, let us restrict to the definition of Γ in steps from $step_{1,0}$ to $step_{k,0}$. Since c is bounded from above by $s < k$, for each considered step c can only assume values in the range $(k,0]$. Note that, in this case, the number of steps covered by flipping the bit positions contained in the sequence is $\frac{1}{2}k(k+1)$, instead of k.

Definition 2. (Γ_0-**sequence**) The sequence $\Gamma_0 = (k,0]$ corresponds to bit positions that Algorithm 1 flips when it performs the comparisons within the $\frac{1}{2}k(k+1)$ steps starting from $step_{1,0}$.

To resume, given a generic element $A[r]$, with $0 \leqslant r < n$, and considering a superstep of the bitonic network, the only bits of r flipped by Algorithm 1 to identify the corresponding elements to compare with are those identified by the sequence Γ of bit positions. Then, bit positions that do not occur in Γ are identical for the elements compared with $A[r]$ in such superstep. By definition of the Γ-sequence, we can retrieve the following claim.

Claim 2. *Let $A[r]$ and $A[q]$ be two elements of A. Given a superstep and its Γ-sequence, $A[r]$ and $A[q]$ belong to the same partition if and only if $\forall i \notin \Gamma \cdot r_{[i]} = q_{[i]}$, where the notation $r_{[i]}$ denotes the ith bit of the binary representation of r.*

From the previous claims, we can also retrieve the size of each partition as function of Γ.

Lemma 1. *Each part is composed by $2^{|\Gamma|}$ items.*

Proof. By induction on the length of Γ. When $|\Gamma| = 1$, an item is compared with only another one, by Claim 1. So each part is made up of two items. For the inductive step, let us consider the next step in the superstep. Each of the previous items is compared with an element not yet occurred, due to the new value of c that implies to flip a new bit position. Since each item forms a new pair to compare, the number of items to include in the part doubles, namely it is $2 \times 2^{|\Gamma|} = 2^{|\Gamma|+1}$. □

From the above lemma, and because each partition covers all the elements of A, it follows directly that

Corollary 1. *The number of parts for covering all the comparisons in the superstep is $2^{\log n - |\Gamma|}$.*

The previous claim can be extended to define the Γ-partition procedure.

Definition 3 (*Γ-partition*). Given a k-superstep, the relative Γ-partition is the set of parts $\mathscr{P} = \{p_i\}$, for $0 \leqslant i < 2^{\log n - |\Gamma|}$ where each part is constructed by means of Algorithm 2.

[3] In the rest of the paper we denote a sequence of integers by putting the greater value on the left of the range. For example, the sequence formed by the elements in $\{i | m \leqslant i < M\}$ is denoted by $(M,m]$.

Algorithm 2. BuildPartition (A, n, k, Γ)

```
/*create a bit-mask corresponding to the fixed $\log n - k$ bits whose positions are not in $\Gamma$ */
$j = 0, m = 0$;
for $b = 0$ to $\log n - 1$ do
  if $b \notin \Gamma$ then $m_{[b]} = i_{[j]}, j = j + 1$;
/*populate the partition using the bit-mask $m$ defined in the previous step */
for $e = 0$ to $2^k - 1$ do
  $j = 0, r = m$;
  for $b = 0$ to $\lceil \log n \rceil - 1$ do
    if $b \in \Gamma$ then $r_{[b]} = e_{[i]}, i = i + 1$;
  $p_i$=$p_i \cup A[r]$;
```

Now, let us make some consideration about the communication overhead discussed above. Each time we perform a stream, the computation is charged of the time spent to fetch all n elements divided among the different parts, then to write them back. In order to minimize this overhead, we need to minimize the number of streams needed to cover all the network, i.e. to maximize the number of steps performed within each partition. Because each Γ-sequence is made up of $2^{|\Gamma|}$ items, see Lemma 1, and in the K-model the data of each part has to fit in the local memory of σ locations, the optimal size for Γ is $\log \sigma$. Then, each Γ-partition forms a stream that feeds an appropriate kernel. Due to the mapping we design, each part is modeled as a bitonic network (see Algorithm 3). It is possible to show that such a modeling allows to always keep the k executors active. At the same time, the contention to access the k on-chip memory banks is balanced. Note that, in the K-model rule, by balancing the contention the latency of the accesses is reduced because the maximum contention is lower.

The pseudo-code in Algorithm 3 discards some side aspects, to focus the main technique. In particular it takes a part (A_p) and the related Γ-sequence (Γ), then performs all due comparisons in-place. The procedure InsAt (N, x, p) inserts the bit x at the position c of the binary representation of N, for example InsAt $(7, \mathbf{0}, 1) = 1\,1\,\mathbf{0}\,1 = 13$. The procedure Compare & Swap performs the comparisons between the argument elements and, if needed, swaps them. Note that, each RUNSTREAMELEMENT execution is free from conditional branch instructions. This is a very important feature for a SIMD algorithm, avoiding, in fact, divergent execution paths that are serialized by the (single) instruction unit of the processor.

Algorithm 3. RunStreamElement (A_p, Γ)

```
for each $id \in [0, k - 1]$ parallel do
$n = \log_2(\sigma)$
for $i = 0$ to $n - 1$ do
  $c = \Gamma[i]$
  for $j = id$ to $n/2 - 1$ step $k$ do
    $p$= InsAt $(j, 0, c)$
    $q$= InsAt $(j, 1, c)$
    Compare & Swap $(A_p[p], A_p[q])$
```

In a previous work we argued that minimizing the number of data-transfers is not enough (Capannini et al., 2009). In particular, in the cache-based model, proposed by Frigo, Leiserson, Prokop, & Ramachandran (1999), the bandwidth needed to replace a cache-line, in the case of cache-miss, is constant. Following the K-model rules (Capannini et al., 2010), the memory bandwidth is fully exploited when simultaneous memory accesses can be coalesced into a single memory transaction. This means that it is possible to reduce the latency of data transfer by reorganizing in the proper manner the accesses to the off-chip memory.

In the rest of the section we will refer a sequence of k consecutive addresses with the term *k-coalesced set*, and we will say that a part, or the associated Γ-sequence, satisfies the *k-coalesced condition* when its values are related only to sets that are k-coalesced. Specifically, for Definition 3, such a Γ-sequence satisfies the *k-coalesced condition* when it contains all the values in the range from 0 to $\log k - 1$.

Let us analyze the k-coalesced condition in the K-model. By definition of Γ-sequence, when we fall into a $\Gamma_<$ case and $c > \log k$, the k-coalesced condition is verified because the Γ-partition accesses to 2^{c+1}-coalesced subsets of positions. When $c \leqslant \log k$, and we are still in the $\Gamma_<$ case, we need to access to longer consecutive sequences of addresses to satisfy the k-coalesced condition. On the other hand when we fall into a $\Gamma_\geqslant$-sequence, no consecutive addresses are included in the relative partitions, because the value 0 cannot be included in such type of sequence, for Definition 1. Eventually, the Γ_0 sequence is composed of a unique sequence of contiguous addresses.

To satisfy the k-coalesced condition for all the generated Γ-partitions, we move some pairs of items from a part of the current partition to another part. The aim is to group in the same memory transaction items having consecutive addresses,

Please cite this article in press as: Capannini, G., et al. Sorting on GPUs for large scale datasets: A thorough comparison. *Information Processing and Management* (2011), doi:10.1016/j.ipm.2010.11.010

whenever we need longer sequences of consecutive memory addresses. To do that, each Γ-sequence is initialized with the values in the range $]\log k, 0]$. Then, the values of c related to the next steps to perform are pushed in the Γ-sequence as far as it contains $\log\sigma$ distinct values.

This operation augments the coalescing-degree of the data transfer, still it forces to remove from Γ some elements related to the next values of c. The best possible communication bandwidth is attained at the cost of decreasing the length of some supersteps. This means to perform more supersteps to cover the whole bitonic network.

6. *K*-Model-based sorting network evaluation

The solution we propose is evaluated theoretically and experimentally by comparing its complexity and performance with those obtained by Cederman & Tsigas (2008) with their version of Quicksort (hereinafter referred to as Quicksort), and the Radixsort based solution proposed by Sengupta et al. (2007) (hereinafter referred to as Radixsort). Quicksort exploits the popularly known divided-and-conquer principle, whereas Radixsort exploits the processing of key digits.

6.1. Theoretical evaluation

BSN The number of steps to perform is $(\log^2 n + \log n)/2$. To estimate the number of *memory transaction* needed to compute a sorting network for an array of size n, we have to count the number of Γ-partitions needed to cover all the network. That means to know how many stream elements are computed, then the number of fetch/flush phases, and the number of memory transactions.

From Definition 2, it follows that the first partition covers the first $(\log^2\sigma + \log\sigma)/2$ steps.

Let us call $stage_s$ the loop at line 2 of Algorithm 1. In the remaining steps $s > \sigma$, $\log n - \log\sigma$ stages remain, and each of them has the last Γ-partition covering $\log\sigma$ steps. On the other hand the $s - \log\sigma$ steps are performed with partitions covering $\log(\sigma/k)$ steps. Resuming, the number of partitions needed to cover all the network is

$$1 + \sum_{s=\log\sigma+1}^{\log n}\left(\left\lceil\frac{s-\log\sigma}{\log(\sigma/k)}\right\rceil + 1\right) = O\left(\frac{\log^2 n}{\log k}\right)$$

Since, each element fetches and flushes only coalesced subset of elements, the number of transactions is

$$O\left(\frac{n}{k}\cdot\frac{\log^2 n}{\log k}\right)$$

The *time complexity* is

$$O\left(\frac{n\log^2 n}{k}\right)$$

as it is obtained by Algorithm 3 which equally spreads the contentions among the k memory banks and maintains active all elements.

Regarding the *computational complexity* it is known and it is

$$O(n\log^2 n)$$

Quicksort It splits the computation in $\log n$ steps. For each step it performs three kernels. In the first one, it equally splits the input and counts the number of elements greater than the pivot, and the number of the elements smaller than the pivot. In the second, it performs twice a parallel prefix sum of the two set of counters in order to know the position where to write the elements previously scanned. In the final kernel, it accesses to the data in the same manner that in the first kernel, but it writes the elements to the two opposite heads of an auxiliary array beginning at the positions calculated in the previous kernel.

The first kernel coalesces the access to the elements and, since the blocks are equally sized, also the computation is balanced. Then the counters are flushed, and the second kernel starts. Supposing that $n/k < \sigma$, each prefix sum can be computed within a unique stream element. Consequently, for each prefix sum we need n/k^2 *memory transactions* to read n/k counters. The *time complexity* is logarithmic in the number of elements, on the contrary the *computational complexity* is linear. Last kernel is similar to the first one, except for flushing the data into the auxiliary array. In particular, because each thread accesses to consecutive memory locations, the main part of the requests is not coalesced, requesting one memory transaction per element.

The following table contains the evaluation of the three type of kernel in the K-model. In order to compute the complexity of the whole algorithm, the sum of such formulas have to be multiplied by $\log n$:

	Memory transactions	Time complexity	Computational complexity
kernel #1	$n/k + 2$	n/k	n
kernel #2	$4n/k^2$	$4 \cdot \log \frac{n}{k}$	$2n/k$
kernel #3	$n/k + n$	n/k	n
OVERALL	$O(n \log n)$	$O(\frac{n}{k} \log n)$	$O(n \log n)$

Radixsort. It divides the sequence of n items to sort into h-sized subsets that are assigned to $p = \lceil n/h \rceil$ blocks. Radixsort reduces the data transfer overhead exploiting the on-chip memory to locally sort data by the current radix-2^b digit. Since global synchronization is required between two consecutive phases, each phase consists of several separate parallel kernel invocations. Firstly, each block loads and sorts its subset in on-chip memory using b iterations of binary-split. Then, the blocks write their 2^b-entry digit histogram to global memory, and perform a prefix sum over the $p \times 2^b$ histogram table to compute the correct output position of the sorted data. However, consecutive elements in the subset may be stored into very distant locations, so coalescing might not occur. This sacrifices the bandwidth improvement available, which in practice can be as high as a factor of 10.

In their experiments, the authors obtained the best performance by empirically fixing $b = 4$ and $h = 1024$. That means each stream is made up of $\lceil n/1024 \rceil$ elements. Once the computation of a stream element ends, the copies of the sorted items may access up to $O(2^b)$ non-consecutive positions. Finally, considering 32-bit words, we have $32/b$ kernels to perform. This leads to formalize the total number of *memory transactions* performed as follows:

$$O\left(\frac{32}{b} \cdot \frac{n}{h} \cdot 2^b\right)$$

Regarding *computational* and *time complexity*, Radixsort does not use expensive patterns and it does not increase the contention in accessing shared memory banks. Therefore, the *time complexity* is given, by $b \cdot n/k$, and the *computational complexity* is linear with the number of input-elements, i.e. $b \cdot n$.

6.2. Experimental evaluation

The experimental evaluation is conducted by considering the execution time and amount of memory required by running BSN, Quicksort and Radixsort on different problem size. The different solutions have been implemented and tested on an Ubuntu Linux Desktop with an Nvidia 8800GT, that is a device equipped with 14 SIMD processors, and 511 MB of external memory. The compiler used is the one provided with the Compute Unified Device Architecture (CUDA) SDK 2.1 (NVIDIA, 2008). Even if the CUDA SDK is "restricted" to Nvidia products, it is conform to the K-model. To obtain stable result, for each distribution, 20 different arrays were generated.

According to Helman, Bader, & JáJá (1995), a finer evaluation of sorting algorithms should be done on arrays generated by using different distributions. We generate the input array according to uniform, gaussian and Zipfian distributions. We also consider the special case of sorting an all-zero array.[4] These tests highlight the advantages and the disadvantages of the different approach tested. The computation of Radixsort and BSN is based on a fixed schema that uses the same number of steps for all type of input dataset; on the contrary, Quicksort follows a divide-and-conquer strategy, so as to perform a varying number of steps depending on the sequence of recursive procedures invoked. The benefits of the last approach are highlighted in the all-zero results.

The experiments confirm our theoretical ones. Fig. 5 shows the means, the standard deviation, and the maximum and the minimum of the execution time obtained in the conducted tests by our solution, the Cederman & Tsigas (2008) Quicksort, and Sengupta et al. (2007) Radixsort, respectively. Radixsort results to be the fastest and this is mainly due to its complexity in terms of the number of memory transactions that it needs, see Table 1.

This confirms our assumption that the number of memory transactions is dominant w.r.t the other two complexity measures, i.e. *computational* and *time*. This is particularly true whenever the cost of each operation is small if compared to the number of memory access operations (like in the case of data-intensive algorithms).

Radixsort, in fact, has a $O(n)$ number of memory transactions, that is smaller than $O(n \log^2 n/k \log k)$ of the BSN and than $O(n \log n/k)$ of the Quicksort.

Considering the specifications of real architectures, which related to the parameter k of the K-model, and considering the capacity of the external memory available on real devices (order of Gigabytes), Quicksort results to be the least performing method analyzed, see Fig. 6.

On the other hand, our BSN approach is comparable to Radixsort and it is always faster than Quicksort, mainly because the mapping function proposed allows the full exploitation of the available memory bandwidth.

[4] All elements are initialized equal to 0.

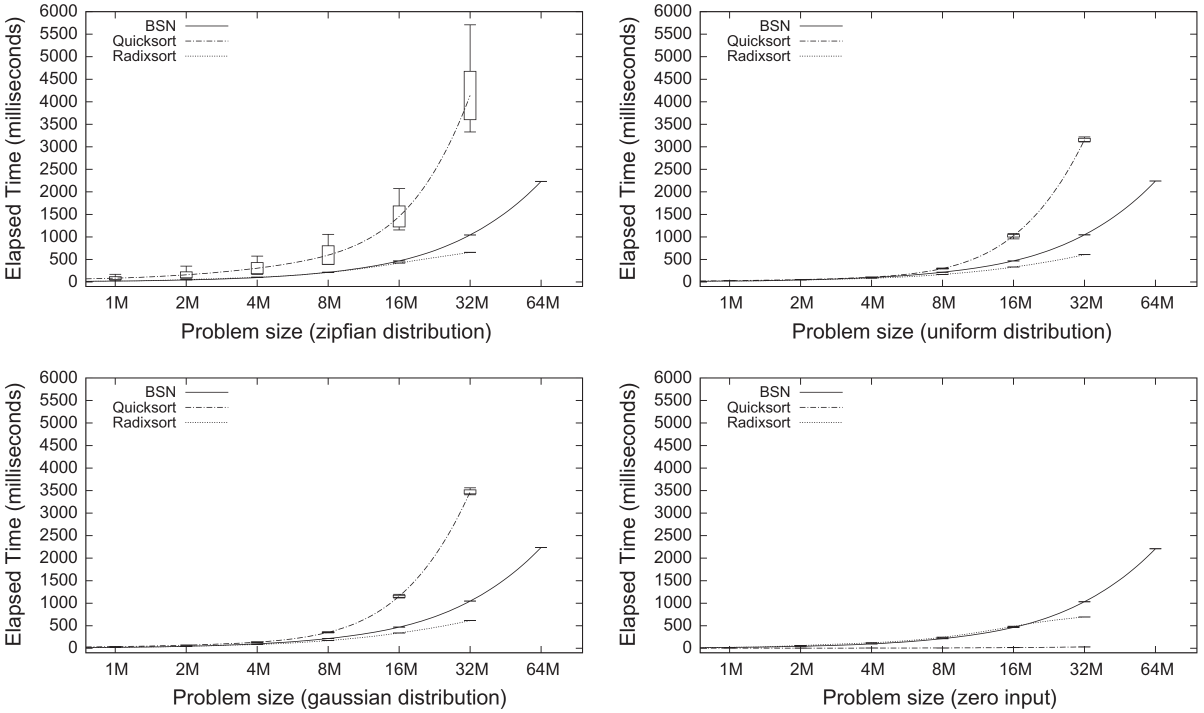


Fig. 5. Elapsed sorting time for varying input size. We represent variance, maximum, and minimum of elapsed times by using candlesticks.

Table 1
Performance of BSN, Radixsort and Quicksort in terms of number of memory transactions, memory contention, and number of divergent paths. Results are related to uniform distribution. "n.a." means that computation is not runnable for lack of device memory space.

Problem size		Memory transactions	Memory contention	Divergent paths
2^{20}	Bitonicsort	796,800	34,350	0
	Quicksort	4,446,802	123,437	272,904
	Radixsort	965,791	132,652	29,399
2^{22}	Bitonicsort	4,119,680	151,592	0
	Quicksort	18,438,423	379,967	1,126,038
	Radixsort	3,862,644	520,656	122,667
2^{24}	Bitonicsort	20,223,360	666,044	0
	Quicksort	85,843,422	1,379,155	1,126,038
	Radixsort	15,447,561	2,081,467	492,016
2^{26}	Bitonicsort	101,866,786	2,912,926	0
	Quicksort	n.a.	n.a.	n.a.
	Radixsort	n.a.	n.a.	n.a.

A last word has to be spent regarding the memory consumption of the methods. Quicksort and Radixsort are not able to sort large arrays, as it is pointed out, see Table 1. Being an in-place solution, in fact, BSN can thus devote all the available memory to store the dataset. This has to be carefully considered since sorting large datasets will require less passes than the other solutions. They need, in fact, to split the sorting process in more steps, then to merge the partial results. Moreover, merge operation may require further transfers for copying the partial results to the device memory if this operation is performed on manycores. Otherwise, CPU can perform the merging step, but exploiting a bandwidth lower than the GPU's one.

Table 1 measures the memory contention, and the number of divergent paths. The first value measures the overhead due to the contention on the on-chip memory banks as *K*-model expects. The second value measures how many times threads of the multiprocessors can not work simultaneously. These two last metrics together show the efficiency of the algorithms tested. Keeping low both values corresponds to a better exploitation of the inner parallelism of the SIMD processor. All the memory banks and all the computational, in fact, are allowed to work simultaneously.

Moreover, since direct manipulation of the sorting keys as in Radixsort is not always allowed, it is important to provide a better analysis of the comparison-based sorting algorithms tested. Due to the in-place feature and due to the best

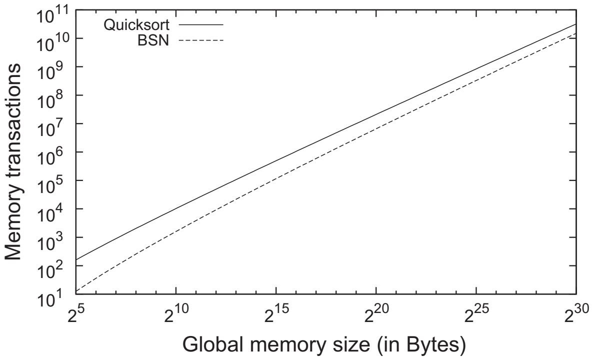


Fig. 6. Theoretical number of memory transactions for BSN and Quicksort considering the specifications of real architectures, i.e. $k \simeq 16$, and the capacity of the external memory available as order of Gigabytes.

performance resulting from the test conducted, BSN seems more preferable than Quicksort. Furthermore, BSN exposes lower variance in the resulting times, it is equal to zero in practice. On the contrary, depending on the distribution of the input data, Quicksort's times are affected by great variance. Probably, this is due to how the divide-et-impera tree grows depending on the pivot chosen, and on the rest of the input. For example, on system based on multi-devices (i.e. more than one GPU), this result increases the overhead of synchronization among the set of available devices.

7. Indexing a real dataset

We present experiments showing the performance of a *Blocked Sort-Based Indexer* (Manning et al., 2008) (BSBI), modified with a GPU-based sort step, to index six different collections of Web documents. Each collection has been obtained by crawling a portion of the web. The biggest collection contains 26 million documents and each document has an average size of 200 KB. The first column of Table 2 reports the different sizes of each collection tested.

Our goal is to show that using a GPU-based solution we can speed-up the indexing process due to, mainly, two reasons: (i) GPU-based sort is faster than CPUs-based one, (ii) CPUs and GPUs can be pipelined to obtain synergistic effort to efficiently exploit all the available resources. We have already shown, in previous section, that GPU-based sorting is more efficient than CPU-based sorting. On the other hand, to show that we can synergistically exploit a pipelined CPU/GPU indexer, we design in this section a novel *blocked* indexing schema where each block of pages is firstly parsed by the CPU to generate a run of termID–docID pairs and then, the generated run is sorted by the GPU while the CPU carries out a new parsing step.

Fig. 7 shows the three different indexing algorithms we have tested. $Index_{cpu}$ is the classical CPU-based BSBI algorithm that takes as input a dataset D and generate the index f_{out}. $Index_{gpu}$ is the GPU-based counterpart, obtained by replacing the CPU-based sorting algorithm with our BSN. $Index_{gpu^+}$, instead, is our pipelined version of a BSBI indexer. We shall explain in more details this last algorithm in the remaining part of the section.

We have already pointed out in the introduction of this paper that GPUs are considerably more powerful from a computational point of view. The major drawback of GPU architectures, though, is represented by a limited memory size (511 MB in our experimental setting) and, in our case, this limits the maximum number of pairs we can generate in each run (blocks are referred as B in Fig. 7). This aspect could affect the overall indexing time. In particular the smaller the block size the greater the number of intermediate indexes to merge. This number affects the merge algorithm complexity of a logarithmic factor (see below). However, the complexity of merging is dominated by the latency of data transferring from/to disk. Since the amount of data to transfer (namely the *termId–docId* pairs) is the same in both the CPU-based and the GPU-based solution, we expect the time needed by the merging phase to be similar as well.

Table 2 shows how indexing time varies with the size of the collection processed. We consider the two main components of the computational time, namely "Parse+[Gpu]Sort" and "Merge". "Parse+[Gpu]Sort" is the time needed to create the n

Table 2
Comparison of the computational times referring the different approach for the indexer architecture. Times are in seconds.

$\lvert D\rvert_{(\times 10^6)}$	$Index_{cpu}$		$Index_{gpu}$	$Index_{gpu^+}$	$Index_{gpu,gpu^+}$
	Parse+Sort	Merge	Parse+GpuSort	Parse+GpuSort	Merge
0.81	42.96	16.87	12.61	9.02	19.87
1.63	94.51	35.50	24.99	20.27	42.07
3.27	185.24	88.75	52.04	38.09	100.90
6.55	394.57	193.47	107.98	73.96	244.68
13.10	783.56	482.38	221.36	151.59	599.84
26.21	1676.82	1089.26	456.49	346.88	1291.78

```
$\text{Index}_{cpu}$ $(D)$ ::
  $n \leftarrow 0$
  while $D \neq \emptyset$ do
      $n \leftarrow n + 1$
      $B \leftarrow$ NextBlock()
      Parse($B$)
      Sort($B$)
      toDisk($B$, $f_n$)
      $D \leftarrow D \setminus B$
  $f_{out} \leftarrow$ Merge($f_1, \ldots, f_n$)

$\text{Index}_{gpu}$ $(D)$ ::
  $n \leftarrow 0$
  while $D \neq \emptyset$ do
      $n \leftarrow n + 1$
      $B \leftarrow$ NextBlock()
      Parse($B$)
      GpuSort($B$)
      toDisk($B$, $f_n$)
      $D \leftarrow D \setminus B$
  $f_{out} \leftarrow$ Merge($f_1, \ldots, f_n$)

$\text{Index}_{gpu^+}$ $(D)$ ::
  ○ Cpu($D$) ::
      while $D \neq \emptyset$ do
          $B \leftarrow$ NextBlock()
          Parse($B$)
          SyncSend($B$, toGpu)
          $D \leftarrow D \setminus B$
      SyncSend($\emptyset$, toGpu)
      Recv($n$, fromGpu)
      $f_{out} \leftarrow$ Merge($f_1, \ldots, f_n$)
  ○ Gpu() ::
      $n \leftarrow 0$
      while $B \neq \emptyset$ do
          Recv($B$, fromCpu)
          if $B \neq \emptyset$ then
              $n \leftarrow n + 1$
              GpuSort($B$)
              toDisk($B$, $f_n$)
      SyncSend($n$, toCpu)
```

Fig. 7. Description of the different versions of the indexer architecture.

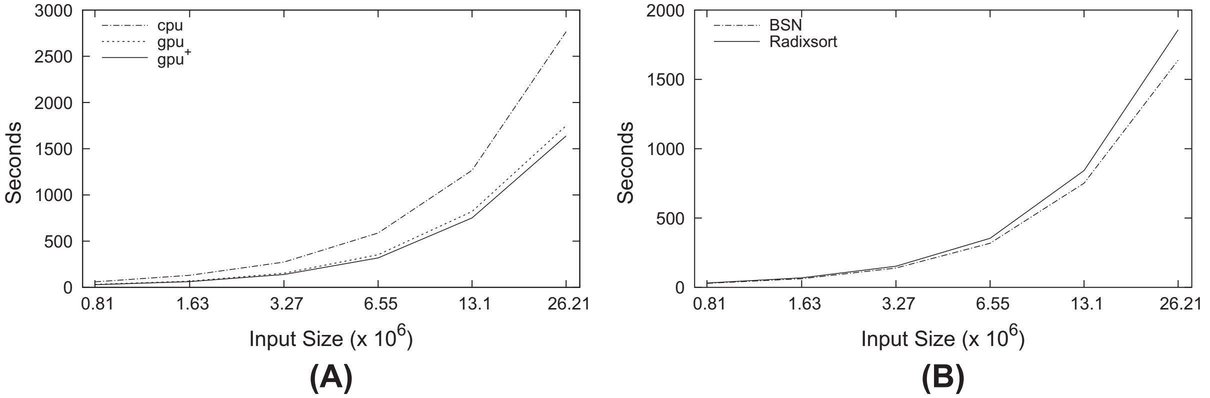


Fig. 8. (A) Computational time for different indexer architectures. (B) Computational time measured by performing $Index_{gpu^+}$ with different GpuSort() solutions.

sorted files. Instead, "Merge" represents the time spent in merging sorted files. Concerning the two GPU-based methods, "Merge" column of Table 2 is shown only once since the solutions perform the same merge procedure on exactly the same input.

Fig. 8A shows the overall computational time in the case of the three algorithms we have tested. First of all, we point out the clear benefit, in terms of efficiency, of using a GPU-based sorting step in a BSBI indexer. Indeed, the sorting phase does not represent a bottleneck anymore of the indexing process. In contrast, in GPU-based indexing, the computational time is dominated by the latency of the merging phase.

Let us consider the pair building, and the merging steps of algorithms in Fig. 7. The complexity of the former is linear in the size of B, i.e. $O(n|B|)$. On the other hand, the complexity of the merging phase is equal to $O(n|B|\log n)$, because the n-way merger is implemented by using a heap of size n, i.e. the number of runs to merge. At each step, Merge extracts the smallest termID–docID pair [5] (t,d) from the heap and a new element, taken from the same run from (t,d) was extracted, is inserted. Extraction takes constant time. Insertion is logarithmic in the size of the heap. Therefore, the complexity of the merging phase

[5] With respect to a precedence relation defined as follows: $(t_1,d_1) < (t_2,d_2)$ iff $(t_1 < t_2) \vee ((t_1 = t_2) \wedge (d_1 < d_2))$.

is equal to the number of pairs collected multiplied by $\log n$, i.e. $O(n|B|\log n)$. In other words, the complexity of merging is higher than the complexity of the parsing step.

Fig. 8B shows the execution times obtained by different versions of $Index_{gpu^+}$ by varying the GPU-based sorting algorithm[6]. As said in the previous section, Radixsort is faster than BSN. However, due to its space requirements we are forced to use a number of blocks which is twice as big as the number of blocks required by the BSN solution. Therefore, the merging phase is more expensive since the number of runs to merge doubles. This result confirms the consideration in Section 6. Namely, space efficiency has to be carefully considered for sorting large datasets. Inefficient solutions need, in fact, to split the sorting process in more steps, then to merge the partial results.

8. Conclusion and future work

This paper focuses on using GPUs as co-processors for sorting. We propose a new mapping of bitonic sorting network on GPUs. We started from its traditional algorithm, and we extend it to adapt to our target architecture. Bitonic sorting network is one of the fastest sorting networks to run on parallel architectures. The proposed solution was evaluated both theoretically and experimentally, by comparing its complexity and performance with those obtained by two others state-of-the-art solutions (Quicksort and Radixsort).

The theoretical algorithms complexity was evaluated by using *K*-model a novel computational model to specifically designed to capture important aspects in stream processing architectures.

The experimentally evaluation was conducted using input streams generated according to different distributions. This kind of experiments highlighted the behavior of the analyzed algorithms particularly regarding the variance of the performance obtained for different data distributions. Regarding the execution time, our solution is outperformed by Radixsort for input arrays made up of up to *8* Million of integers. On the other hand, our solution requires one half the memory used by the other ones and it is able to efficiently exploit the high bandwidth memory interface available on GPUs making it viable for sorting large amounts of data.

Accordingly the results of the experiments, we have chosen bitonic sorting network and Radixsort to develop an indexer prototype to evaluate the possibility of using an hybrid CPU–GPU indexer in the real world. The time results obtained by indexing tests are promising and suggest to move also others computational intensive procedure on the GPUs.

References

Al-Hajery, M.Z., & Batcher, K.E. (1993). Multicast bitonic network. In *SPDP*. pp. 320–326.

Baeza-Yates, R., Castillo, C., Junqueira, F., Plachouras, & V., Silvestri, F. (2007). Challenges on distributed web retrieval. In *ICDE*. IEEE.

Barroso, L.A., & Hölzle, U. (2009). *The datacenter as a computer: an introduction to the design of warehouse-scale machines*. Morgan & Claypool. <http://dx.doi.org/10.2200/S00193ED1V01Y200905CAC006>.

Batcher, K. E. (1968). Sorting networks and their applications. In *AFIPS '68*. New York, NY, USA: ACM.

Bilardi, G., & Nicolau, A. (1986). *Adaptive bitonic sorting: an optimal parallel algorithm for shared memory machines*. Tech. Rep. Cornell University, Ithaca, NY, USA.

Bingsheng, H., Wenbin, F., Qiong, L., Naga, K.G., & Tuyong, W. (2008). Mars: a MapReduce framework on graphics processors. In *PACT '08*. ACM, New York, NY, USA. <http://dx.doi.org/10.1145/1454115.1454152>.

Capannini, G., Silvestri, F., & Baraglia, R. (2010). *K-model: a new computational model for stream processors*. Tech. Rep. CNR–ISTI, Via Moruzzi, 1, Pisa, Italy.

Capannini, G., Silvestri, F., Baraglia, R., & Nardini, F.M. (2009). Sorting using bitonic network with CUDA. In *7th Workshop on LSDS-IR*. <http://lsdsir09.isti.cnr.it/lsdsir09-4.pdf>.

Cederman, D., & Tsigas, P. (2008). A practical Quicksort algorithm for graphics processors. In *ESA '08*. Berlin, Heidelberg: Springer-Verlag.

Dowd, M., Perl, Y., Rudolph, L., & Saks, M. (1989). The periodic balanced sorting network. *Journal of the ACM, 36*(4).

Frigo, M., Leiserson, C.E., Prokop, H., & Ramachandran, S. (1999). Cache-oblivious algorithms. In *FOCS '99*. Washington, DC, USA. <http://portal.acm.org/citation.cfm?id=796479>.

Govindaraju, N., Gray, J., Kumar, R., & Manocha, D. (2006). Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD '06*. New York, NY, USA, pp. 325–336.

Govindaraju, N. K., & Manocha, D. (2007). Cache-efficient numerical algorithms using graphics hardware. *Parallel Computing, 33*(10–11).

Govindaraju, N. K., Raghuvanshi, N., & Manocha, D. (2005). Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *SIGMOD '05*. New York, NY, USA: ACM.

Greß, A., & Zachmann, G. (2006). Gpu-abisort: optimal parallel sorting on stream architectures. In *IPDPS '06* (April).

Helman, D.R., Bader, D.A., & JáJá, J. (1995). A randomized parallel sorting algorithm with an experimental study. Tech. Rep. *Journal of Parallel and Distributed Computing*.

Kapasi, U. J., Dally, W. J., Rixner, S., Mattson, P. R., Owens, J. D., & Khailany, B. (2000). Efficient conditional operations for data-parallel architectures. In *Proceedings of the 33rd annual ACM/IEEE international symposium on microarchitecture*. ACM Press.

Khailany, B., Dally, W. J., Kapasi, U. J., & Mattson, P. (2001). Imagine: media processing with streams. *IEEE Micro, 21*(2).

Kipfer, P., Segal, M., & Westermann, R. (2004). Ubernow: a GPU-based particle engine. In *In HWWS G04: proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on graphics hardware* (pp. 115G–122). ACM Press.

Kipfer, P., & Westermann, R. (2005). Improved GPU sorting. In M. Pharr (Ed.), *GPUGems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley.

Knuth, D. E. (1973). *The art of computer programming. Sorting and searching* (Vol. III). Addison-Wesley.

Kruijf, M., & Sankaralingam, K. (2007). Mapreduce for the cell B.E. architecture. *IBM Journal of Research and Development*.

Kumar, V. (2002). *Introduction to parallel computing*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc..

[6] We did not consider the GPU version of Quicksort because results of the previous section pointed out that such solution is both slower and space inefficient than the other two.

Please cite this article in press as: Capannini, G., et al. Sorting on GPUs for large scale datasets: A thorough comparison. *Information Processing and Management* (2011), doi:10.1016/j.ipm.2010.11.010

Lester, N., 2005. Fast on-line index construction by geometric partitioning. In *Proceedings of the 14th ACM conference on information and knowledge management (CIKM 2005)*. ACM Press.

Manning, C.D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval*. Cambridge University Press (July). <http://www.amazon.ca/exec/obidos/redirect?tag=09-20&amp;path=ASIN/0521865719>.

NVIDIA (2008). *CUDA Programming Guide* (June).

Purcell, T.J., Donner, C., Cammarano, M., Jensen, H.W., & Hanrahan, P. (2003). Photon mapping on programmable graphics hardware. In *SIGGRAPH conference on graphics hardware*.

Sengupta, S., Harris, M., Zhang, Y., & Owens, J.D. (2007). Scan primitives for GPU computing. In *GH '07*. Eurographics Association, Aire-la-Ville, Switzerland.

Witten, I.H., Moffat, A., & Bell, T.C. (1999). *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann Publishers, San Francisco, CA. <http://citeseer.ist.psu.edu/witten96managing.html>.

Please cite this article in press as: Capannini, G., et al. Sorting on GPUs for large scale datasets: A thorough comparison. *Information Processing and Management* (2011), doi:10.1016/j.ipm.2010.11.010