Computer Science • 14 (4) 2013 http://dx.doi.org/10.7494/csci.2013.14.4.679

Dominik Żurek
Marcin Pietroń
Maciej Wielgosz
Kazimierz Wiatr

THE COMPARISON OF PARALLEL SORTING ALGORITHMS IMPLEMENTED ON DIFFERENT HARDWARE PLATFORMS

Abstract *Sorting is a common problem in computer science. There are a lot of well-known sorting algorithms created for sequential execution on a single processor. Recently, many-core and multi-core platforms have enabled the creation of wide parallel algorithms. We have standard processors that consist of multiple cores and hardware accelerators, like the GPU. Graphic cards, with their parallel architecture, provide new opportunities to speed up many algorithms. In this paper, we describe the results from the implementation of a few different parallel sorting algorithms on GPU cards and multi-core processors. Then, a hybrid algorithm will be presented, consisting of parts executed on both platforms (a standard CPU and GPU). In recent literature about the implementation of sorting algorithms in the GPU, a fair comparison between many core and multi-core platforms is lacking. In most cases, these describe the resulting time of sorting algorithm executions on the GPU platform and a single CPU core.*

Keywords parallel algorithms, GPU, OpenMP, CUDA, sorting networks, merge-sort

1. Introduction

Sorting is one of the most fundamental problems in computer science, as it is used in most software applications. Data-driven algorithms especially use sorting to gain an efficient access to data. Many sorting algorithms with distinct properties for different architectures have been developed. It should be noted that some sorting algorithms for sequential hardware platforms should be modified when implemented on parallel architectures. Some algorithms have a parallel structure, making them easier to adapt for parallel hardware architecture. The best-known sorting algorithms with parallel structures are merge sort, parallel quick-sort (hyper-quick-sort), odd-even sort and bitonic sort as an example of sorting networks. The main challenge in constructing optimized algorithms on hardware accelerators is choosing and adapting appropriate algorithms for specific hardware architectures. Such algorithms are known as hybrid sorting algorithms. There are a few well-known hybrid algorithms on GPU cards and multi-core systems [1, 2, 3, 5, 9]. The main drawback of them is the lack of a fair time comparison between hardware platforms. The goal of this article is to show a short review and analysis of sorting algorithms which we used while working with data mining algorithms. It can give only advice which algorithms are appropriate for building parallel algorithms for specific hardware, and to show the regularity in which efficient GPU sorting can be faster than single-core sorting (mostly described in lot of articles) yet less efficient than modern multi-core processors. This research is not sufficient enough and will be further investigated (especially sorting based on histogram computing as fast no-comparison algorithm on multi-core platform and adaptive bitonic [5] on a GPU).

2. Parallel hardware platforms

Modern processors consist of two or more independent central processing units. This architecture enables multiple CPU instructions (add, move data, branch etc.) to run at the same time. The cores are integrated into a single integrated circuit. The manufacturers AMD and Intel have developed several multi-core processors (dual-core, quad-core, hexa-core, octa-core etc.). The cores may or may not share caches, and they may implement message passing or shared memory inter-core communication. Homogeneous multi-core systems include only identical cores, and heterogeneous multi-core systems have cores that are not identical. The single cores in multi-core systems may implement architectures such as vector processing, SIMD, or multi-threading. These techniques offer another aspect of parallelization (implicit to high level languages, used by compilers). The performance gained by the use of a multi-core processor depends on the algorithms used and their implementation. Most important are how many parts of code can be run simultaneously as well as the frequency of communication between parallel threads or processes.

The graphical processor unit has a multiprocessor structure. The cores in each multiprocessor share an Instruction Unit with other cores. Multiprocessors (SM in

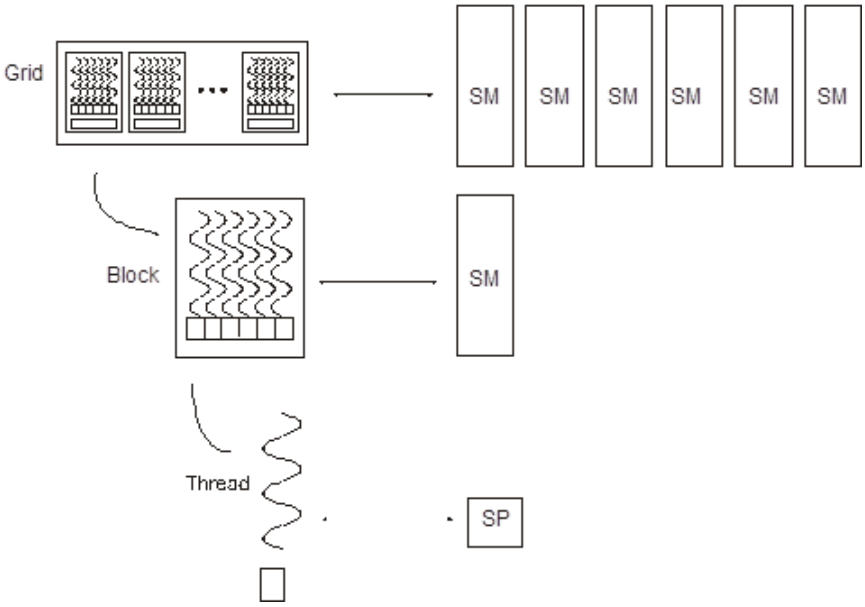

Figure 1. Relation between Stream Multiprocessors and Cuda threads grouped in blocks and grids.

Fig. 1) have read-only constant/texture memory and shared memory which are much faster than global memory (common to all multiprocessors). GPU cards enable thousands of parallel threads to run (SP in Fig. 1), which are grouped in blocks with shared memory. The blocks are grouped in a grid (Fig. 2). The main aspects are the usage of memory, an efficient dividing code to parallel threads, and thread communications. As mentioned earlier, constant/texture and shared memories are the fastest. Therefore, programmers should optimally use them to speedup access to data on which an algorithm operates.

3. Parallel programming models

Parallel hardware platforms can be programmed by high-level programming frameworks. These programming frameworks are based on high-level languages like C language, with built-in mechanisms to exploit parallelism from specific hardware platforms. In our implementation, the OpenMP environment and CUDA framework were used.

3.1. CUDA programming framework

CUDA is a software architecture that enables the graphics processing unit (GPU), to be programmed using high-level languages such as C and C++. CUDA requires NVIDIA GPU cards [7]. CUDA provides three key mechanisms to parallelize programs: thread group hierarchy, shared memories, and barrier synchronization. These

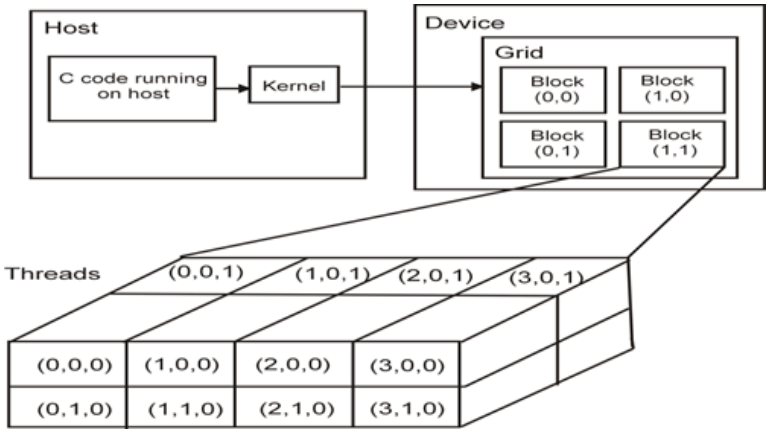

Figure 2. Threads groups in blocks and threads [7].

mechanisms provide fine-grained parallelism nested within coarse-grained task parallelism. Creating the optimized code on GPU cards is not a trivial task, so thorough knowledge about this hardware accelerator architecture is needed. The main issues to solve are usage of memory, efficiency of dividing code to parallel threads, and thread communications. Programmers should optimally use them to speedup access to data on which an algorithm operates. Another important aspect is to optimize synchronization and communication between the threads. Synchronization of the threads between blocks is much slower than within a single block. If necessary, it should be solved by the sequential running of multiple kernels. Another important aspect is the fact that recursive function calls are not allowed in CUDA kernels. Providing stack space for all the active threads requires substantial amounts of memory.

3.2. OpenMP programming environment

OpenMP[8] is a concurrency platform for multi-threaded, shared-memory parallel processing multi-core architectures for C, C++ and Fortran languages. By using OpenMP, the programmer does not need to create the threads nor assign tasks to each thread. The programmer inserts directives to assist the compiler into generating threads for the parallel processor platform. OpenMP is a higher-level programming model compared to pthreads in the POSIX library. The OpenMP consists of the following major components:

1) compiler directives – which allow the programmer to instruct the compiler thread creation, data management, thread synchronization, etc. Most popular are: atomic (memory location that must be updated atomically), barrier (synchronization of all threads in the same region), critical (defines critical section executed by single thread at a time), for (defines for loop iterations should be run in parallel), and parallel (defines region of the code that will be run by multiple threads).

Each OpenMP directive can be followed by a collection of clauses which mainly define thread variables and their access policy,

2) runtime library functions – control the parallel execution environment, control and monitor threads, control and monitor processors,
3) environment variables – variables to alter the execution of OpenMP applications.

The most important advantage of the OpenMP framework is that the programmer does not have to restructure the sequential source code. The process of making parallel version only consists of insertion appropriate compiler directives to restructure the serial program to a parallel one.

4. Implemented sorting algorithm on a GPU

There are a few parallel sorting algorithms adapted to GPU cards. Most of them are modifications of standard, well-known sorting algorithms adapted to GPU hardware architecture. Sintorn [1] proposed a modified merge-sort, Peters [9] implemented an adaptive bitonic sorting algorithm with a bitonic tree based on a tables structure. Cederman [5] adapted a quick sort for the GPU platform. Our sorting algorithm is based on the bitonic sort as the most efficient way of sorting elements on GPU cards. The reason is that the sequence of comparisons in this sorting algorithm does not depend on the order in which the data is presented (in most sorting algorithms, stages depend on the order). It also allows easy and flexible partitioning a group of sorted elements between multiprocessors (blocks). Bitonic sort is a sorting-network algorithm developed by Batcher [6]. A sorting network is a sorting algorithm where the sequence of comparisons is predetermined and data-independent. The Bitonic algorithm sorts a sequence of N elements and consists of logN stages. In the first stage, pairs of subsequences of length 1 are merged, which results in sorted subsequences of length 2. In the second stage, pairs of these sorted subsequences are merged, resulting in sorted subsequences of length 4. In each stage, the merging process of two bitonic subsequences is performed. A bitonic sequence is a concatenation of two subsequences sorted in opposite directions. A formal description of the algorithm is defined by the following equations:

If the following sequence $(E_0, E_1, \ldots, E_{M-1})$ is a bitonic sequence of a length M then we can define:

$$L(E) = (\min(E_0, E_{M/2}), \min(E_1, E_{M/2+1}), \ldots, \min(E_{M/2-1}, E_{M-1})) \quad (1)$$

$$U(E) = (\max(E_0, E_{M/2}), \max(E_1, E_{M/2+1}), \ldots, \max(E_{M/2-1}, E_{M-1})) \quad (2)$$

where equation (1) is a bitonic sequence of lower elements, equation (2) is a bitonic sequence of upper elements.

1) bitonic sequence of the lower elements,
2) bitonic sequence of the upper elements.

As it is seen, the steps in each stage can be parallelized. The stages are dependent because of the relation between the data of which they access.

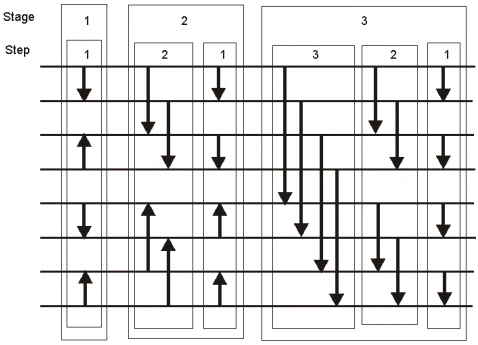

Figure 3. Bitonic sorting.

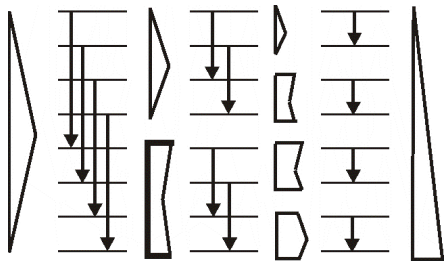

Figure 4. Merging bitonic sequences.

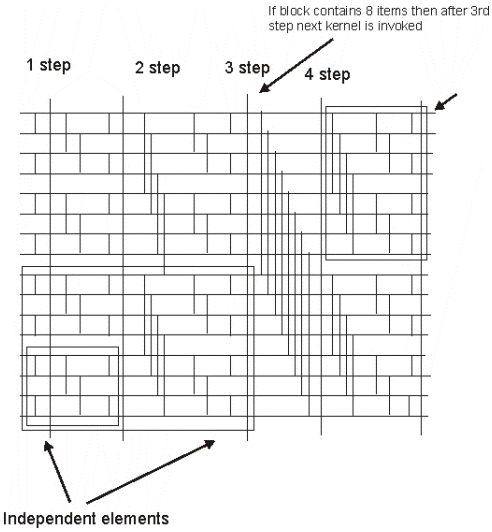

Figure 5. Bitonic sort algorithm on the GPU hardware platform.

In our research, we achieved the best results with algorithms based on sorting networks. The algorithm has only logN stages which must be synchronized compared to N synchronizations in odd-even sort. Each parallel thread reads and compares a constant number of elements in every step, unlike the mergesort and quicksort algorithms where the number of independent comparisons in each step is different. The created sorting algorithm is a kind of hybrid bitonic sort adapted to a many-core architecture. Our GPU sorting algorithm is fully based on the bitonic sorting method as described in Figure 3 and Figure 4. As described in Figure 5, elements in the sorting network can be grouped into independent partitions. Groups in the same stages can be run parallel. Therefore, a set of grouped elements can be mapped to the device block (in example in Fig. 5, each block contains 8 elements) and sorted by

its parallel threads (1, 2, and step 3 in Fig. 5). After that, additional kernel functions must be invoked to merge elements between the sorted blocks until all blocks contain bitonic sequences (in Fig. 5, comparisons between steps 3 and 4). Then, in each block, a merging process is executed to sort the elements. These steps of the algorithm are run until all input data in the global memory is sorted. The best results of the presented algorithm were achieved (Tesla m2090 graphic card) when each thread was responsible for two comparisons in each stage (each thread in one access to memory reads 4 elements which minimize memory delays). The maximal number of threads in each block is 512 (Nvidia Tesla m2090), so the number of elements stored in each block is 4096 integer values. At the beginning, the threads read two pairs of elements from the global memory and then compare each pair. After each stage, the threads are synchronized (Fig. 3 and Fig. 4). After the last step, the threads write the results of the comparison back to the global memory. The obtained computational complexity of the algorithm is the equal sequential complexity divided by the number of parallel multiprocessors – $n/p * log^2 n$. Results of this algorithm are presented in section 7.

5. Implemented sorting algorithms on multi-core hardware platforms

In scientific literature, fair comparisons of sorting algorithms between GPUs and multi-core platforms are lacking. Therefore, an efficient parallel algorithm was implemented in the OpenMP environment to compare its efficiency with GPU-based sorting algorithms.

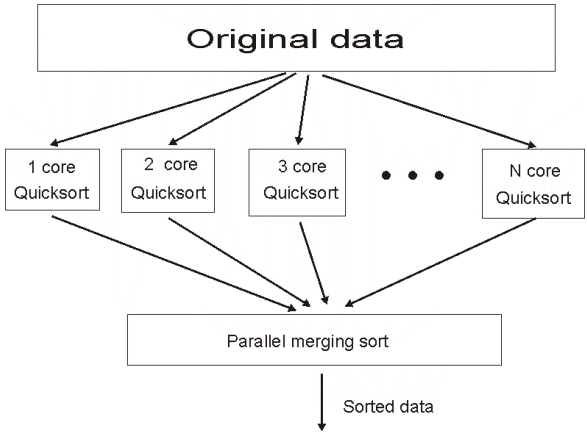

Figure 6. Parallel Merge-sort algorithm.

The most-efficient one is the quick-merge hybrid algorithm (average computational complexity – $n * log(n)$, memory complexity – $2 * n$). In this case, each core sorts partial data (data is divided into equal parts for each core) by the quick-sort algorithm. Then, the results of each core are merged by an efficient merge-sort algorithm

(Fig. 6). Merge-sort is a type of sorting algorithm that, before sorting, the data is separated (divided), each partial component is sorted, then all sorted components are pasted together in a cycle to sort all of the data (Fig. 7). During a cyclical call, it goes through three steps: divide, conquer, and paste. The algorithm has logN steps when sorting N elements. In each step, the size of partial components doubles.

Each stage can be parallelized by dividing the independent sorting of each two components to different computing units (Fig. 7). In our algorithm, the first stage of merge-sort has partial components equal to the number of elements sorted by each core in the previous step (quicksort). The algorithm has computational complexity equal $n * log(c) + n/c * log(n/c)$, where c is the number of cores. Its complexity is equal to parallel quicksort [11]. Its main advantage is that each thread receives equal size partition of input data. In case of the parallel quicksort algorithm, the size of partition in each stage depends on the distribution of sorted data. The next analyzed algorithm is a bucket sort. In this algorithm, minimal and maximal values must be found at the beginning.

Data is divided into buckets (the equal intervals of the range of sorted values, the number of buckets is equal to the number of cores). When dividing the data number of elements stored in each bucket is counted. Then, each core sorts a single bucket using the quick-sort algorithm. The number of elements in a bucket is a parameter of the quick-sort method. The computational complexity is $2 * n + n/c * log(n/c)$, where c is the number of cores. The drawback of this parallel algorithm is that the lots of memory must be reserved for each bucket.

6. Implementation of hybrid sorting algorithms based on the CPU and GPU

In case of a small amount of input data, the GPU-sorting algorithms are faster than multi-core ones; but when the amount of data increases, the multi-core sorting algorithms become more efficient than GPU-based algorithms (section 7). Therefore, an algorithm consisting of two main steps was developed. The first one executed on a GPU and the second on a CPU. The algorithm sends all original data that must be sorted from the host (CPU) to a GPU. All threads from each block read and transfer parts of data to the shared memory (Fig. 8). After that, each block has 4096 integer values to sort. This is the maximal amount of data that can be stored in the shared memory (the number of elements must be to the power of 2). Then, the data in each block is sorted by the bitonic algorithm.

After that, the data is sent to the host and merged by the efficient merge-sort algorithm (Fig. 8). The merge-sort is run with a gap equal to 4096 because this is the size of the started partial components. The computational complexity of the whole algorithm is $n/c * log(c) + n/p * log^2 n$, where c is the number of cores and p is the number of multiprocessors on the GPU. Further algorithms can be modified in such a way that more partial data can be sorted on a GPU. This demands inter-block bitonic merging (multi kernel invocation needed).

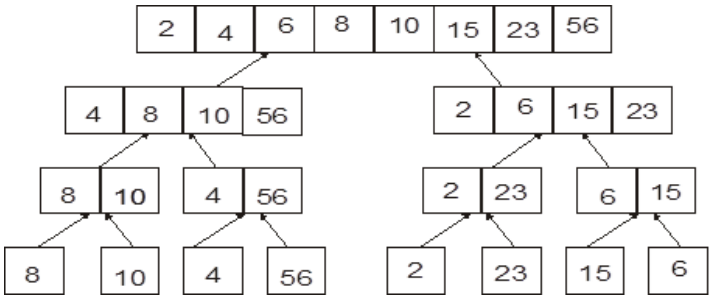

Figure 7. Quick-merge parallel algorithm.

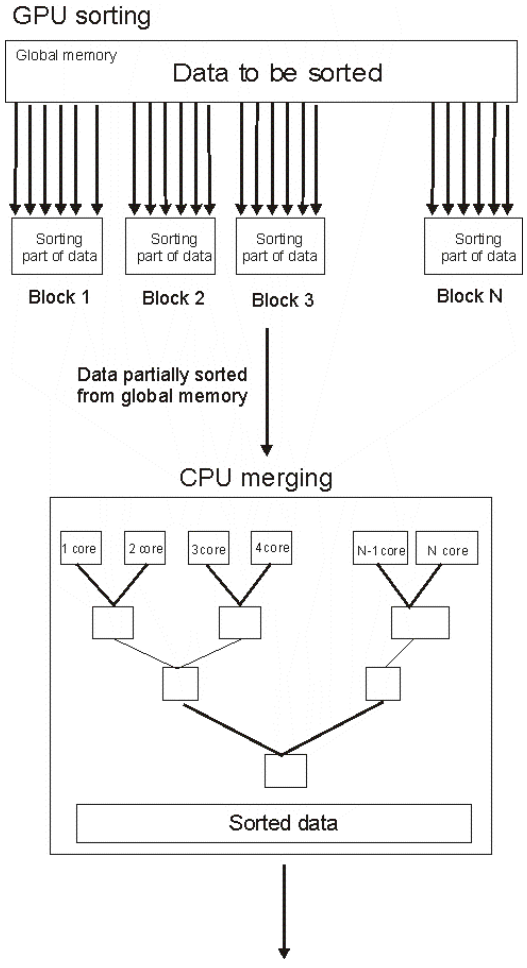

Figure 8. Implemented hybrid sorting algorithm.

7. Results

As shown in Figure 9, our hybrid algorithm (sequential merging on a CPU) is up to four times faster than the optimized sequential quick-sort on a CPU. This comparison is useful when the programmer can use in his application GPU and single-core CPU. In this case, a hybrid algorithm is the most efficient solution.

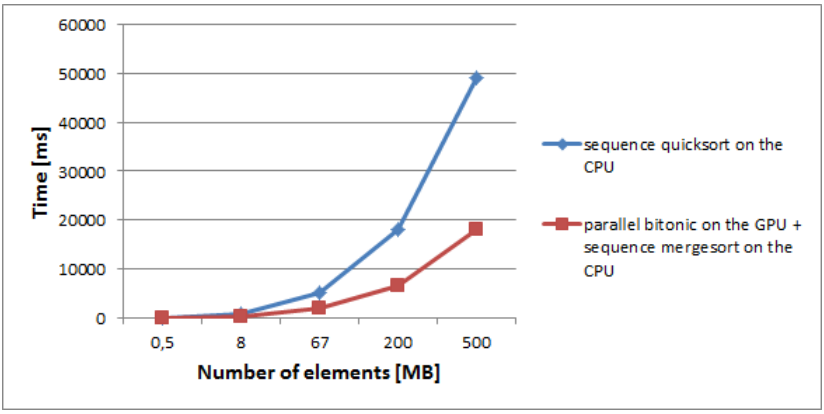

Figure 9. Comparison of hybrid algorithm to quicksort on a single-core CPU.

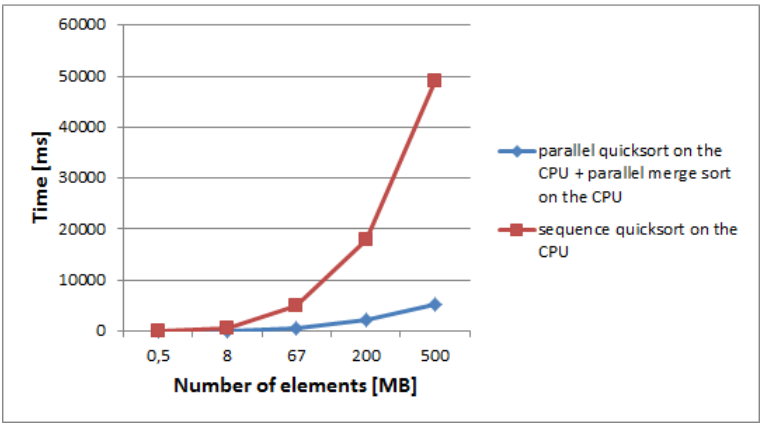

Figure 10. Comparison of multicore parallel quicksort (8-core) to quicksort on a single-core.

Figure 10 presents the results of a quick-merge parallel algorithm on a multi-core processor compared with a sequential quicksort algorithm (shows its scalability). All results were measured on a 8-core processor. Figure 11 compares a hybrid algorithm with a parallel quick-merge algorithm. Figure 12 describes the results of GPU hybrid bitonic sorting (section 4) compared with a sequential quicksort algorithm. The results

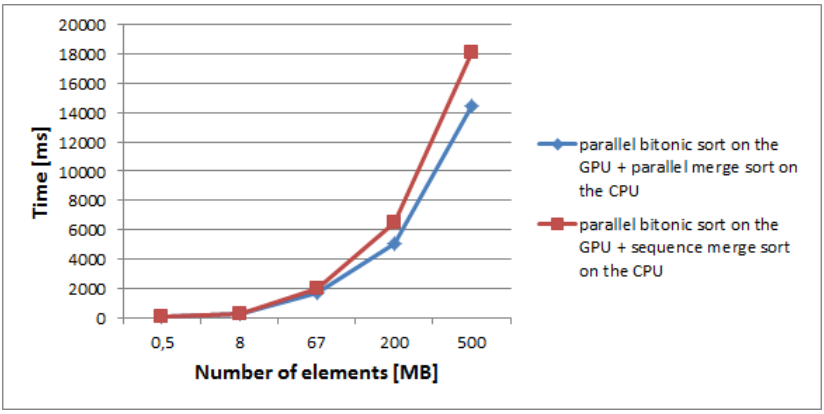

Figure 11. Chart of results of hybrid algorithm and parallel quick-merge.

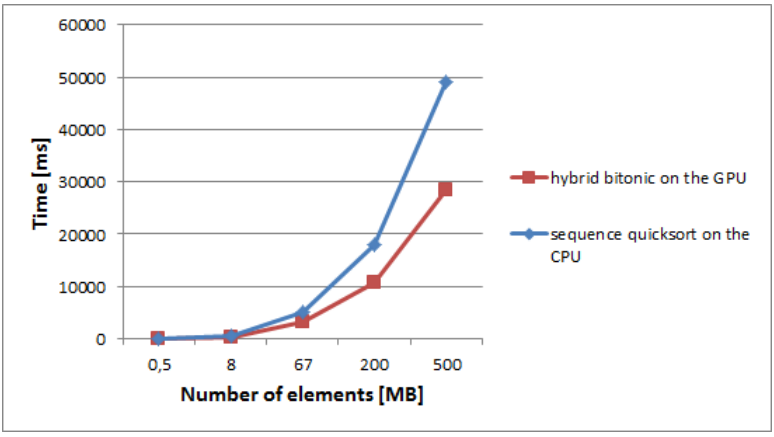

Figure 12. Results of hybrid bitonic sort on the GPU.

achieved by the hybrid bitonic algorithm are similar to the algorithms presented in [2]. The most effcient GPU-sorting algorithms found in the scientific literature are based on an adaptive bitonic sort [4, 5, 9] ($n/p*log(n)$ computional complexity). The analysis of their efficiency shows that they can achieve double speedup compared to algorithms based on an original bitonic sort. All algorithms were run and tested on a NVIDIA Tesla m2090 and an Intel Xeon E5645 8-core processor.

8. Conclusions

Our research in solving the sorting problem on many-core and multi-core hardware platforms shows that multi-core sorting algorithms are the most efficient and best scalable. The GPU sorting algorithms, compared to a single core, are up to four times

faster than the optimized quick sort algorithm. The implemented hybrid algorithm (executed partially on CPU and GPU) is more efficient than algorithms only run on GPU (despite transfer delays) but a little slower than the most efficient quick-merge parallel CPU algorithm.

Acknowledgements

This scholarly work was made thanks to POWIEW project. The project is co-funded by the European Regional Development Fund (ERDF) as a part of the Innovative Economy program (POIG.02.03.00-00-018/08).

References

[1] Sintorn E., Assarson U.: Fast parallel GPU-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, vol. 68, pp. 1381–1388, 2008.

[2] Cederman D., Tsigas P.: A practical quick-sort algorithm for graphics processors. *Proc. of the 16th annual European Symposium on Algorithms*, Heidelberg, pp. 246–258, 2008, Springer-Verlag.

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

[4] Greß N. K., Zachmann G.: Gpu-abisort: Optimal parallel sorting on stream architectures. *Proc. of the 20th IEEE International Parallel and Distributed Processing Symposium IPDPS*, Heidelberg, 2006.

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

[6] Batcher K. E.: Sorting networks and their applications. *Proc. of the AFIPS'68*, April 30-May 2, pp. 307–314, New York, NY, USA.

[7] NVIDIA: *NVIDIA CUDA. Compute Unified Device Architecture-Programming Guide.* 2007.

[8] OpenMP: OpenMP website – `www.openmp.org`.

[9] Peters H., Schulz-Hildebrandt O., Luttenberger N.: A Novel Sorting Algorithm for Many-core Architectures Based on Adaptive Bitonic Sort. *Proc. of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 227–237, IEEE Computer Society Washington, DC, USA, 2012.

[10] Szymczyk M.: Równoległy algorytm typu quicksort. *Automatyka*, vol. 1, z. 1, pp. 402–408, 1997.

[11] Sequential and parallel algorithms – `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/algoen.htm`

Affiliations

Dominik Żurek
ACC AGH Cyfronet, Krakow, Poland, dominik.zurek1102@gmail.com

Marcin Pietroń
ACC AGH Cyfronet, Krakow, Poland, pietron@agh.edu.pl

Maciej Wielgosz
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, ACC AGH Cyfronet, Krakow, Poland, wielgosz@agh.edu.pl

Kazimierz Wiatr
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, ACC AGH Cyfronet, Krakow, Poland, wiatr@agh.edu.pl

Received: 3.09.2013
Revised: 16.10.2013
Accepted: 16.10.2013