CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **28**:21–43
Published online 12 August 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3611

CUDA-quicksort: an improved GPU-based implementation of quicksort

Emanuele Manca[1,*,†], Andrea Manconi[2], Alessandro Orro[2], Giuliano Armano[1] and Luciano Milanesi[2]

[1]*Department of Electrical and Electronic Engineering, The University of Cagliari, Via Marengo, 09123 Cagliari, Italy*
[2]*Institute for Biomedical Technologies, National Research Council, Via Fratelli Cervi, 93 - 20090 Segrate (MI), Italy*

SUMMARY

Sorting is a very important task in computer science and becomes a critical operation for programs making heavy use of sorting algorithms. General-purpose computing has been successfully used on Graphics Processing Units (GPUs) to parallelize some sorting algorithms. Two GPU-based implementations of the quicksort were presented in literature: the GPU-quicksort, a compute-unified device architecture (CUDA) iterative implementation, and the CUDA dynamic parallel (CDP) quicksort, a recursive implementation provided by NVIDIA Corporation. We propose CUDA-quicksort an iterative GPU-based implementation of the sorting algorithm. CUDA-quicksort has been designed starting from GPU-quicksort. Unlike GPU-quicksort, it uses atomic primitives to perform inter-block communications while ensuring an optimized access to the GPU memory. Experiments performed on six sorting benchmark distributions show that CUDA-quicksort is up to four times faster than GPU-quicksort and up to three times faster than CDP-quicksort. An in-depth analysis of the performance between CUDA-quicksort and GPU-quicksort shows that the main improvement is related to the optimized GPU memory access rather than to the use of atomic primitives. Moreover, in order to assess the advantages of using the CUDA dynamic parallelism, we implemented a recursive version of the CUDA-quicksort. Experimental results show that CUDA-quicksort is faster than the CDP-quicksort provided by NVIDIA, with better performance achieved using the iterative implementation. Copyright © 2015 John Wiley & Sons, Ltd.

Received 13 September 2014; Revised 15 June 2015; Accepted 7 July 2015

KEY WORDS: high performance computing; GPU; CUDA; quick sort

1. INTRODUCTION

Sorting is a very important task in computer science and becomes a critical operation when dealing with huge amounts of data. Several algorithms have been devised and implemented aimed at reducing the computing time needed to sort large amounts of data. Some are problem-specific, while others exploit parallelization strategies. The latter usually has strong dependence on the adopted hardware and software architecture.

General-purpose computing on GPUs is increasingly used to deal with computationally intensive algorithms. Recently, GPUs have also been used to parallelize some sorting algorithms. GPU-quicksort [1] is one of the first GPU-based implementations of the original quicksort algorithm [2]. At the time, GPU-quicksort has been presented, recursion was not supported by GPUs, and their authors devised an iterative implementation of the algorithm. Recently, the NVIDIA laboratories released a recursive GPU-based quicksort implementation called compute-unified device architecture (CUDA) dynamic parallel (CDP)-quicksort [3].

*Correspondence to: Emanuele Manca, Department of Electrical and Electronic Engineering, University of Cagliari, Via Marengo, 09123 Cagliari, Italy.
†E-mail: Emanuele.manca@diee.unica.it

Both implementations repeatedly perform two steps on the sequence in hand. In the first step, a pivot (say P) is picked, and the sequence is partitioned, so that several thread groups can work in parallel on different parts of the sequence (Figure 1.1A and B). Each thread group calculates the coordinates of two partial subsequences by separating the items with value $< P$ from the items with value $> P$. Then, two subsequences are generated to separate the items with value $< P$ from those with value $> P$. Each thread group will be responsible to move the related items in to the appropriate subsequence (Figure 1.1C). The process is then repeated on the generated subsequences (Figure 1.1D). The second step starts when the size of each subsequence is so small that the overhead of using quicksort becomes too high. Then, a GPU-based bitonic sort is used to top off the sorting (Figure 1.2).

Two main issues must be dealt with to provide an efficient implementation of quicksort on GPUs. As an efficient synchronization is hardly achievable given the high quantity of threads used in GPUs, the communication among thread groups and the thread-block synchronization are critical operations. Moreover, the sorting may require to change the position of a very high number of items. As GPUs have a high-latency global memory and very limited caches (compared to the number of cores on a chip), to optimize the memory, access patterns become an important issue.

In the GPU-quicksort, both synchronization and communication issues are addressed through a block-oriented iterative implementation, where each partition is processed by one thread block. To keep the inter-block communication low, the first step of the algorithm is divided in two phases: the first phase proceeds as the aforementioned step, whereas the second one starts when the size of a

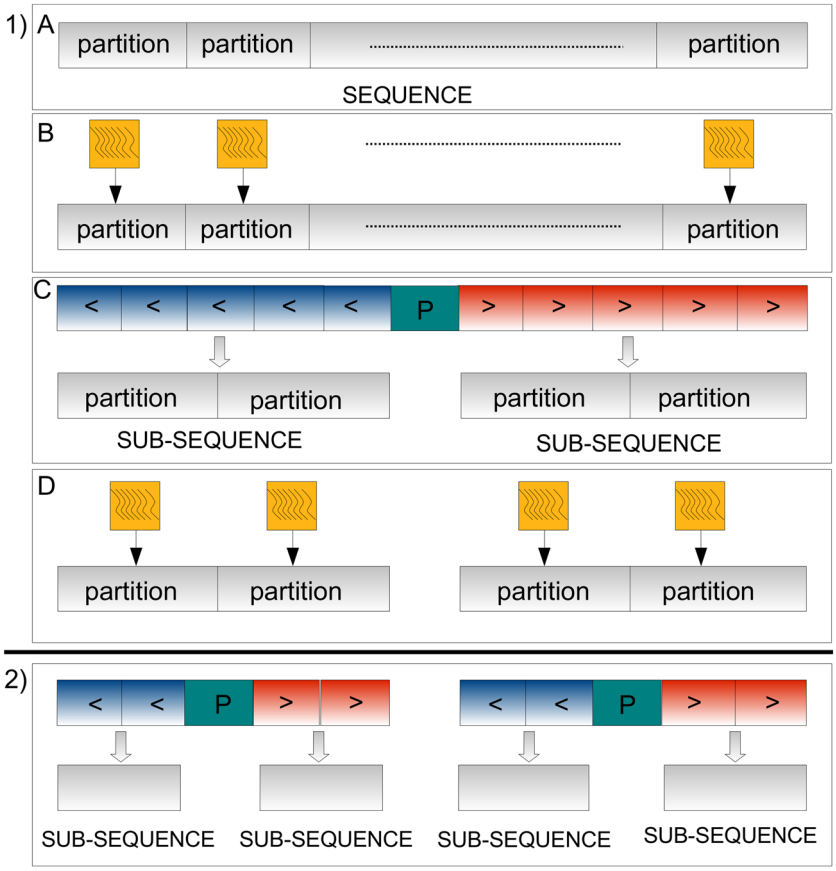


Figure 1. (1) First step: (A) The given sequence is partitioned so that several thread blocks can work in parallel on different parts; (B) a thread block is assigned to each different partition element; (C) the partial result of each thread block is merged in two subsequences; and (D) new subsequences are partitioned and assigned to thread blocks. (2) Second step: small subsequences assigned to a single thread block. As a final step, bitonic sort is used to finalize the sorting (not represented in the figure).

DOI: 10.1002/cpe

subsequence can be entirely processed by a thread block. In this phase, there is no need to partition the sequence and to provide communication among blocks. Differently, CDP-quicksort exploits a warp-oriented recursive implementation that uses an inter-warp communication based on atomic primitives [4] (i.e., barrier-functions supported starting from the NVIDIA Fermi architecture).

As for memory accesses, we observed that both implementations have been properly designed to optimize the reading phases, whereas an analogous optimization has not been provided for the writing phases.

In this article, we propose CUDA-quicksort, a new block-oriented iterative GPU-based implementation of quicksort, which uses atomic primitives to perform communication among blocks while ensuring an optimized access to the GPU memory. Experimental results show that our implementation outperforms both GPU-quicksort and CDP-quicksort in terms of computing time. The rest of the article is organized as follows: In the next section, the architecture and programming model of GPUs is recalled. Section 3 gives an overview of related work. Section 4 describes the proposed implementation of quicksort. Section 5 illustrates and discusses experimental results. Section 6 reports conclusions and future work.

The software is freely available for non-commercial use at http://sourceforge.net/projects/cuda-quicksort/.

2. ARCHITECTURE AND PROGRAMMING MODEL OF GRAPHICS PROCESSING UNITS

In the NVIDIA GPU-based architecture, parallelization is obtained through the execution of tasks in a number of stream processors or CUDA cores. Cores are grouped in multiprocessors that execute in parallel. The NVIDIA GPU (Figure 2) features several CUDA cores organized in streaming multiprocessors. CUDA cores can be used to execute integer and floating point instructions, and all cores in a streaming multiprocessor execute the same instruction at the same time. This computational paradigm, called single instruction, multiple thread, can be considered as an advanced form of

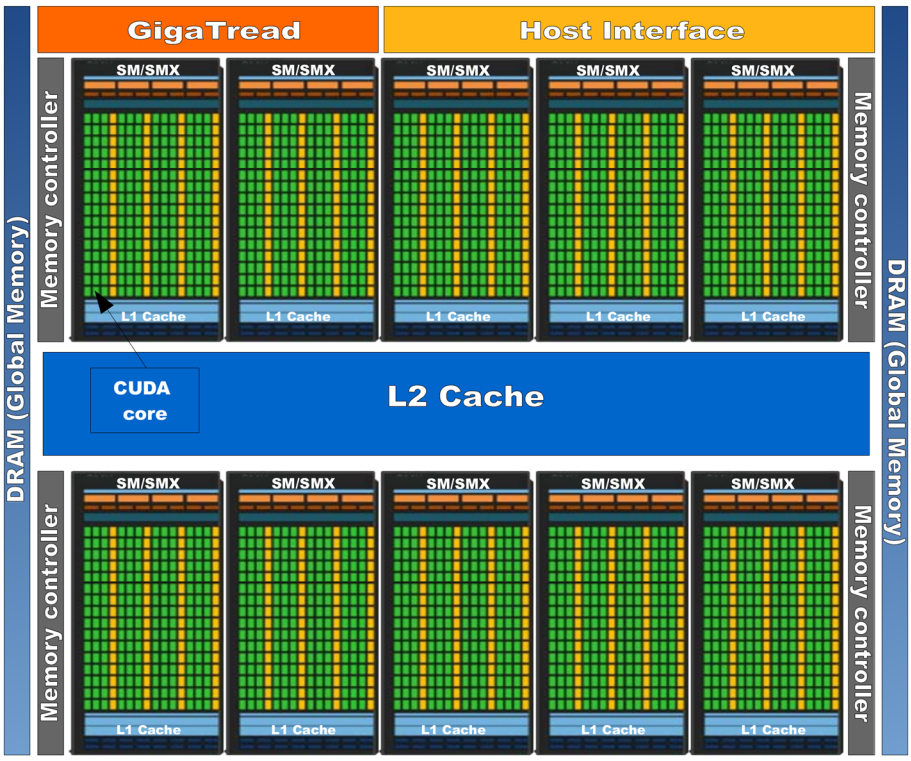


Figure 2. GPU architecture: The GPU is a many-core processor equipped with hundreds of cores able to simultaneously handle thousands of threads. DRAM, dynamic random access memory; CUDA, compute-unified device architecture. GPU, graphics processing unit.

DOI: 10.1002/cpe

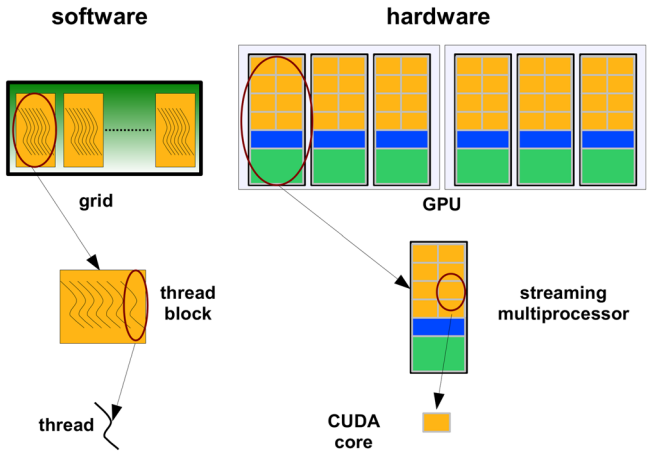


Figure 3. Compute-unified device architecture (CUDA) execution model: Threads are grouped in blocks in a grid; each thread has a private memory and runs in parallel with the others in the same block [5]. GPU, graphics processing unit.

the single instruction, multiple data paradigm. The code is executed in groups of 32 threads called warps. The GPU architecture supports a dynamic random access memory, with a low-latency L2 cache and a high-bandwidth L1 cache per streaming multiprocessor. Each multiprocessor has also a programmable L1 cache (called shared memory), managed by a specific software.

The parallel programming model of the CUDA architecture [4] provides a set of APIs that allows programmers to access the underlying hardware infrastructure and to exploit the fine-grained and coarse-grained parallelism of data and tasks. Summarizing, the CUDA execution model (Figure 3) can be described as follows: The GPU creates an instance of the kernel program that is made of a set of threads grouped in blocks in a grid. Each thread has a unique identifier within its block, a private memory, and registers, and runs in parallel with other threads of the same block. All threads in a block execute concurrently and cooperatively by sharing memory and exchanging data. A block, identified by a unique identifier within the block grid, can execute the same kernel program on different data, which can be read from/written to a global shared memory. Each block in the grid is assigned to a streaming multiprocessor in a cyclical manner. A streaming multiprocessor can host up to eight or 16 blocks. When a block is assigned to it, there is no possibility to migrate on other ones.

3. RELATED WORK

A first try to design a GPU-based quicksort has been proposed in [6]. In this case, results were not encouraging: computing time was an order of magnitude slower than other sorting algorithms they used for comparison. GPU-quicksort was the first high-performance implementation of the quicksort for GPU. It is well known that quicksort is a recursive algorithm based on the divide-and-conquer paradigm. It recursively picks a pivot from a sequence and successively moves items with value lower than the pivot to the 'left' and items with value higher than the pivot to the 'right'. GPU-quicksort works in a similar manner and has been designed with the goal of minimizing the amount of bookkeeping and inter-thread synchronization. It parallelizes quicksort by exploiting a straightforward approach [7, 8] that divides the sequence in hand in different partitions and dynamically assigns them to available processors.

NVIDIA Corporation proposed the CDP-quicksort, a recursive implementation of the sorting algorithm based on the recent CUDA dynamic parallelism technology. This technology allows a GPU kernel to call another GPU kernel so that recursive algorithms can also be implemented. It should be pointed out that the dynamic parallelism is only available for NVIDIA GPUs with computing capability $\geqslant 3.5$.

DOI: 10.1002/cpe

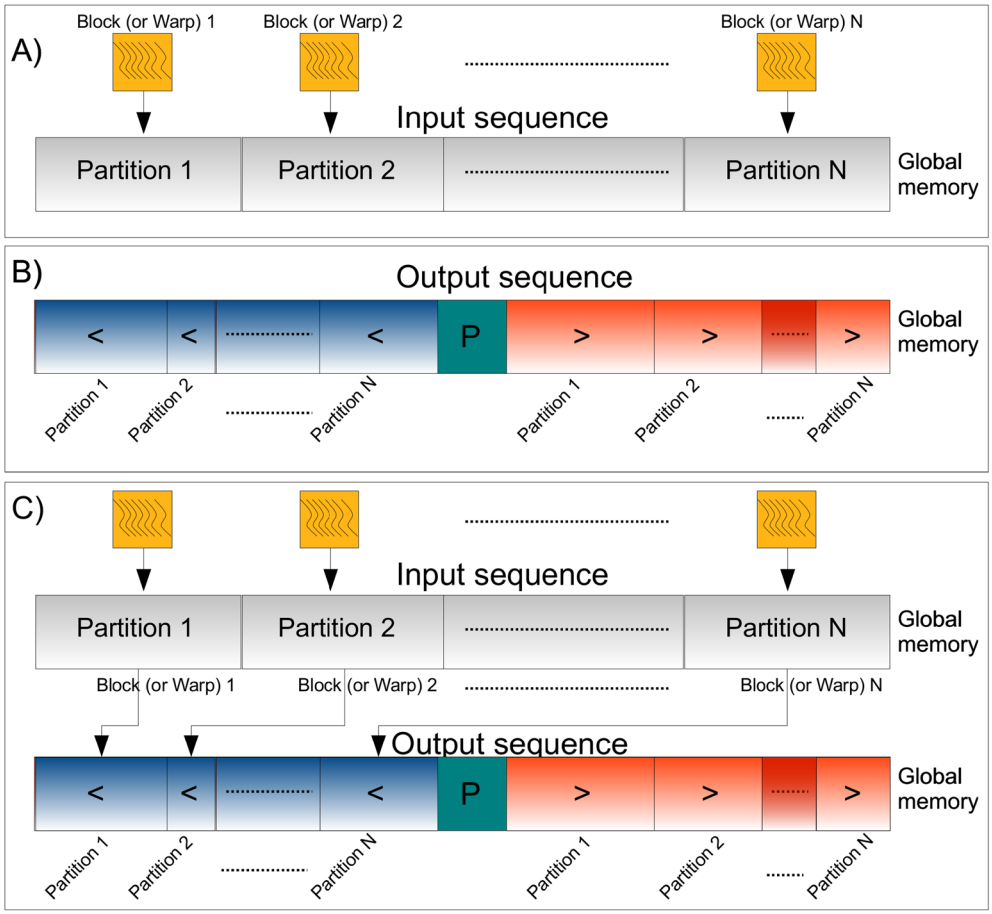


Figure 4. (A) The sequence is partitioned so that several thread blocks (or warp) can work in parallel on different parts. A thread block (or warp) is assigned to each different partition element; (B) the two output subsequences are partitioned; and (C) each block items whose value is higher or lower than the pivot are moved in their respective partitions.

GPU-quicksort and CDP-quicksort implementations are very similar. As in the GPU-quicksort, CDP-quicksort partitions the sequence to be sorted. However, unlike GPU-quicksort, in CDP-quicksort, each partition is processed by a thread warp (Figure 4A).

Both algorithms return two subsequences: one containing all items $< P$ and one containing all items $> P$. In turn, also the two generated subsequences must be partitioned so that each thread can write its partial result in its assigned partition. The size of partition element is not fixed nor can be known *a priori*, as the number of items $< P$ or $> P$ is variable in each partition element. For this reason, both algorithms proceed as follows: (i) each warp (or block) counts the number of elements $< P$. This number is used to calculate the size of the subsequence partition element; (ii) the partitioning of the subsequence is performed (Figure 4B); and (iii) each warp (or block) moves items $< P$ in its subsequence partition element (Figure 4C). Analogous tasks are performed to move elements $> P$. As the GPU-quicksort, when subsequences are very small, the sorting is finalized using a GPU-based bitonic sort [9]. The two subsequences are not partitioned in the same way by the two algorithms. In GPU-quicksort, the partitioning is CPU-performed, whereas in CDP-quicksort, a dynamic partitioning of the two subsequences is performed on the GPU using atomic primitives.

It should be pointed out that GPU-based solutions have also been proposed in literature to efficiently parallelize other sorting algorithms. In [10], a GPU-based bitonic merge sort has been presented, based on the implementation proposed in [11]. In [12] and [13], a bitonic and an odd–even merge sort have been presented. In [14], two sorting solutions have been developed: a solution based

DOI: 10.1002/cpe

on the periodic-balanced sorting network [15] and a solution based on the bitonic sorting network [16]. In [17], an approach for parallel sorting on stream processing architectures based on adaptive bitonic sorting, called GPU-ABISort, has been presented. Another algorithm for fast sorting large lists that makes use of GPUs has been presented in [18]. In this work, the authors designed a vector-based mergesort using CUDA, designed to work on four 32-bit floats simultaneously, resulting in a 4-time speed improvement compared to mergesorting. In [19], the authors described the design and implementation of a sample sort algorithm for CUDA-enabled GPU cards.

4. METHODS

CUDA-quicksort is based on the GPU-quicksort implementation. As previously illustrated, in the first step, GPU-quicksort repeatedly performs two phases on the given sequence. In the first phase, GPU-quicksort picks a pivot (P) and partitions the sequence, so that several thread blocks can work in parallel on different parts of the sequence. Each thread in the block iterates through all data of its assigned partition, keeping track of the number of elements that is greater or lesser than P. This information is stored in two arrays on the shared memory (Figure 5A). Then, each thread block calculates the GPU-based prefix-sum [20] of these two arrays, so that each thread knows the relative offset on where to move items that are higher or lower than the pivot (Figure 5B). Then, thread blocks are synchronized, so that each thread block knows the absolute offset where to move the items. In the inter-block synchronization, the CPU waits for the completion of each thread block then calculates another prefix-sum (Figure 5C). Finally, when each thread of a block knows its offset, the items $< P$ and $> P$ are moved in their respective slice (Figure 5D), and the items whose value is equal to P are written between the two subsequences. After a fixed number N of iterations, GPU-quicksort starts the second phase. Typically, in this phase, the size of each subsequence is such that it can be entirely processed by a thread block. Finally, in the second step, a GPU bitonic sort based on [9] is used to obtain the final result.

We propose a new block-oriented iterative implementation of the quicksort that uses atomic primitives to perform inter-block communications while guaranteeing an optimized access to the GPU memory. Similarly that for the GPU-quicksort, our implementation consists of two steps. However, unlike GPU-quicksort, the first step of the algorithm is not divided in two phases (box 4.1). The first step is repeatedly performed on the sequence in hand. In this step, the CPU is only used to control the iterations. The second step starts when the size of the subsequences is so small that overhead of quicksort becomes too high. Also, in the proposed implementation, a GPU-based bitonic sort is used to top off the sorting.

In the first step, a pivot is picked out, and the sequence is partitioned to let several thread blocks work in parallel. As in the GPU-quicksort, an auxiliary buffer is allocated in the GPU global memory, where the items with value higher and lower than the pivot are separated in two subsequences. This process is then repeated on the two subsequences in the auxiliary buffer. Initially, thread blocks sort their assigned partition independently from each other (i.e., from the rest of the sequence); see Figure 6A. In particular, each thread block sorts its assigned partition elements in its local shared memory separating in two different buckets items with value lower than the pivot from those whose value is higher (task 1 of box 4.3 and Figure 6B). Through the atomic primitives, the auxiliary buffer where the subsequences will be written is dynamically partitioned (task 2 of box 4.3 in lines 25–28). Each block is assigned to a partition on the auxiliary buffer, so that they can simultaneously write their partial results (Figure 6C). Then, each block copies the bucket stored in its local shared memory to the assigned partition. In this way, all buckets of each block that keep track of items lower than the pivot are merged into a single subsequence (task 3 of box 4.3 in lines 33–40 and Figure 6C). The same task is performed while merging all buckets that keep track of the items with value higher than the pivot in the other subsequence. Finally, the items whose value is equal to the pivot are written between the two subsequences. These two subsequences are sorted in parallel, and the sorting process can be started over again on each of them independently (box 4.1).

As in both GPU-quicksort and CDP-quicksort, in the second step, which starts when subsequences are very small, a GPU-based bitonic sort is used to complete the sorting (box 4.1). Unlike

DOI: 10.1002/cpe

```
Input: in, size
Output: out
 1: offset ← 0
 2: P ← MEDIAN(in, offset, size)
 3: stack ← NEWSTACK()
 4: gpu_stack ← GPU_NEWSTACK()
 5: gpu_stack ← GPU_PUSH(offset, size, P)
 6:
 7: while true do
 8:     WAIT UNTIL ALL GPU KERNELS AT THIS STAGE HAVE FINISHED        ▷ barrier-function
 9:
10:     stack ← COPYFROMGPU(gpu_stack)
11:
12:     if stack is empty  then
          return
13:     end if
14:
15:
16:     while stack is not empty do                                   ▷ For each subsequence
17:
18:         (offset, len, P) ← POP(stack)              ▷ POP the coordinates of a subsequence
19:
20:         stream ← NEWSTREAM()   ▷ Each GPU kernel of a new stream is parallel processed
21:
22:         if len < LENGHT then
23:             BITONICSORT_KERNEL< stream >(in, out, offset, len)
24:         else
25:             (left, right, lP, rP) ← CUDA-QUICKSORT< stream >(in, out, offset, len, P)
26:                         ▷ (left, right): lengths of subsequences created by the quicksort
27:
28:             gpu_stack ← GPU_PUSH< stream >(offset, left, lP)
29:             gpu_stack ← GPU_PUSH< stream >(offset + len − right, right, rP)
30:
31:         end if
32:     end while
33:
34:     GPU_SWAP< stream >(in, out)
35:
36: end while
37: return out
```

Box 4.1: Pseudo-code of the CUDA-Quicksort algorithm. GPU kernels are asynchronous to the CPU. They return before the GPU has completed its execution. GPU Kernel with the same stream are mutually synchronous. GPU kernel with different streams are parallel processed by the GPU.

from the GPU-quicksort and the CDP-quicksort that use two different implementations based on the same parallel algorithm proposed in [9], CUDA-quicksort uses the bitonic sort released with the NVIDIA CUDA sample 6.0. Experiments aimed at comparing the performance of the three implementations to sort sequences of 1024 items (32-bit integer) show that the bitonic sort released by NVIDIA resulted 2.7 times faster than that used in the CDP-quicksort and 3 time faster than that used in the GPU-quicksort.

In the following of this section, we describe in more details as CUDA-quicksort partitions the sequences, manages the inter-block communication, and optimizes the memory accesses.

DOI: 10.1002/cpe

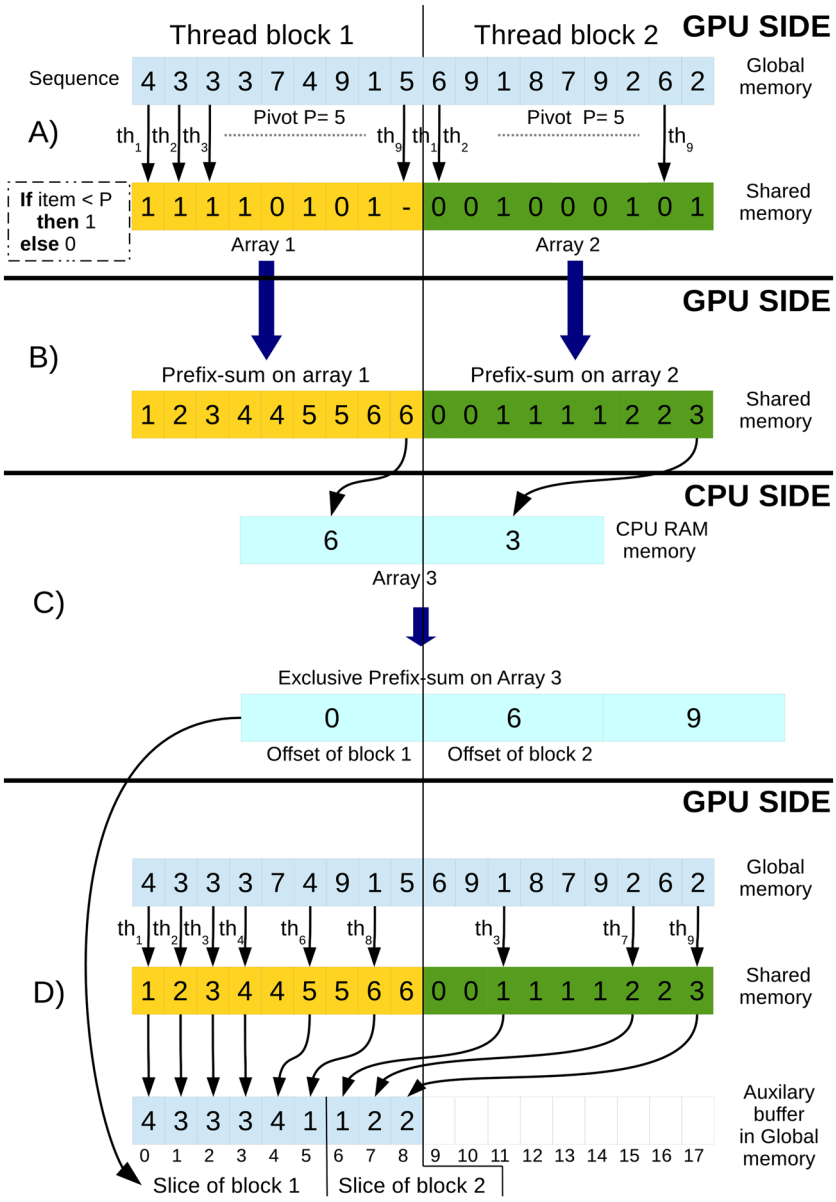


Figure 5. Example of the GPU-quicksort algorithm for a sequence of 18 elements. The sequence is partitioned in two nine-thread blocks. (A) Each thread compares its related item with P. Then, results of comparison are stored in the shared memory (block 1 in array 1 and block 2 in array 2). (B) Each thread block calculates the prefix-sum on array 1 and on array 2. (C) The CPU waits for the completion of each thread block. Then, the CPU stores in array 3 the number of elements that is lesser than the pivot associated to each block. Finally, the CPU calculates the exclusive prefix-sum on array 3 to calculate the slice offset of each block. (D) Each thread of a block gets its offset from the shared memory and moves the thread-associated item $< P$ in its respective slice. The same algorithm is used to move items $> P$. RAM, random access memory. GPU, graphics processing unit.

4.1. Partitioning

In GPU-quicksort sequence partitioning is based on the number of thread blocks, which is fixed *a priori* independently from the sequence size. The partition element size depends on the sequence size and is calculated as the ratio of the sequence size to a fixed number of thread blocks ($partition\_element\_size = sequence\_size/thread\_block\_number$). In our solution, the partition element size is fixed *a priori*, and the number of thread blocks depends on the sequence size. The number of thread blocks is calculated as the ratio of the sequence size to a fixed partition

DOI: 10.1002/cpe

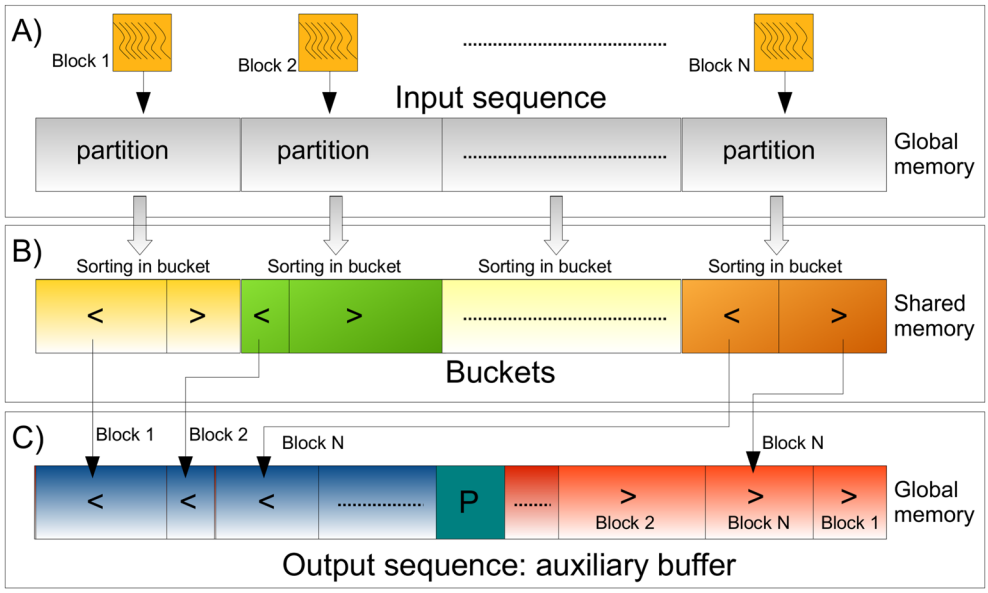


Figure 6. (A) The sequence is partitioned so that several thread blocks can work in parallel on different parts. A thread block is assigned to each different partition element; (B) each thread block sorts its assigned partition element in its local shared memory separating in two different buckets items with value lower than the pivot from those whose value is higher; and (C) through the atomic primitives, the auxiliary buffer is dynamically partitioned. Then, each block copies the bucket stored in its local shared memory to the assigned partition.

element size, which is equal to the size of the shared memory necessary to store elements $< P$ and $> P$. In CUDA-quicksort, the number of threads in a block is lower than the number of items in the partition. In this case, a thread block is unable to process a partition in a single run, so it divides and processes the partition in different tiles.

4.2. Inter-block communication using atomic primitives

In the first phase of the first step of the GPU-quicksort, sequences are typically very large, and several thread blocks may work on different partitions. Thread blocks process their partition independently from the other ones, even though they are responsible to contribute to merge their partial results with all other ones. Then, appropriate communication among thread blocks is required to dynamically partition the two subsequences, gathering all items with value higher or lower than the pivot in order to merge together the partial result of each thread block. Atomic primitives could be used to perform an inter-block communication in this phase. The authors of GPU-quicksort assessed the opportunity to use them. However, as atomic primitives were not widely supported by GPUs, they decided not to use them with the aim to provide a more generalized solution for GPU. Basically, the communication among thread blocks is guaranteed waiting the completion of each thread block. This implies that the host is responsible for the communication among thread blocks, resulting in a decay of the overall performance. This effect depends on (i) the serialization of part of this phase and (ii) the increase of the time spent for transferring data from device to host and vice versa. To keep the inter-block communication low, a second phase has been implemented. Indeed, the second phase starts after N iterations of the first phase, when the size of each subsequence is generally such that it can be processed by a thread block. This phase differs from the first as each thread block is assigned with its own subsequence, eliminating the need for inter-block communications. However, there is no guarantee that the generated subsequences will be sufficiently small to be processed in this phase by one thread block. Of course, in the event that a very big subsequence is processed by only one thread block, a decay of the overall performance occurs.

Nowadays, the access to atomic primitives is widely provided by GPUs. In the NVIDIA CDP-quicksort, partitions are processed by thread warp, and the inter-warp communication is

DOI: 10.1002/cpe

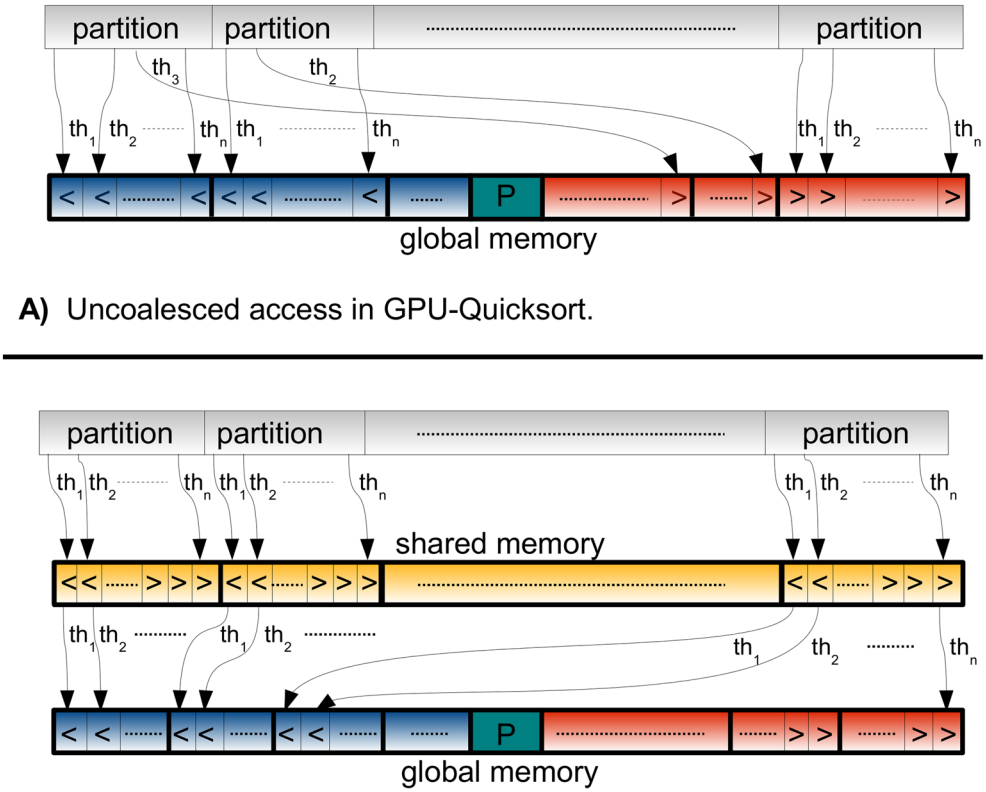


Figure 7. (A) Uncoalesced access in GPU-quicksort. Each thread in a block may move its item in not consecutive global memory addresses. (B) Coalesced access in the proposed solution. Items are sorted in the shared memory before being written in the global memory. CUDA, compute-unified device architecture. GPU, graphics processing unit.

performed by atomic primitives. This solution has high atomic issue rate, leading to a decrease of the overall performance. We propose a block-oriented solution that uses the atomic primitives to perform inter-block communications. In doing so, we use the computing power of the GPU while providing a full parallelization of the first step of the algorithm and reducing the high atomic rate issue.

Each block sorts its assigned partition in the shared memory. Before merging the partial result in the output sequence, an inter-block communication is required so that each block knows where it should write its result. To this end, each block updates an atomic counter with the size of its partial result. The atomic counter increases the counter value and returns the old value. This value represents the offset in the output sequence where the block shall write its result. For example, let us assume of having to deal with two blocks, one having expected to write 100 and the other 200 items to the output sequence. A communication between the blocks is required, to let each block know where to write its outcome. Let us assume that block 2 (200 items) ends writing before block 1. Block 2 updates the atomic counter, which returns the initial value, that is, 0 and increases the counter value by 200. Block 2 now knows where its outcome should be written, that is, between 0 and 200. When block 1 updates the counter, it will return a value of 200 (i.e., the size of block 2 partition) to block 1. Therefore, block 1 will write its partial result between 200 and 300. This allows a dynamic partitioning of the output sequence. This is required for the quicksort parallelization, as the number of items with value higher and lower than the pivot processed by each block is not fixed and cannot be known *a priori*. Therefore, a communication among blocks is required to coordinate writing. For implementation details, see box 4.3.

DOI: 10.1002/cpe

4.3. Optimizing memory access

In GPU-quicksort, each thread block uses a prefix-sum to calculate the new coordinates of the items to be moved to the left or to the right of the pivot. The output of the prefix-sum is stored in the shared memory. An auxiliary region of the global memory is allocated to store the new subsequences in parallel. Then, each thread in a block (i) accesses the shared memory to read the new coordinates of its assigned item; (ii) gets the item value in the sequence; and (iii) writes it to the proper subsequence on the global memory. In doing so, global memory is accessed in writing mode without guaranteeing a coalesced access. It is commonly known that uncoalesced access to global memory may substantially affect the overall performance. Coalesced memory access is guaranteed if consecutive threads access consecutive global memory addresses. However, as each thread in a block may be assigned to an item lower or higher than the pivot, consecutive threads may access very distant memory regions (Figure 7A). CDP-quicksort proceeds in a quite similar way, with the difference that calculations are more fine-grained as they are performed at a thread-warp level rather than at a thread-block level. Sequence partitioning in thread warps allows using warp vote functions that permit a thread-block synchronization without using the shared memory. In particular, in the CDP-quicksort, each thread of a warp exploits a warp prefix-sum to calculate the offset of the items to be moved to the left or to the right of the pivot. Then, when each thread of a warp knows its offset, it writes its items in the appropriate subsequence in the global memory. Also, in this case, access to the global memory is uncoalesced.

In CUDA-quicksort, the issue has been addressed sorting items in the shared memory before writing them to the global memory. In particular, in the proposed solution, (i) each thread performs the prefix-sum (lines 17–18 in box 4.3) to calculate the new coordinates of its assigned items; (ii) gets the items value in the sequence and writes it to the proper subsequence in the shared memory (task 1 in box 4.3 in line 21 and lines 20–34 in box 4.2); and (iii) updates the atomic counter (task 2 in box 4.3 in lines 25–28) and writes its new assigned item to the proper subsequence in the global memory (task 3 in box 4.3 in lines 33–40). In this way, consecutive threads in a block read sorted items from the shared memory and write them to consecutive addresses of the global memory (Figure 7B).

5. RESULTS

Experiments have been carried out on a Intel Xeon CPU E5-2667 (2.90 GHz, 12 cores)(Intel, Santa Clara, CA, USA) and a GPU NVIDIA Tesla Kepler k20c. We used the same six sorting benchmark distributions (Figure 8) used in [1] to evaluate GPU-quicksort. These benchmarks are commonly used in literature to compare the performance of different sorting algorithms [21]. Performance was evaluated on a 32-bit integer and 64-bit floating point sequences, varying their size from 1M to 32M elements (only power-of-2 sizes).

Initially, we carried out experiments aimed at assessing to what extent the proposed changes improve the computing time required to perform the different tasks of the first step of CUDA-quicksort. In doing this, we analyzed the computing time required to carry out the first step of CUDA-quicksort and CDP-quicksort and the first phase of the first step of GPU-quicksort.

Summarizing, both the first step of the first two algorithms and the first phase of the last algorithm consist of the following tasks: (i) picking out the pivot; (ii) partitioning the sequence; (iii) reading subsequences and calculating their coordinates; (iv) partitioning of the output sequence; (v) writing subsequences in the global memory; and (vi) preparing for the next quicksort execution.

In our evaluation, we deemed appropriate to sum together the time required to perform the first four tasks. In the following of this section, we will refer to the first four tasks as the sorting task.

Figure 9 shows a comparison of the computing time required to perform the different tasks obtained using a uniform distribution of the data. The figure shows that CUDA-quicksort performs better than GPU-quicksort and CDP-quicksort, in both sorting and writing tasks. However, it can be observed that in CUDA-quicksort, the time required to perform the sorting task tends to become higher than the one required in the GPU-quicksort for very long sequences (represented *in blue* in Figure 9). This worsening is mainly due to the greater amount of blocks used in CUDA-quicksort.

DOI: 10.1002/cpe

```
1:  Tile ← Block_id * SHARED_SIZE                                   ▷ Partitioning
2:  index ← Tile + thread_id                                        ▷ Partitioning
3:
4:  function READ_DATA(indata, offset, index, size, P)
5:      low ← 0
6:      hi ← 0
7:      while offset + index < size do
8:          data ← indata[offset+index]
9:          if data<P then
10:             low ← low + 1
11:         end if
12:         if data>P then
13:             hi ← hi + 1
14:         end if
15:         index ← index + BLOCK_SIZE      ▷ BLOCK_SIZE is the size of the threads block
16:     end while
17:     return (low,hi)
18: end function
19:
20: function WRITE_SM(indata, offset, index, size, left_cordinates, right_cordinates )
21:     __shared_memory__ subsequences_left, subsequences_right
22:     while offset + index < size do
23:         data ← indata[offset+index]
24:         if data<P then
25:             subsequences_left[left_cordinates] ← data
26:             left_cordinates ← left_cordinates + 1
27:         end if
28:         if data>P then
29:             subsequences_right[right_cordinates] ← data
30:             right_cordinates ← right_cordinates + 1
31:         end if
32:         index ← index + BLOCK_SIZE      ▷ BLOCK_SIZE is the size of the threads block
33:     end while
34:     return (subsequences_left, subsequences_right)
35:
36: end function
```

Box 4.2: READ_DATA() and WRITE_SM() functions of CUDA-Quicksort

However, in doing so, CUDA-quicksort optimizes the computing time required to perform the fifth task (represented *in orange* in Figure 9) yielding an overall improvement of performance.

It should also be observed that the time to perform the sixth task in CDP-quicksort (represented *in yellow* in Figure 9) is very high if compared with that required in CUDA-quicksort and GPU-quicksort. This bulk of performance occurs when a kernel executing quicksort calls other two kernels to execute quicksort on the (two) output subsequences. The calling kernel can only execute the quicksort on subsequences once all blocks are executed. In GPUs, the barrier-synchronization is not allowed among threads in different blocks. Therefore, CDP-quicksort implements the inter-block barrier-synchronization by using another atomic counter, so that it knows which is the last block. This inter-block barrier-synchronization entails the bulk of performance in CDP-quicksort.

We also performed experiments aimed at assessing to which extent the proposed changes affect the performance of the first step. To this end, we implemented two different releases of CUDA-quicksort: (i) CUDA-quicksort-AP, a release implemented exploiting only atomic primitives without providing any optimized memory access and (ii) CUDA-quicksort-MAO, a release developed to

DOI: 10.1002/cpe

```
Input: indata, offset, size, P
Output: outdata, left_length, right_length, left_newP, right_newP
 1:
 2: ****************** TASK 1 ******************
 3:
 4: Tile ← Block_id * SHARED_SIZE                                   ▷ Partitioning
 5: index ← Tile * Block_id + thread_id                             ▷ Partitioning
 6:
 7: (low, hi) ← 0                  ▷ number of elements < P and > P calculated by one thread
 8: (low_total, hi_total) ← 0              ▷ total number of elements of a block < P and > P
 9: (left_length, right_length) ← 0  ▷ size of the subseq., shared references in the global memory
10:
11:                                ▷ each thread calculates the number of elements < P and > P
12: (low, hi) ←READ_DATA(indata, offset, index, size, P)
13:
14: WAIT UNTIL ALL THREAD OF A BLOCK HAVE FINISHED                  ▷ barrier-function
15:
16:        ▷ each thread calculates the coordinates of its items to be moved into the subsequences
17: (left_offset, low_total) ←PREFIX_SUM(low, shared_array)
18: (right_offset, hi_total) ←PREFIX_SUM(hi, shared_array)
19:
20:                                ▷ WRITE_SM() writes the subsequences in the shared memory
21: (subseq_left,subseq_right) ← WRITE_SM(indata,offset,index,size, left_offset,right_offset)
22:
23: ****************** TASK 2 ******************
24:
25: if (thread_id == 0) then                        ▷ only a thread is active for each thread block
26:     left_slice_offset ← offset + ATOMICADD(low_total, left_length)
27:     right_slice_offset ← size - (ATOMICADD(hi_total , right_length) + hi_total )
28: end if
29:
30: ****************** TASK 3 ******************
31:
32:                                          ▷ writting the subsequences in the global memory
33: while index < low_total do
34:     outdata [left_slice_offset + index] ← subseq_left [index]
35:     index ← index + BLOCK_SIZE          ▷ BLOCK_SIZE is the size of the threads block
36: end while
37: while index < hi_total do
38:     outdata [right_slice_offset + index] ← subseq_right[index]
39:     index ← index + BLOCK_SIZE          ▷ BLOCK_SIZE is the size of the threads block
40: end while
41: ************************************************
42: WAIT UNTIL ALL THREAD BLOCK HAVE FINISHED                       ▷ barrier-function
43: outdata [offset + left_length + index] ← WRITE_PIVOT(P, size - left_length - right_length)
44:
45: (left_newP, right_newP) ←NEW_PIVOTS(indata,offset,size)
46: return left_length, right_length, left_newP, right_newP
```

Box 4.3: CUDA-Quicksort kernel algorithm. *AtomicAdd(new val, memory address)* reads the old value located at the address in memory, computes old value + new val, and stores the result back to memory at the same address. These three operations are performed as an atomic transaction. It returns the old value. READ_DATA and WRITE_SM have been defined in box 4.2.

DOI: 10.1002/cpe

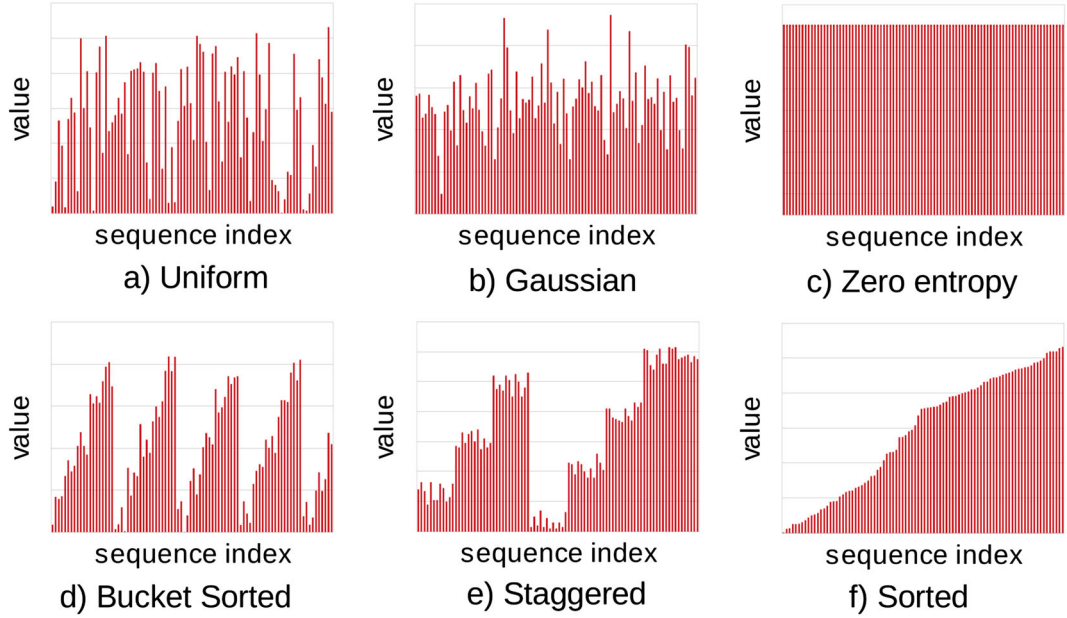


Figure 8. Sorting benchmarks distributions: (a) a uniformly distributed input obtained with random values from 0 to 2^{31}; (b) a Gaussian distributed random input created calculating the average of four randomly generated values; (c) a zero entropy input, created by setting every value to a random constant value; (d) an input sorted into p buckets, such that the first $\frac{n}{p^2}$ elements in each bucket are random numbers in [0, $\frac{2^{31}}{p} - 1$] and the second $\frac{n}{p^2}$ elements in [$\frac{2^{31}}{p}$, $\frac{2^{32}}{p} - 1$], and so forth; (e) after that, a dataset is divided into p partitions. Then, if the partition index i-th is $\leqslant \frac{p}{2}$ ($> \frac{p}{2}$) to their items will be assigned a random value between $(2i-1)\frac{2^{31}}{p}$ and $(2i)(\frac{2^{31}}{p} - 1)$ ($(2i-p-2)\frac{2^{31}}{p}$ and $(2i-p-1)\frac{2^{31}}{p} - 1$); and (f) sorted uniformly distributed values.

enforce memory access optimization while refrain from using atomic primitives. Experiments have been carried out using uniformly distributed benchmarks.

Results plotted in Figure 10 report the performance of CUDA-quicksort-AP, CUDA-quicksort-MAO, CUDA-quicksort run using all the proposed changes, and GPU-quicksort.

In particular, Figure 10a details the computing time required to perform the tasks required at the first step for sorting 1M and 2M items, whereas Figure 10b shows the performance improvement achieved by the different releases of CUDA-quicksort with respect to GPU-quicksort, for all sizes of the sorted sequences.

The computing time required to sort items (in blue in Figure 10a) and to write them (in orange in Figure 10a) highlights to which extent atomic primitives and memory access optimization improve the performance of CUDA-quicksort with respect to GPU-quicksort. Figure 10b shows that CUDA-quicksort-AP outperforms GPU-quicksort with a performance improvement between 26% and 8% depending on the size of the sequence to be sorted out; the greater the size, the lesser the improvement. Analyzing the performance of CUDA-quicksort-MAO, we observe an improvement in the writing task and a worsening in the sorting task with respect to GPU-quicksort. This worsening of the computing time is because increasing the number of blocks entails a tighter inter-block communication managed by the CPU that leads to a decay of the performance. Figure 10b shows that in the first step, CUDA-quicksort-MAO outperforms GPU-quicksort first phase, with a performance improvement between 32% and 52%, depending on the size of the sequence to be sorted. As for CUDA-quicksort versus CUDA-quicksort-MAO (Figure 10a), let us note that using atomic primitives and increasing the number of blocks lead to a speed-up in the writing task higher than the one achieved just using the atomic primitives.

As shown in Figure 10b, all the proposed changes lead to an overall improvement of about 80% (over the first phase of GPU-quicksort) (Table I). In particular, the first step of CUDA-quicksort leads to an improvement of about 50% over the second phase of GPU-quicksort. In this phase, each

DOI: 10.1002/cpe

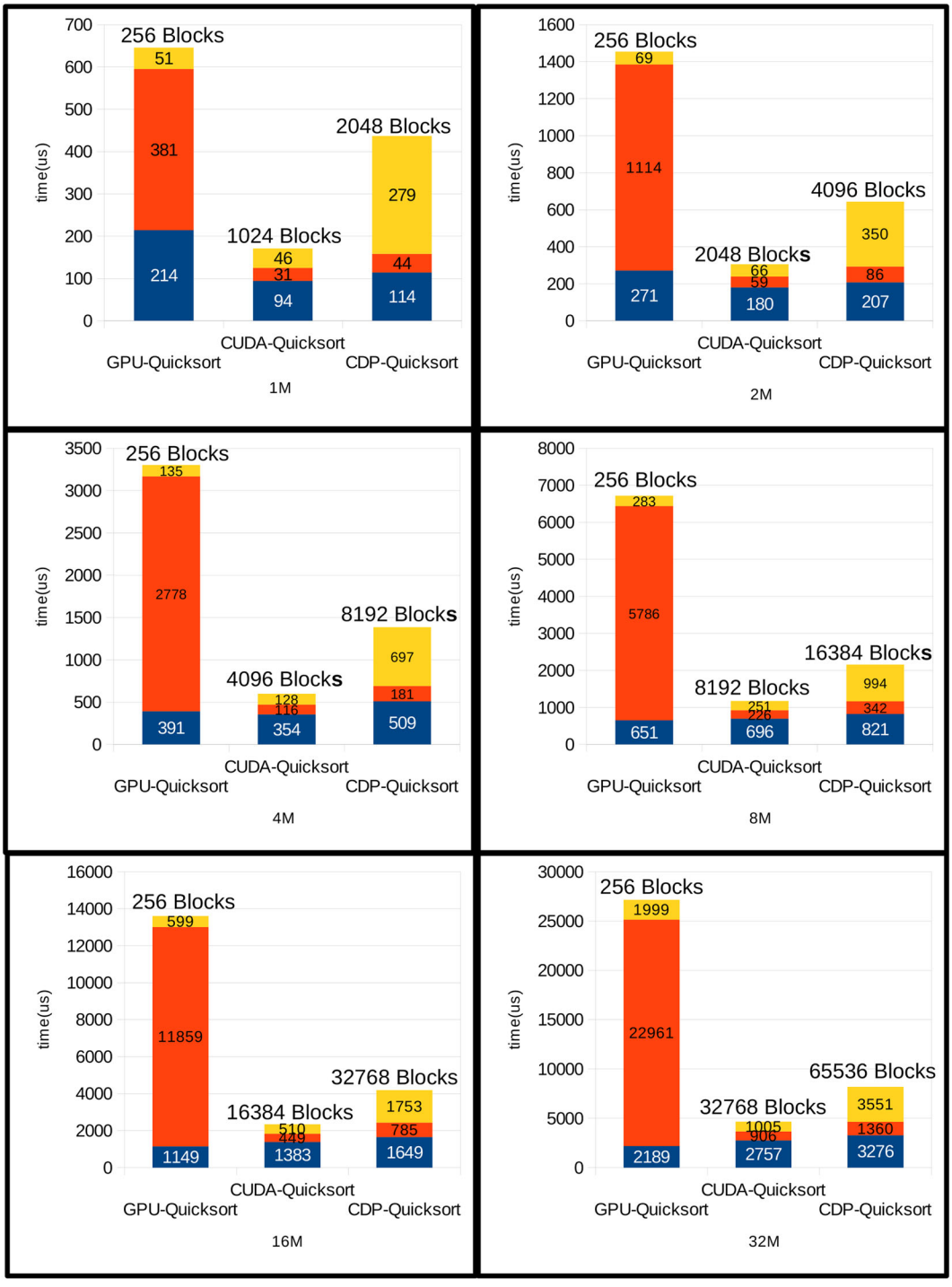
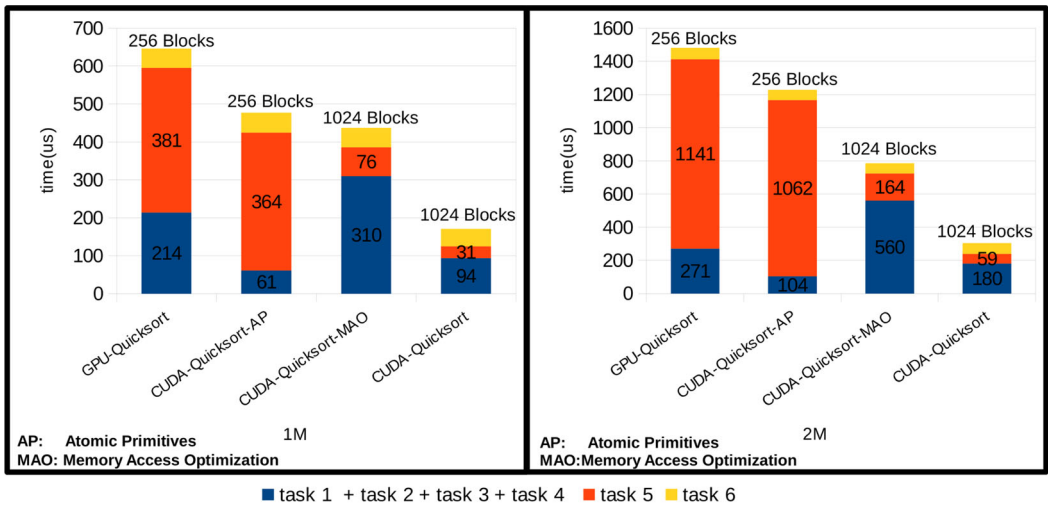
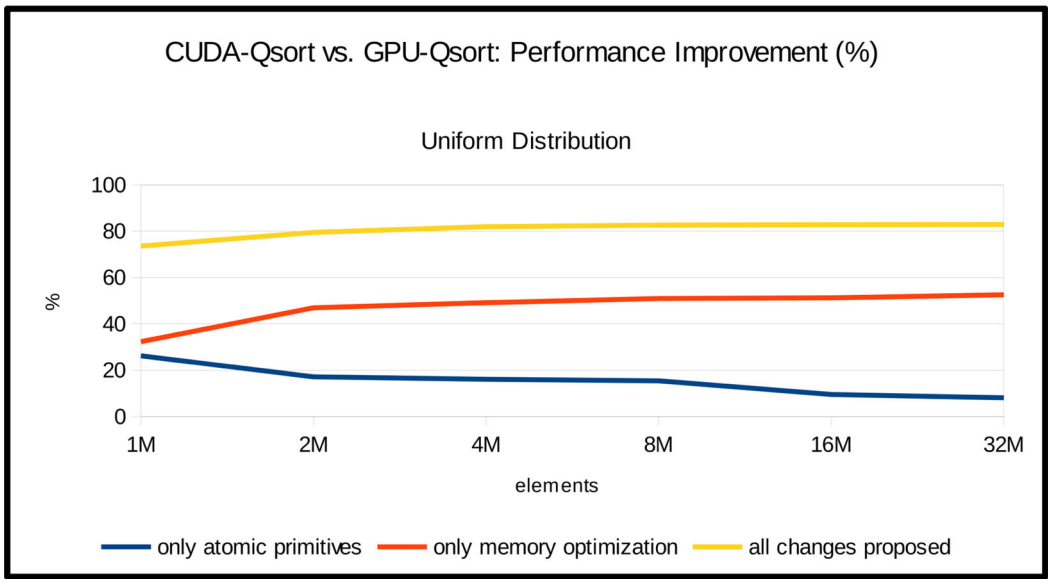


Figure 9. Time required to perform the following tasks in the first step when using a sorting benchmark uniform distribution. Task 1: picking out of the pivot; Task 2: sequence partitioning; Task 3: creation of two subsequences for each thread block; Task 4: thread block synchronization; Task 5: writing subsequences in the global memory; and Task 6: preparation for the next quicksort execution. Kernel size: GPU-quicksort and CUDA-quicksort (128 threads per block) and CDP-quicksort (512 threads per block). CUDA, compute-unified device architecture; CDP, CUDA dynamic parallel. GPU, graphics processing unit.

DOI: 10.1002/cpe



(a) Time required to perform the following tasks in the first step when using a sorting benchmark uniform distribution. *Task 1*: picking out of the pivot; *Task 2*: sequence partitioning; *Task 3*: creation of two subsequences for each thread block; *Task 4*: thread block synchronization; *Task 5*: writing subsequences in the global memory; *Task 6*: preparation for the next quicksort execution. CUDA-Qsort-256 uses only atomic primitives, CUDA-Qsort-1024 uses only memory access optimization, CUDA-Qsort uses all proposed changed. These three solutions and GPU-Qsort are perfomed by 128 threads per block.



(b) Percentage of performance improvement achieved by CUDA-Quicksort over GPU-Quicksort.

Figure 10. Comparison between the first step of the CUDA-quicksort and the first phase of the GPU-quicksort. GPU, graphics processing unit; CUDA, compute-unified device architecture.

Table I. Percentage of performance improvement achieved by CUDA-quicksort over GPU-quicksort.

Step 1	Step 2	Step 1 + Step 2
$\frac{60\%}{\text{Phase1:80\%Phase2:50\%}}$	15%	75%

CUDA, compute-unified device architecture. In the first column, CUDA-quicksort step 1 and GPU-quicksort step 1 are compared. Moreover, in the same column, CUDA-quicksort step 1 and GPU-quicksort phases 1 and 2 are compared. GPU, graphics processing unit.

DOI: 10.1002/cpe

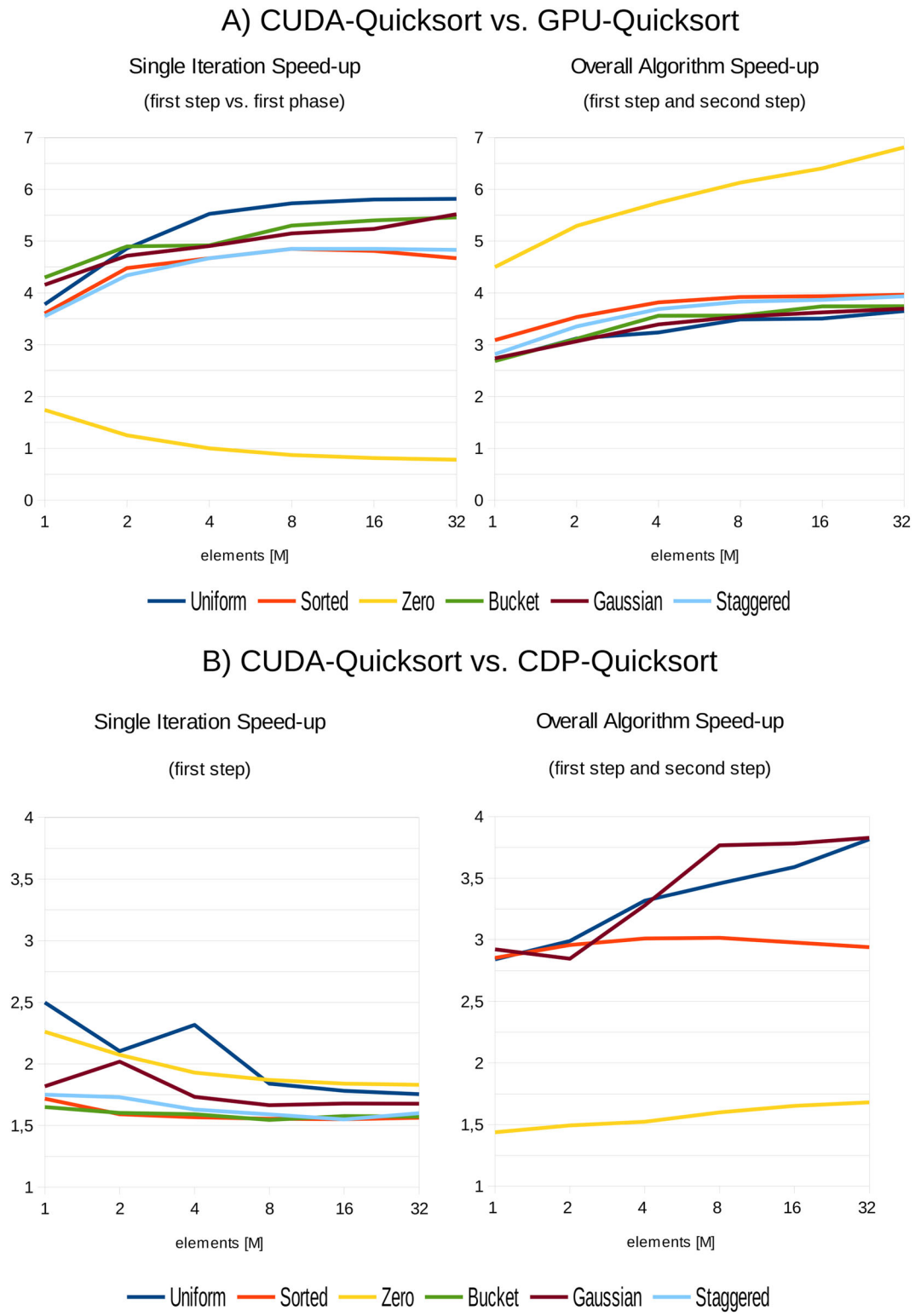


Figure 11. The single iteration speed-up of the CUDA-quicksort first step against the CDP-quicksort first step and the GPU-quicksort first phase is reported for different benchmarks on the left side. The speed-up of the overall algorithm (first and second steps) is reported on the right side. The Gaussian and staggered distributions are not shown in (B), as CUDA-quicksort is 60 times faster than CDP-quicksort. GPU, graphics processing unit; CUDA, compute-unified device architecture; CDP, CUDA dynamic parallel.

DOI: 10.1002/cpe

thread block is assigned to its own subsequence, eliminating the need for inter-block communications. Therefore, the performance improvement depends on memory optimization only. Indeed, the percentage of performance improvement achieved by the first step of CUDA-quicksort over the second phase of GPU-quicksort is the same obtained by the first step of CUDA-quicksort over the first phase of GPU-quicksort using only a memory optimization (orange curve of Figure 10b).

Figure 11 compares speed-up obtained over a single iteration with the overall one obtained with CUDA-quicksort – for all analyzed benchmarks. CUDA-quicksort resulted up to 5.8 times faster than GPU-quicksort and up to ∼1.9 times faster than CDP-quicksort on the first step, when sorting 32M uniformly distributed elements. Apart from the zero entropy one, similar results have also been obtained with the other distributions. However, considering the overall performance, CUDA-quicksort resulted ∼3.7 times faster than both GPU-quicksort and CDP-quicksort to sort 32M uniformly distributed items. In general, the overall speed-up achieved by CUDA-quicksort with respect to GPU-quicksort decreases with respect to that obtained in the first step for almost all benchmarks. This is mainly because the first step of CUDA-quicksort is only twice faster than the second phase of GPU-quicksort (i.e., 50% of performance improvement, see Table I). Conversely, the overall speed-up achieved by CUDA-quicksort versus CDP-quicksort increases in the first step

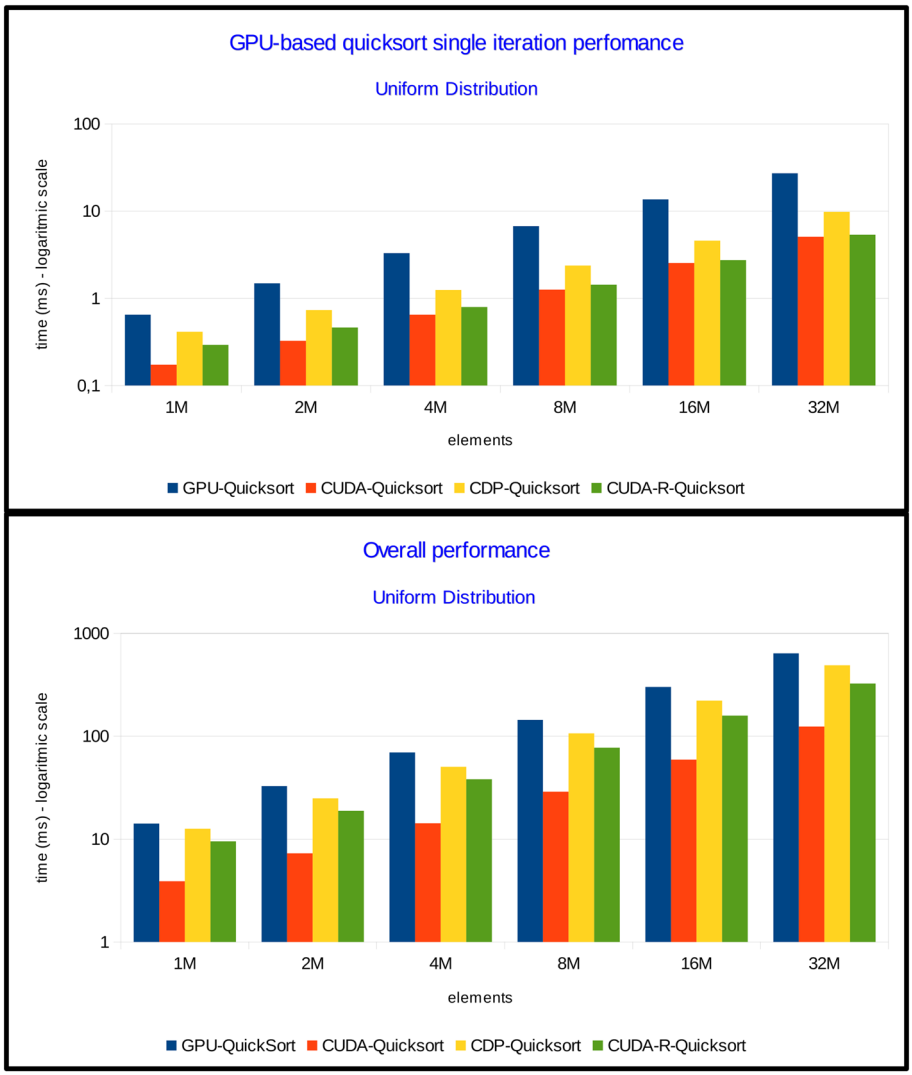


Figure 12. Comparison among CUDA-quicksort, CUDA-R-quicksort, GPU-quicksort, and CDP-quicksort using the uniform sorting benchmark distribution. GPU, graphics processing unit.

DOI: 10.1002/cpe

for almost all benchmarks. For the sake of completeness, it should be observed that the overall speed-up achieved by CUDA-quicksort versus CDP-quicksort is also related to the different policies used to choose the pivot (depending on the value of the pivot, the quicksort time complexity varies between $n \cdot log(n)$ and n^2).

In order to perform a straightforward comparison between CUDA-quicksort and CDP-quicksort and to assess the opportunity of exploiting the dynamic parallelism technology provided by CUDA, we developed a recursive version of the proposed algorithm called CUDA-R-quicksort.

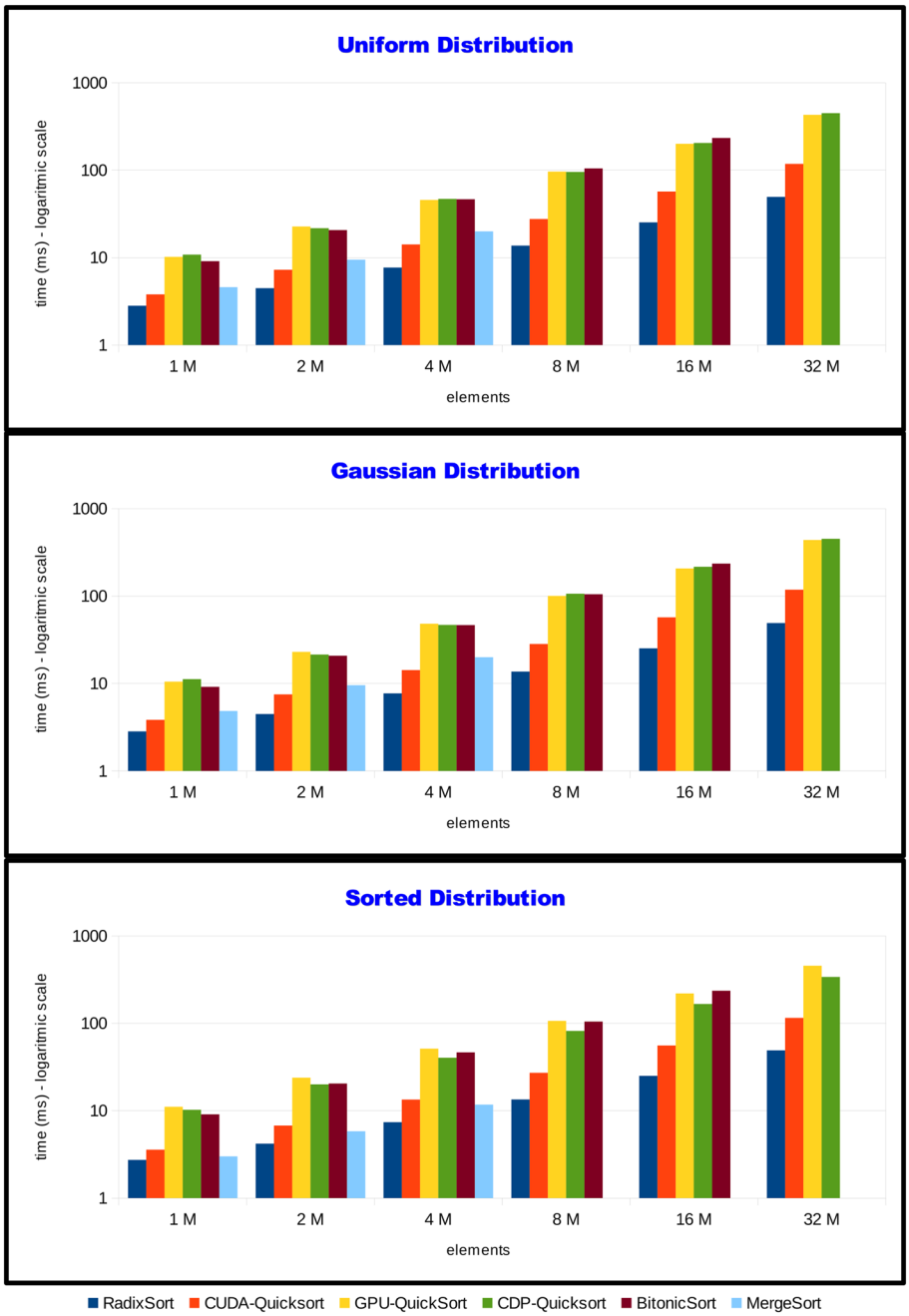


Figure 13. Comparison among CUDA-quicksort and the other GPU-based sorting algorithms using the uniform, gaussian, sorted sorting benchmark distributions. GPU, graphics processing unit; CUDA, compute-unified device architecture; CDP, CUDA dynamic parallel. MergeSort does not work when sorting 8M elements or more. BitonicSort does not work when sorting 16M elements or more. CDP, CUDA dynamic parallel. GPU, graphics processing unit; CUDA, compute-unified device architecture.

DOI: 10.1002/cpe

Experimental results (Figure 12) show that CUDA-quicksort and CUDA-R-quicksort are faster than the NVIDIA solution both on single-iterations and on the overall performance. CUDA-quicksort and CUDA-R-quicksort exhibit a similar computing time on single-iterations, whereas the overall performance of CUDA-quicksort resulted faster than its recursive version. Figure 12 reports the performance for the uniform distribution that is deemed representative of the behavior of other distributions.

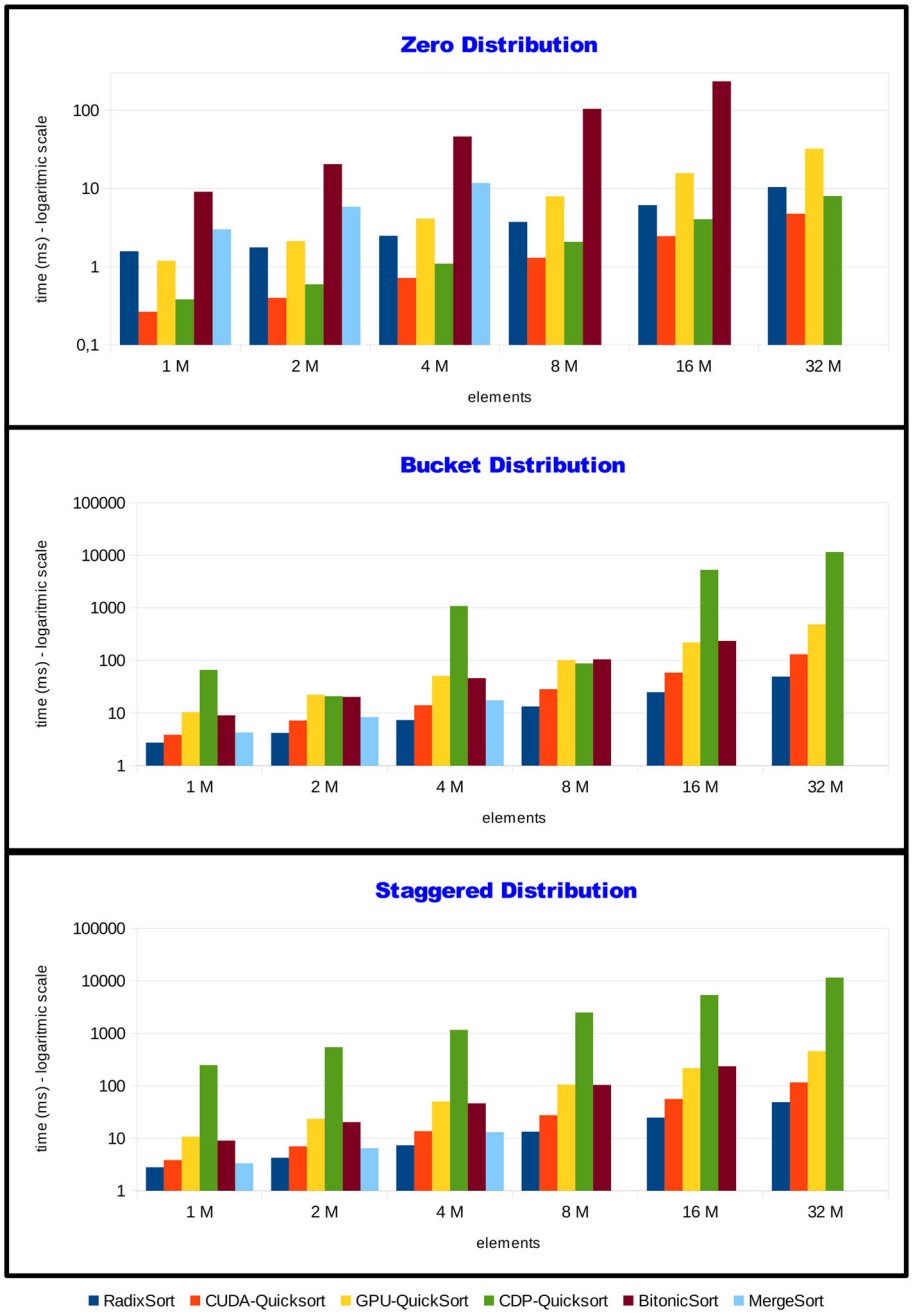


Figure 14. Comparison among CUDA-quicksort and the other GPU-based sorting algorithms using the zero, bucket, staggered sorting benchmark distributions. MergeSort does not work when sorting 8M elements or more. BitonicSort does not work when sorting 16M elements or more. CDP, CUDA dynamic parallel. GPU, graphics processing unit; CUDA, compute-unified device architecture.

DOI: 10.1002/cpe

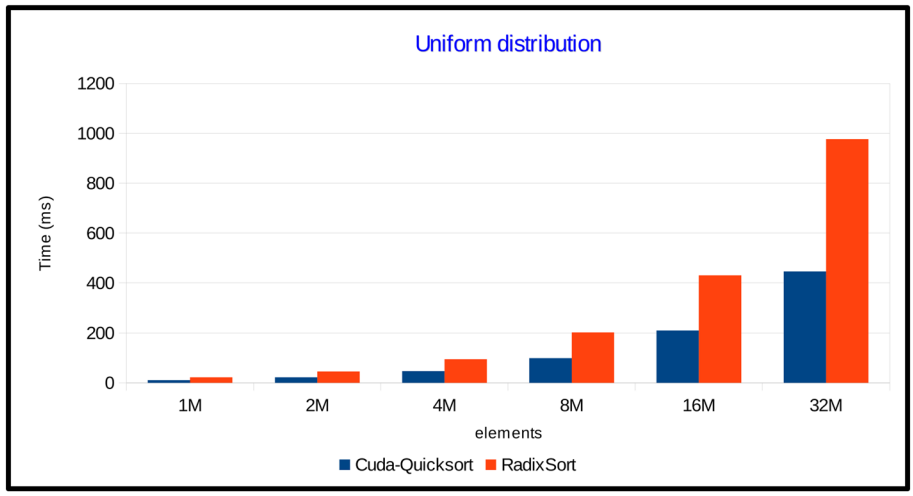


Figure 15. Comparison between CUDA-quicksort and RadixSort. CUDA, compute-unified device architecture.

Further experiments have been performed, aimed at comparing the performance of CUDA-quicksort with those of other cutting-edge GPU-based sorting algorithms. In particular, we compared CUDA-quicksort with the Radix Sort of the Thrust Library [22], based on [23], Bitonic Sort, and Merge Sort [23]. Bitonic Sort and Merge Sort are provided in the NVIDIA CUDA sample 6.0. All sorting algorithms used for benchmarking purposes are implemented in CUDA.

Figures 13 and 14 show that CUDA-quicksort outperforms almost all cited algorithms, but the Thrust Radix Sort. For the sake of completeness, let us recall that the computational complexity of the not comparison-based radixsort is $O(k \cdot n)$ for n keys with k or fewer digits whereas of quicksort is $O(n \cdot log(n))$. In the event that keys are longer than $O(log(n))$, quicksort outperforms radixsort. Conversely, if $k < O(log(n))$, radixsort performs better than quicksort.

We performed some experiments aimed at comparing the CUDA-quicksort and the Thrust Radix Sort to sort items with long keys (19 digits). Experiments performed on the uniform distribution benchmark show that CUDA-quicksort outperforms the Thrust Radix Sort achieving a speed-up ranging from 1.58x to 2.18x depending on the size of the dataset (Figure 15). Similar behavior has also been obtained with the other benchmarks.

6. CONCLUSIONS

We presented CUDA-quicksort, a new GPU-based implementation of quicksort based on the state-of-the-art GPU-quicksort. In order to improve the performance, we made three major changes to GPU-quicksort: (i) we designed a block-oriented iterative GPU-based implementation of quicksort that uses atomic primitives to perform inter-block synchronization; (ii) we optimized memory access; and (iii) we used a different third-party bitonic sort implementation to finalize the sorting process. The authors of GPU-quicksort mentioned the use of atomic primitives. However, at the time they released GPU-quicksort, atomic primitives were not widely supported, and they did not used them. Nowadays, atomic primitives are widely supported and have also been adopted by NVIDIA to design their CDP-quicksort.

Experimental results show that CUDA-quicksort is about four times faster than GPU-quicksort and that the major improvement depends on the optimized access to the global memory.

Results also show that CUDA-quicksort is about three times faster than the most recent CDP-quicksort of NVIDIA. We also implemented a recursive version of our CUDA-quicksort using the CDP technology used in the CDP-quicksort. Experiments show that also, in this case, our solution is faster than the CDP-quicksort although better performance are obtained with the iterative implementation. We also compared CUDA-quicksort with other cutting-edge GPU-based sorting algorithms.

DOI: 10.1002/cpe

CUDA-quicksort outperformed almost all of them. As expected, only Thrust Radix Sort outperformed CUDA-quicksort while sorting large sequences of integers. However, CUDA-quicksort outperformed Thrust Radix Sort when sorting large sequences of structured data.

CUDA-quicksort has also been successfully used to address a specific challenge in bioinformatics [24]. Based on the memory constraints we encountered in the cited work, we are planning to redesign the algorithm with the aim of providing a new release able to run on multiple GPUs.

ACKNOWLEDGEMENTS

The work has been supported by the Italian Ministry of Education and Research through the Flagship InterOmics (PB05) and HIRMA (RBAP11YS7K) projects and by the European Community through MIMOmics (305280) project.

REFERENCES

1. Cederman D, Tsigas P. A practical quicksort algorithm for graphics processors. *In the ACM Journal of Experimental Algorithmics (JEA)* 2009; **4**(14).
2. Hoare CAR. Quicksort. *Computer Journal* 1962; **4**(5):10–15.
3. CUDA Toolkit 6.0, documentation. (Available from: http://docs.nvidia.com/cuda/cuda-samples/index.html) [accessed on 10 September 2014].
4. NVIDIA Corporation. *NVIDIA CUDA Programming Guide, Version 6.0*, 2014.
5. Manconi A, Orro A, Manca E, Armano G, Milanesi L. *GPU-BSM: A GPU-Based Tool to Map Bisulfite-Treated Reads*. PloS one, 9(5), Public Library of Science, pp. e97277, 2014.
6. Sengupta S, Harris M, Zhang Y, Owens JD. Scan primitives for GPU computing. In Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, San Diego, California, 2007; 97–106.
7. Heidelberger P, Norton A, Robinson JT. Parallel quicksort using fetch-and-add. *IEEE Transactions on Computers* 1990; **39**(1):133–138.
8. Tsigas P, Zhang Y. A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. *Proceedings of the 11th Euromicro Conference on Parallel Distributed and Network-based Processing*, Genova, Italy, 2003; 372–381.
9. *Parallel bitonic sort algorithm*. (Available from: http://www.tools-of-computing.com/tc/CS/Sorts/bitonic_sort.htm) [accessed on 10 September 2014].
10. Purcell J, Donner C, Cammarano M, Jensen HW, Hanrahan P. Photon mapping on programmable graphics hardware. *Proceedings of the 2003 Annual ACM SIGGRAPH/Eurographics Conference on Graphics Hardware (EGGH 03)*, Aire-la-Ville, Switzerland, 2003; 41–50.
11. Kapasi UJ, Dally WJ, Rixner S, Mattson PR, Owens JD, Khailany B. Efficient conditional operations for data-parallel architectures. *In Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (Micro-33)*, New York, USA, 2000; 159–170.
12. Kipfer P, Segal M, Westermann R. Uberow: a GPU-based particle engine. *In Proceedings of the 2004 ACM SIGGRAPH/Eurographics Conference on Graphics Hardware (HWWS '04*, New York, USA, 2004; 115–122.
13. Kipfer P, Westermann R. GPU Gems 2: In *Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, Ch. Improved GPU sorting, 2005; 205–222.
14. Govindaraju NK, Raghuvanshi N, Manocha D. Fast and approximate stream mining of quantiles and frequencies using graphics processors. *In Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*, New York, USA, 2005; 611–622.
15. Dowd M, Perl Y, Rudolph L, Saks M. The periodic balanced sorting network. *Journal of the ACM* 1989; **36**(4): 738–757.
16. Govindaraju NK, Raghuvanshi N, Henson M, Manocha D. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, University of North Carolina, Chapel Hill, 2005.
17. Greß A, Zachmann G. GPU-Abisort: Optimal Parallel Sorting on Stream Architectures. *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06. IEEE Computer Society, Washington, DC, USA, 2006; 45–45. (Available from: http://dl.acm.org/citation.cfm?id=1898953.1898980) [accessed on 10 September 2014].
18. Sintorn E, Assarsson U. Fast parallel GPU-sorting using a hybrid algorithm. *Journal Parallel Distributed Computing* October 2008; **68**(10):1381–1388. (Available from: http://dx.doi.org/10.1016/j.jpdc.2008.05.012) [accessed on 10 September 2014].
19. Leischner N, Osipov V, Sanders P, 2009. Gpu Sample Sort. CoRR abs/0909.5649.
20. Harris M, Sengupta S, and Owens JD. Parallel prex sum (scan) with CUDA. In *H. Nguyen, Editor, GPU Gems 3*. Addison Wesley, 2007.
21. Helman DR, Bader DA, Jaja J. A randomized parallel sorting algorithm with an experimental study. *Journal Parallel Distributed Computing* 1998; **1**(52):1–23.
22. Hoberock J, Bell N. Thrust: A Parallel Template Library, 2010.

DOI: 10.1002/cpe

23. Satish N, Harris M, Garland M. Designing efficient sorting algorithms for manycore GPUs. In *Parallel Distributed Processing. IPDPS 2009. IEEE International Symposium on,* Rome, Italy, May 2009; 1–10.
24. Manconi A, Manca E, Orro A, Armano G, Gnocchi M, Moscatelli M, Milanesi L. *G-CNV: A GPU-Based Tool for Preparing Data to Detect CNVs with Read Depth Methods*. Front. Bioeng. Biotechnol, 2015. 3:28. doi: 10.3389/fbioe.2015.00028.

DOI: 10.1002/cpe